

Neuroevolution Through Erlang

Gene I. Sher

Department of EECS, University of Central Florida

CorticalComputer@gmail.com

Outline

- Introduction
- Creating The Perfect Neural Network Programming Language
- From Telecommunications Networks To Neural Networks
- DXNN: A Case Study
- Beyond The Horizon
- Conclusion

Objectives

- Elaborate on why Erlang is such a great fit for the field of computational intelligence, and its future.
- Discuss the first of such Erlang implemented, general topology and parameter evolving universal learning networks.
- Promote Erlang within the Scientific community for Neural Network, Computational Intelligence, and Multi-Agent Based System research and applications.

Introduction

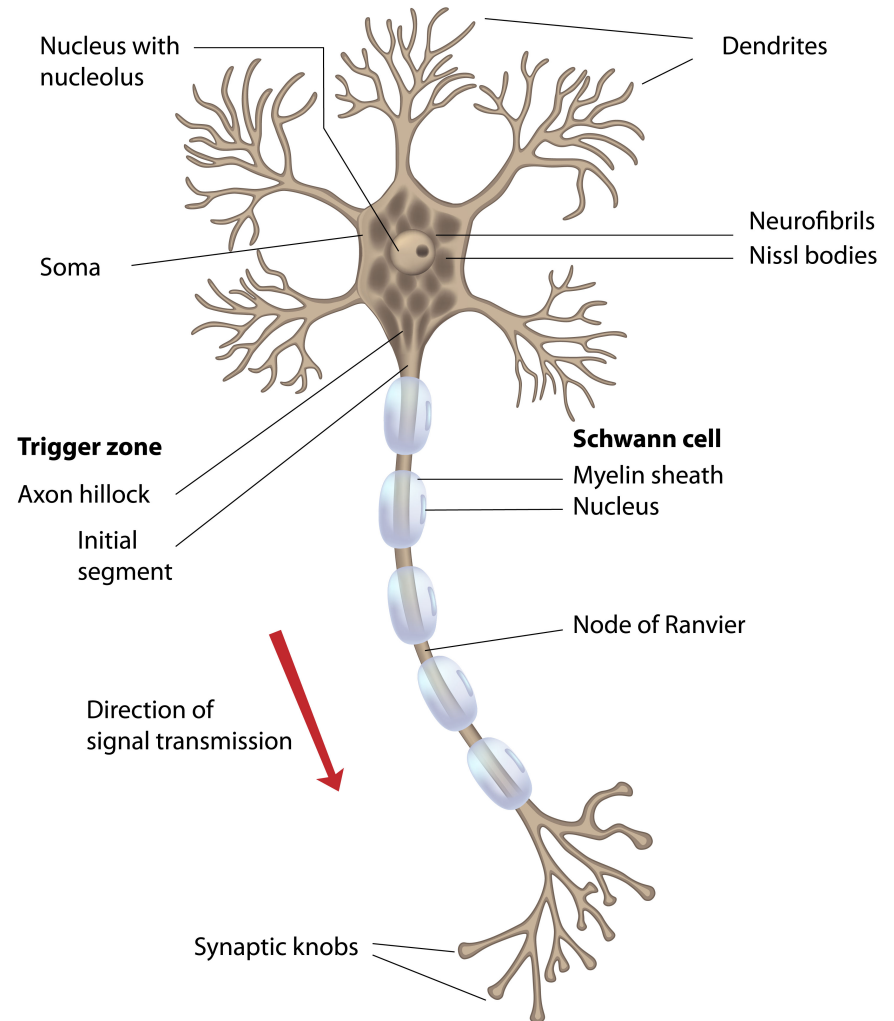
- Neural Networks are graph based, distributed, learning systems
- Hardware is moving forward, towards many-core architectures
 - Xeon Phi
 - Tiler64
 - ...
- Other programming languages leave a conceptual gap between themselves and the problem domain of neural network based computational intelligence
- What is the right programming language architecture?

Biological Neural Network

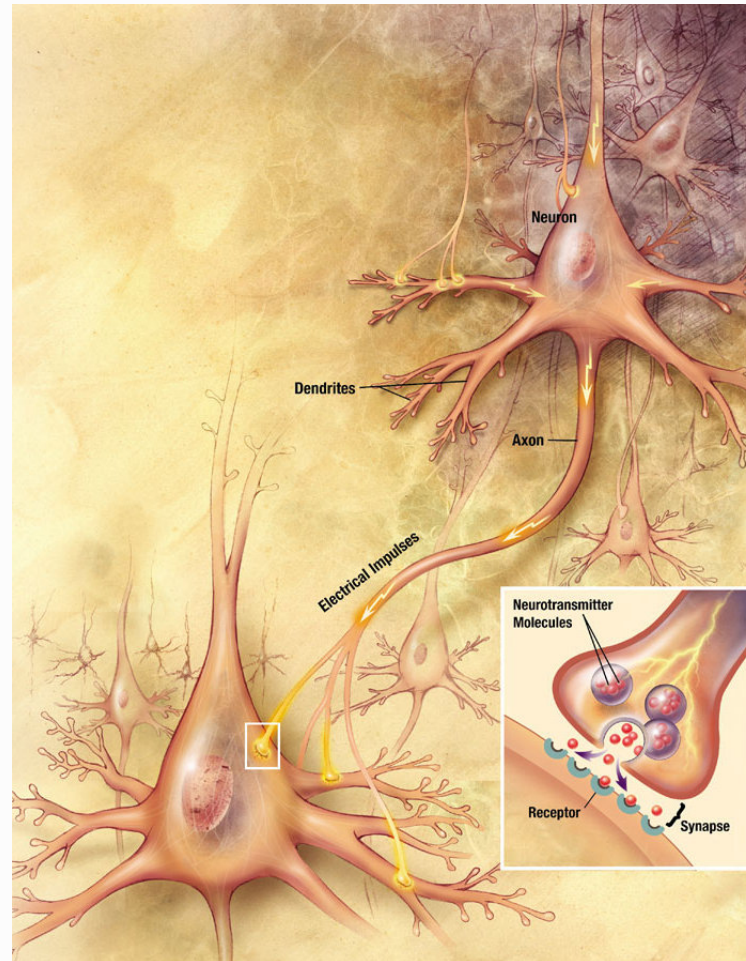
Biological neuron

- An biological processing node
- Signal integration
- Spatiotemporal signal processing
- Frequency encoding
- Biological limitations

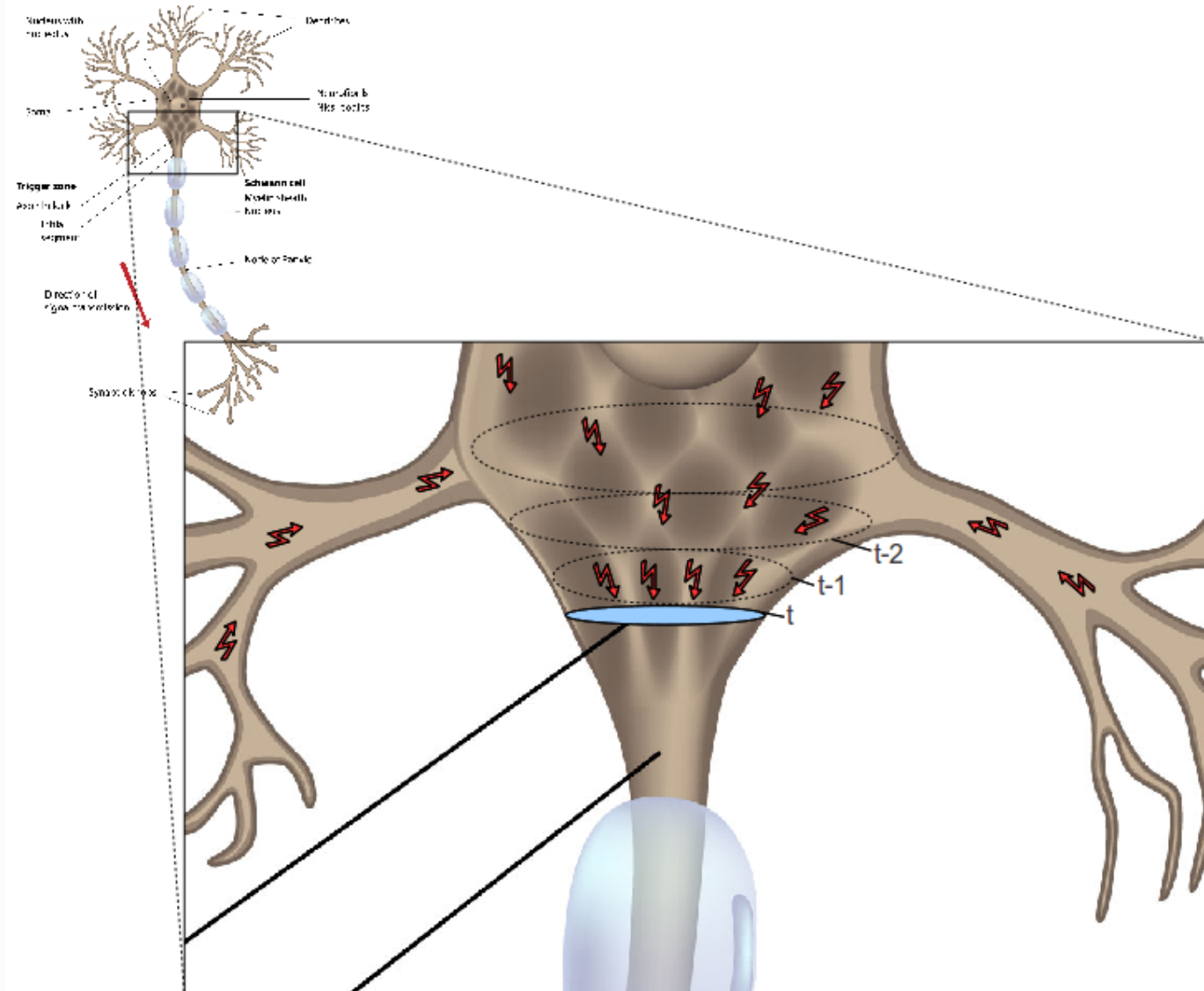
A multipolar neuron (Ex. spinal motor neuron)



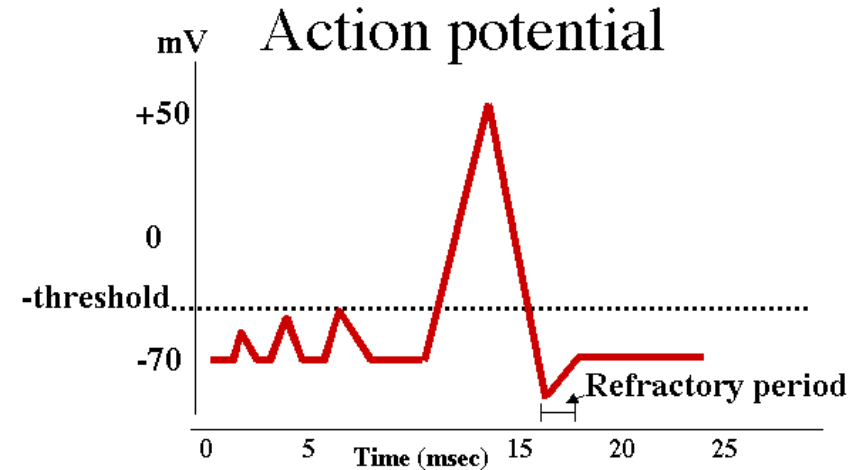
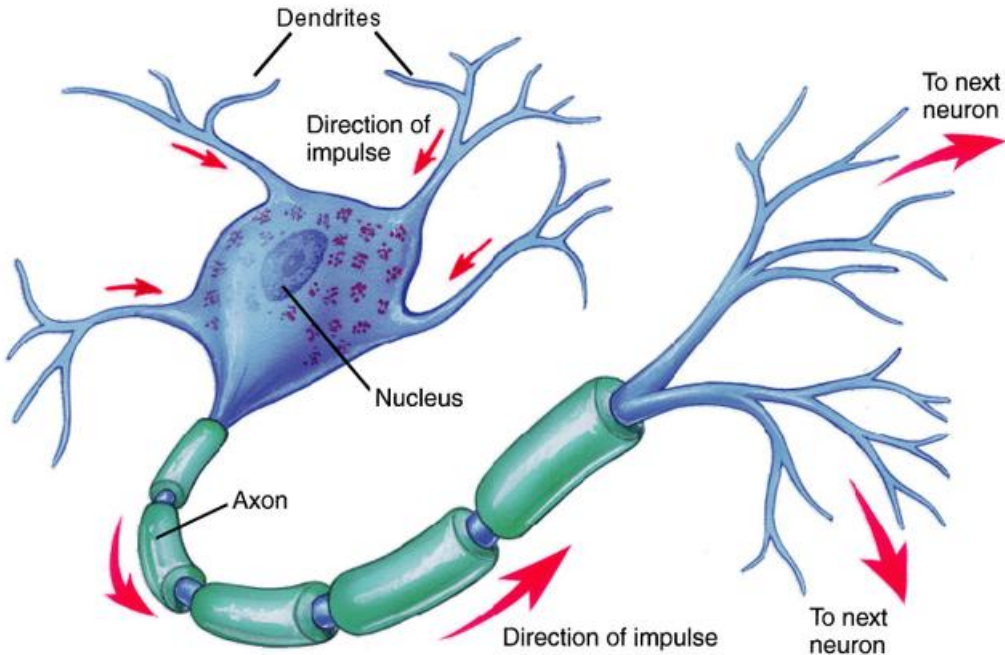
Signal Integration



Spatiotemporal Processing



Frequency Encoding



Threshold- to generate an action potential an axon requires a stimulus of a certain minimum strength

All or none - each action potential has the same amplitude independently from the strength of the stimulus

Refractory period- a second action potential cannot occur during this period

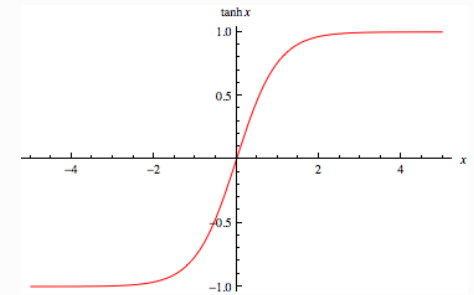
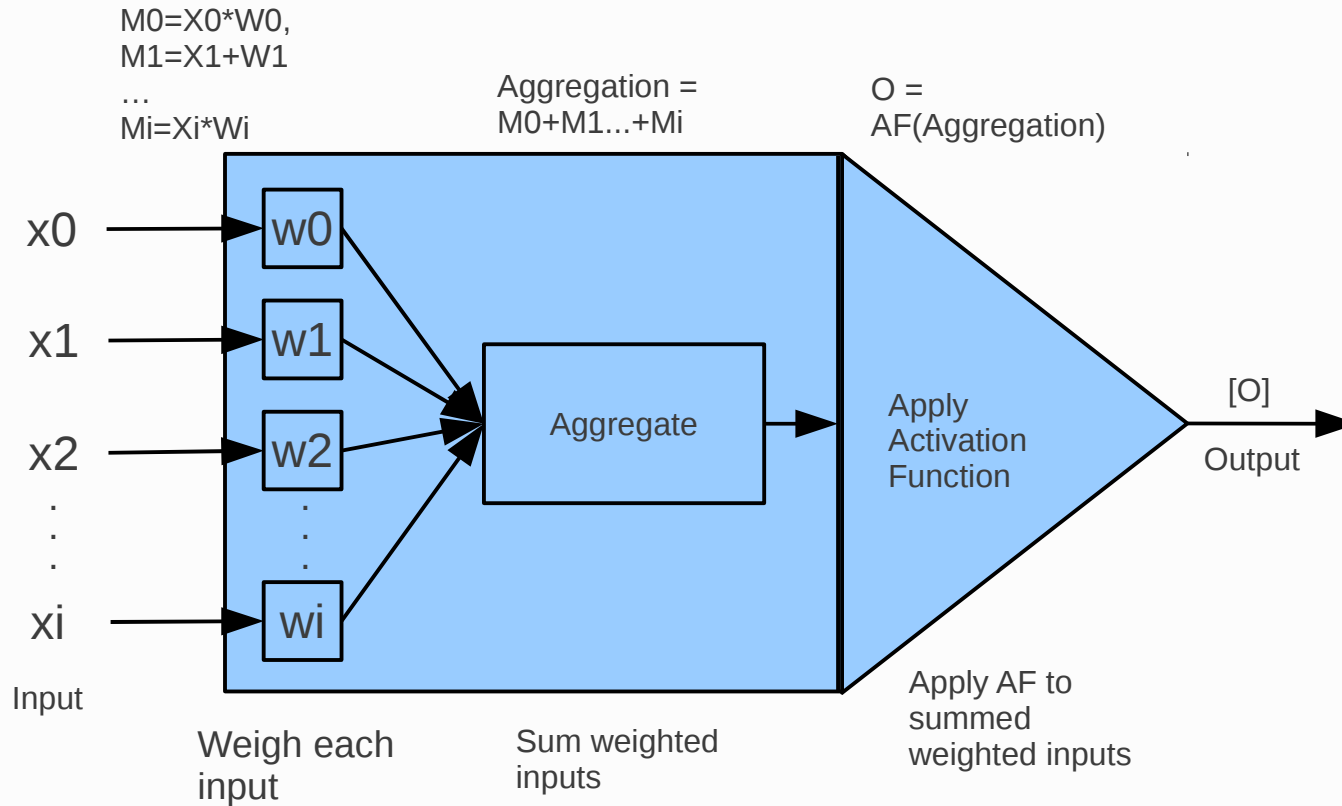
Whether the dendrites experience excitatory or inhibitory signals, depends not only on the actual signal sent by the presynaptic neuron, but also on the dendrites, their chemistry, receptors...

Plasticity

- Axon extension
- New dendrite branches
- More/less receptors

Artificial Neural Network

Artificial neuron



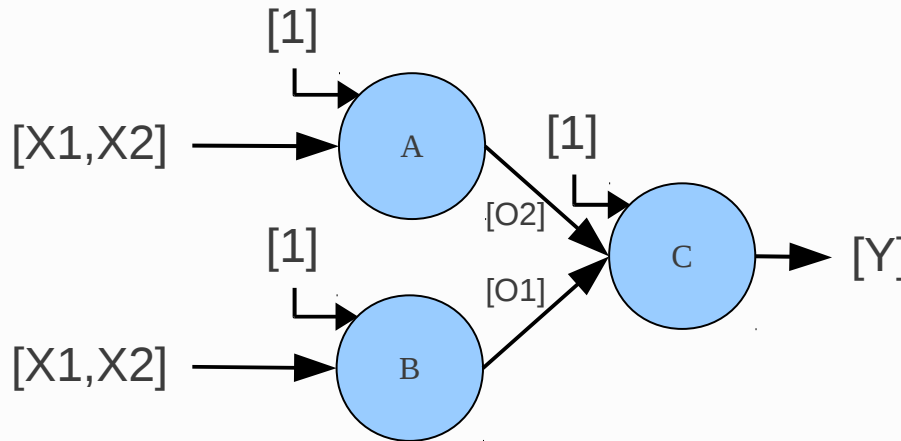
The Input is Just a Vector

1. Dot product:
 $DP = (0.5 * -1) + (0.2 * 1)$
Threshold = $(0 * 1)$

2. Activation strength:
Output = $\tanh(DP + \text{Threshold})$



Neural Circuit In Action



Input [X1,X2]: [-1,-1]

A: O1 = -0.9704 = $\tanh(-1*2.1081 + -1*2.2440 + 1*2.2533)$

B: O2 = 0.9922 = $\tanh(-1*3.4964 + -1*-2.7464 + 1*3.5200)$

C: Y = **-0.99** = $\tanh(0.9922*-2.5983 + -0.9704*2.7354 + 1*2.7255)$

Input [X1,X2]: [-1,1]

A: O1 = 0.9833 = $\tanh(-1*2.1081 + 1*2.2440 + 1*2.2533)$

B: O2 = -0.9914 = $\tanh(-1*3.4964 + 1*-2.7464 + 1*3.5200)$

C: Y = **0.99** = $\tanh(-0.9914*-2.5983 + 0.9833*2.7354 + 1*2.7255)$

Input [X1,X2]: [1,-1]

... C: Y = **0.99**

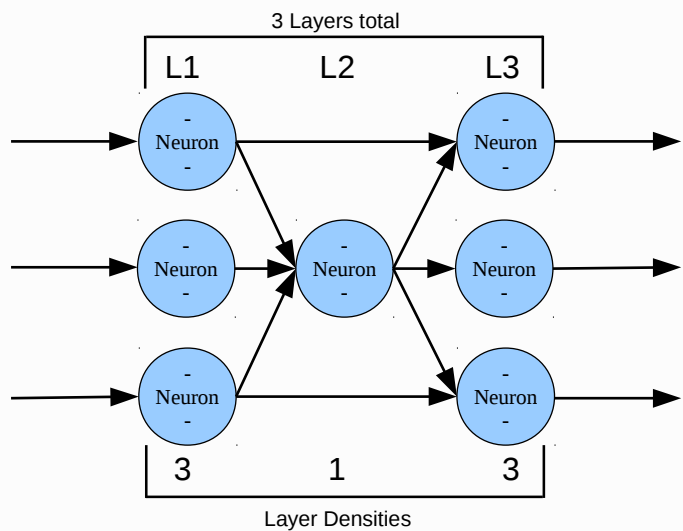
Input [X1,X2]: [1,1]

... C: Y = **-0.99**

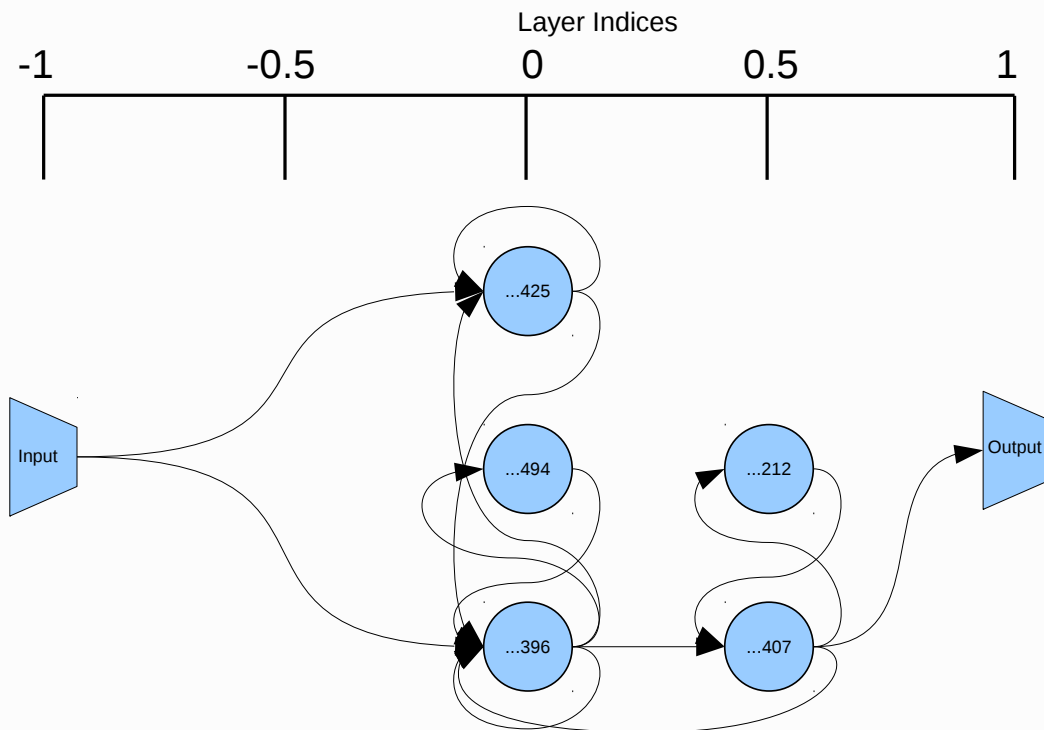
$$Y = C(A(X1*Wa_1 + X2*Wa_2 + 1*Wa_3)*Wc_1 + B(X1*Wb_1 + X2*Wb_2 + 1*Wb_3)*Wc_2)$$

