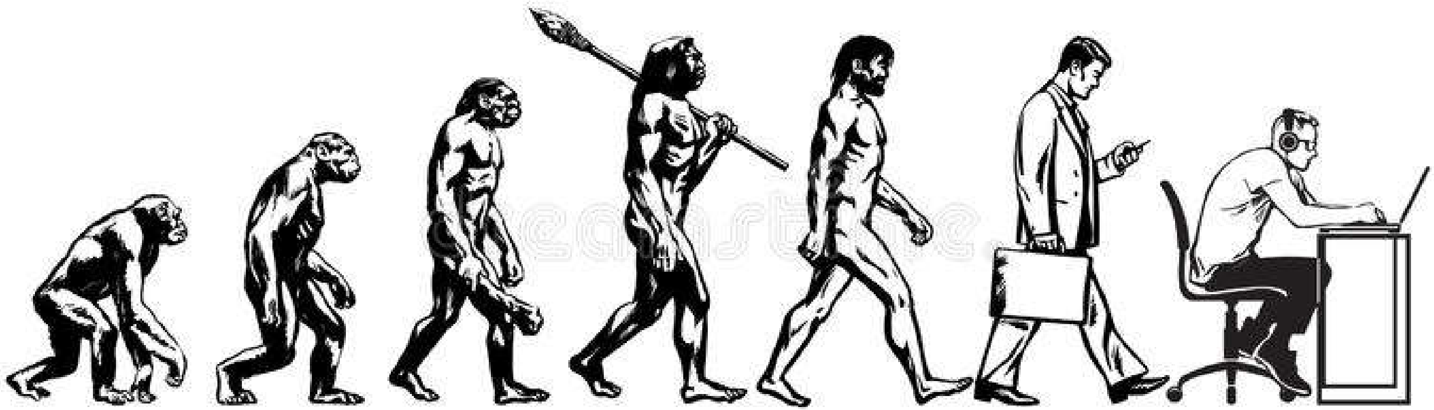
UPPSALA UNIVERSITET

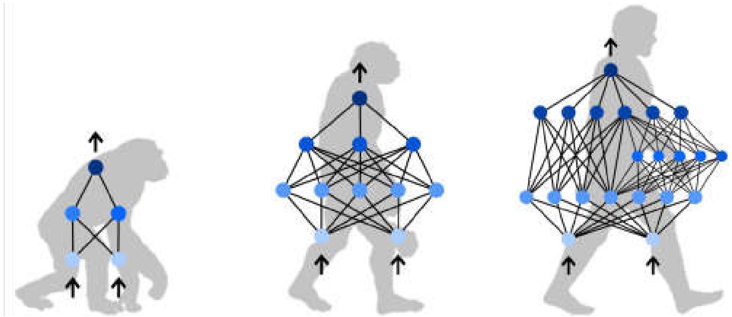
# Natural Computation Methods in Machine Learning (NCML)

## Lecture 12: Evolutionary Computing 2



# Neuroevolution

- Genetic algorithms can be used to evolve, rather than train, artificial neural networks
  - This is one of the most common applications of GAs
- Not only to find weight values. We can also evolve
  - Structure
  - Size
  - Activation functions
  - Other meta-parameters



- Current popular method: NEAT
  - NeuroEvolution of Augmented Topologies
  - Many cool application examples on YouTube

UPPSALA UNIVERSITET

# Neuroevolution

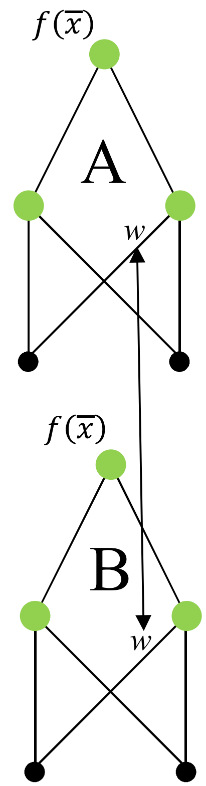
- Could combine the two forms of learning (evolutionary and neural)
  - Evolve networks, but also train the individuals using conventional neural network algorithms
    - From the GAs perspective, the ANN training is then part of the evaluation of individuals
    - From the ANNs perspective, the GA provides initial network configurations (instead of just randomizing weights)
  - ≈ Lamarckian evolution: The old idea (before Darwin), that learned experience can be inherited No-longer-to-be-ridiculed
- EC does not have to follow natural laws!
  - If you want to have more than two parents, or Lamarckian evolution, by all means, go for it
    - But maybe first consider why nature did <u>not</u> choose this solution

