Biologically Inspired Computation

Dr Marta Vallejo

m.vallejo@hw.ac.uk

Lecture 9

Neuroevolution

- Neuroevolution
- Why evolution for training ANNs?
- Applications in different Domains
- Different types of Evolution
- Evolution of Weights
- Training an ANN with a fix topology
- Evolution by Composition
- Evolving Topologies: NEAT
- Network Encoding
  - NEAT: Mutation
  - NEAT: Crossover
  - NEAT: Example & Code
- Neuroevolution for Deep Learning Networks
- More material about neuroevolution

Neuroevolution

Despite the advantages and good results achieved by deep learning, their **success** depends on finding an architecture to fit the task

Pulina, L., Tacchella, A.: NeVer: a tool for artificial neural networks verification

Deep learning has scaled up to be applied to more challenging tasks

Architectures have become difficult to design by hand

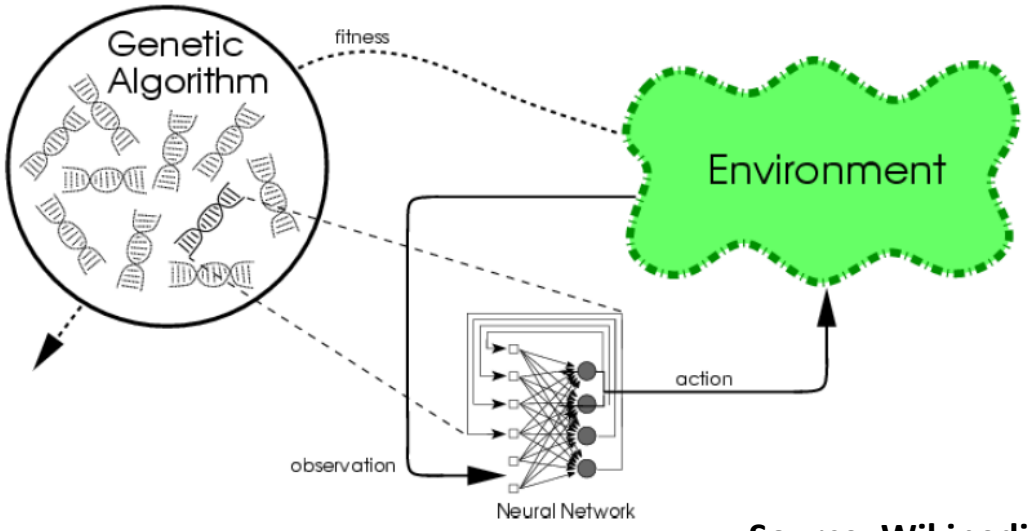
Three challenges:

- how to design the components of the architecture
- how to put them together into a full network topology
- how to set the hyperparameters for the components and the global design

These three aspects need to be addressed **separately** for each new task

Neuroevolution

Neuroevolution is a method for modifying neural network weights, topologies, hyperparameters or ensembles to learn a specific task.



Source: Wikipedia

Investigated for more than 30 years. Can be applied to all types of ANN:

- Feedforward neural networks (supervised learning)
- Recurrent neural networks
- Neural networks trained by reinforcement learning.

More successful for harder problems: unsupervised & reinforcement learning

Why evolution for training ANNs?

Using evolution has several advantages in comparison with backpropagation. Benefits due to the general nature of EA:

- You can decide previously the architecture and learn only the weights (like in backpropagation), but you can learn both things at the same time: **topology + weights**
- The population-nature of the algorithm makes easier to find the global optimum instead of getting stuck in any local minima.
- Some models will perform better than others, just because how they have been initialised. A population of models can minimise this problem.
- Highly general: allows learning without explicit targets
- Remove the constraints of gradient descent: we can make use of **non-differentiable** activation functions (more global search) and we don't have to worry about vanishing gradient problem.

Application in Different Domains

- Powerful method for sequential decision tasks
  Rawal, MiikkulainenFrom Nodes to Networks: Evolving Recurrent Neural Networks

- Also may be useful in supervised tasks, specially when network topology is important
  Montana, Davis: Training feedforward neural networks using genetic algorithms

- Can be applied to other learning paradigms: like a policy search method for reinforcement-learning problems.
  https://blog.openai.com/evolution-strategies/

Different types of Evolution

You can apply evolution at different levels:

- Use a predefined fixed topology and activation functions and evolve the weights.
- Focus on both weights & topology
- Concentrate only on the topology

When both architecture and weight information are encoded into individuals, the impact of random initial weights and training algorithm will be considerably reduced

Y. Liu and X. Yao, "Evolutionary design of artificial neural networks with different nodes,"

Evolution of Weights

GOAL: finding the parameter settings that results in the lowest cost

Chromosomes are strings of connection weights (bits or real)

- E.g. 10010110101100101111001
- Usually fully connected, fixed topology
- Initially random

Parallel search for a good neural network solution

- Each ANN is evaluated by the fitness function
- Good ANNs reproduce through crossover & mutation
- Bad ones are thrown away

Evolution converges the population, however:

- Diversity is lost; progress stagnates
- Different, incompatible encodings for the same solution
- Too many parameters to be optimised simultaneously