Neural Network

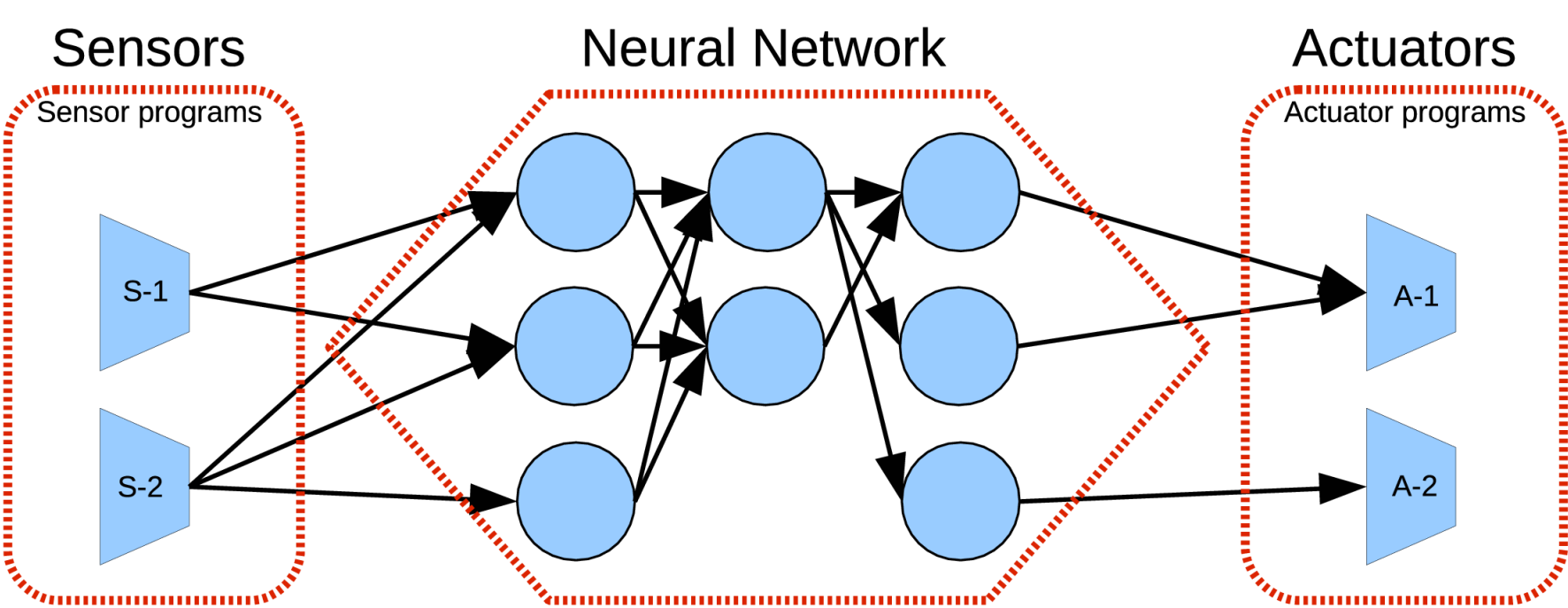
Feedforward



Recurrent



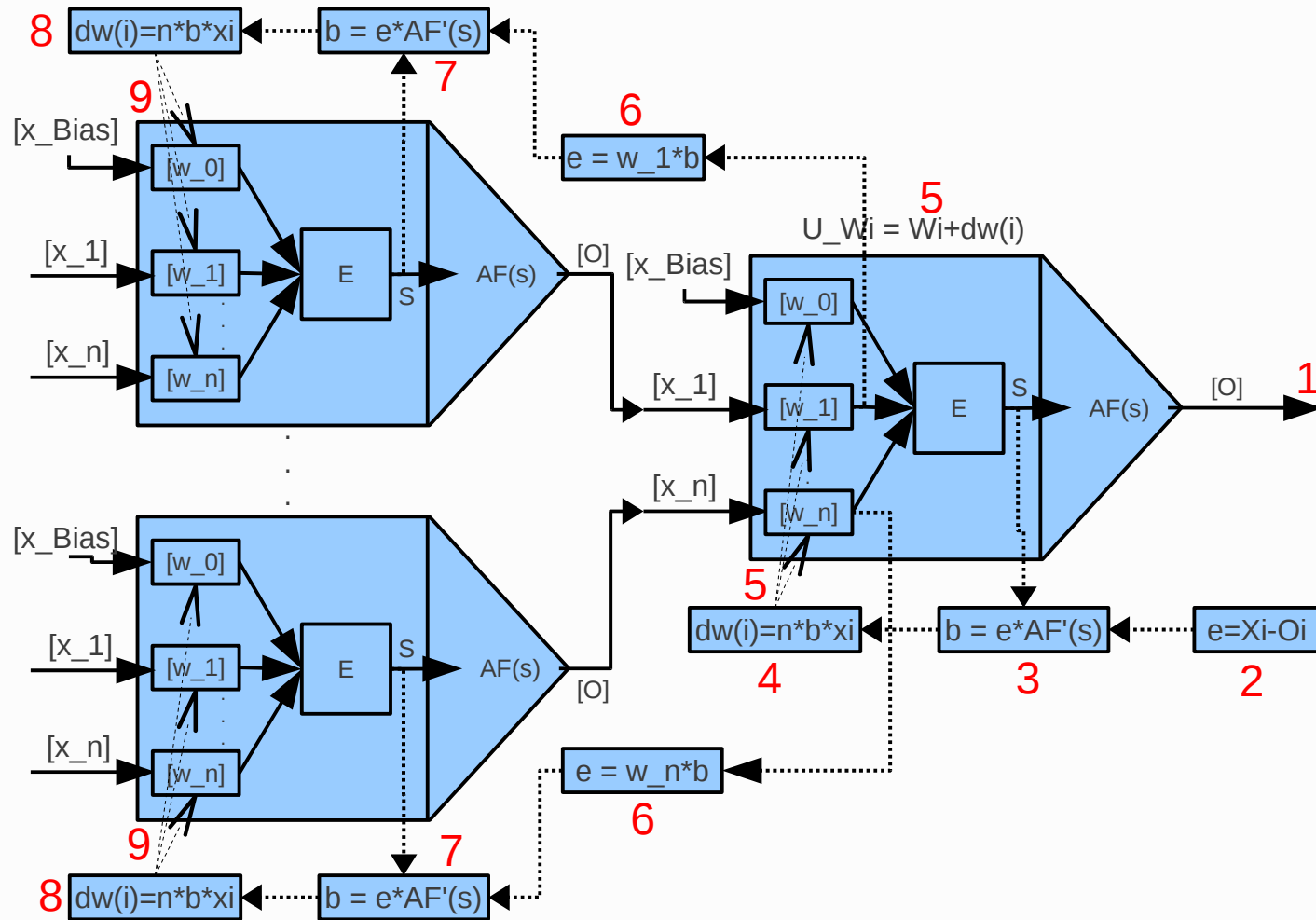
Neural Network Based Agent



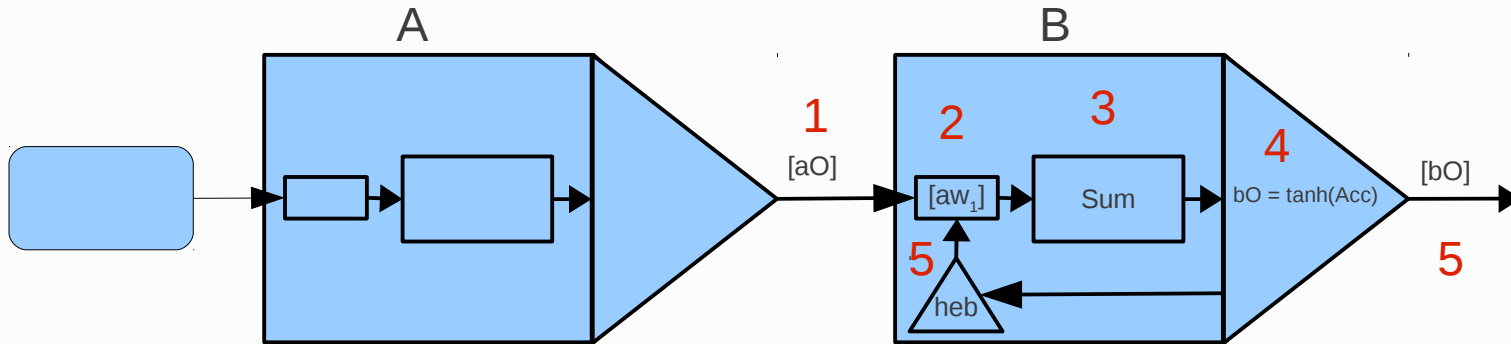
Learning Vs. Training

- Supervised
 - Backpropagation
 - ...
- Unsupervised
 - Kohonon (Self-organizing) map
 - Adaptive Resonance Theory
 - Hebbian
 - "The general idea is an old one, that any two cells or systems of cells that are repeatedly active at the same time will tend to become 'associated', so that activity in one facilitates activity in the other." (Hebb 1949, p. 70)
 - Modulated
 - Evolutionary
 -

Error Backpropogation



Hebbian Learning



1. Neuron A sends the vector signal [aO] to B.
2. Signal aO is weighted with B's synaptic weight aw_1 .
3. The weighted signals (in this case just one) are summed together to produce the value: Acc.
4. Activation function is applied to Acc to produce B's output signal bO.
5. B outputs vector signal [bO], while at the same time uses the Hebbian rule to produce a delta w , and update the synaptic weight aw_1 .

Update Rule: $U_W_i = W_i + n * X_i * O$
 Where X_i is the presynaptic signal associated with synaptic weight W_i , and where O is the postsynaptic neuron's output, and n the learning parameter.

Example: $aw_1 = 0.5$, $aO = 1$, $n = 1$

1. Neuron A sends the vector signal [1] to B.
2. Signal 1 is weighted with B's synaptic weight 0.5 to produce $Y_1 = x_1 * aw_1 = 1 * 0.5 = 0.5$.
3. The weighted signals (in this case just one, Y_1) are summed together: $Acc = Sum(Y_1) = 0.5$.
4. Activation function tanh is applied to Acc to produce B's output signal $bO = \tanh(Acc) = 0.46$.
5. B outputs the vector signal [bO] = [0.46], while at the same time uses the Hebbian rule to produce: $dw = 0.46 * 1 = 0.46$, and update the synaptic weight aw_1 . Thus, the updated $aw_1 = 0.5 + 0.46 = 0.96$. The new synaptic weight is: **$aw_1 = 0.96$** .

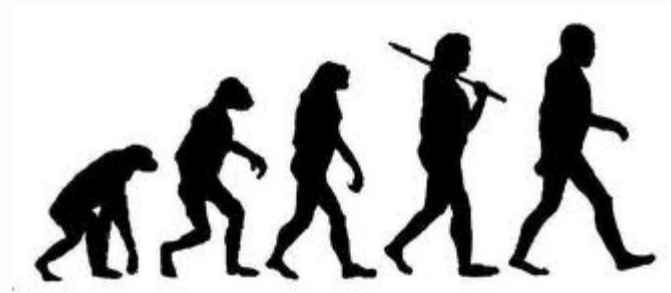
If we now continue running this update rule, with A firing signals of the same magnitude, 1, the sequence of B's weight aw_1 is: **0.5, 0.962, 1.71, 2.64, 3.63, 4.63**

The synaptic weight continues to increase in magnitude over time.

Ok ok... But what about the topology, and the new learning parameters? How do I set them to the values that produce useful system for some problem?

Evolutionary Computation

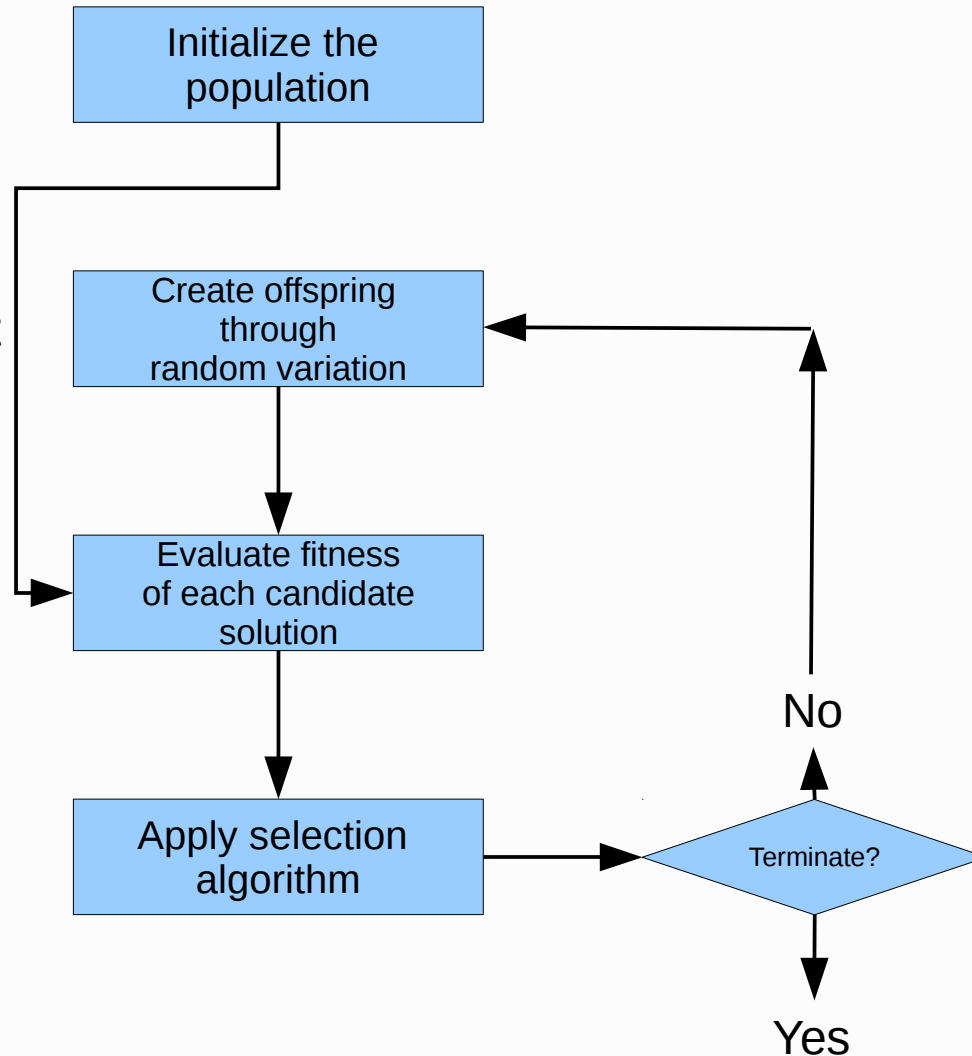
- Based on evolutionary principles
- Stochastic search with a purpose
 - Create as many copies of yourself as possible
 - Some copies (offspring) will have errors when being copied
 - Others are competing for resources
 - Push towards finding an advantage
 - Survival of the fittest
- Genotype to Phenotype
- Mutation and crossover



Evolutionary Computation Flowchart

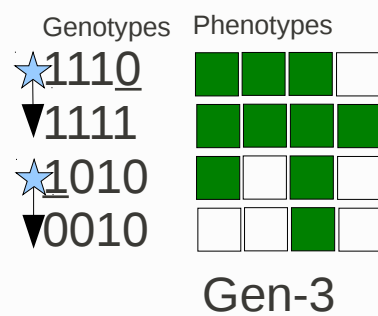
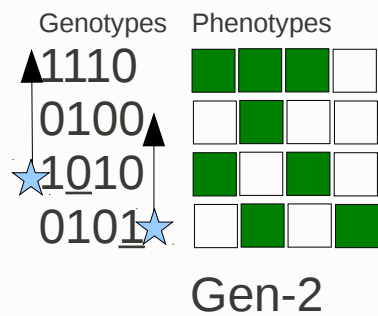
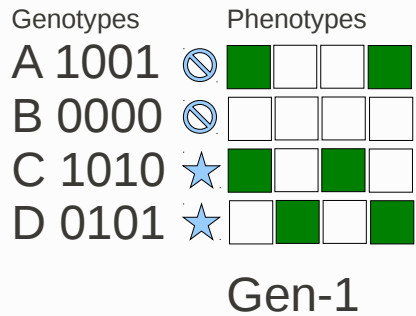
Extracting the most important parts:

1. Replication.
2. Variation: Mutation.
3. Competition: Those that are more fit, will survive and make more mutant copies of themselves.

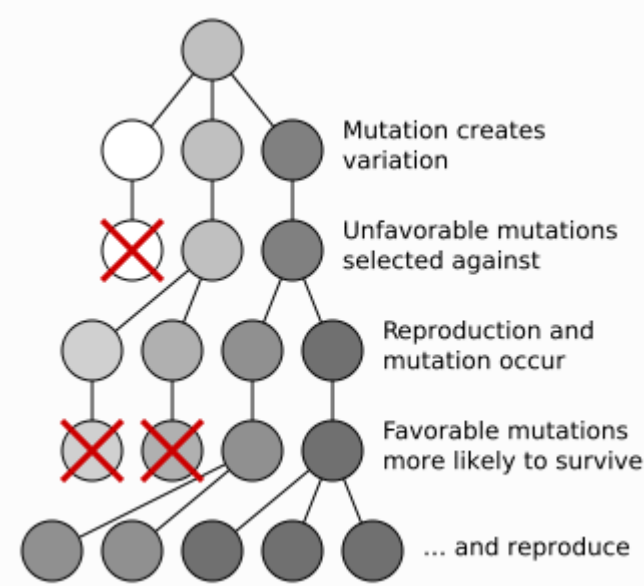
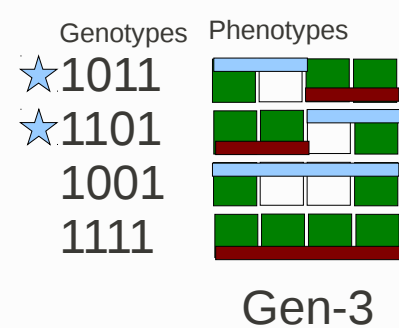
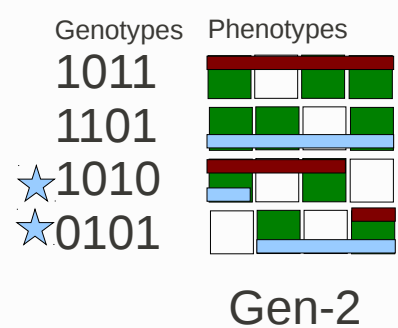
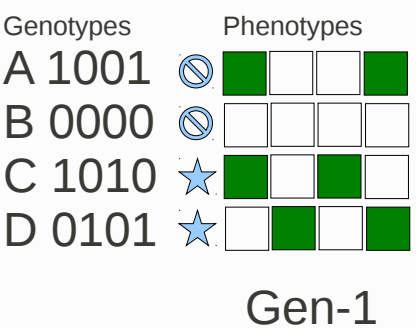


Simple Genetic Algorithm Example

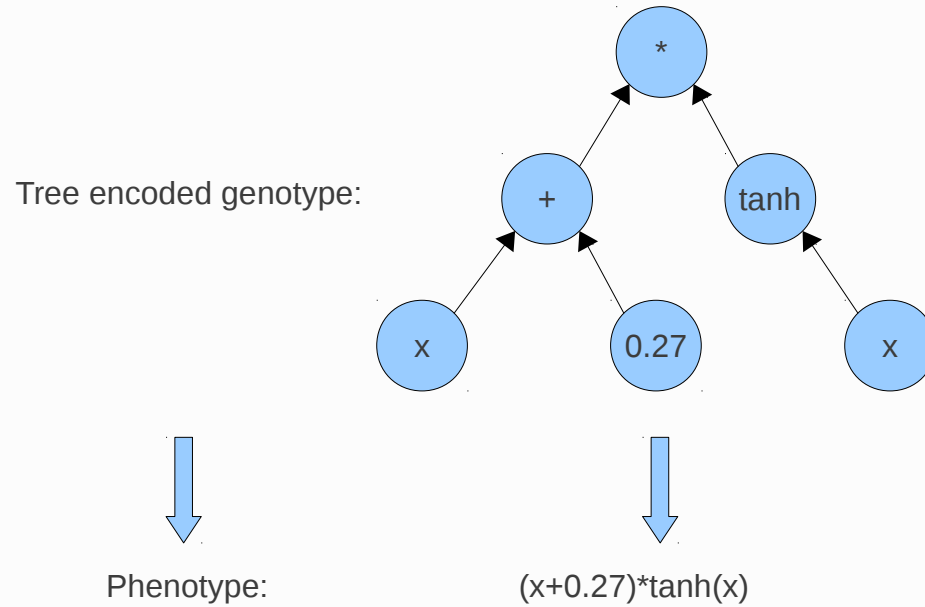
Simple Mutations



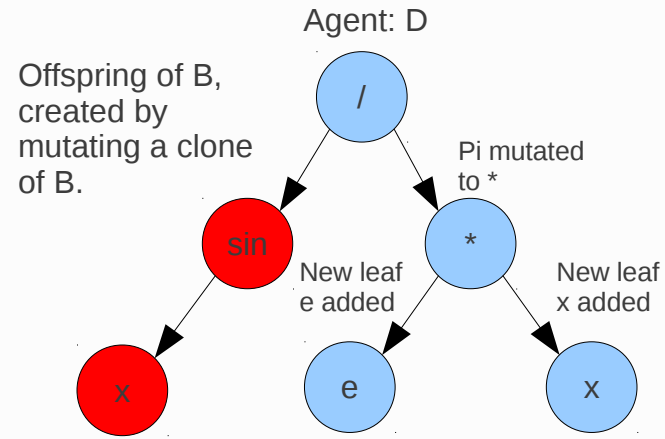
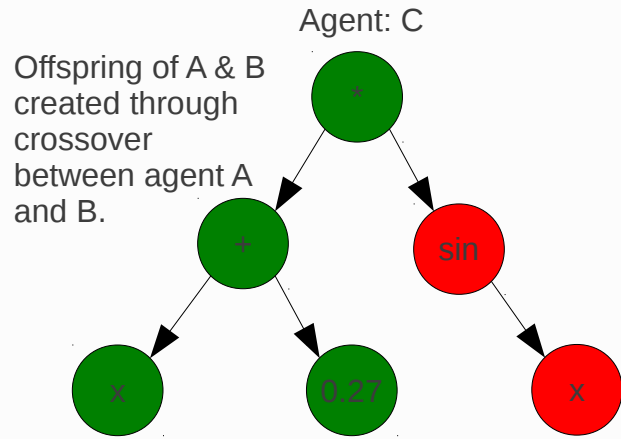
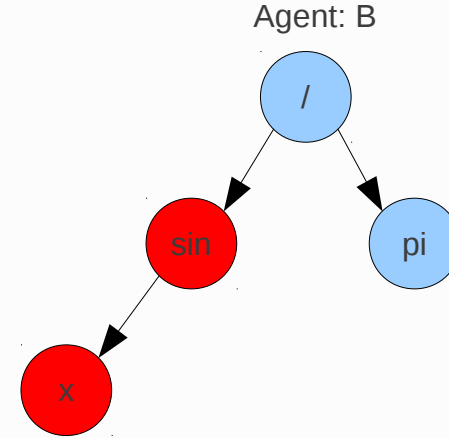
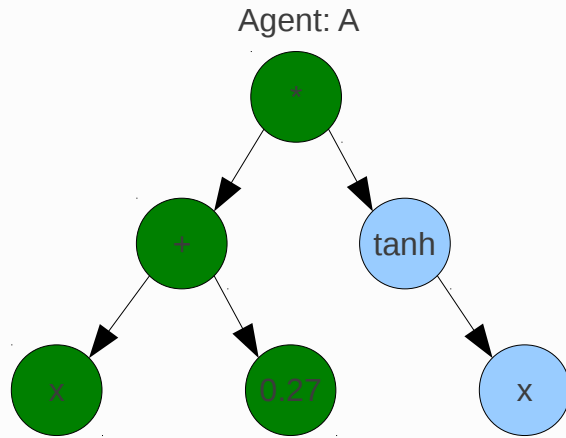
Crossover