or discover that it actually did

# Crossover for training ANNs

is very difficult to define in a non-destructive way

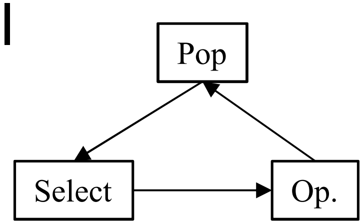
- Two networks that solve the same problem approximately equally well, have probably found very different solutions
  - Their weights are likely to be very different
  - Their 'knowledge' is distributed over all of them
- We can think of crossover as swapping some of the weights between network A to network B
  - Do they still have the same meaning in the other network? (No, not likely)
- The naïve approach with one- or two point crossover is not likely to work (any better than random search)
- There are, of course, many suggested solutions
  - Sorting the hidden nodes actually helps a fair bit
  - Study NEAT for a better (but more complex) solution



# EC issues

Some of which are still open to debate

- Generational or steady state
  - Different time scales
  - The previous lecture presented the generational view
    - Each lap in the loop replaces the whole population
  - Steady state is more careful
    - Each lap in the loop is to add/modify one or two individuals




- Population size, constant or variable?
  - If variable, how?
- Introns ('junk DNA'), useful or not?
- The importance of mutation
- Is crossover really constructive or just some kind of mutation macro?

# EC issues

## Some of which are still open to debate

- Crossover moves parts of genotypes
  - Finding a good encoding for this is difficult
  - Is a substring moved from A to B still meaningful in B?
    - The encoding and the crossover operator should always be designed together, and with a strong focus on this question
- Evaluation (the fitness function) should be simple and efficient
  - This is where EC spends most computation time
  - Good argument against the Lamarckian idea above
  - Is it always necessary to evaluate the whole population every generation?

# Genetic Programming

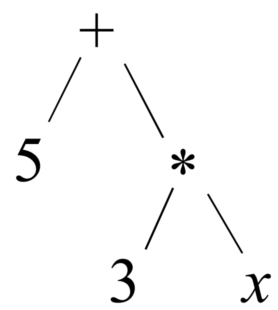
- In GP, genotypes are expressions in a programming language
- Semi-automatic programming
- The most common approach: Operate on representations of parse-trees (Koza, 1990)
- Example: The expression 5 + 3x

*As a Lisp expression*

(+ 5 (* 3 *x*))