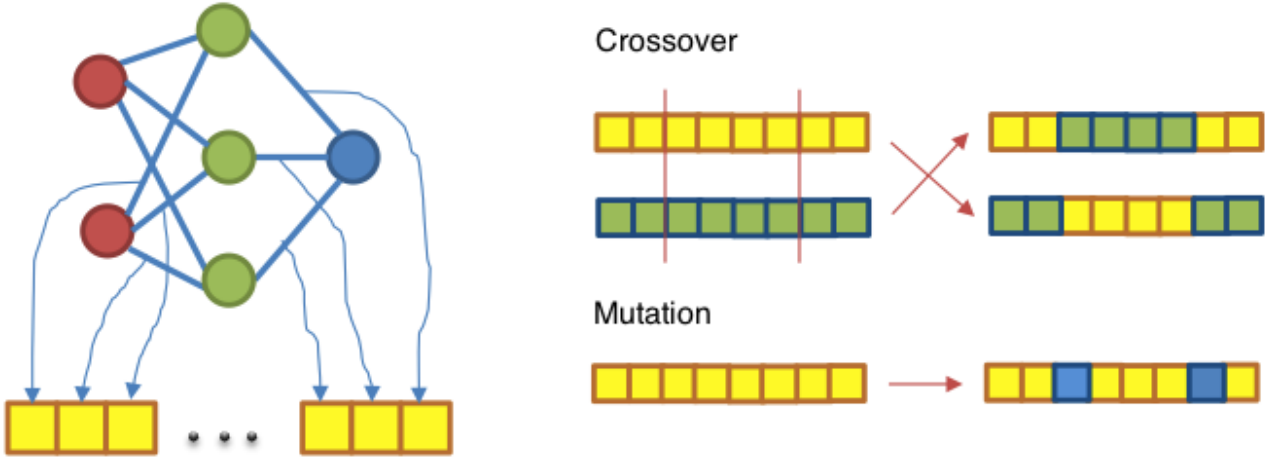
Training an ANN with a fix Topology

Choose your topology (hidden layers and corresponding units). This gives you the number of weights to evolve

The main idea is to transform the weights of the neural network into a **linear chromosome**

Decide the size of your population

Initialise the weights for each of them randomly



Transforming neural network into linear chromosome

Training an ANN with a fix Topology

Decide if you are going to use batch or online learning

Use the error generated by a forward pass as a fitness function

Selection + Mutation + Update or Selection + Mutation + Crossover + update

For the mutated/recombined chromosome: use the error generated by a forward pass as a fitness function to asses new offspring

Run until convergence: threshold or maximum number of iterations

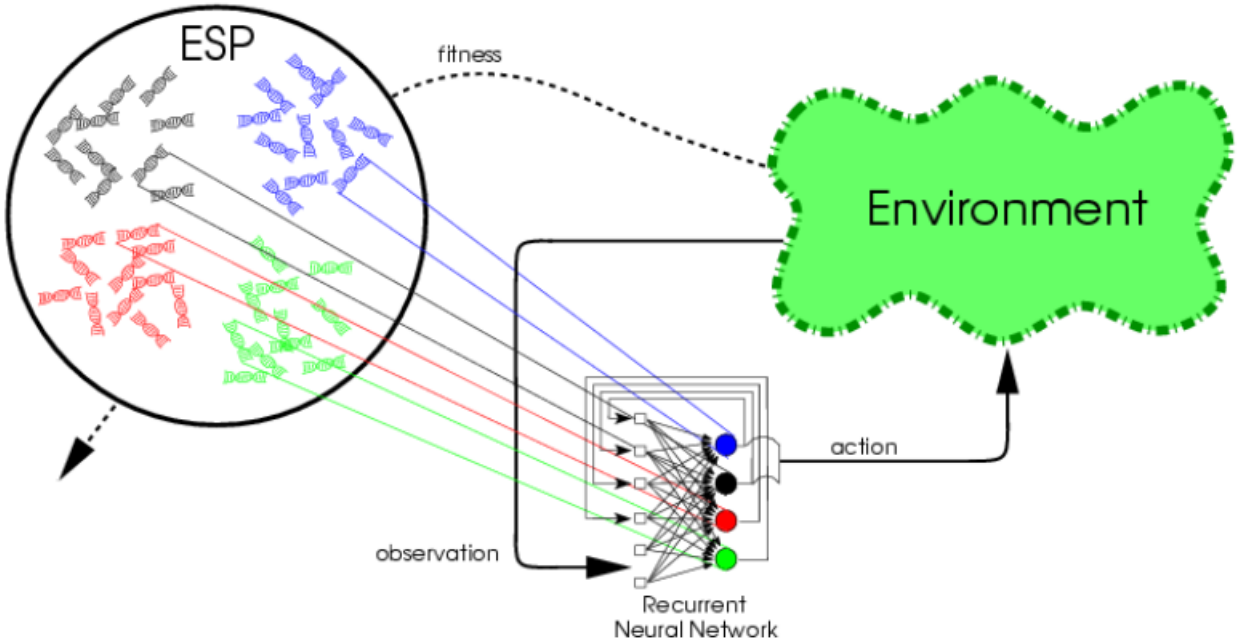
Evolution by Composition

Another approach: evolve partial neural networks and combine them into fully functional network. IDEA: Evolving individual neurons to cooperate in networks. Examples: SANE, ESP, and CoSyNE

Potter, Jong - Cooperative coevolution: An architecture for evolving coadapted subcomponents

E.g. Enforced Sub-Populations (ESP)