Genetic Programming



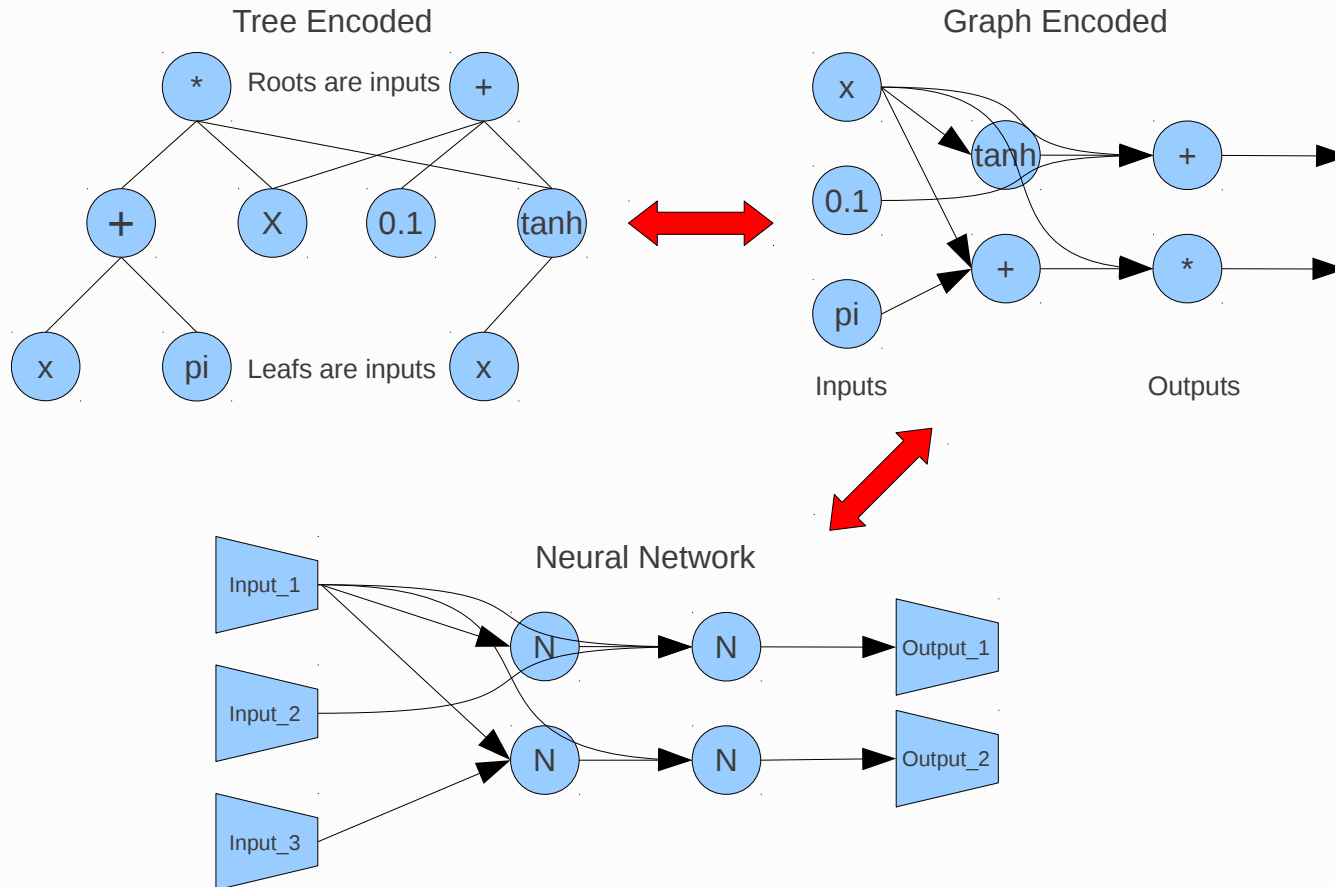
Genetic Programming



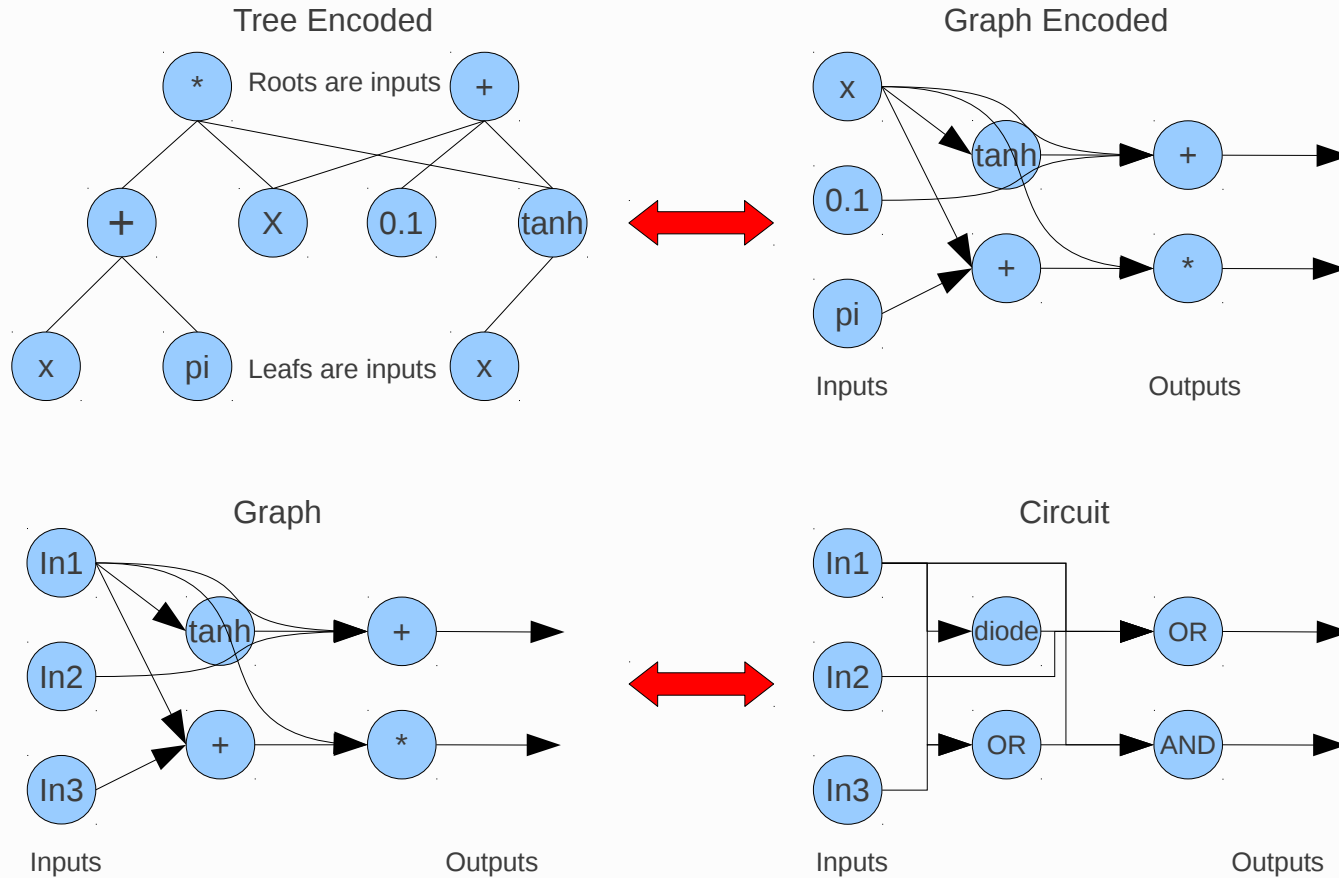
Evolutionary Computation Approaches

- **Genetic Algorithms (John Holland, 73-75)**
 - Population of fixed length genotypes, bit strings, evolved through perturbation/crossing
- **Genetic Programming (John Koza, 92)**
 - Variable sized chromosome based programs represented as treelike structures, with specially crafted genetic operators
- **Evolutionary Strategies (Ingo Rechenberg, 73)**
 - Normal distribution based, adaptive perturbations (self-adaptation)
- **Evolutionary Programming (L. & D. Fogel, 63)**
 - Like ES, but for evolution of state transition tables for finite-state machines (FSMs)

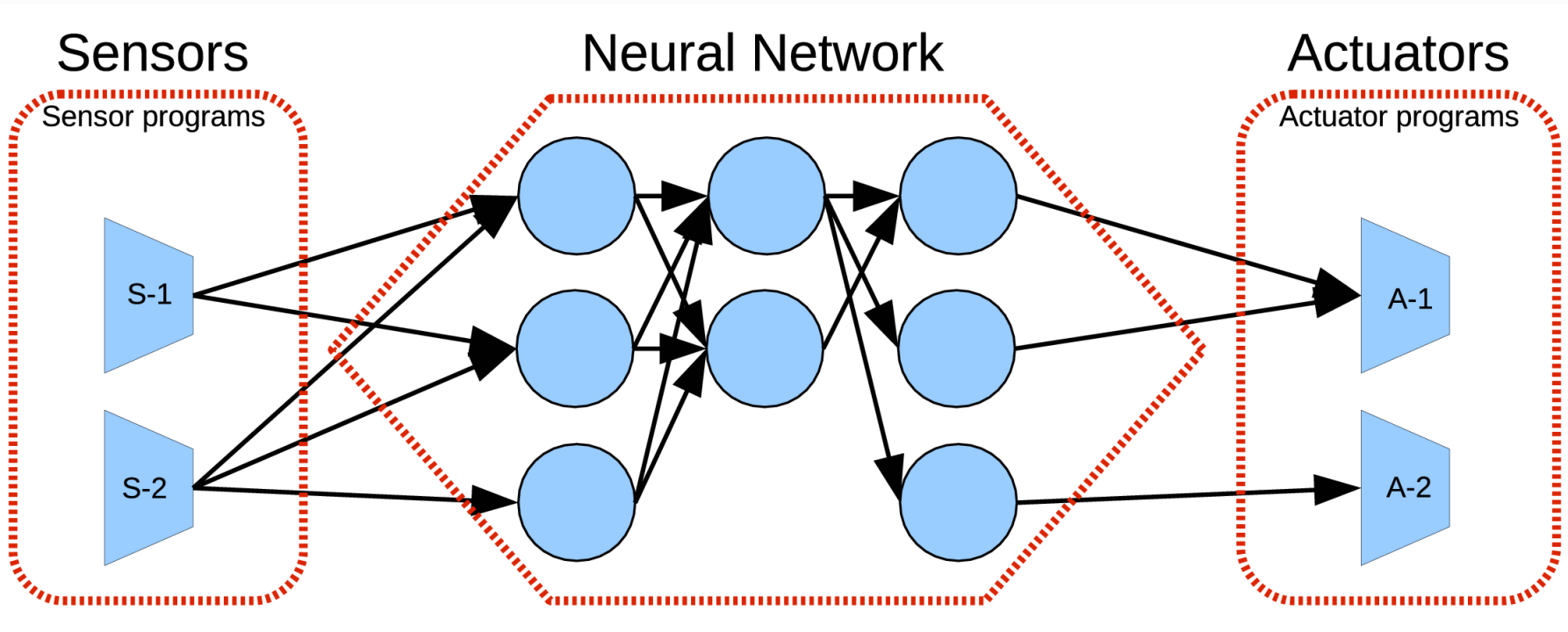
Towards Neuroevolution




Different sides of the same coin



In Search For A Neural Network Programming Language



Hardware is advancing, scaling outward, perfect for distributed and concurrent systems; Software is lagging behind



1997: THE FIRST INTEL® TERAFLUP COMPUTER
consisted of:

9,298 INTEL PROCESSORS

and occupied:


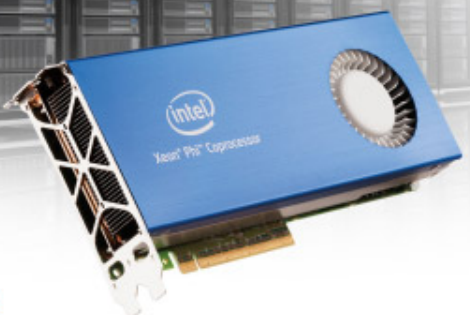
72 SERVER CABINETS

THE INTEL® XEON® PHI™ COPROCESSOR
will provide:

1 TERAFLUP OF PERFORMANCE

and occupy:

1 PCIe SLOT



[Click to learn more](#)

Other Programming/Scripting Languages

- Standard procedural and object oriented programming languages do not have the perfect architectures for NN based systems.
 - C/C++
 - Java
 - Python
 - Perl
- What are the needed features to remove the conceptual gap between the programming language architecture and the distributed NN based computational intelligence problem domain?

Creating The Perfect Neural Network Programming Language

A list of features that a neural network based computational intelligence system needs, as quoted from the list made by Bjarne Däcker [1], is as follows:

1. The system must be able to handle very large numbers of concurrent activities.
2. Actions must be performed at a certain point in time or within a certain time.
3. Systems may be distributed over several computers.
4. The system is used to control hardware.
5. The software systems are very large.
6. The system exhibits complex functionality such as, feature interaction.
7. The systems should be in continuous operation for many years.
8. Software maintenance (reconfiguration, etc) should be performed without stopping the system.
9. There are stringent quality, and reliability requirements.
10. Fault tolerance

Surprisingly enough, Däcker was not talking about a neural network based general computational intelligence systems when he made this list, he was talking about a telecom switching systems.

[1] Bjarne Däcker. Concurrent functional programming for telecommunications: A case study of technology introduction. November 2000. Licentiate Thesis.

Erlang: From Telecom Networks To Neural Networks

- A list of features
- Architectural 1:1 mapping, no conceptual gaps

Erlang Features

The features that Erlang possesses, as quoted from Armstrong's thesis [2], is as follows:

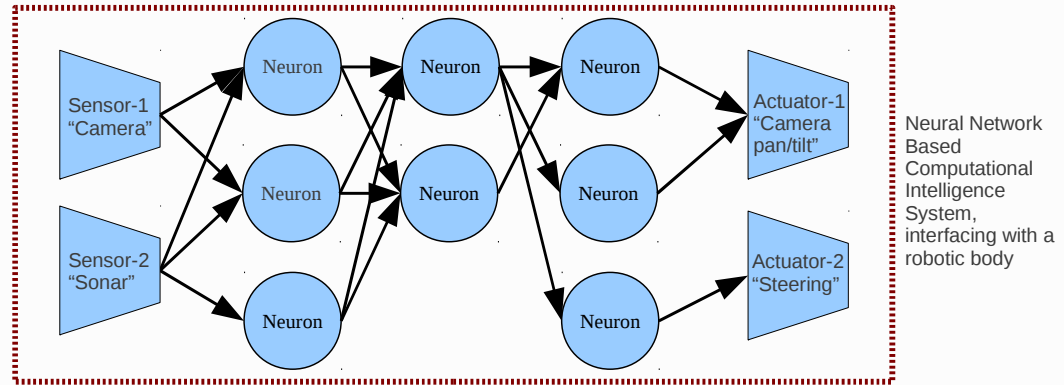
1. Encapsulation primitives — there must be a number of mechanisms for limiting the consequences of an error. It should be possible to isolate processes so that they cannot damage each other.
2. Concurrency — the language must support a lightweight mechanism to create parallel process, and to send messages between the processes. Context switching between process, and message passing, should be efficient. Concurrent processes must also time-share the CPU in some reasonable manner, so that CPU bound processes do not monopolize the CPU, and prevent progress of other processes which are 'ready to run.'
3. Fault detection primitives — which allow one process to observe another process, and to detect if the observed process has terminated for any reason.
4. Location transparency — If we know the PId of a process then we should be able to send a message to the process.
5. Dynamic code upgrade — It should be possible to dynamically change code in a running system. Note that since many processes will be running the same code, we need a mechanism to allow existing processes to run “old” code, and for “new” processes to run the modified code at the same time.

With a set of libraries to provide:

6. Stable storage — this is storage which survives a crash.
7. Device drivers — these must provide a mechanism for communication with the outside world.
8. Code upgrade — this allows us to upgrade code in a running system.
9. Infrastructure — for starting, and stopping the system, logging errors , etc.”

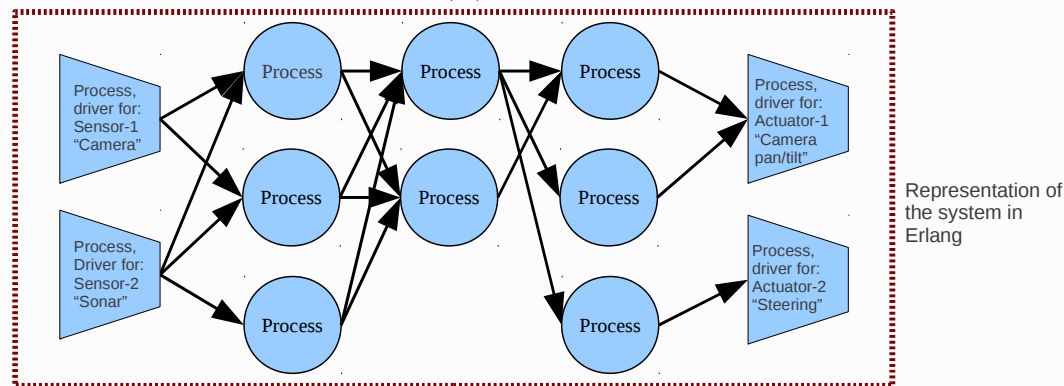
[2] Joe Armstrong, “Making reliable distributed systems in the presence of software errors ” A Dissertation submitted to the Royal Institute of Technology Stockholm, Sweden

The Architectural 1:1 Mapping



Neural Network Based Computational Intelligence System, interfacing with a robotic body

↕ 1:1 Mapping

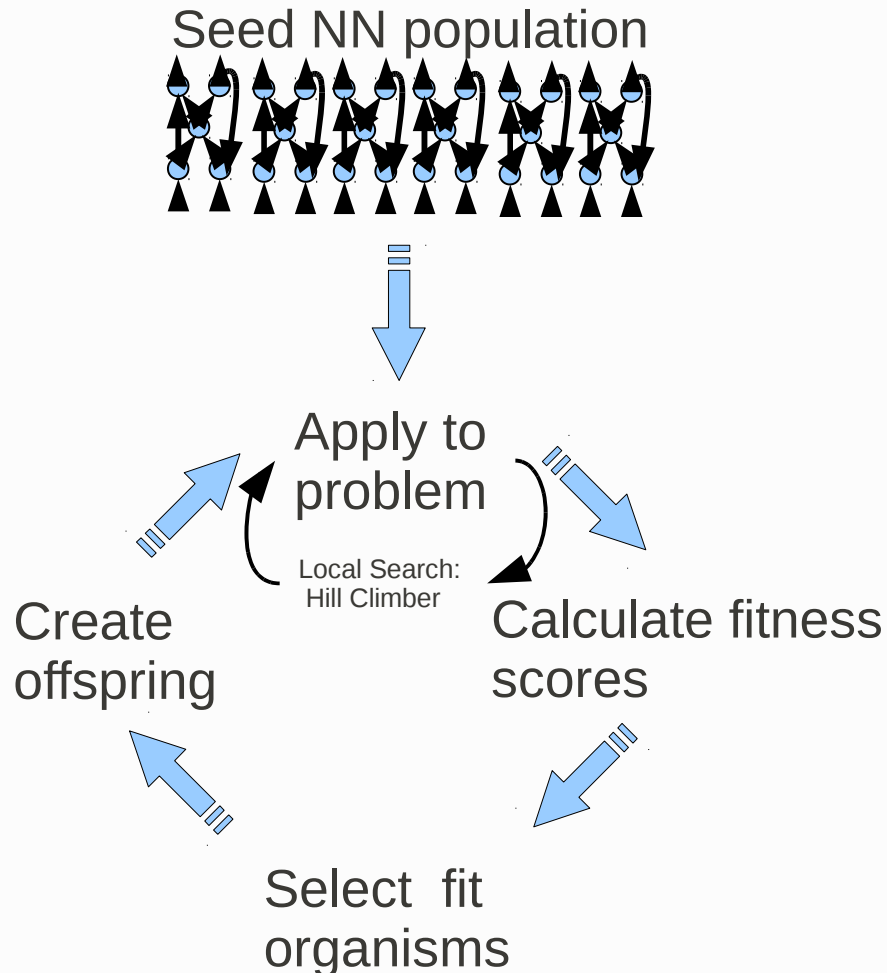


Representation of the system in Erlang

DXNN: A Case Study

- The evolutionary loop
- NN based agent architecture
- Platform architecture
- Neuron Architecture
- Genotype Encoding
- Implementing Mutations
- Incorporating Modularity
- Handbook of Neuroevolution Through Erlang

Memetic Algorithm Based TWEANN



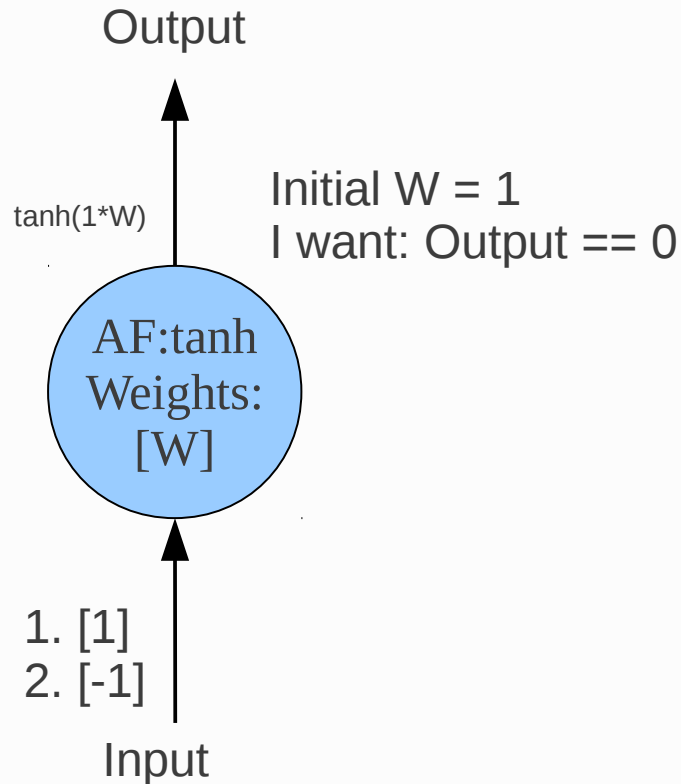
The Learning algorithm is as follows:

0. Create seed population of NN agents.
1. Spawn (convert genotype to phenotype) a population of agents.
2. Each agent interacts with the environment or some problems.
3. Each agent gets a fitness evaluation.
4. A process called exoself perturbs agent's synaptic weights.
5. Applies it to the problem again.
6. And if its performance increases, then this new synaptic weight combination is considered best, and we again perturb the synaptic weights. If the new performance is worse, then we revert to previous best, and perturb the synaptic weights.
7. Eventually all agents have had their synaptic weights tuned, and the fitness scores of the agents is compared.
8. Fitter agents allowed to create more offspring.
9. Goto: 1

Parametric Mutation

- Parameter lists available to the specie:
 - Plasticity_List = [none, hebb...]
 - Activation_Function_List = [none, tanh, sin, gauss...]
 - ...
- Let different species have access to different lists of parameters.
- When you create a new function, simply add its function name to the list, without taking the system offline... offspring agents will begin incorporating the new features.
- Choosing which mutation operators to apply:
 - [mutate:MO(Agent_Id) || MO ← [MOperator || {MOperator, Prob} ← Operators, Prob <random:uniform()]]

Stochastic Hill Climber



1. $\text{Output1} = \tanh(1*1) = 0.76$
 $\text{Output2} = \tanh(1*-1) = -0.76$

2. Weight Perturbation

Perturbation = -0.5
Try $W = 0.5 = 1 - 0.5$
 $\text{Output1} = \tanh(0.5*1) = 0.46$
 $\text{Output2} = \tanh(0.5*-1) = -0.46$
That's closer! New $W = 0.5$