*Why Lisp?*

*As a parse tree*



# Crossover in GP

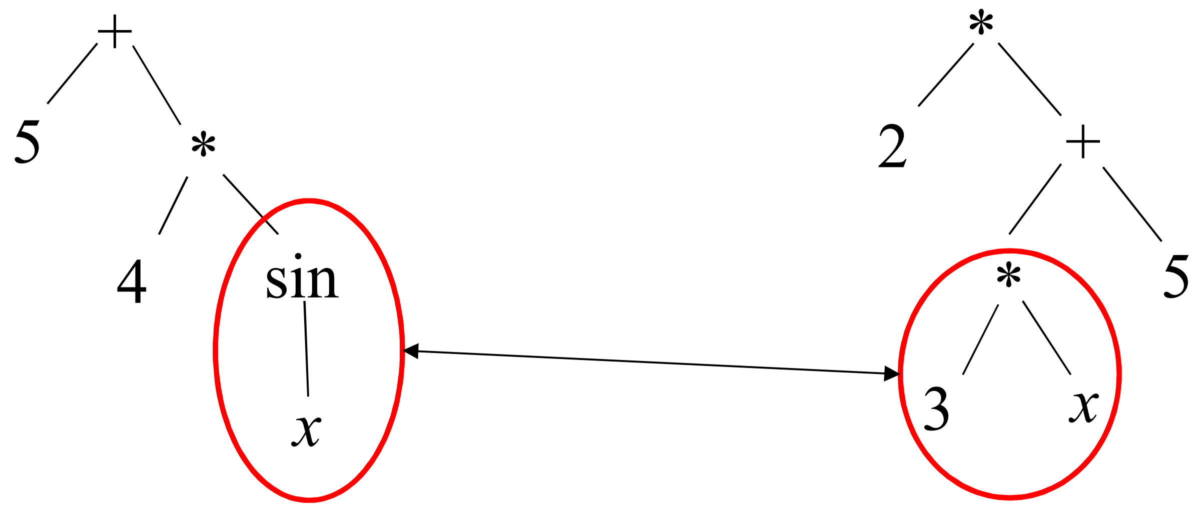
Swap randomly selected sub-trees

Parent A: 5 + 4*sin(*x*)

In Lisp: (+ 5 (* 4 (sin *x*)))

Parent B: 2*(3*x*+5)

In Lisp: (* 2 (+ (* 3 *x*) 5))




Offspring: 5 + 4 * 3 * *x*

2*(sin(*x*) + 5)

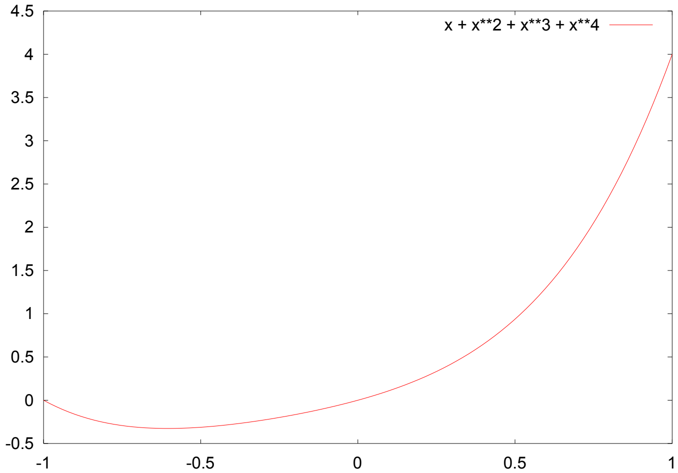
# Mutation in GP
Huge number of variants

- Terminal node mutation
  - Replace a terminal with another terminal
- Swap mutation
  - Swap the arguments of a function
- Grow mutation
  - Replace a node by a new random subtree
- Trunc or Cut mutation
  - Replace a non-terminal node with a terminal
- Gaussian mutation
  - Jog a constant
- ...

# Function approximation in GP
A very simplified example

- Task: Given a training set, discover the function

$$f(x) = x + x^2 + x^3 + x^4, \text{ for } x \in [-1,1]$$

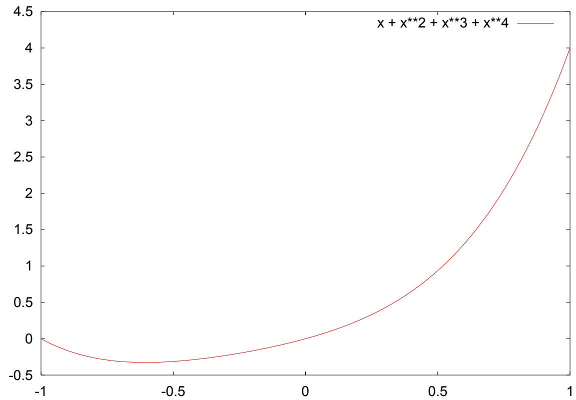



- A neural network would do a *numerical* approximation
- GP is a *combinatorial* method – it should be able to find the *exact* function
  - if given the building blocks required to express that function