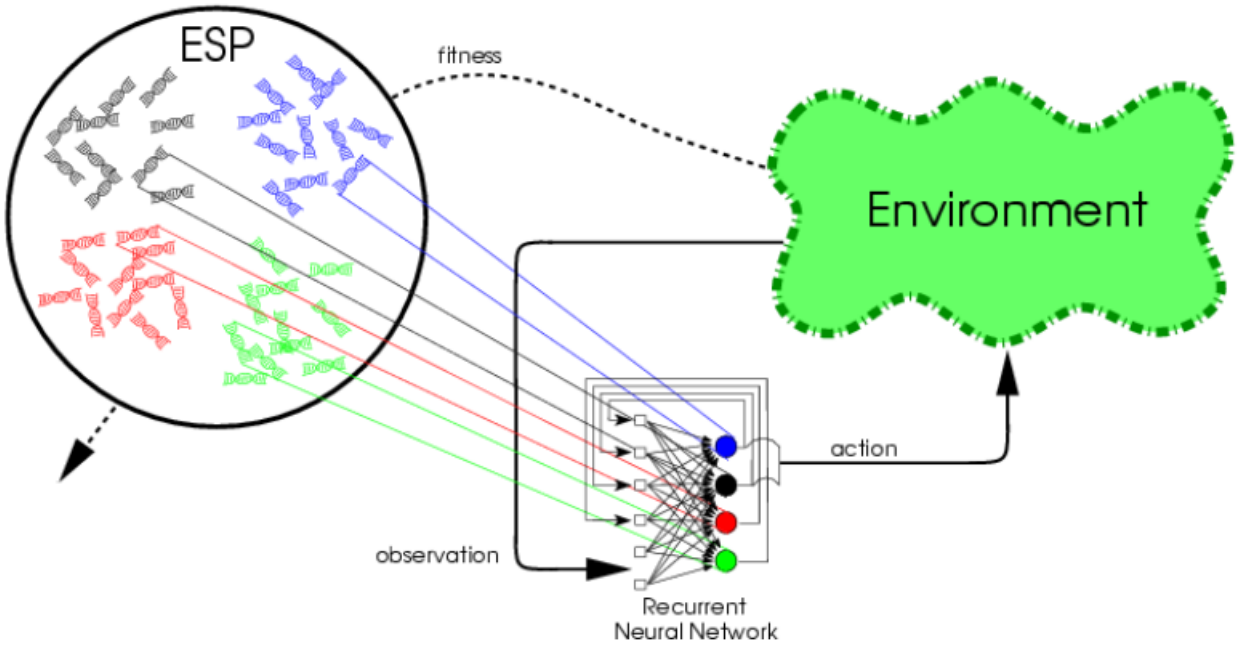
- Each (hidden) neuron in a separate subpopulation
- Fully connected; weights of each neuron evolved
- Populations learn compatible subtasks



Evolution by Composition

Enforced Sub-Populations (ESP)

- Large search space explored due to the division into subtasks
- Evolution encourages diversity automatically since good networks require different types of neurons
- Neurons optimised for compatible roles



Evolving Topologies

Optimising connection weights and network topology like in Cellular Encoding and NEAT

Neuroevolution of Augmenting Topologies (NEAT)

Stanley, Miikkulainen, Evolving Neural Networks Through Augmenting Topologies

Based on **complexification of networks**:

Keeps the search tractable by starting with a simple architecture and adding more sophistication with the use of mutations to add nodes and connections

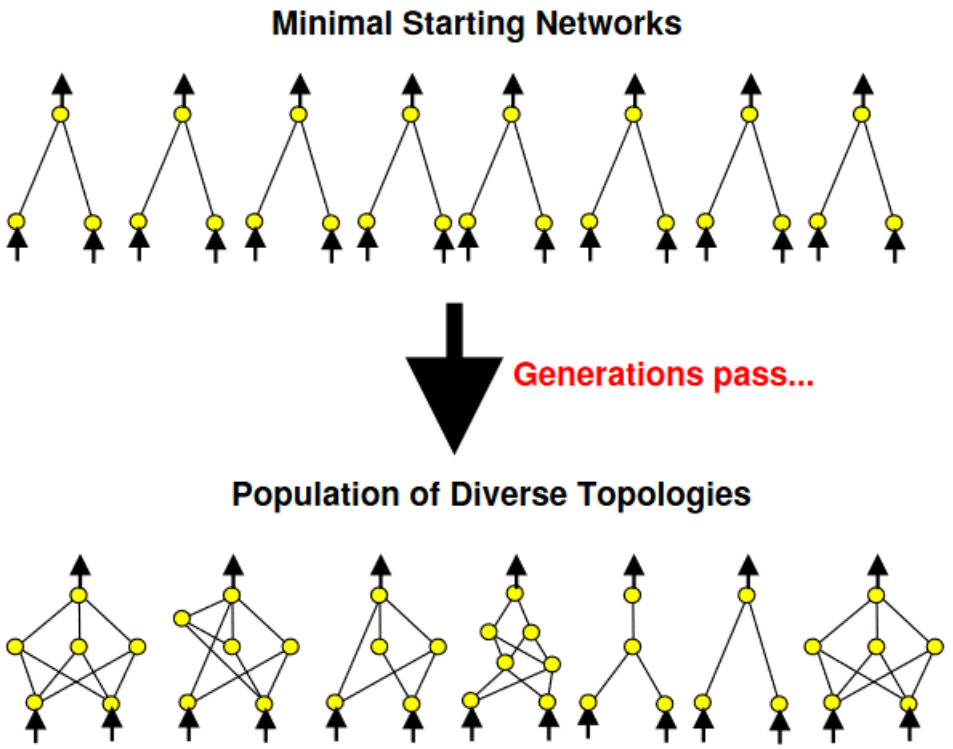
Just as organisms in nature increased in complexity since the first cell, so do neural networks in NEAT. This process of continual elaboration allows finding highly sophisticated and complex neural networks.