3. Weight Perturbation

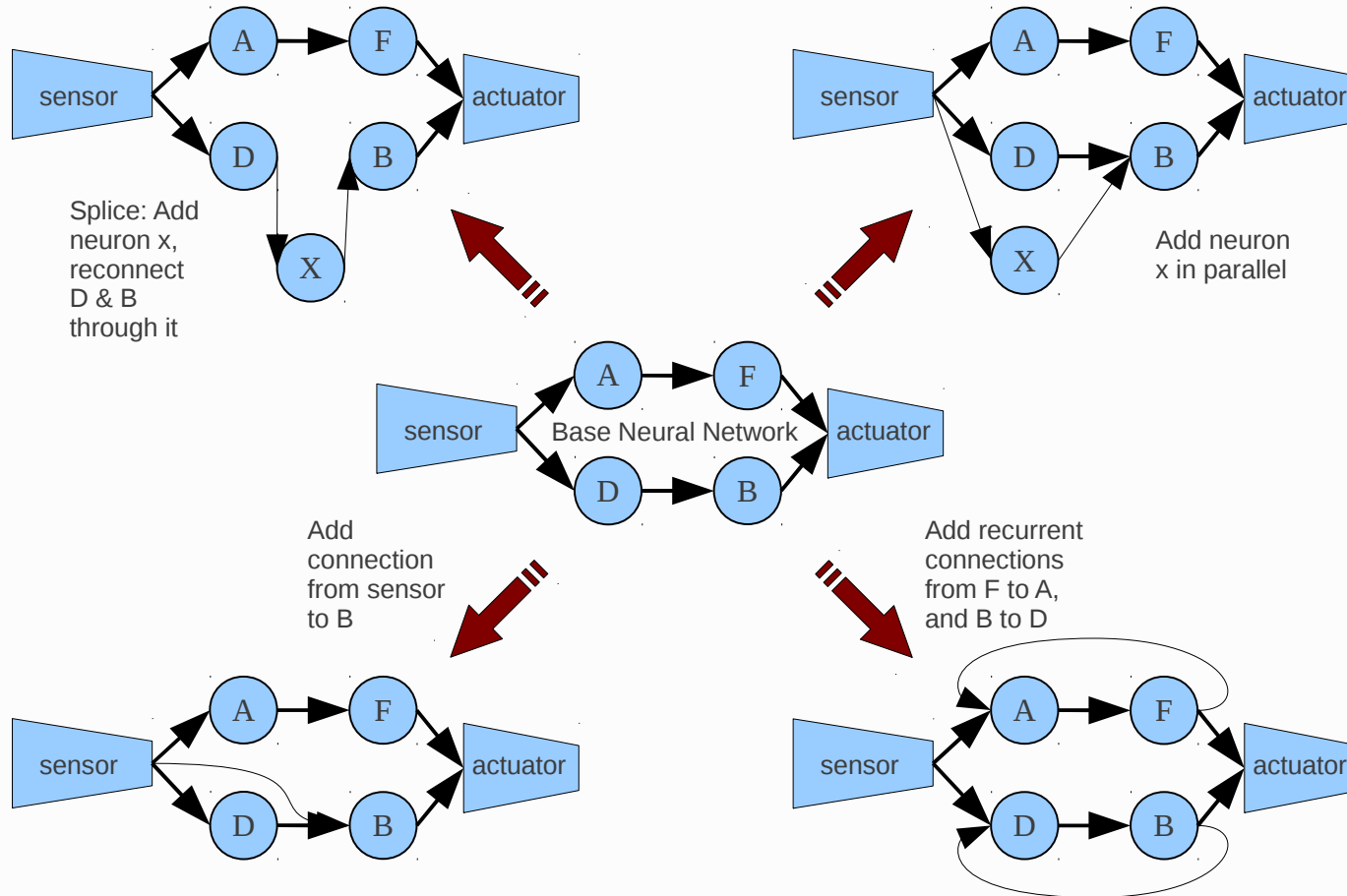
Perturbation = $+0.2$
Try $W = 0.7 = 0.5 + 0.2$
 $\text{Output1} = \tanh(0.7*1) = 0.60$
 $\text{Output2} = \tanh(0.7*-1) = -0.60$
Not as good as before, New $W = 0.5$

4. Weight Perturbation

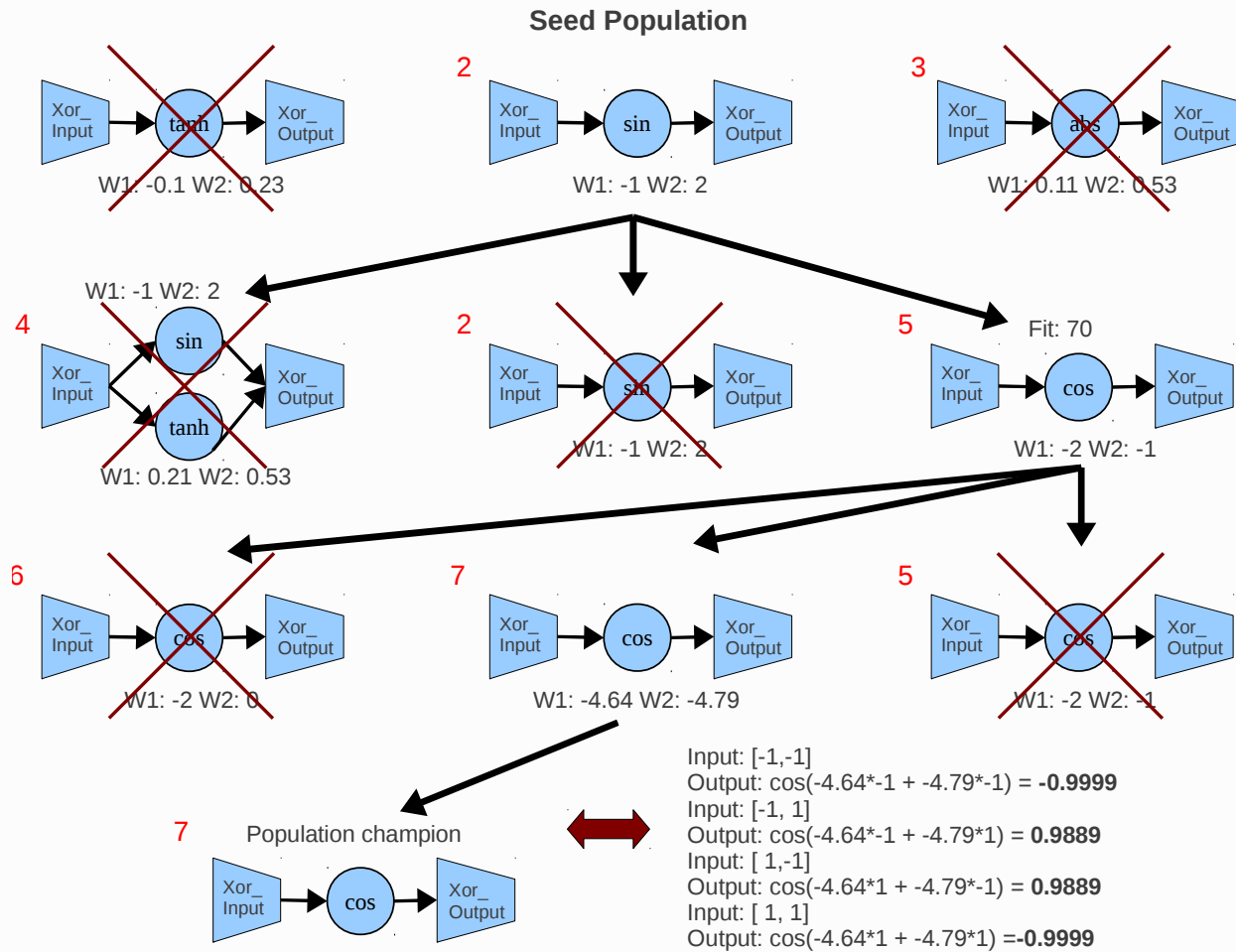
Perturbation = -0.5
Try $W = 0 = 0.5 - 0.5$
 $\text{Output1} = \tanh(0*1) = 0 !!!$
 $\text{Output2} = \tanh(0*-1) = 0 !!!$

The right weight is 0.

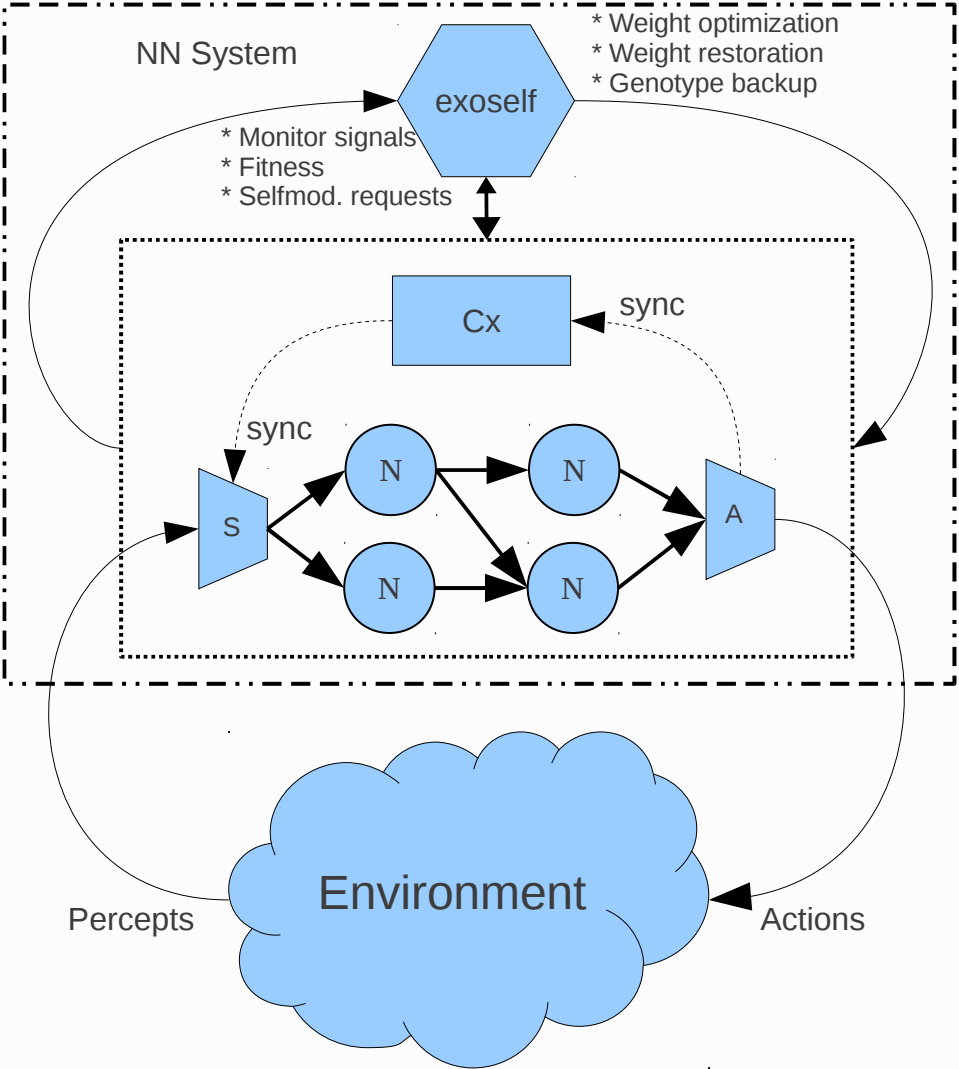
Topological Mutation Operators



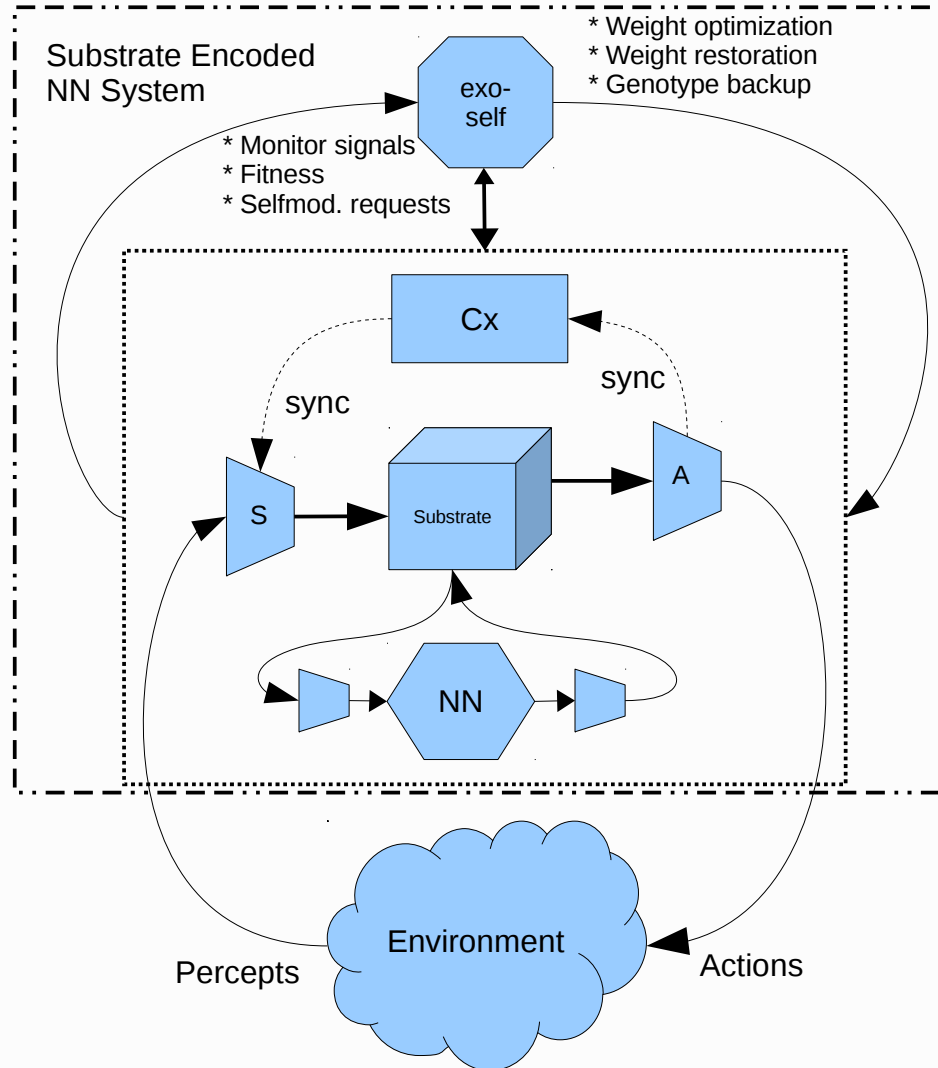
Instead of evolving a single NN, let's evolve a population



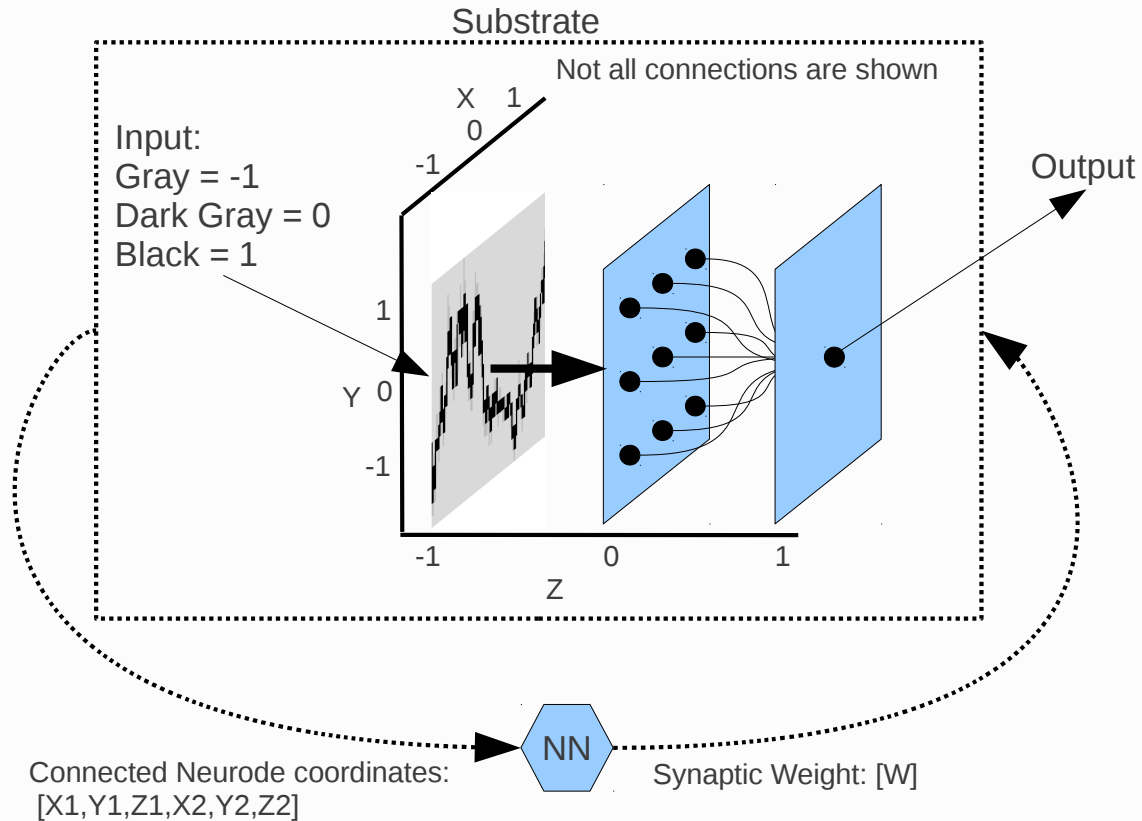
Neural Network Agent Architecture



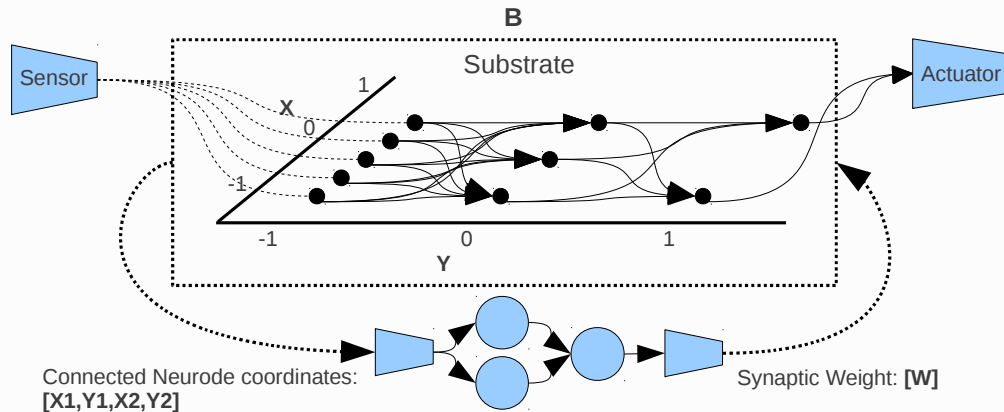
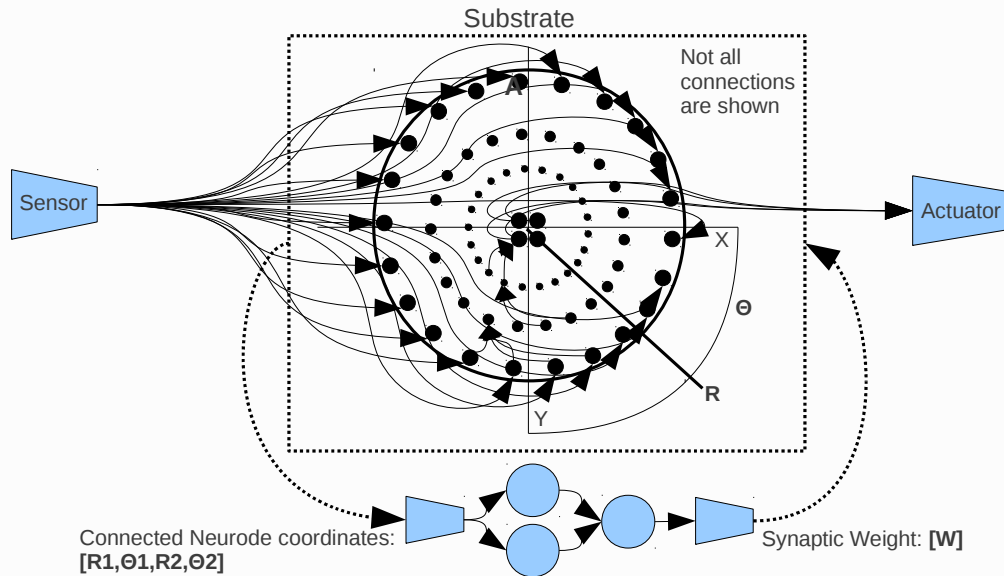
The Infomorph's Phenotype (Substrate)



