NEURO-EVOLUTION

Automating operational decisions in management by applying Neuro-evolution algorithms

Dimitar Lyubchev

TODAY'S GAME PLAN

01
Operational Decisions
Definition; Examples

02
Evolutionary Algorithms
What are they? Types; How they work

03
NEAT - Theory
Definition; Elements

04
NEAT - Sokoban
Sokoban Problem; Setup; Possible solutions (RL vs NEAT)

05
Analyses, Issues, Lessons
Crossover; Mutation; Speciation; Computation

06
Future Implementations
Phase 1 – "Incremental improvements"
Phase 2 – "Evolution"
Phase 3 – "Revolution"

01

Operational Decisions in management

01

Operational Decisions in management

Definition

Short-term decisions – typically made on a weekly, daily or hourly basis.

Concerned with operational details:

Examples

- Daily/weekly work allocation - e.g. Sprint planning
- Process design – creation of products
- Hiring – screening of CV's,
- Facility – Seating plan
- IT Support – Scheduling of system updates

02

Evolutionary algorithms (EA)

02

Definition

An evolutionary algorithm is a type of **optimization algorithm that is inspired by the process of natural evolution**.

Providing approximate solutions for optimization problems

02

Evolutionary algorithms

Types

Genetic Algorithm (GA)

- Uses operators like **selection**, **crossover**, and **mutation**.
- Great for combinatorial and optimization problems.

Genetic Programming

- **Evolves actual programs or expressions** instead of parameter sets.
- Solutions are often represented as trees.

Evolution Strategies

- Focuses more on **mutation** and **selection**; less on crossover.
- Works well for continuous parameter optimization.

02

Evolutionary algorithms

Key elements

- Contain populati… genomes/chromo… …nomes (parents) to reproduce
- Genomes are eva… (generations) – fi… …produce offsprings (children)
- Usually the fittes… …changing a trait of …andom occurrence)

Evolutionary algorithms
Key elements

Genomes are evaluated during stages (generations) – fitness score

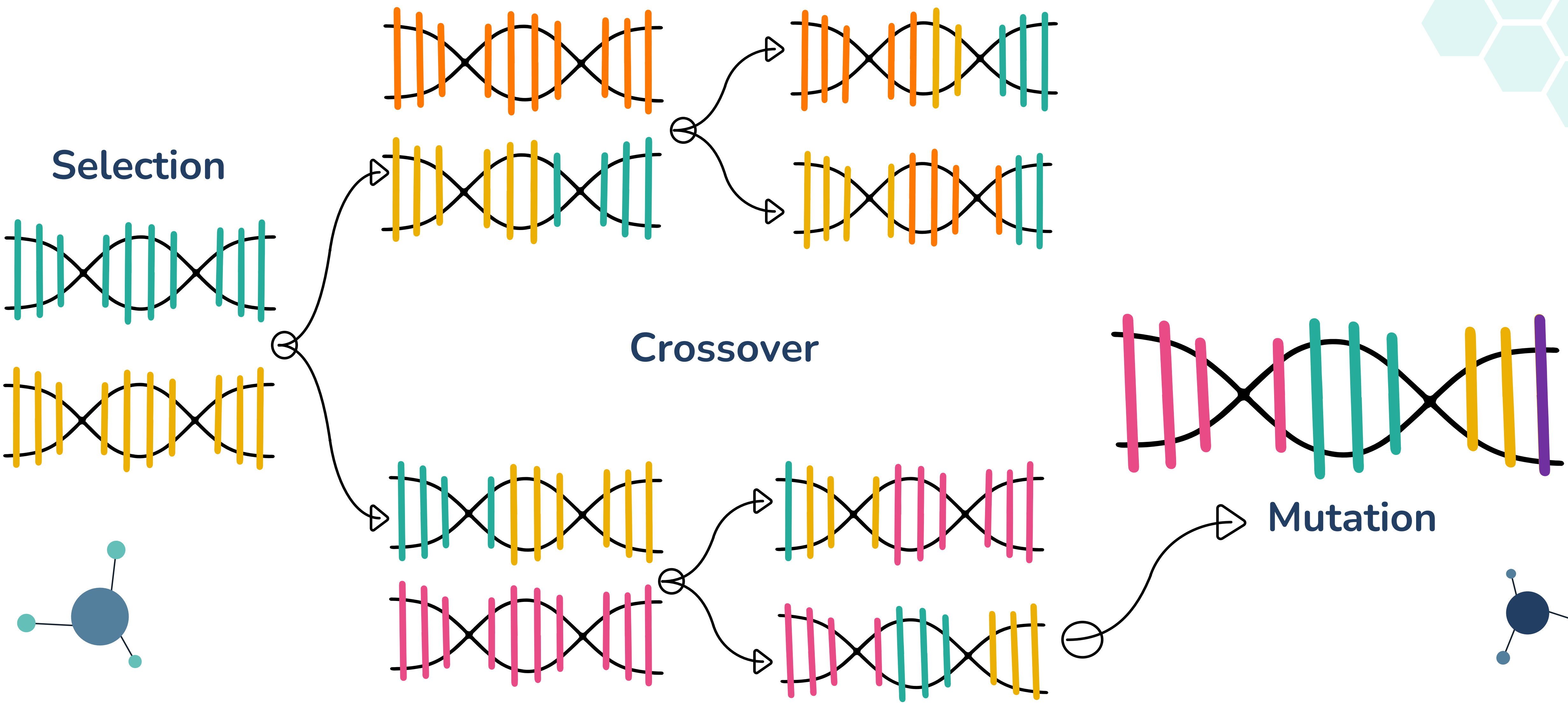
Genome
(solution)