Cellular Encoding and NEAT particularly effective in determining the required recurrence

Evolving Topologies

Complexification:

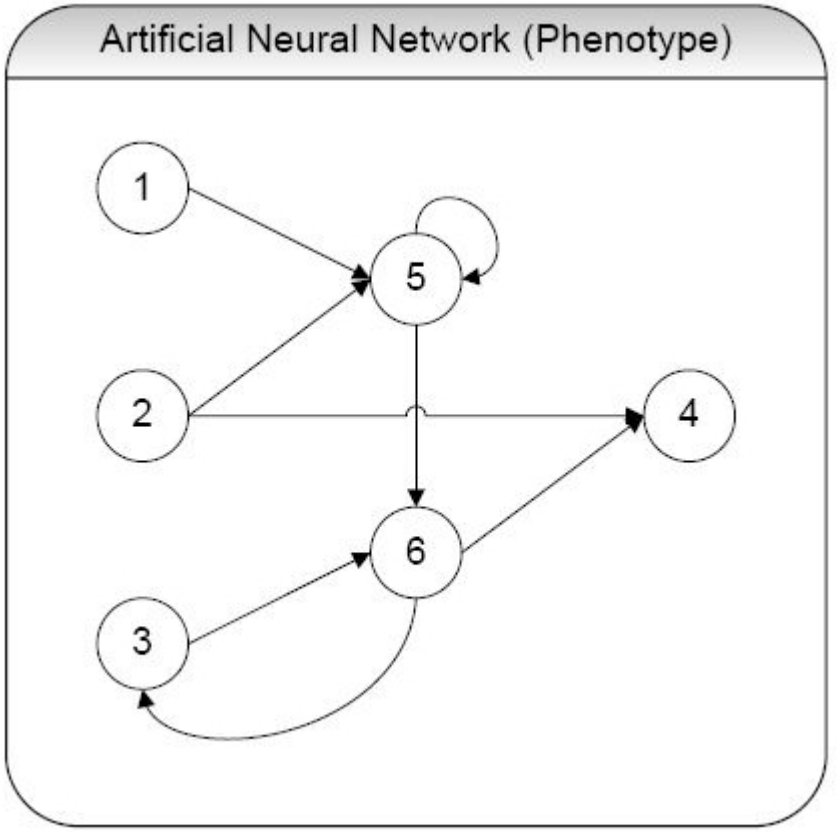
First, a population of **chromosomes** (each represented by a graph) with minimal complexity is created.



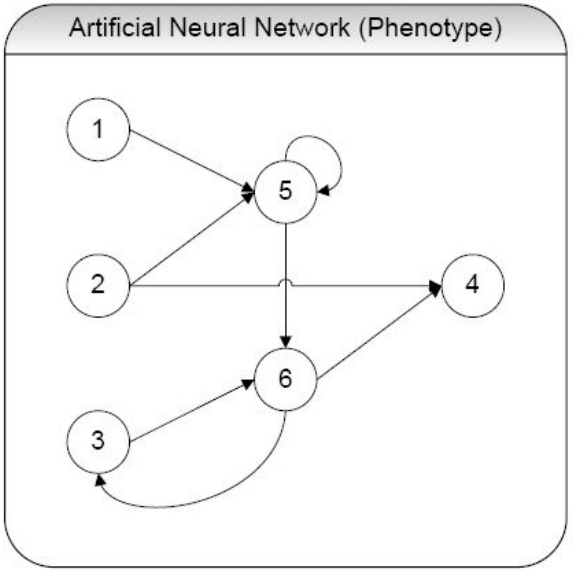
Problem with NEAT: Search space is too large (number of weights to be optimised) → limited to relatively small networks

Network Encoding

Phenotype (Network):



Network Encoding



Genotype (Network):

- List of Nodes Genes
- List of Link Genes

Genome (Genotype)

Node Genes

Node ID: 1 Type: Input Enabled bit: On Innovation: 1	Node ID: 2 Type: Input Enabled bit: On Innovation: 2	Node ID: 3 Type: Input Enabled bit: On Innovation: 3	Node ID: 4 Type: Output Enabled bit: On Innovation: 4	Node ID: 5 Type: Hidden Enabled bit: On Innovation: 5	Node ID: 6 Type: Hidden Enabled bit: On Innovation: 6