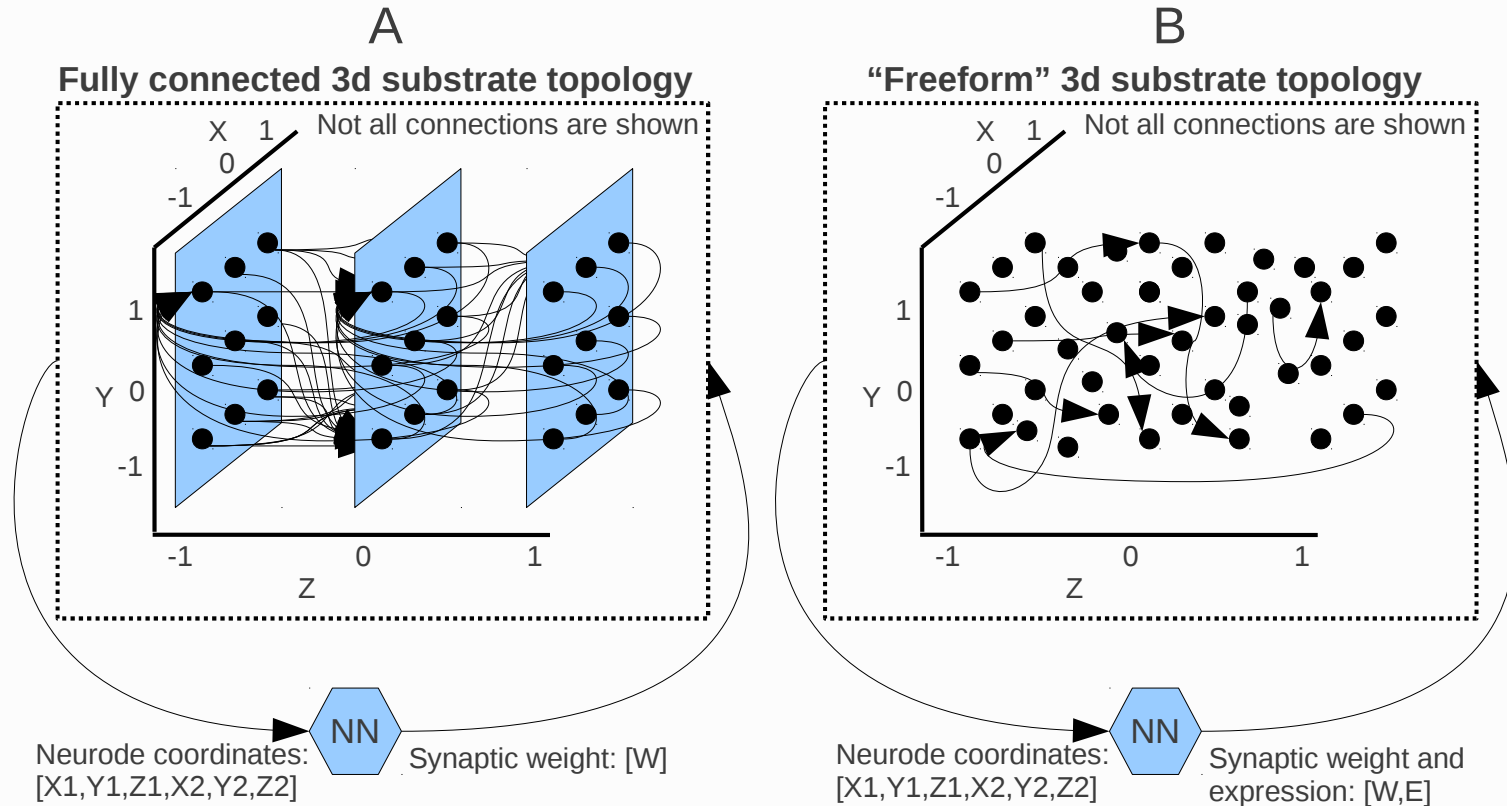
Substrate Encoding



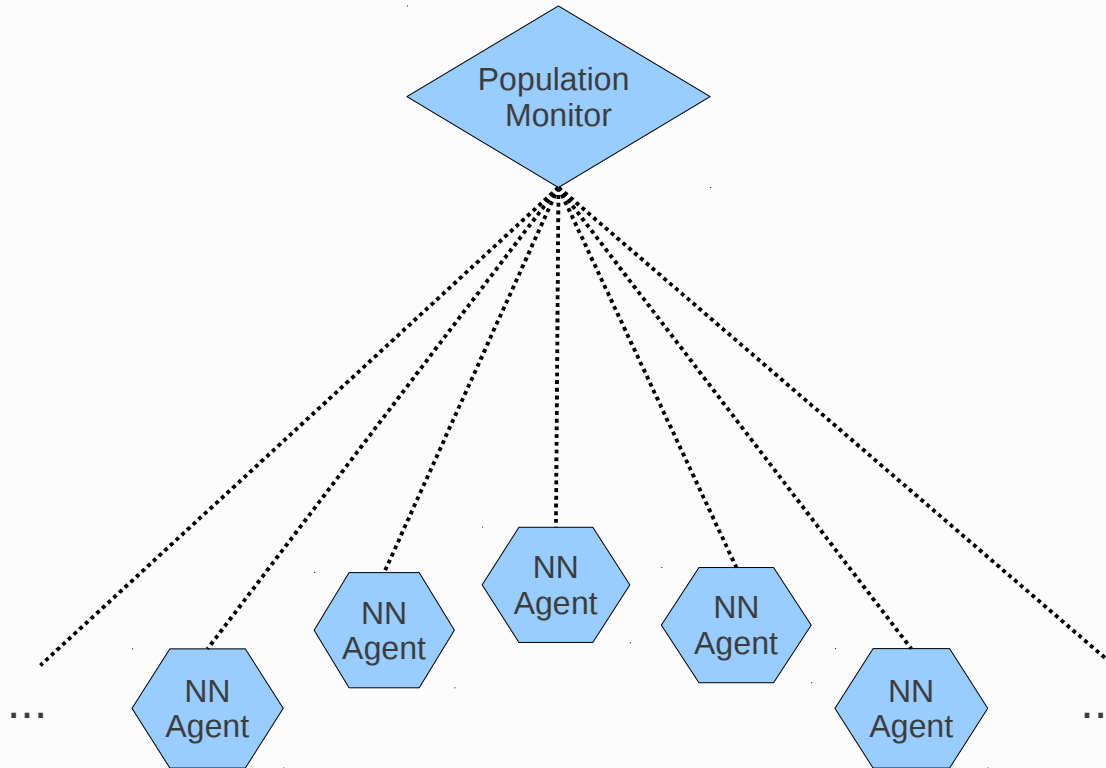
Substrate Encoding (continued)



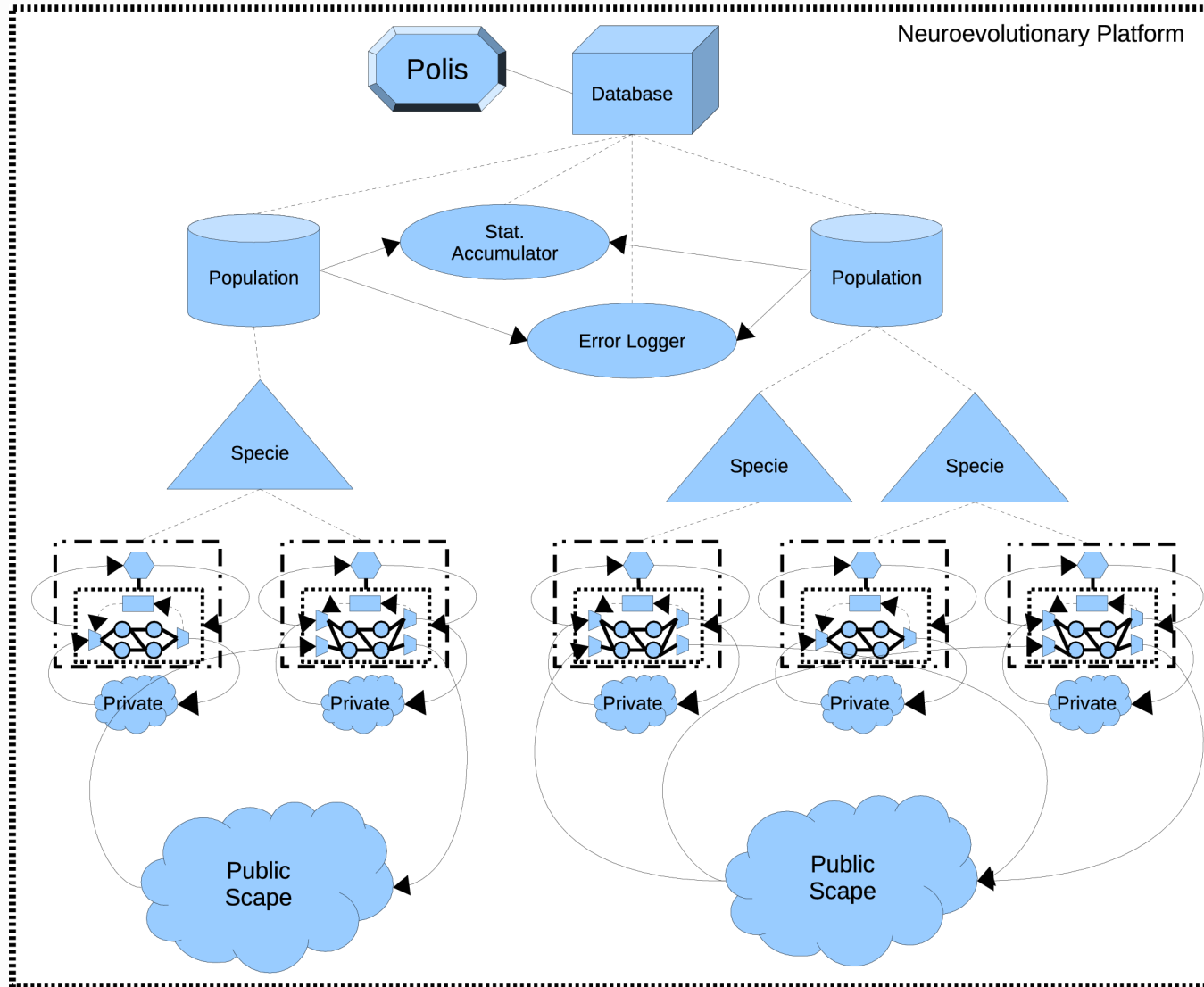
Substrate Encoding (continued)



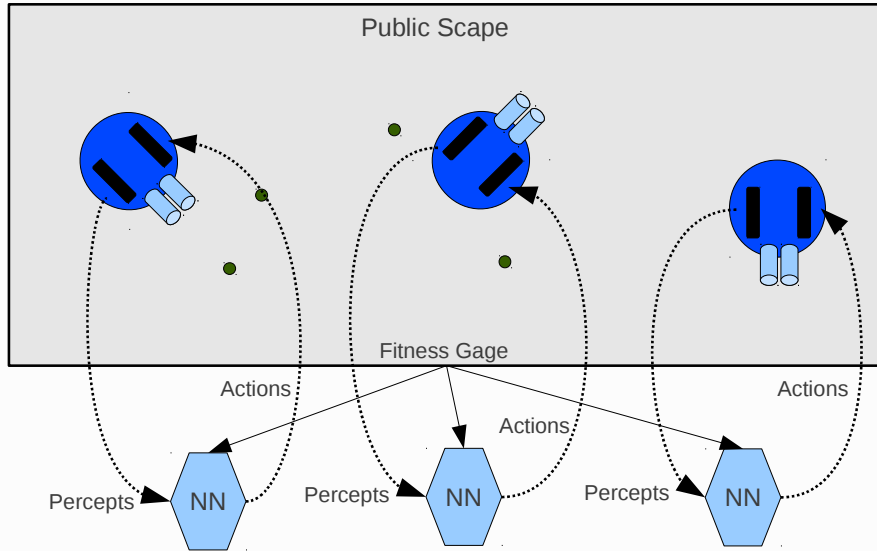
An evolving NN population



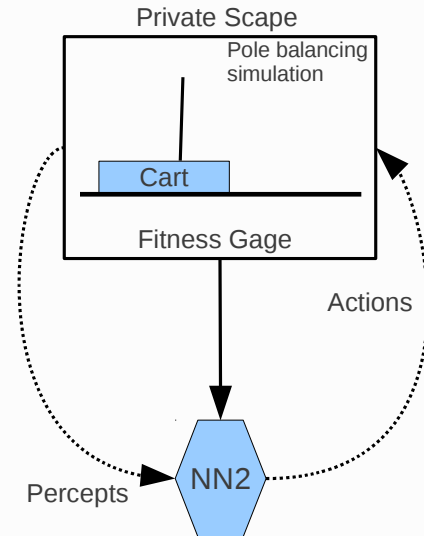
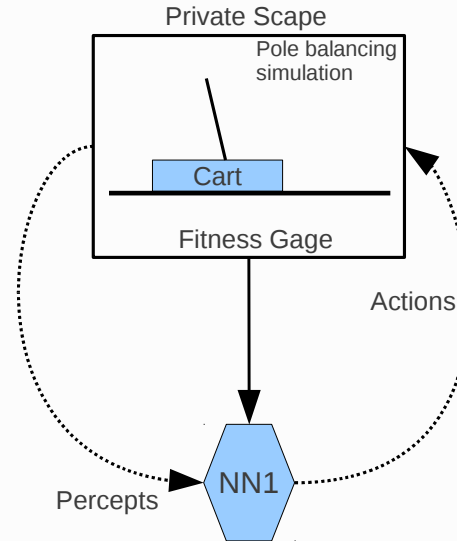
Platform Architecture



Scape & Morphology



This is how my NN based agents interact with problems/simulated environments.



Morphology Specification

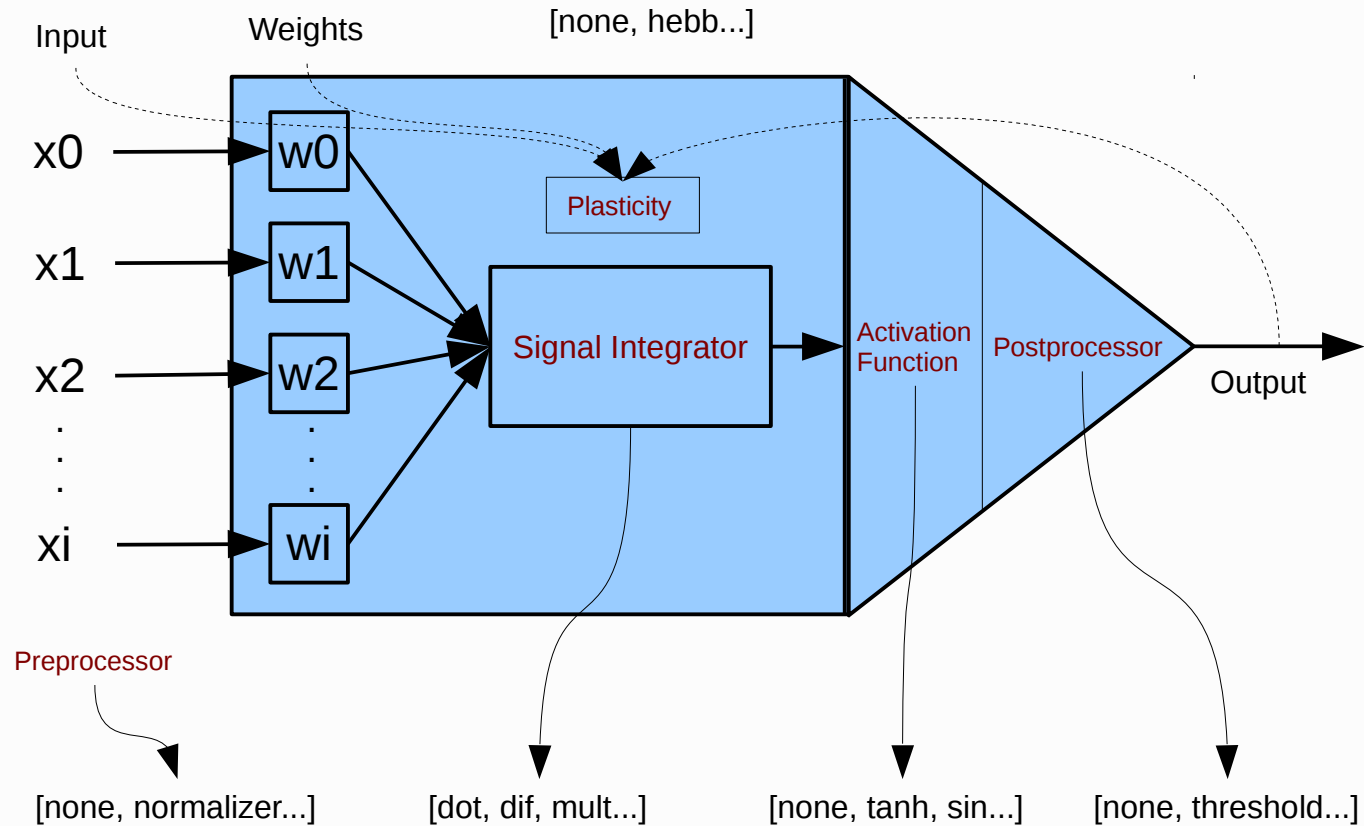
flatlander(actuators)->

```
Movement = [#actuator{name=two_wheels,id=cell_id,format=no_geo,tot_vl=2,parameters=[2]}],  
Cloning = [#actuator{name=create_offspring,id=cell_id,format=no_geo,tot_vl=1,parameters=[1]}],  
Weapons = [#actuator{name=spear,id=cell_id,format=no_geo,tot_vl=1,parameters=[1]}],  
Communications = [#actuator{name=speak,id=cell_id,format=no_geo,tot_vl=1,parameters=[1]}],  
Movement++Weapons++Communications;
```

flatlander(sensors)->

```
Pi = math:pi(),  
Distance_Scanners =  
[#sensor{name=distance_scanner,id=cell_id,format=no_geo,tot_vl=Density,parameters=[Spread,Density,ROffset]} ||  
    Spread<-[Pi/2],Density<-[5], ROffset<-[Pi*0/2]],  
Color_Scanners =  
[#sensor{name=color_scanner,id=cell_id,format=no_geo,tot_vl=Density,parameters=[Spread,Density,ROffset]} ||  
    Spread <-[Pi/2], Density <-[5], ROffset<-[Pi*0/2]],  
Communciations=[#sensor{name=Name,id=cell_id,format=no_geo,tot_vl=Density,parameters=[Spread,Density,ROffset]} ||  
    Name <- [sound_scanner], Spread <-[Pi/2], Density <-[10], ROffset<-[Pi*0/2]],  
Distance_Scanners++Communciations++Color_Scanners.
```

Neural Processing



Output = postproc:PoF(af:AFF(sigint:SIF(preproc:PrF(Input),Weights))),
 Updated_W = plasticity:PlastF(Input,Output,Weights).

StandardOutput = postproc:none(af:tanh(sigint:dot(preproc:none(Input),Weights))),
 Updated_W = plasticity:none(Input,Output,Weights).

ART_N = postproc:threshold(af:none(sigint:dif(preproc:normalizer(Input),Weights))),
 Updated_W = plasticity:hebb(Input,Output,Weights).

DXNN Neural Process In under 80 lines

```

loop(S,ExoSelf_PId,[ok],[ok],SIAcc,MIAcc)->
  {PFName,PFPParameters} = PF = S#state.pf,
  AF = S#state.af,
  AggrF = S#state.aggrf,
  Ordered_SIAcc = lists:reverse(SIAcc),
  SI_PIdPs = S#state.si_pidps_current,
  SOutput = sat(functions:AF(signal_aggregator:AggrF(Ordered_SIAcc,SI_PIdPs)),?OUTPUT_SAT_LIMIT),
  Output_PIds = S#state.output_pids,
  [Output_PId ! {self()},forward,[SOutput]] || Output_PId <- Output_PIds,
  case PFName of
    none ->
      U_S=S;
    _ ->
      Ordered_MIAcc = lists:reverse(MIAcc),
      MI_PIdPs = S#state.mi_pidps_current,
      MAggregation_Product = sat(signal_aggregator:dot_product(Ordered_MIAcc,MI_PIdPs),?SAT_LIMIT),
      MOutput = functions:tanh(MAggregation_Product),
      U_SI_PIdPs = plasticity:PFName([MOutput|PFPParameters],Ordered_SIAcc,SI_PIdPs,SOutput),
      U_S=S#state{
        si_pidps_current = U_SI_PIdPs
      }
  end,
  SI_PIds = S#state.si_pids,
  MI_PIds = S#state.mi_pids,
  neuron:loop(U_S,ExoSelf_PId,SI_PIds,MI_PIds,[],[]);
loop(S,ExoSelf_PId,[SI_PId|SI_PIds],[MI_PId|MI_PIds],SIAcc,MIAcc)->
  receive
  {SI_PId,forward,Input}->
    loop(S,ExoSelf_PId,SI_PIds,[MI_PId|MI_PIds],[{SI_PId,Input}|SIAcc],MIAcc);
  {MI_PId,forward,Input}->
    loop(S,ExoSelf_PId,[SI_PId|SI_PIds],MI_PIds,SIAcc,[{MI_PId,Input}|MIAcc]);
  {ExoSelf_PId,weight_backup}->
    U_S=case S#state.hereditiy_type of
      darwinian ->
        S#state{
          si_pidps_backup=S#state.si_pidps_bl,
          mi_pidps_backup=S#state.mi_pidps_current
        };
      lamarckian ->
        S#state{
          si_pidps_backup=S#state.si_pidps_current,
          mi_pidps_backup=S#state.mi_pidps_current
        }
    end,
    loop(U_S,ExoSelf_PId,[SI_PId|SI_PIds],[MI_PId|MI_PIds],SIAcc,MIAcc);
  {ExoSelf_PId,weight_restore}->
    U_S = S#state{
      si_pidps_bl=S#state.si_pidps_backup,
      si_pidps_current=S#state.si_pidps_backup,
      mi_pidps_current=S#state.mi_pidps_backup
    },
    loop(U_S,ExoSelf_PId,[SI_PId|SI_PIds],[MI_PId|MI_PIds],SIAcc,MIAcc);

```

```

{ExoSelf_PId,weight_perturb,Spread}->
  Perturbed_SIPIIdPs=perturb_IPIdPs(Spread,S#state.si_pidps_backup),
  Perturbed_MIPIIdPs=perturb_IPIdPs(Spread,S#state.mi_pidps_backup),
  U_S=S#state{
    si_pidps_bl=Perturbed_SIPIIdPs,
    si_pidps_current=Perturbed_SIPIIdPs,
    mi_pidps_current=Perturbed_MIPIIdPs
  },
  loop(U_S,ExoSelf_PId,[SI_PId|SI_PIds],[MI_PId|MI_PIds],SIAcc,MIAcc);
{ExoSelf_PId,reset_prep}->
  neuron:flush_buffer(),
  ExoSelf_PId ! {self(),ready},
  RO_PIds = S#state.ro_pids,
  receive
  {ExoSelf_PId, reset}->
    fanout(RO_PIds,{self(),forward,[?RO_SIGNAL]})
  end,
  loop(S,ExoSelf_PId,S#state.si_pids,S#state.mi_pids,[],[]);
{ExoSelf_PId,get_backup}->
  NId = S#state.id,
  ExoSelf_PId ! {self(),NId,S#state.si_pidps_backup,S#state.mi_pidps_backup},
  loop(S,ExoSelf_PId,[SI_PId|SI_PIds],[MI_PId|MI_PIds],SIAcc,MIAcc);
{ExoSelf_PId,terminate}->
  ok
end.

```

Mnesia as Storage for Genotypes

- Robust and safe
- Tuple friendly
- Easy atomic mutations
 - If any part of the mutation fails, the whole mutation is just retracted automatically

Genotype Record Relations

-record(population,{id, polis_id, specie_ids=[], morphologies=[], innovation_factor, evo_alg_f, fitness_postprocessor_f, selection_f, trace=#trace{}}).

-record(specie,{id, population_id, fingerprint, constraint, agent_ids=[], dead_pool=[], champion_ids=[], fitness, innovation_factor={0,0},stats=[]}).

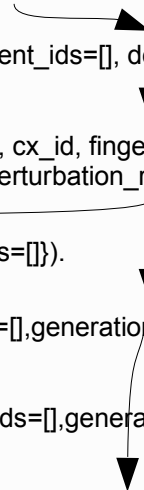
-record(agent,{id, encoding_type, generation, population_id, specie_id, cx_id, fingerprint, constraint, evo_hist=[], fitness=0, innovation_factor=0, pattern=[], tuning_selection_f, annealing_parameter, tuning_duration_f, perturbation_range, mutation_operators,tot_topological_mutations_f,heredity_type,substrate_id}).

-record(cortex, {id, agent_id, neuron_ids=[], sensor_ids=[], actuator_ids=[]}).

-record(sensor,{id,name,type,cx_id,scape,vl,fanout_ids=[],generation,format,parameters,gt_parameters,phys_rep,vis_rep,pre_f,post_f}).

-record(actuator,{id,name,type,cx_id,scape,vl,fanin_ids=[],generation,format,parameters,gt_parameters,phys_rep,vis_rep,pre_f,post_f}).

-record(neuron, {id, generation, cx_id, pre_processor,signal_integrator,af, post_processor, pf, aggr_f, input_idps=[], input_idps_modulation=[], output_ids=[], ro_ids=[]}).



Genotype

- record(agent,{id, encoding_type, generation, population_id, specie_id, cx_id, fingerprint, constraint, evo_hist=[], fitness=0, innovation_factor=0, pattern=[], tuning_selection_f, annealing_parameter, tuning_duration_f, perturbation_range, mutation_operators,tot_topological_mutations_f,heredity_type,substrate_id}).
- record(cortex, {id, agent_id, neuron_ids=[], sensor_ids=[], actuator_ids=[]}).
- record(substrate, {id, agent_id, densities, linkform, plasticity=none, cpp_ids=[],cep_ids=[]}).
- record(sensor,{id,name,type,cx_id,scape,vl,fanout_ids=[],generation,format,parameters,gt_parameters,phys_rep,vis_rep,pre_f,post_f}).
- record(actuator,{id,name,type,cx_id,scape,vl,fanin_ids=[],generation,format,parameters,gt_parameters,phys_rep,vis_rep,pre_f,post_f}).
- record(neuron, {id, generation, cx_id, pre_processor,signal_integrator,af, post_processor, pf, aggr_f, input_idps=[], input_idps_modulation=[], output_ids=[], ro_ids=[]}).
- record(population,{id, polis_id, specie_ids=[], morphologies=[], innovation_factor, evo_alg_f, fitness_postprocessor_f, selection_f, trace=#trace{}}).
- record(specie,{id, population_id, fingerprint, constraint, agent_ids=[], dead_pool=[], champion_ids=[], fitness, innovation_factor={0,0},stats=[]}).
- record(avatar,{id,sector,morphology,energy=0,health=0,food=0, age=0, kills=0, loc, direction,r, mass, objects=[], state,actuators,sensors}).