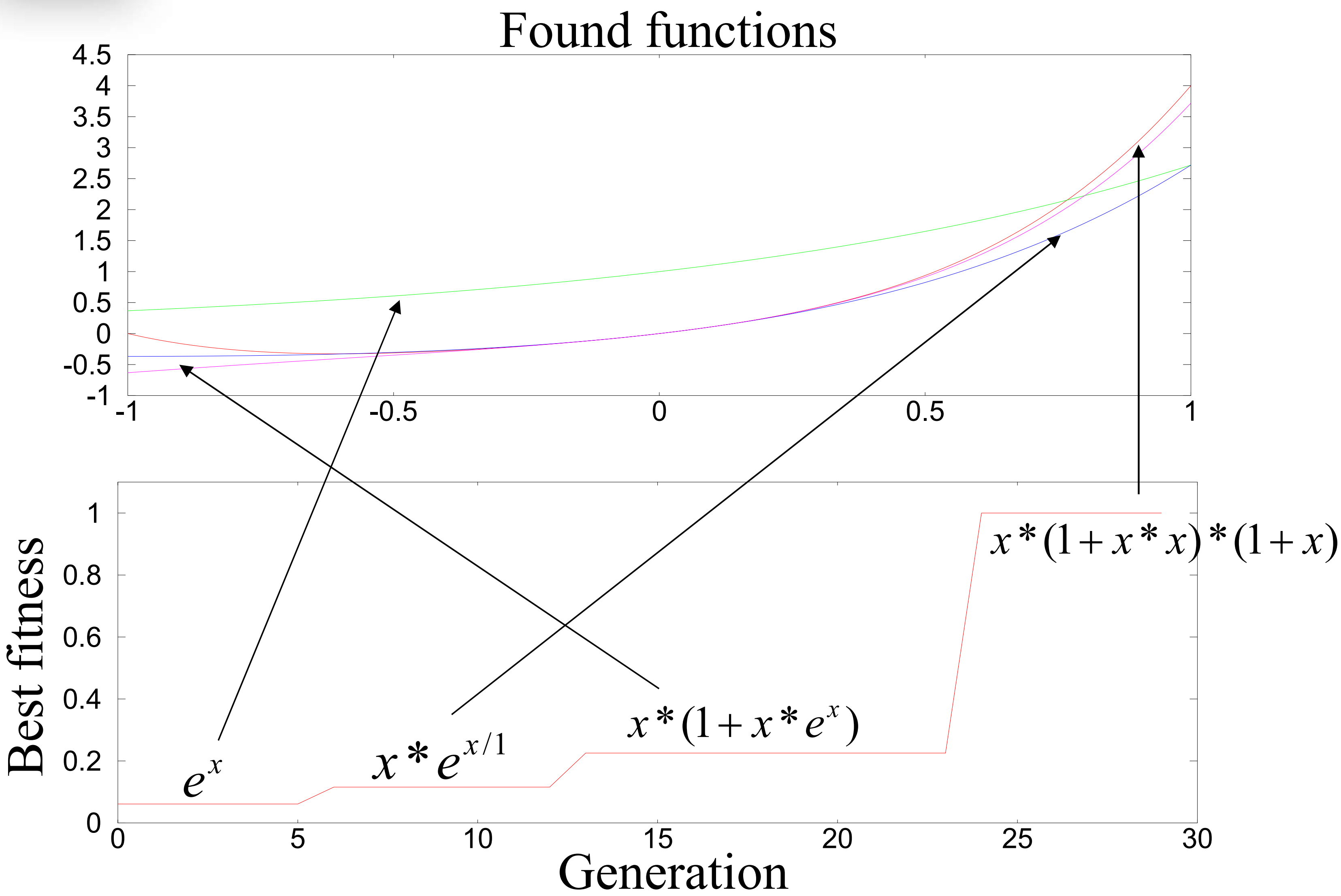
# Function approximation in GP

A very simplified example

- Create a population of random expressions, $y(x)$, using the functions $+$, $-$, $*$, $/$, $\sin$, $\cos$, $\exp$ and $\log$, and the terminals $1$ and $x$
- Many lures (*exp* in particular)
- Fitness: 0 if illegal expression, else $1/(d+1)$, where $d = |f(x) - y(x)|$
- This way, fitness will stay in $[0,1]$)



# Test run (100 individuals)



# Target Languages

- Any language can be used
  - but Lisp is particularly suitable (functional, syntax, untyped)
- Assembler/machine code
  - e.g. AIM-GP (Nordin et al, 1997)
  - Assembler instructions = integers (so GA can be used)
  - Very efficient evaluation
    1. Store the expression in an unsigned integer array
    2. Cast it to a function pointer, and just call it
  - RedCode (Core War)
- Most methods are specialized for their language
- Decision trees! (common application)
- Computer network protocols
- See [geneticprogramming.com](geneticprogramming.com) for an overview

# Grammatical Evolution

Ryan & O'Neill, 1998

- GP methods often specific to their target language
- Grammatical Evolution is language independent
  - It takes a BNF grammar as input,
  - and can therefore generate programs in any language
- The genotype is just a sequence of numbers
  - For example, 8-bit integers: 25, 11, 4, 16, 13, ...
  - So, it's actually GA, not GP (by Koza's definition)
    
    + two extra, GE-specific operators
  - How to map a sequence of integers to an expression, given a grammar?