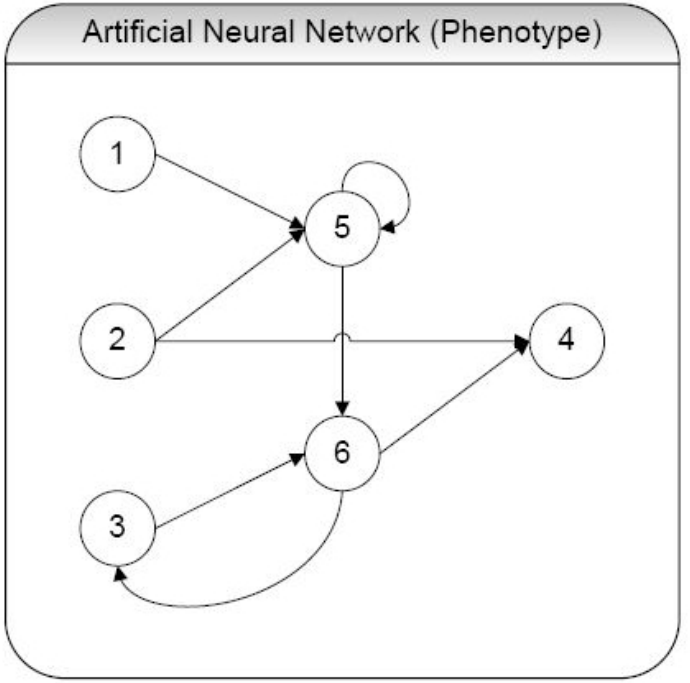
Link Genes

Input Node: 1 Output Node: 5 Weight: 0.6 Enabled bit: On Innovation: 7	Input Node: 2 Output Node: 5 Weight: 0.2 Enabled bit: On Innovation: 8	Input Node: 2 Output Node: 4 Weight: -0.3 Enabled bit: On Innovation: 9	Input Node: 3 Output Node: 6 Weight: 0.7 Enabled bit: On Innovation: 10	Input Node: 5 Output Node: 5 Weight: 0.5 Enabled bit: On Innovation: 11	Input Node: 5 Output Node: 4 Weight: 0.24 Enabled bit: Off Innovation: 12	Input Node: 5 Output Node: 6 Weight: 0.67 Enabled bit: On Innovation: 14	Input Node: 6 Output Node: 3 Weight: -0.23 Enabled bit: On Innovation: 16	Input Node: 6 Output Node: 4 Weight: -0.12 Enabled bit: On Innovation: 17

Network Encoding

Each link contains:

- values for its input node
- values for its output node
- the connection weight
- information about whether the link is enabled or disabled
- an innovation number



disabled

Link Genes

Input Node: 1 Output Node: 5 Weight: 0.6 Enabled bit: On Innovation: 7	Input Node: 2 Output Node: 5 Weight: 0.2 Enabled bit: On Innovation: 8	Input Node: 2 Output Node: 4 Weight: -0.3 Enabled bit: On Innovation: 9	Input Node: 3 Output Node: 6 Weight: 0.7 Enabled bit: On Innovation: 10	Input Node: 5 Output Node: 5 Weight: 0.5 Enabled bit: On Innovation: 11	Input Node: 5 Output Node: 4 Weight: 0.24 Enabled bit: **Off** Innovation: 12	Input Node: 5 Output Node: 6 Weight: 0.67 Enabled bit: On Innovation: 14	Input Node: 6 Output Node: 3 Weight: -0.23 Enabled bit: On Innovation: 16	Input Node: 6 Output Node: 4 Weight: -0.12 Enabled bit: On Innovation: 17

recursion

NEAT: Mutation

Mutation operations in NEAT can change:

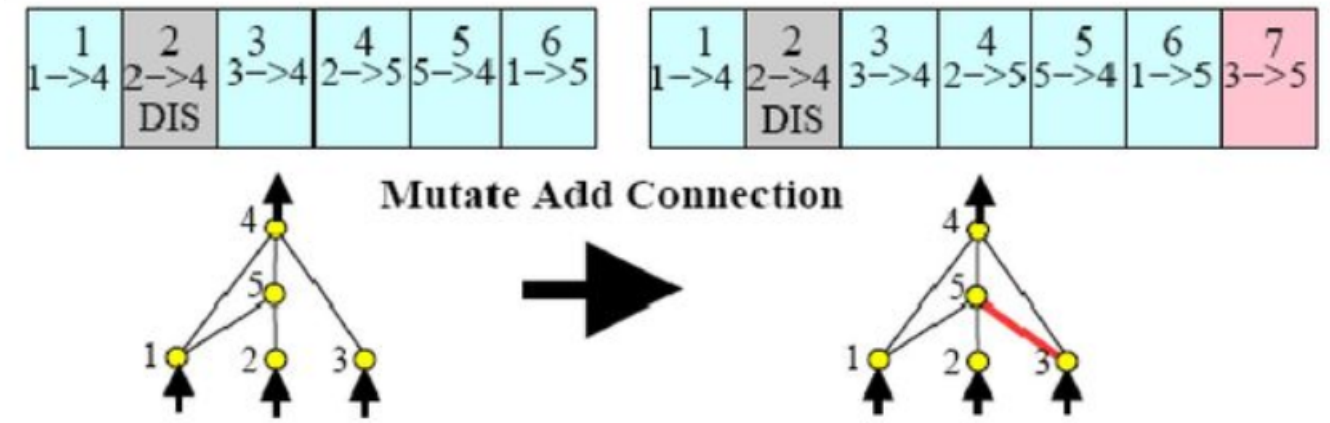
- Connection weights
- Node activation values
- Network topology

All these mutations occur randomly, constrained by a fixed probability which is defined for each individual simulation

- Weight and activation value mutations occur when a random node or link is chosen and its weight is perturbed by a constrained random value.
- NEAT mutation operations change network topology by adding **links** and **nodes**.

NEAT: Mutation

NEAT mutation operations change network topology by adding **links** and **nodes**.