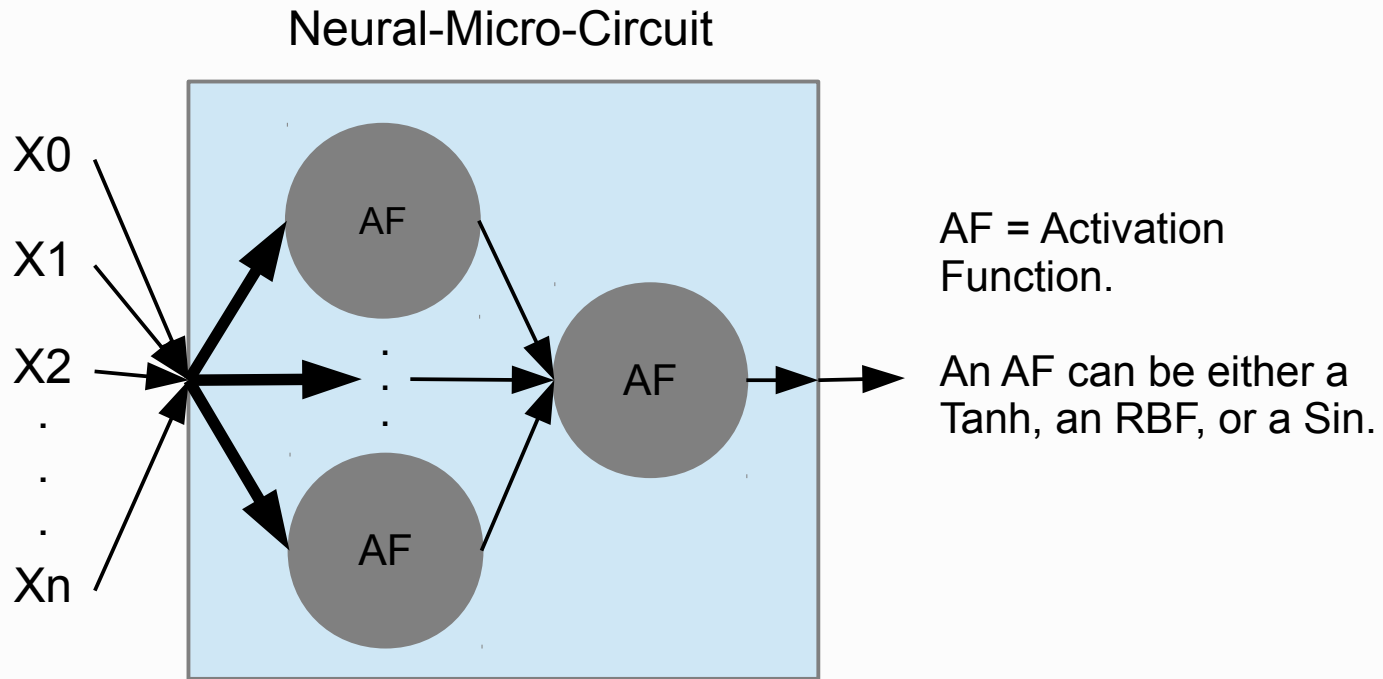
```
[[{agent,...."ids and general agent information"...},  
{cortex,{{origin,7.427859664144057e-10},cortex},  
test,  
[{{0.0,7.427859664110573e-10},neuron}},  
[{{-1,7.427859664112002e-10},sensor}},  
[{{1,7.427859664111848e-10},actuator}}}  
{sensor,{{-1,7.427859664112002e-10},sensor},  
pb_GetInput,standard,  
{{origin,7.427859664144057e-10},cortex},  
{private,pb_sim},  
3,  
[{{0.0,7.427859664110573e-10},neuron}},  
0,undefined,  
[3],  
undefined,undefined,undefined,undefined,undefined}]
```

```
{neuron,{{0.0,7.427859664110573e-10},neuron},  
0,  
{{origin,7.427859664144057e-10},cortex},  
undefined,undefined,tanh,undefined,  
{none,[]},  
dot_product,  
[{{{-1,7.427859664112002e-10},sensor},  
[{{0.15516645684354882,[]},  
{0.4631980138130717,[]},  
{0.4869749390984265,[]}}]}],  
[],  
[{{1,7.427859664111848e-10},actuator}},  
[]}  
{actuator,{{1,7.427859664111848e-10},actuator},  
pb_SendOutput,standard,  
{{origin,7.427859664144057e-10},cortex},  
{private,pb_sim},  
1,  
[{{0.0,7.427859664110573e-10},neuron}},  
0,undefined,  
[with_damping,1],  
undefined,undefined,undefined,undefined,undefined}]
```

Recent Updates

- Hall-Of-Fame/Archiving
- Multi-Objective Optimization
- Novelty Search
- Neural-Micro-Circuit
- Adaptive-Resonance-Theory

Larger Basic Building Blocks: Neural-Micro-Circuits



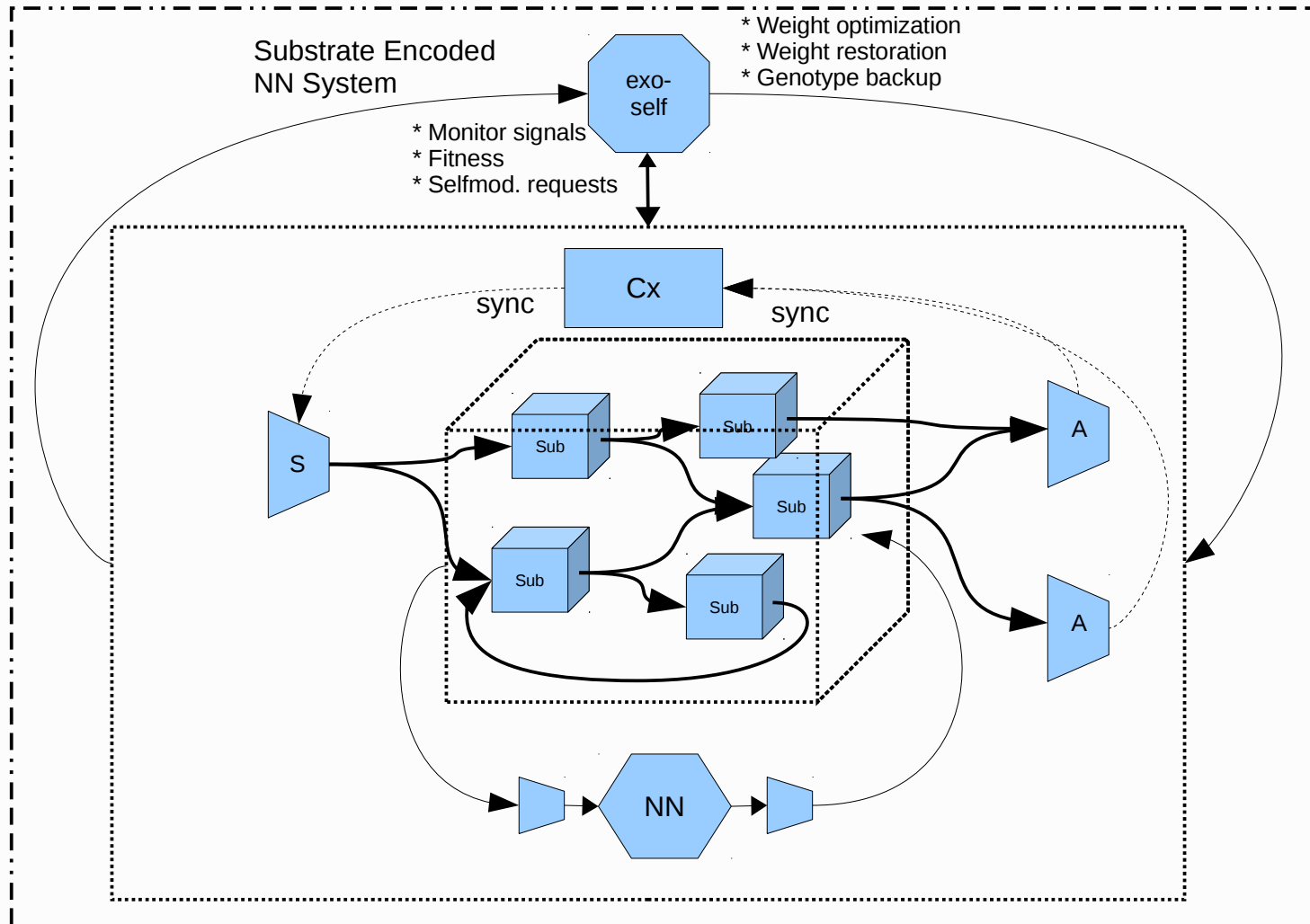
NMC Node

```
create_circuit(IVL,Densities,AF)->create_circuit(IVL,Densities,AF,[]).
create_circuit(IVL,[VL|Densities],AF,Acc)->
  {Weights,Parameters} = case AF of
    rbf ->
      {[random:uniform()-0.5|| _<-lists:seq(1,IVL)], [random:uniform()]};
    _ ->
      {[random:uniform()-0.5|| _<-lists:seq(1,IVL)], []}
  end,
  Layer=[#neurode{id=technome_constructor:generate_UniqueId(),af=AF,weights=Weights} || _<-lists:seq(1,VL)],
  create_circuit(length(Layer),Densities,AF,[Layer|Acc]);
create_circuit(_IVL,[],_AF,Acc)->
  lists:reverse(Acc).

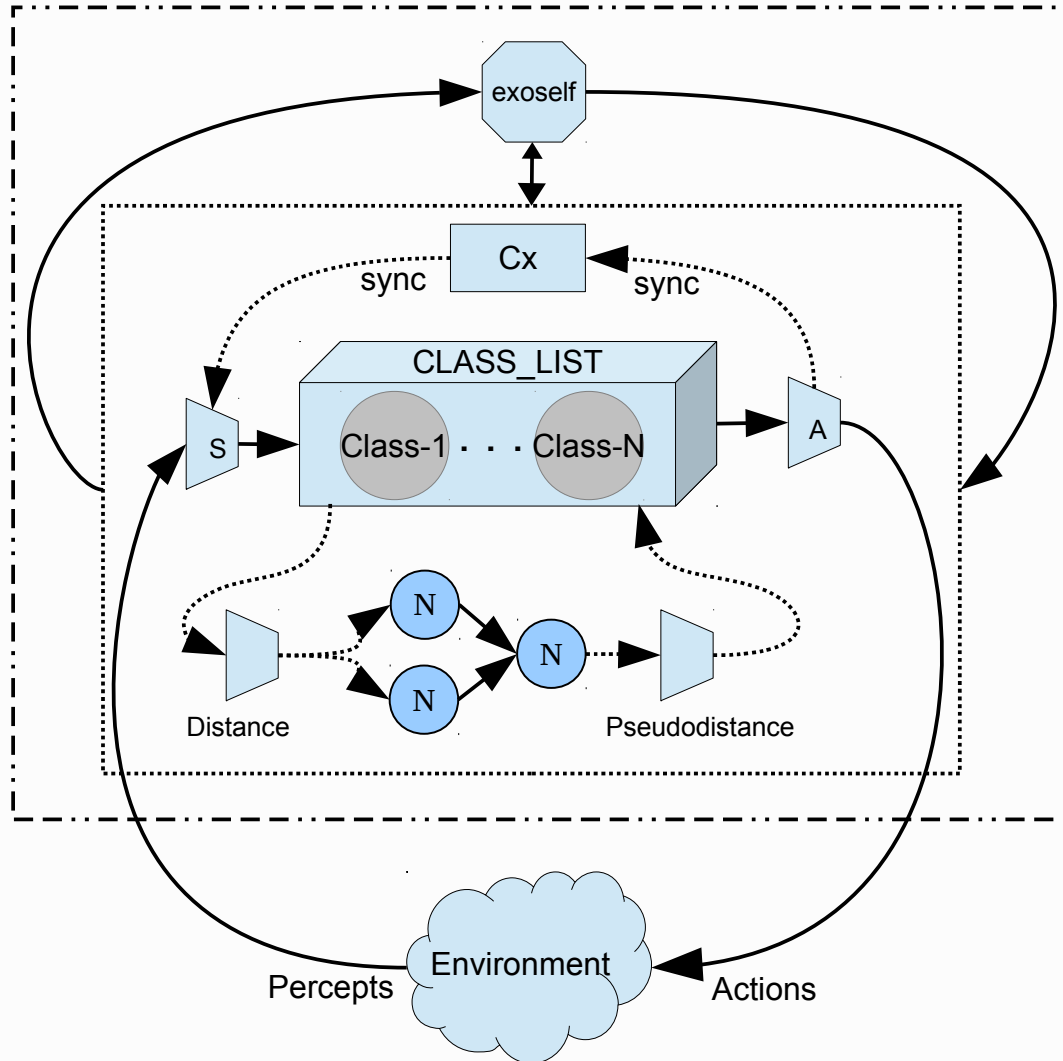
calculate_output_std(IVector,[Cur_NeurodeLayer|Circuit])->
  U_IVector = [calculate_neurode_output_std(IVector,N#neurode.weights,N#neurode.bias,0) || N <- Cur_NeurodeLayer],
  calculate_output_std(U_IVector,Circuit);
calculate_output_std([Output],[])->
  Output.

calculate_neurode_output_std([I|IVector],[Weight|Weights],Bias,Acc)->
  calculate_neurode_output_std(IVector,Weights,Bias,I*Weight+Acc);
calculate_neurode_output_std([],[],undefined,Acc)->
  functions:tanh(Acc);
calculate_neurode_output_std([],[],Bias,Acc)->
  functions:tanh(Acc+Bias).
```

Distributed Substrates

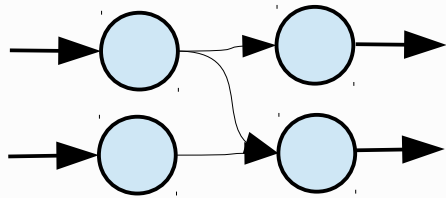


ARTMAP Agent

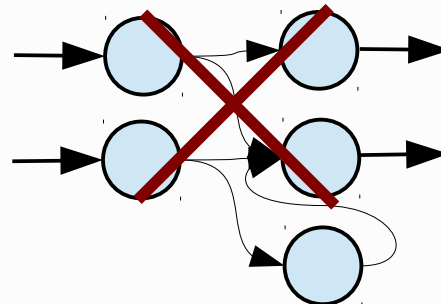
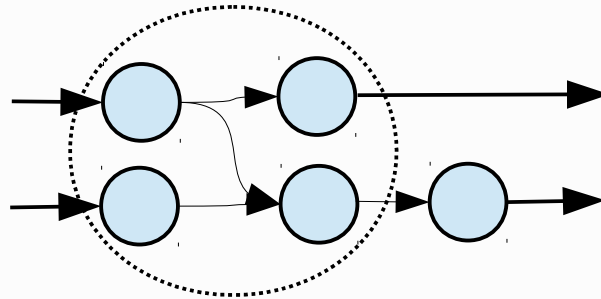
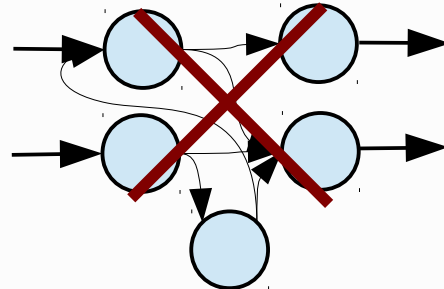


Crystallization/ADFs

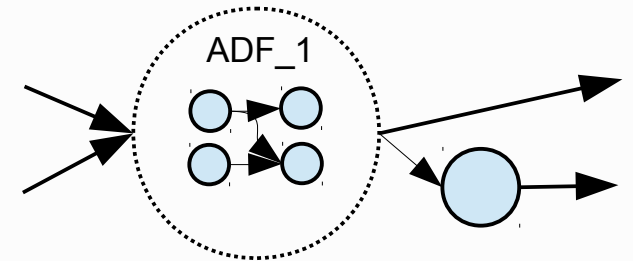
Automatically Defined Functions (ADFs), subgraphs not mutated in a long while, become treated as units.



Generation: N

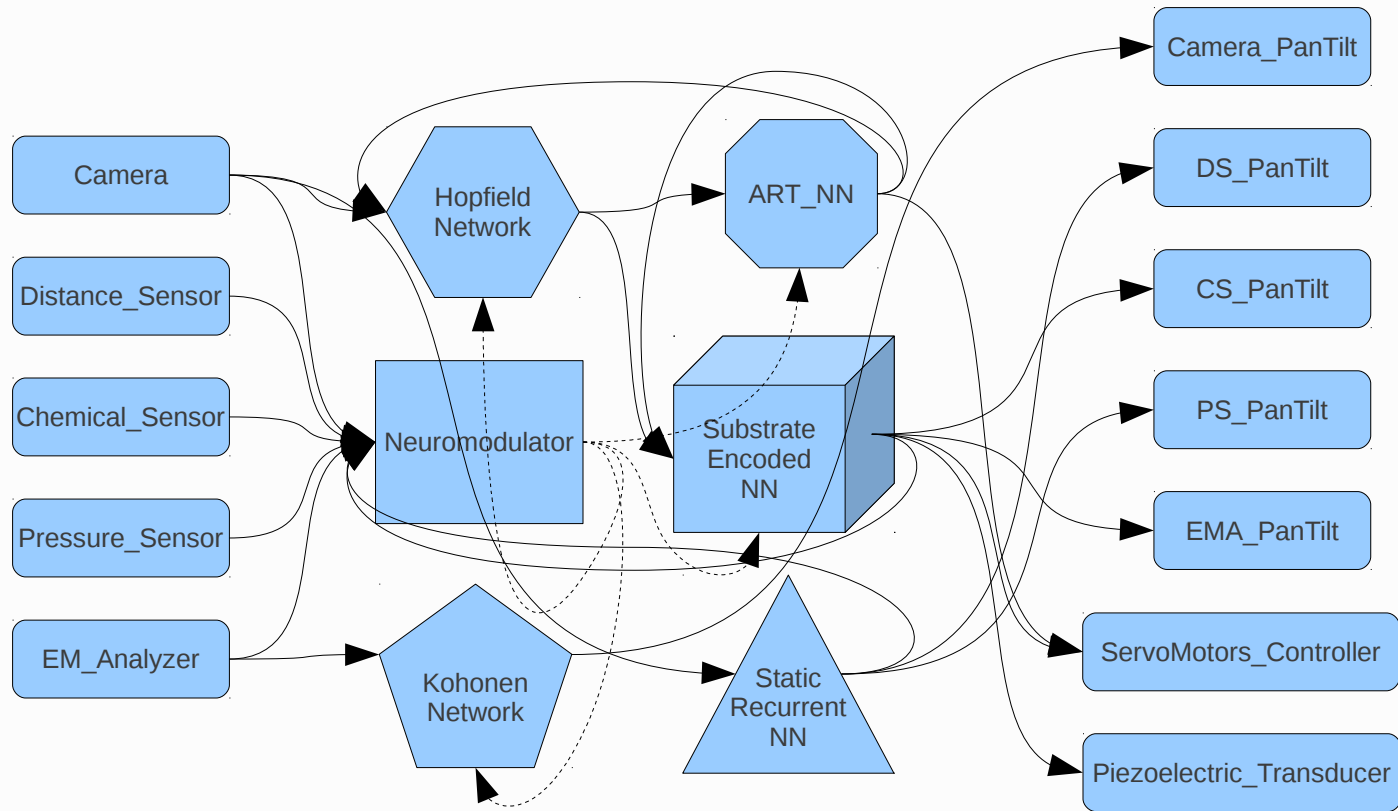


Generation: N+1



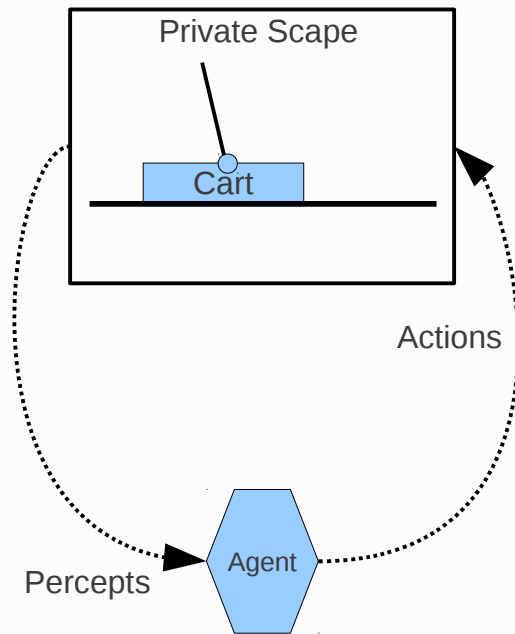
Generation: N+2

Modular NNs

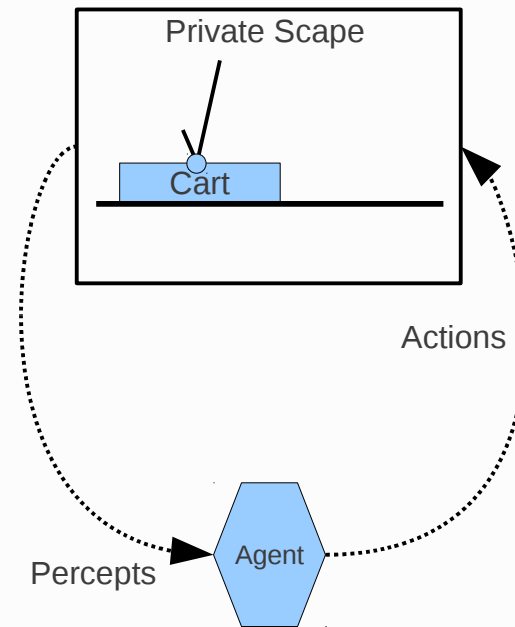


The Pole Balancing Benchmark

A. Single pole balancing simulation



B. Double pole balancing simulation



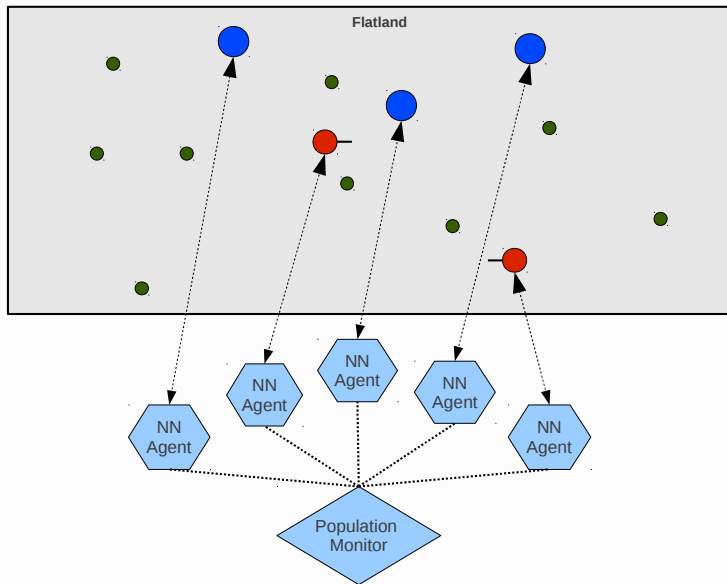
Double Pole Balancing Benchmark

Method	Without-Damping	With-Damping
RWG	415209	1232296
SANE	262700	451612
CNE	76906	87623
ESP	7374	26342
NEAT	---	6929
CMA-ES*	3521	6061
CoSyNE*	1249	3416
<i>DXNN (old)</i>	2359	2313
DXNN	1289	1830
DXNN:NMC	1618	1703

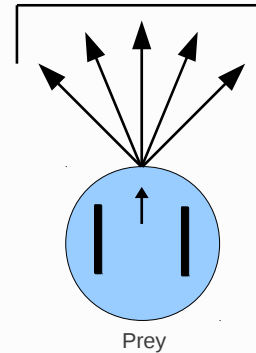
Benchmark data taken from: Faustino “Gomez, Jurgen Schmidhuber, Risto Miikkulainen,: Accelerated Neural Evolution through Cooperatively Coevolved Synapses. Journal of Machine Learning Research 9 (2008) 937-965”

Artificial Life

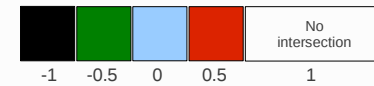
- Simple Food Gathering
- Dangerous Food Gathering
- Predator Vs. Prey