UPPSALA UNIVERSITET

# Grammatical Evolution (example)

BNF grammar

<expr> ::= <num> | <num> <op> <num> | <preop>(<num>)

<num> ::= 7 | 17 | 42

<op> ::= + | -

<preop> ::= sin | cos | exp

Genotype: A sequence of integers

ex) 25, 11, 4, 16, 13, ...

# Grammatical Evolution (example)

BNF grammar

<expr> ::= <num> | <num> <op> <num> | <preop>(<num>)

<num> ::= 7 | 17 | 42

<op> ::= + | -

<preop> ::= sin | cos | exp

Genotype: A sequence of integers

ex) **25**, 11, 4, 16, 13, ...

25 mod 3 = 1 → "<num><op><num>"

UPPSALA UNIVERSITET

# Grammatical Evolution (example)

BNF grammar

<expr> ::= <num> | <num> <op> <num> | <preop>(<num>)

<num> ::= 7 | 17 | 42

<op> ::= + | -

<preop> ::= sin | cos | exp

Genotype: A sequence of integers

ex) 25, **11**, 4, 16, 13, ...

11 mod 3 = 2 → "42 <op><num>"

# Grammatical Evolution (example)

BNF grammar

<expr> ::= <num> | <num> <op> <num> | <preop>(<num>)

<num> ::= 7 | 17 | 42

<op> ::= + | -

<preop> ::= sin | cos | exp

Genotype: A sequence of integers

ex) 25, 11, **4**, 16, 13, ...

4 mod 2 = 0 → "42 + <num>"

UPPSALA UNIVERSITET

# Grammatical Evolution (example)

BNF grammar

<expr> ::= <num> | <num> <op> <num> | <preop>(<num>)

<num> ::= 7 | 17 | 42

<op> ::= + | -

<preop> ::= sin | cos | exp

Genotype: A sequence of integers

ex) 25, 11, 4, **16**, 13, ...

16 mod 3 = 1 → "42 + 17"

# Grammatical Evolution (example)

[25, 11, 4, 16, 13, ...] → "42+17"

- The rest of the genotype (13, ...) is not used
  - Introns ...
- Wrap around if the sequence is too short
  - The first integers (now reused) are likely to produce a different expression this time, since we are now in a different point in the grammar
    - an advantage in this case
    - but also the source of the big problem with GE
- Evolve the string using Genetic Algorithms
  - Two additional GE specific operators – Prune and Duplicate

# The problem
## with Grammatical Evolution

- We usually strive for 'locality' in EC
  - Similar genotypes should produce similar phenotypes
- This is not the case in GE
  - The meaning of codons (the integers) is extremely context dependent
  - This makes conventional crossover very destructive
    - A subsequence moved from A to B will almost certainly <u>not</u> generate the same expression in B
  - There are suggested solutions to this, of course
- More recent variant - Grammatical Swarm
  - Same basic idea, but trained by Particle Swarm Optimization (Lecture 14) instead of GA
  - There is no crossover in PSO ...

# Challenges in GP

- How and when to define new functions
  - Encapsulation of knowledge, to prevent its destruction
  - For example ADF (Automatically Defined Functions)
- How to deal with types, if-statements, etc.
- GP is a Software Engineering nightmare!
  - Would you want to review the code produced?
- Does it scale?
  - the search space is enormous!
  - a valid concern also for EC in general
- Personal thought (feel free to disagree):
  - Scalability requires constructive crossover!
  - Without it, EC is close to random search
    - and arguably not 'evolutionary' at all, but that's just a name
  - Crossover is indeed difficult, so it's tempting to move away from it, as some branches of EC has, but I think that's a dead end

(a local optimum in the search for scalable methods)