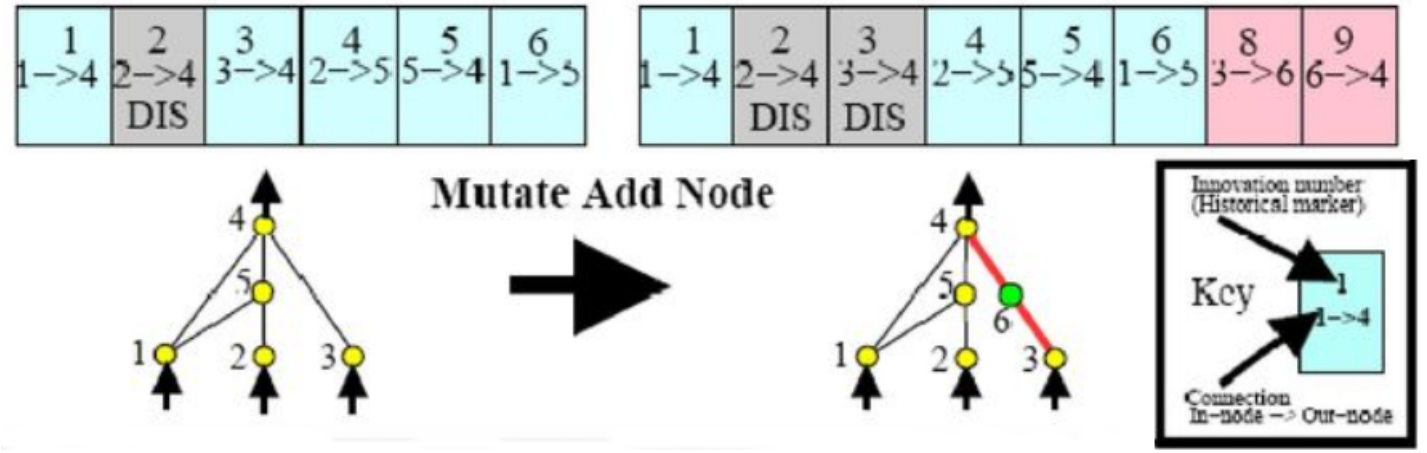


It randomly chooses two nodes and inserts a new link gene with an initial weight of **one**.

- If a link already existed but was **disabled**, it is re-enabled.
- If there is no link between them and an equivalent link has already been created in this population, this link is created with the **same innovation number** as the previously created link as it is not a newly emergent innovation.

NEAT: Mutation

NEAT mutation operations change network topology by adding **links** and **nodes**.

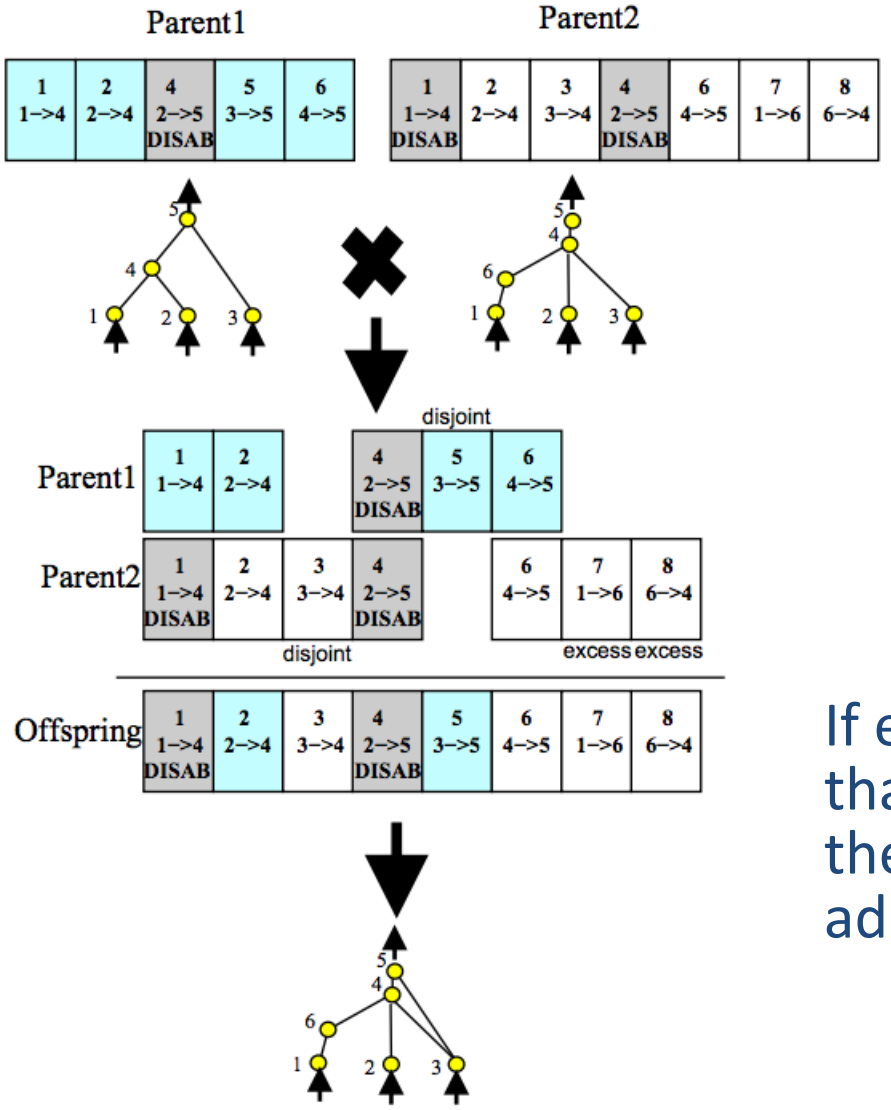


NEAT chooses and **disables** an existing link and inserts a node with a random activation value, as well as two link genes to connect the node to the now-disabled link's previous input and output nodes.

NEAT transfers the weight from the disabled link to the new link **connected to the old output** neuron. The weight of the link gene inserted **between the new neuron and the old input** node is set to one so as not to disturb any learning that has already occurred in this connection.