90 Degree Coverage
Resolution: 5

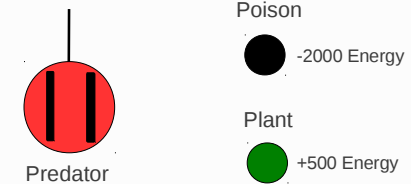


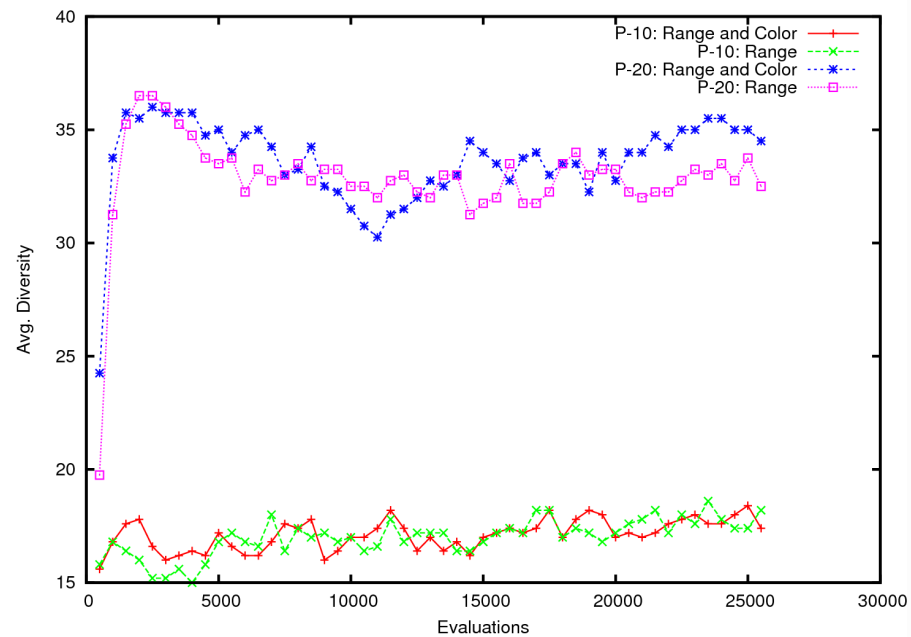
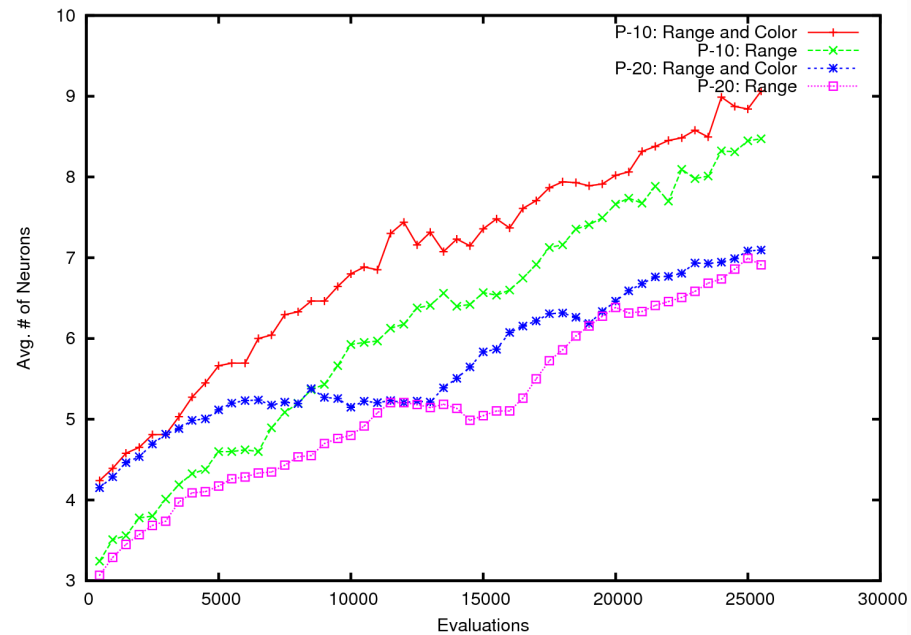
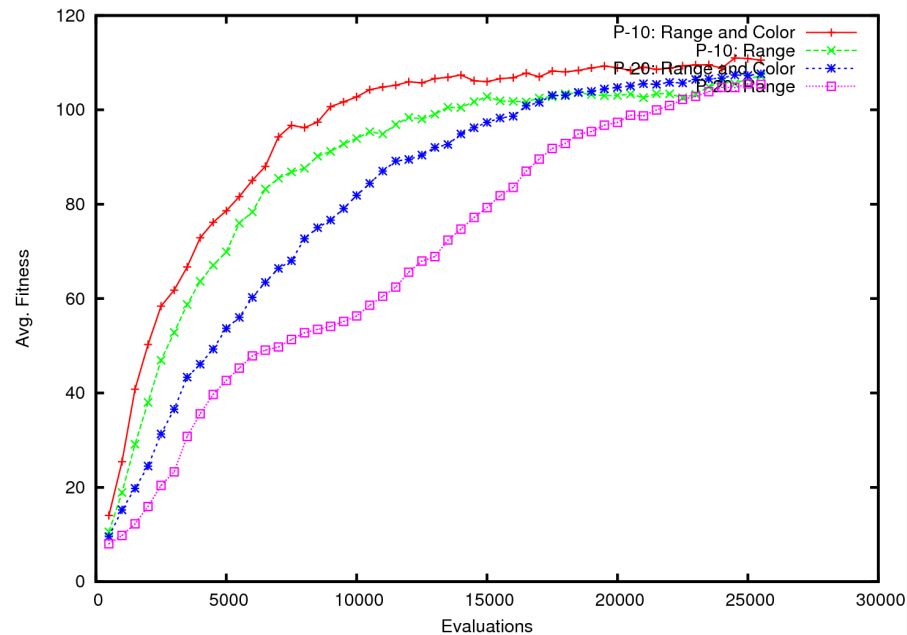
Color to floating point encoding:



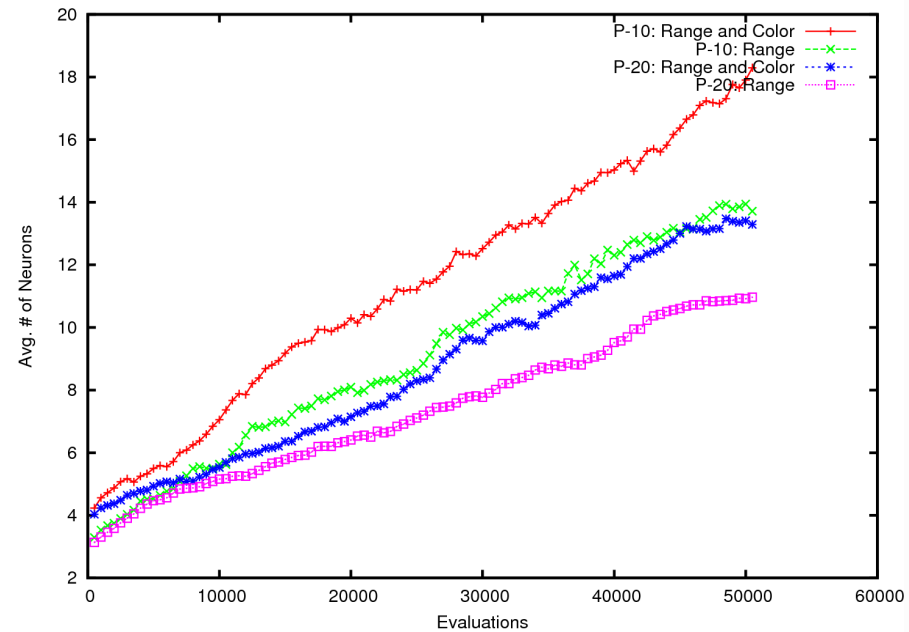
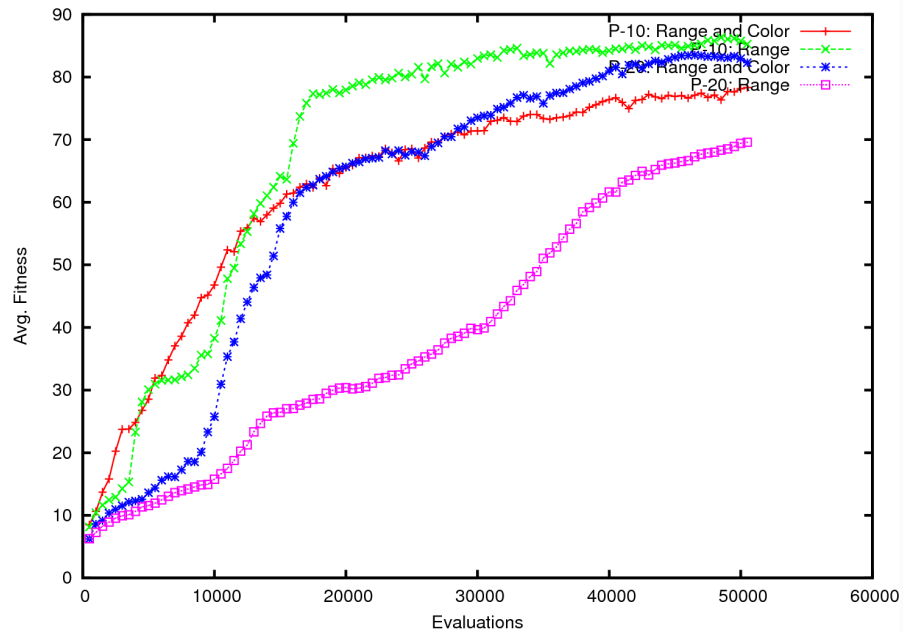
Sensors Available:
* Range Sensor:
Resolution-5
* Color Sensor:
Resolution-5

Actuators Available:
* Differential Drive

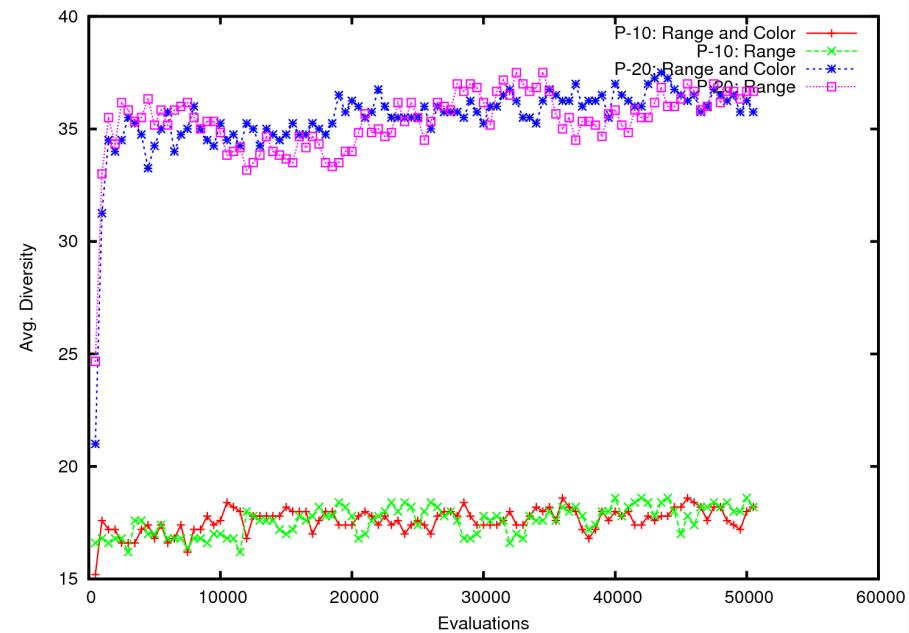




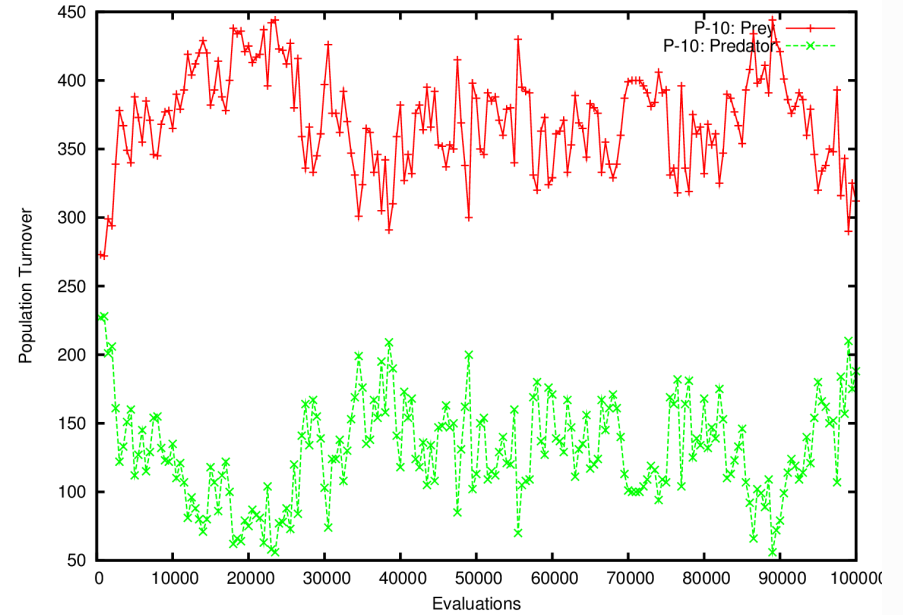
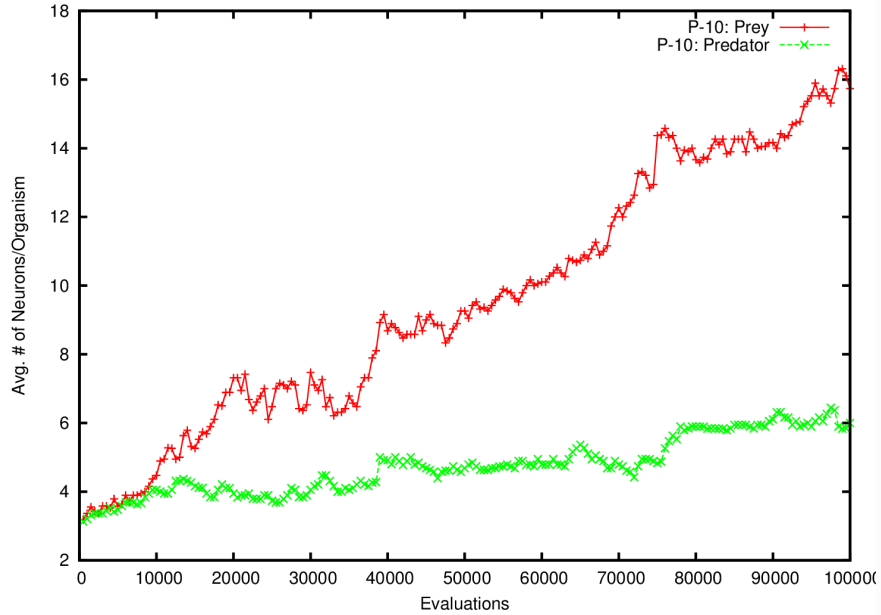
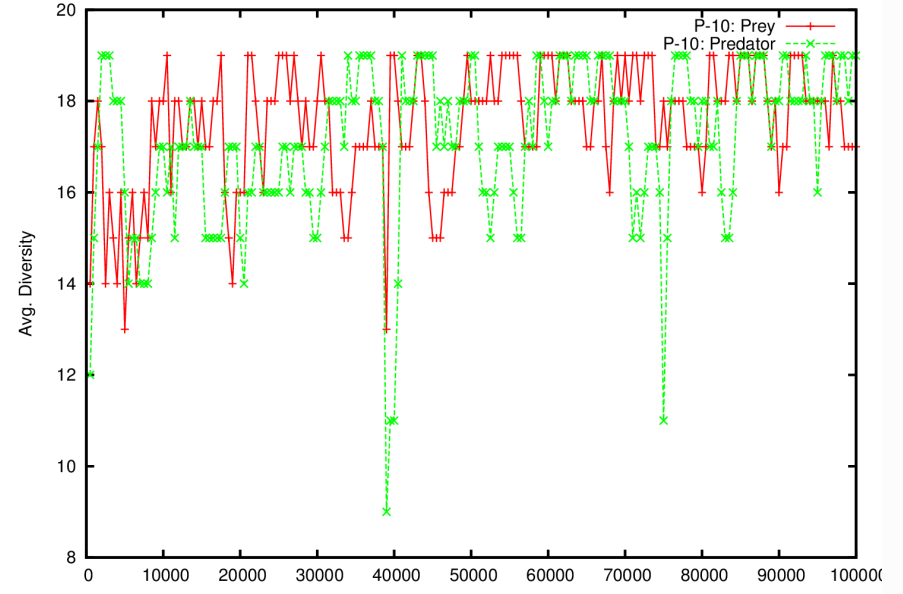
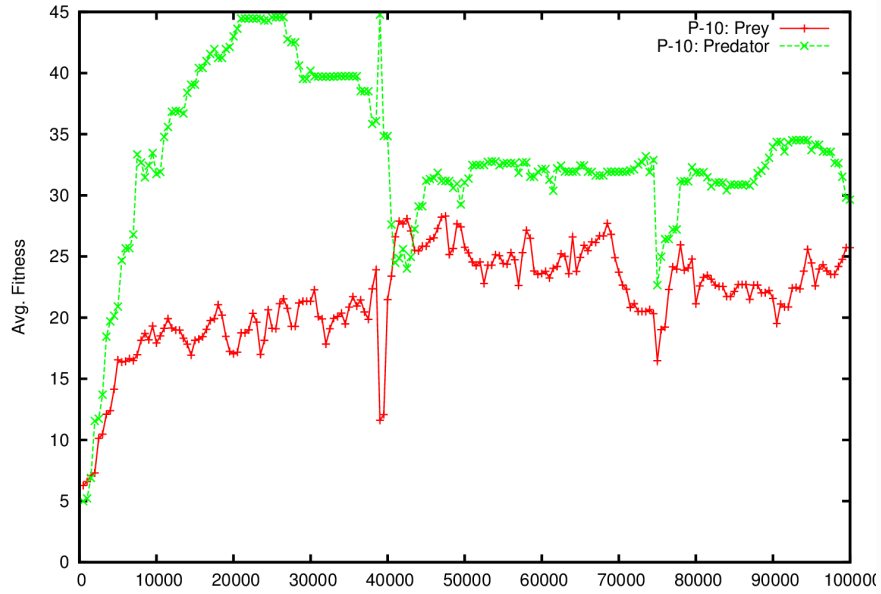
Simple Food Gathering



Dangerous Food Gathering

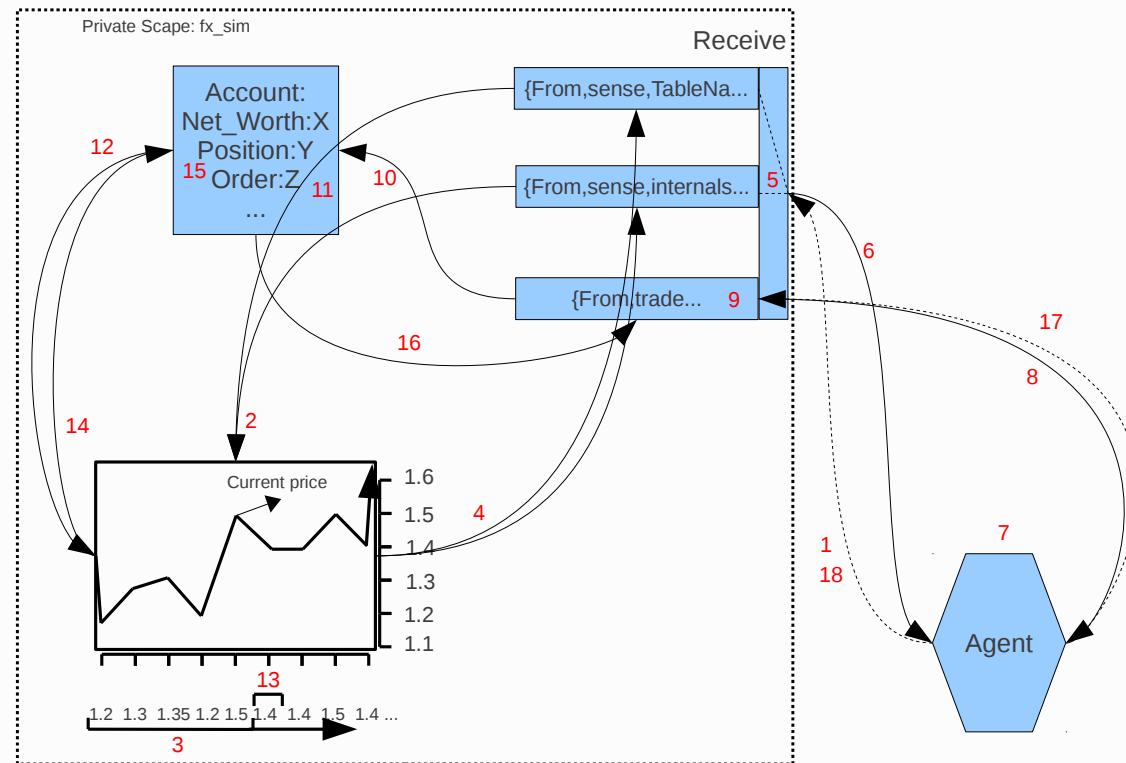


Predator Vs. Prey

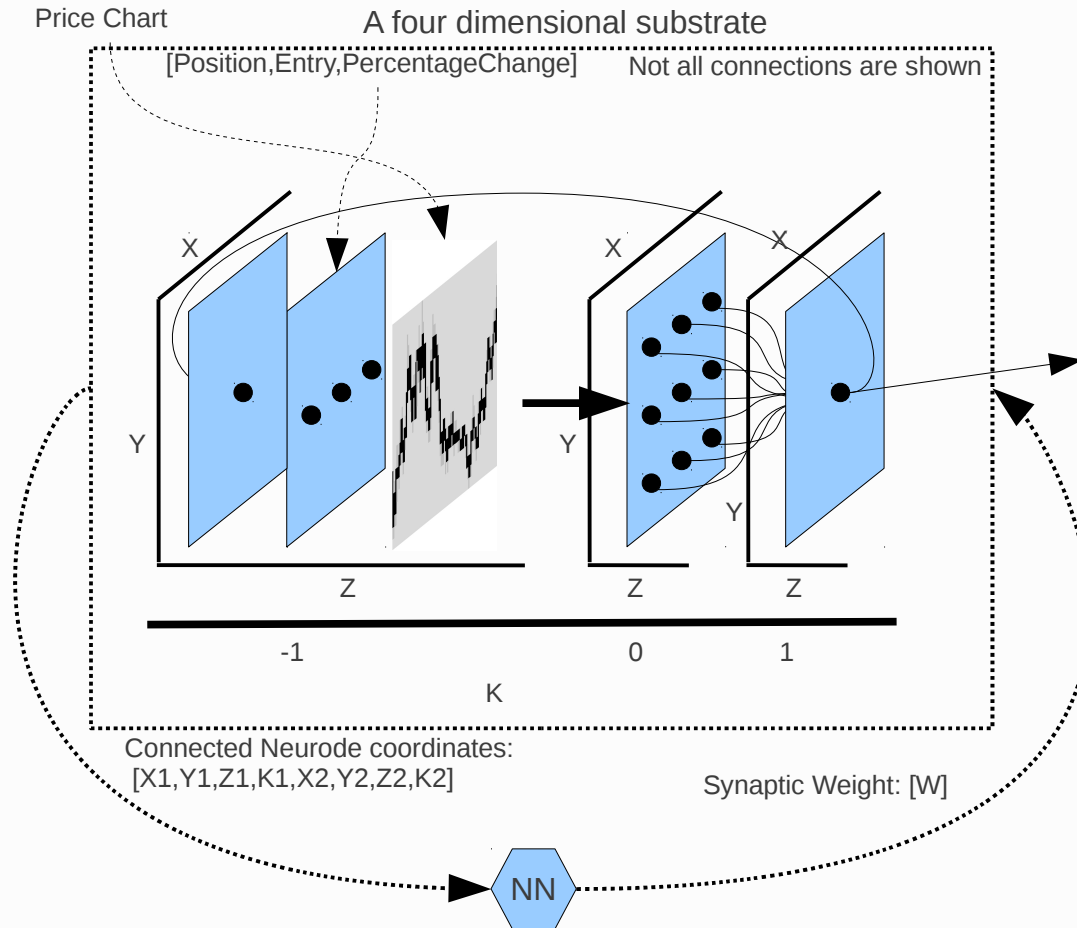


Forex Trading

- Trading using sliding window
- Trading using chart window



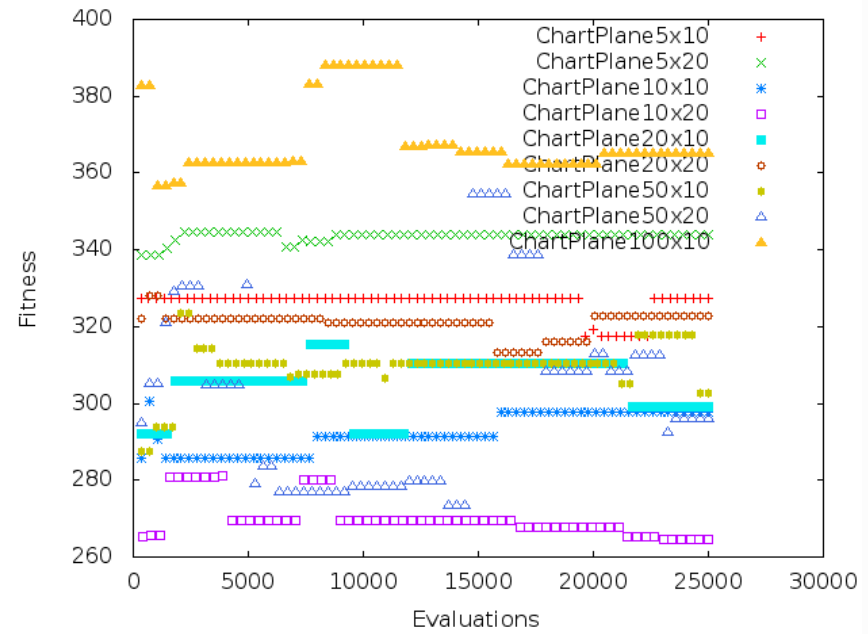
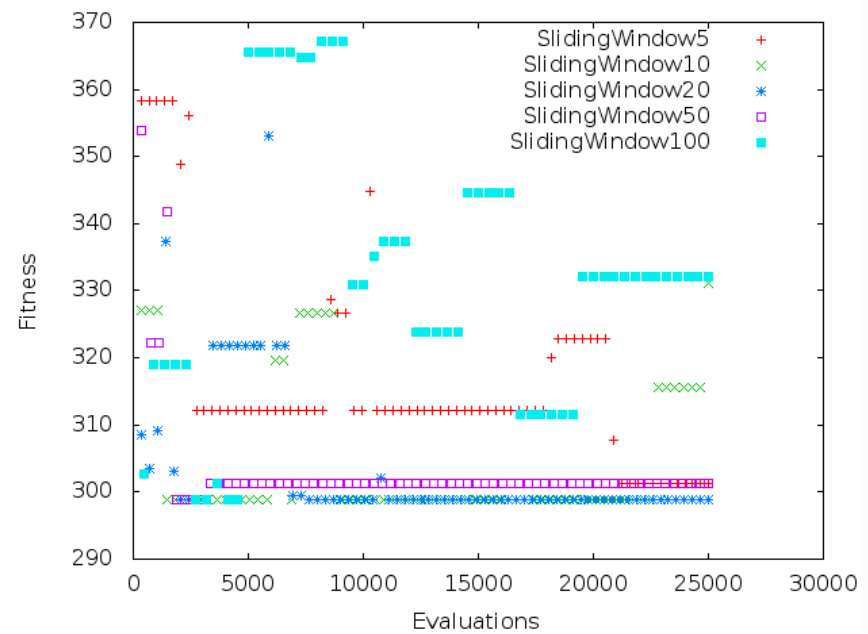
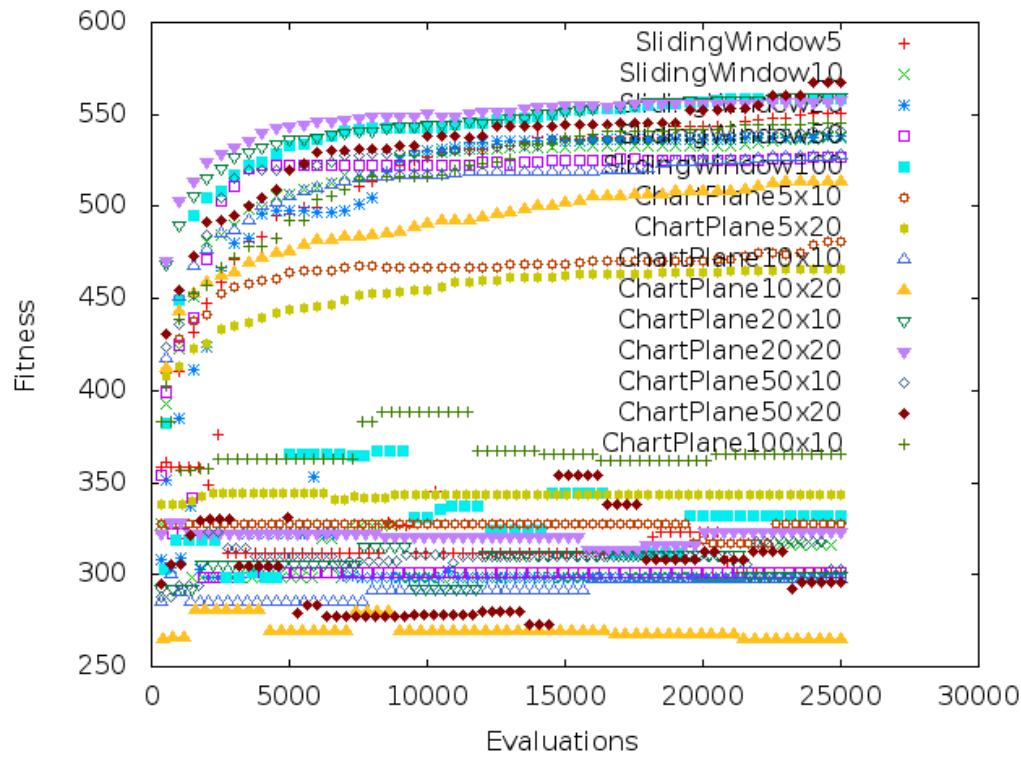
The Substrate Topology



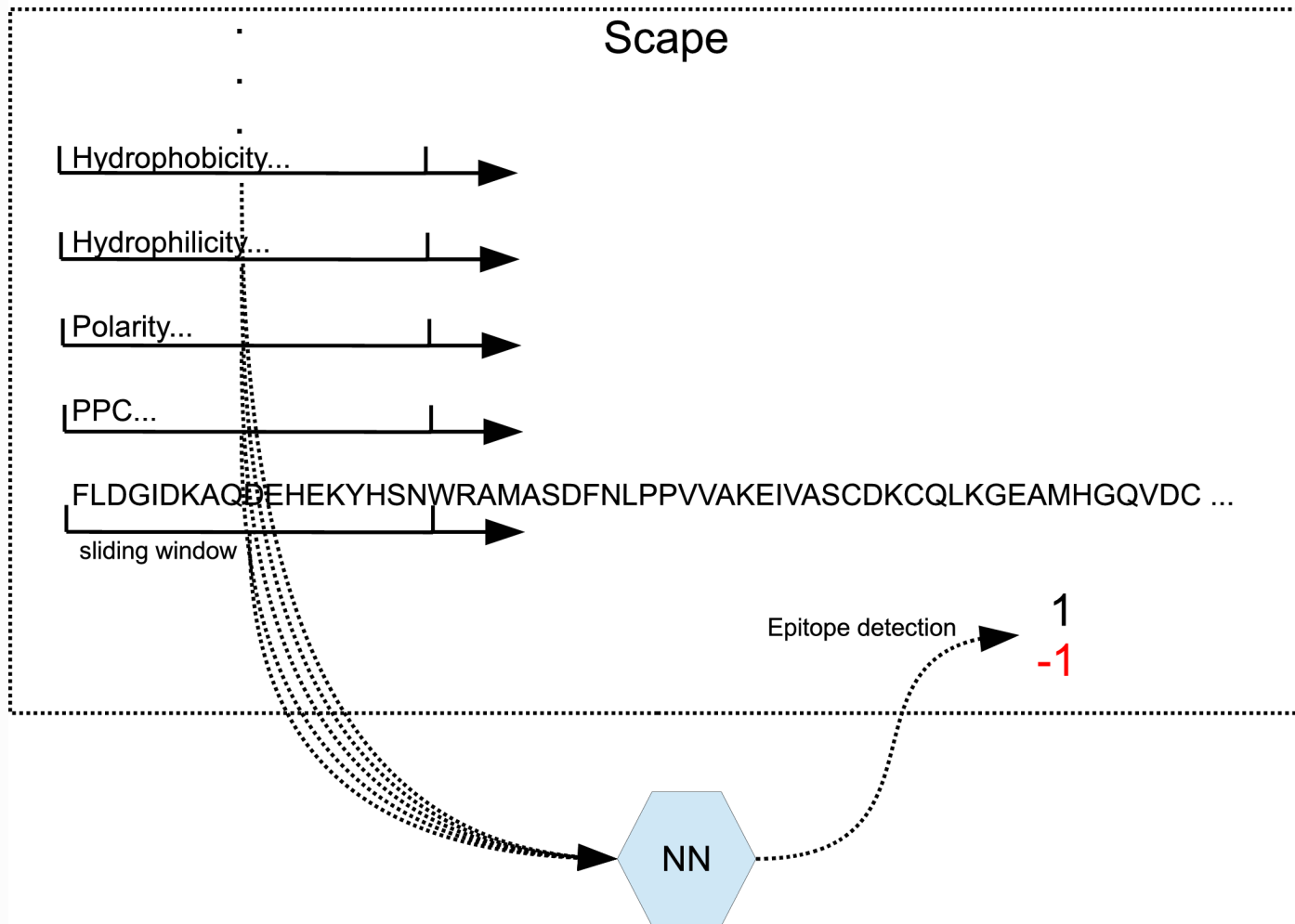
Forex Trading Results

TrnAvg	TrnBst	TstWrst	TstAvg	TstStd	TstBst	Price Vector Sensor Type
540	550	225	298	13	356	[SlidWindow5]
523	548	245	293	16	331	[SlidWindow10]
537	538	235	293	15	353	[SlidWindow20]
525	526	266	300	9	353	[SlidWindow50]
548	558	284	304	14	367	[SlidWindow100]
462	481	214	284	32	346	[ChartPlane5X10]
454	466	232	297	38	355	[ChartPlane5X20]
517	527	180	238	32	300	[ChartPlane10X10]
505	514	180	230	26	292	[ChartPlane10X20]
546	559	189	254	29	315	[ChartPlane20X10]
545	557	212	272	36	328	[ChartPlane20X20]
532	541	235	279	23	323	[ChartPlane50X10]
558	567	231	270	20	354	[ChartPlane50X20]
538	545	256	310	37	388	[ChartPlane100x10]
311	N/A	N/A	300	N/A	N/A	Buy & Hold
N/A	704	N/A	N/A	N/A	428	Max Possible

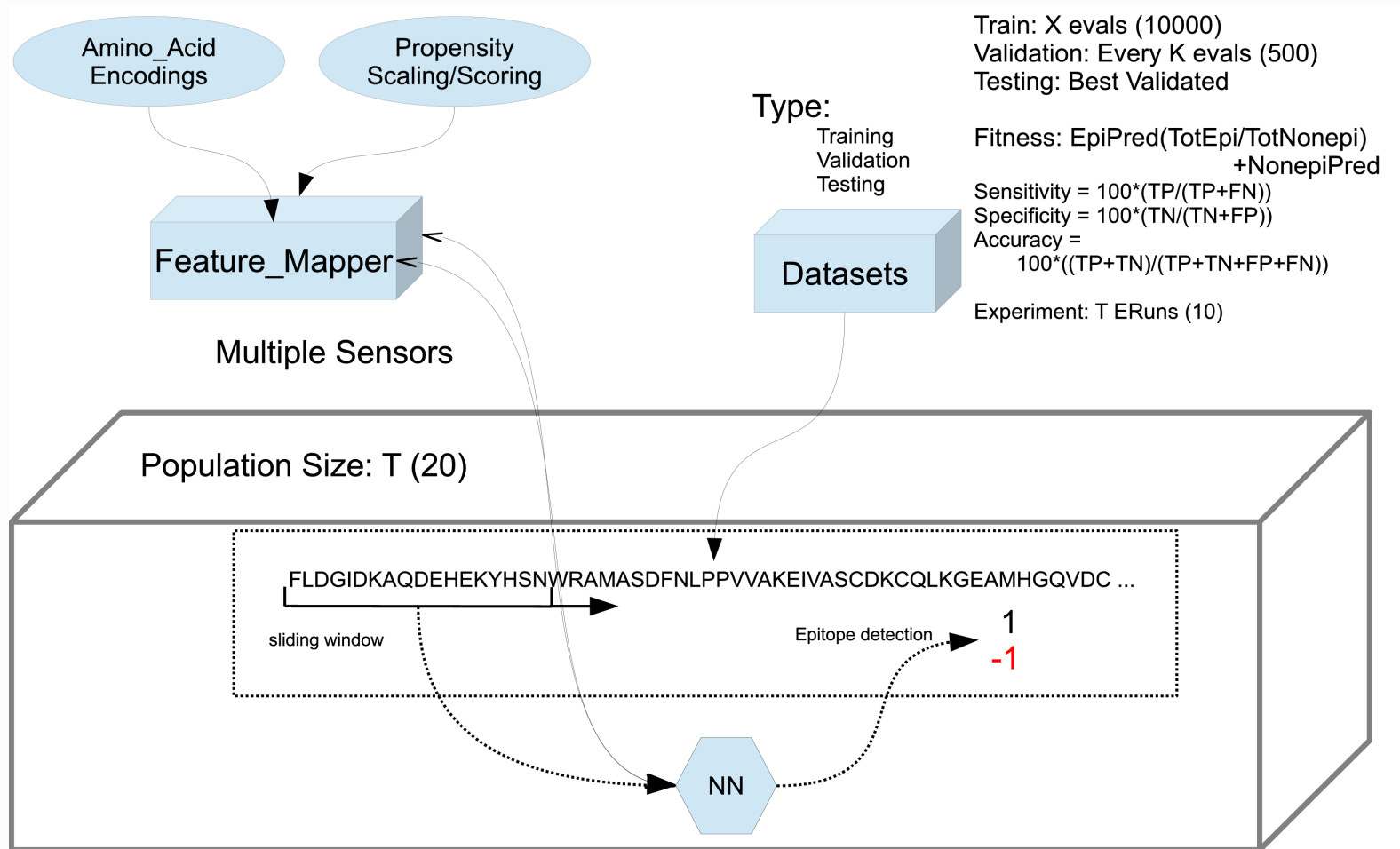
Generalization Results



Epitope Prediction



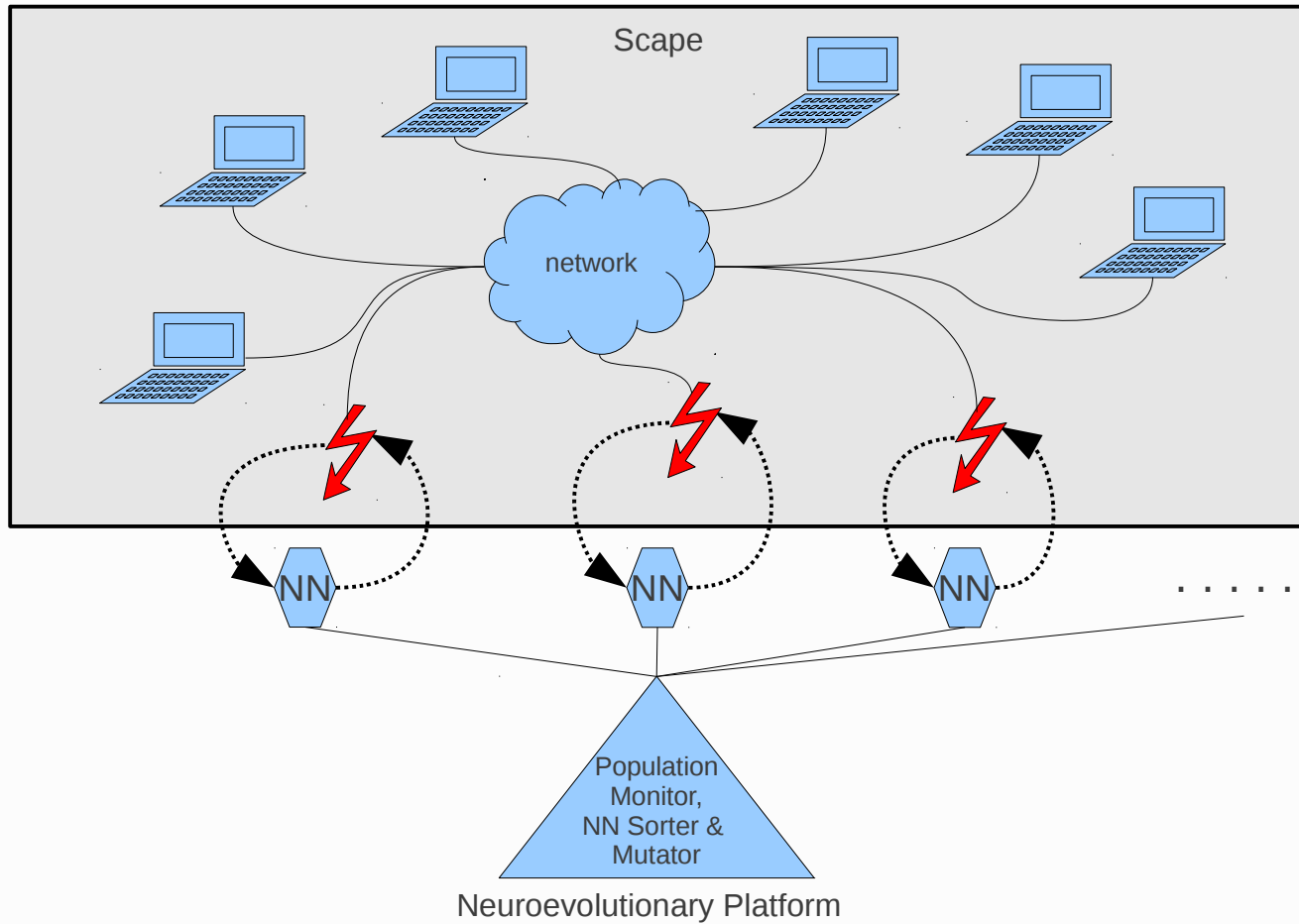
Epitope Prediction Platform



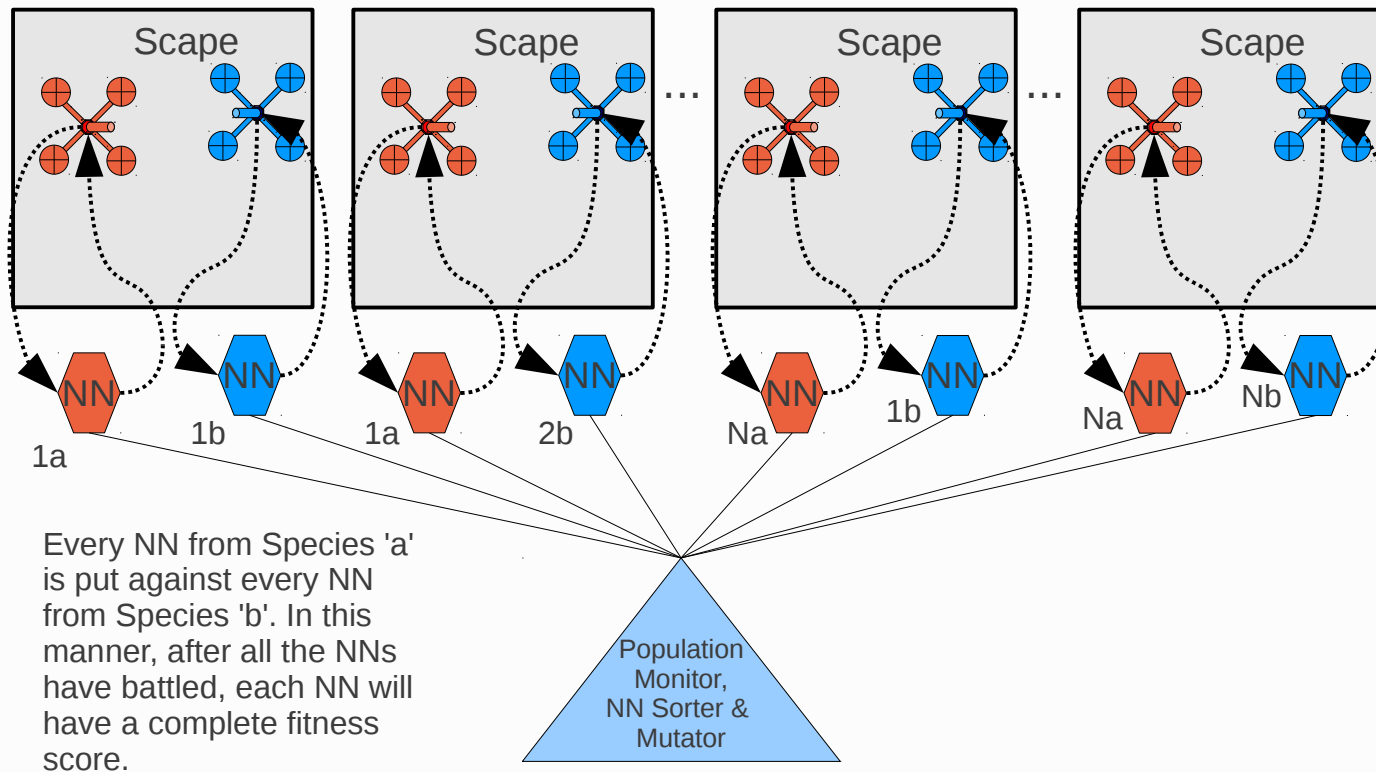
Beyond The Horizon

- Trivial to distribute a NN over the Internet
- This adds an enormous amount of robustness and computational power
- Erlang's natural code hot-swapping ability, potentially allows neural networks to rewrite their own source code, without going off-line. If something goes wrong, if the rewriting causes a crash to the network, the exoself can recover the system...
- Building modular neural networks, composed of very different structures, becomes trivial
- Other scientific applications in the multi-agent based field
 - Cyberwarfare
 - Circuit
 - Economic multiagent based simulations
 - ...

Cyberwarfare



Evolving UCAV Neurocontrollers



Conclusion & Summary

- Common programming language do not have the architecture that is perfect for modern Neural Network based Computational Intelligence
- A perfect functional programming language already exists, it is Erlang, the quintessential NN programming language, with a 1:1 mapping.
- The first of fully general Topology and Parameter Evolving Universal Learning Networks in Erlang has been created, called DXNN.
- New horizons have opened up that can now be explored with ease. Experiments within self recovery, global distribution of a NN, self rewriting... are all easily accomplished due to the features Erlang possesses
- It is essential for the scientific community to begin utilizing this language, as the hardware will only continue to scale outwards, and whereas languages like Scala are Java extensions, Erlang was built from the start for robustness, scalability, distribution...

References

- Joe Armstrong, “Making reliable distributed systems in the presence of software errors ” A Dissertation submitted to the Royal Institute of Technology Stockholm, Sweden
- Bjarne Däcker, “Concurrent functional programming for telecommunications: A case study of technology introduction” November 2000. Licentiate Thesis
- Gene Sher (2012), “Handbook of Neuroevolution Through Erlang” Springer-Verlag, New York

Thank You

Questions?