NEAT: Crossover

During a crossover operation, NEAT can quickly determine how to line up the two parents' genes

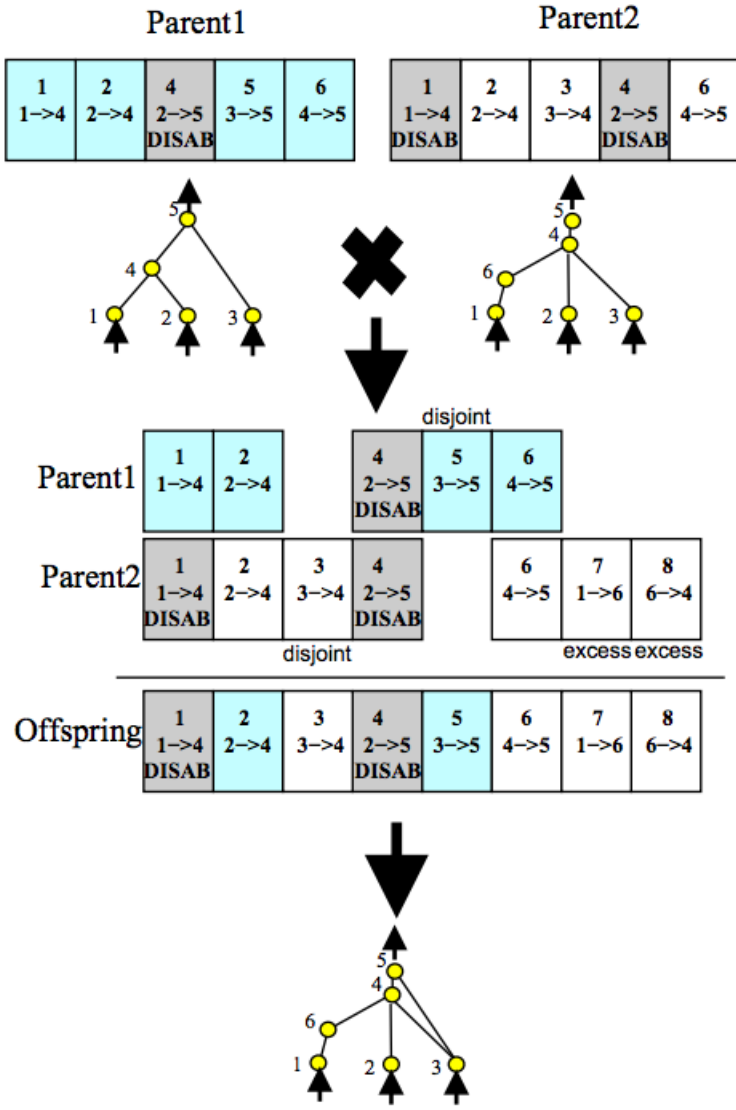


Any genes that do not share innovation numbers with genes in the other parent's genome are referred to as **disjoint** and are added to the child during crossover.

If either parent has genes that are newer than any of the genes in the other parent, they are considered **excess** and are also added to the child during crossover.

NEAT: Crossover

During a crossover operation, NEAT can quickly determine how to line up the two parents' genes

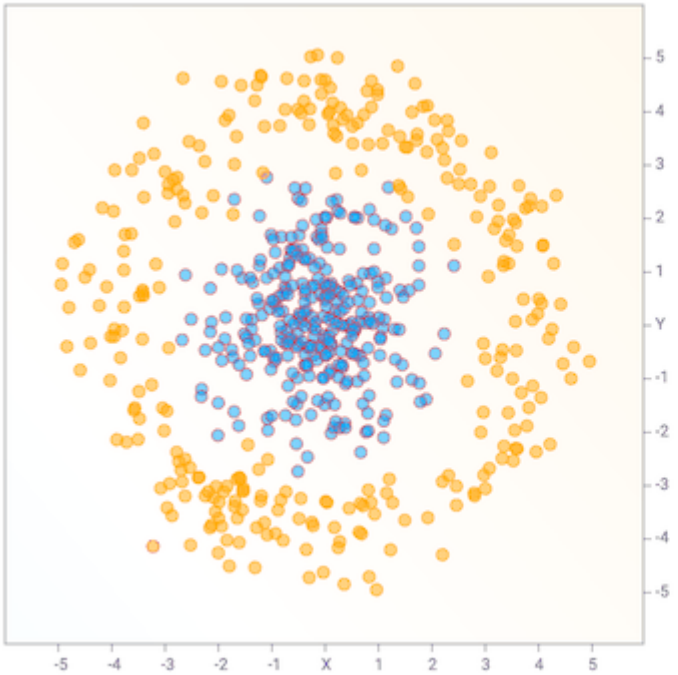


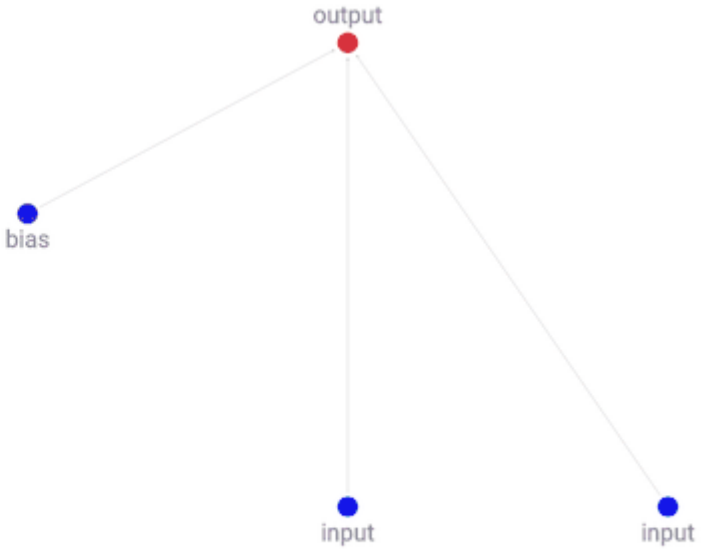
All genes that are **shared** by both parents are inherited by the child from the parent with the **highest fitness**.

A gene that is **disabled** in one parent but enabled the other has a chance of being re-enabled in the offspring.

NEAT: Example and Code

Neural Network Evolution Playground with Backprop NEAT





Code: https://github.com/hardmaru/backprop-neat-js

Demo: http://otoro.net/ml/neat-playground/

There are many **extensions** proposed for NEAT: Real time NEAT (rtNEAT), Hypercube based NEAT (HyperNEAT), cgNEAT (Content-Generating NEAT), Evolutionary Acquisition of Neural Technologies (EANT)

Neuroevolution for Deep Neural Networks

NEAT method is first extended to evolving network topology and hyperparameters of deep neural networks in **DeepNEAT**

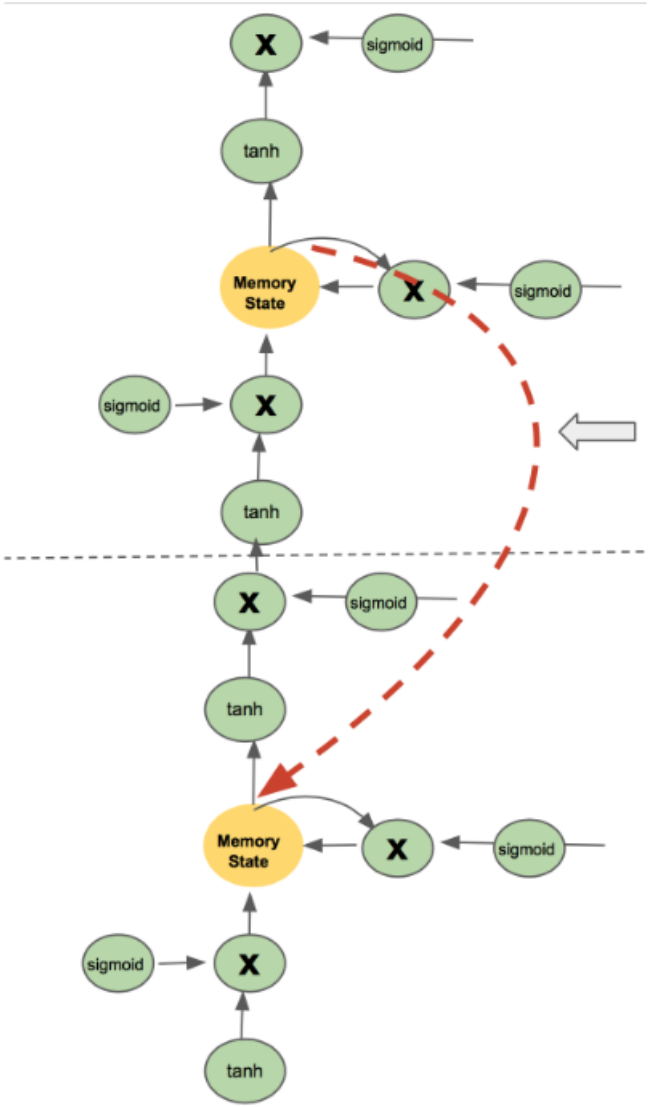
"Evolving Deep Neural Networks" by Risto Miikkulainen

DeepNEAT differs from NEAT in that each node in the chromosome no longer represents a neuron, but a **layer** in a DNN.

The gene also contains a table of hyperparameters that may be mutated during the evolution context.

Neuroevolution for Deep Neural Networks

A variant of DeepNEAT, called Coevolution DeepNEAT (**CoDeepNEAT**)



It separately evolves two subpopulations:

- network structure
- composing modules structure

Authors used it for CNN and LSTM networks.

"Evolving Deep Neural Networks" by Risto Miikkulainen

Neuroevolution for Deep Neural Networks

Multi-node Evolutionary Neural Networks for the Deep Learning (**MENNDL**) framework is proposed to address the optimization of CNN hyperparameters using GAs

The hyperparameters that were optimized are the **filter size** and the **number of filters** for each convolutional layer, which was restricted to three layers. The resulting network is fully-trained and no further post-processing is required.

"Optimizing deep learning hyper-parameters through an evolutionary algorithm" by Young, S. Rose

Reduction of **training time** of Deep Neural Networks (DNNs) over the regular approach by evolving optimised DNNs instead of adopting heuristic random initial architecture using MNIST dataset.

"Evolving deep neural networks: A new prospect" by SS Tirumala

Used CMA-ES to optimise the hyperparameters of existing DNNs

Iloshchilov, Hutter. 2016 CMA-ES for Hyperparameter Optimization of Deep Neural Networks

More material about Neuroevolution

Incremental methods:

Wang, J., Wang, H., Chen, Y., Liu, C.: A constructive algorithm for unsupervised learning with incremental neural network.

The University of Texas: Neural Network Research Group

http://nn.cs.utexas.edu/

Article: Welcoming the Era of Deep Neuroevolution (UBER Engineering)

https://eng.uber.com/deep-neuroevolution/

Example: NeuroPong - Learning to play pong using Neuroevolution

https://www.youtube.com/watch?v=GVDwmp9RZo4