Genome
Evaluation

02

Evolutionary algorithms

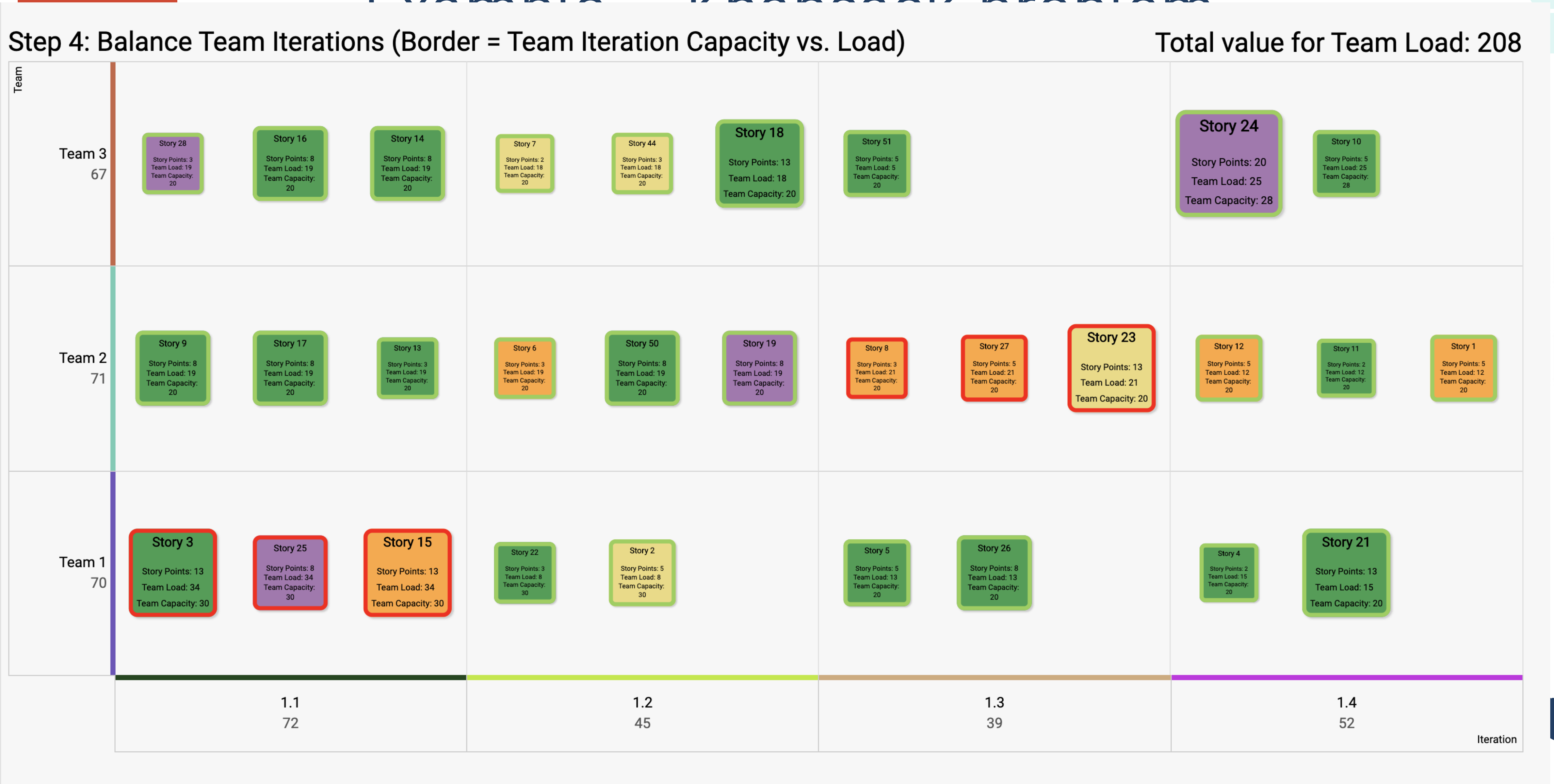
Key elements



02

Evolutionary algorithms

Example – Knapsack problem



02

Evolutionary algorithms
Key advantages

Global search capacity
- Good at exploring entire solution space
- Less likely to get stuck in local optima

No need for Gradient info
- EA don’t require derivative calculations
- Useful for non-differentiable, discontinuous, or “black-box” functions.

Parallelism
- Naturally suited for parallel processing
- Multiple candidate solutions are evaluated simultaneously

Flexibility
- Can handle multi-objective, constrained, and combinatorial problem (e.g. car design)

Robustness
- Performs well even with **noisy, dynamic**, or **complex environments** (e.g. adapting stock trading strategies).

02

Evolutionary algorithms
Key disadvantages

❌ Computational cost
- Can be slow due to large population and many generations.
- High number of function evaluations needed.

❌ Parameter Sensitivity
- Requires careful tuning of parameters (e.g., mutation rate, crossover rate).
- Performance highly depends on these settings.

❌ No Guarantee of Optimality
- Stochastic in nature, so results may vary.
- No formal guarantee of finding the global optimum.

❌ Premature Convergence
- Risk of converging to suboptimal solutions if diversity is not maintained.

03

NEAT Theory

03

Definition

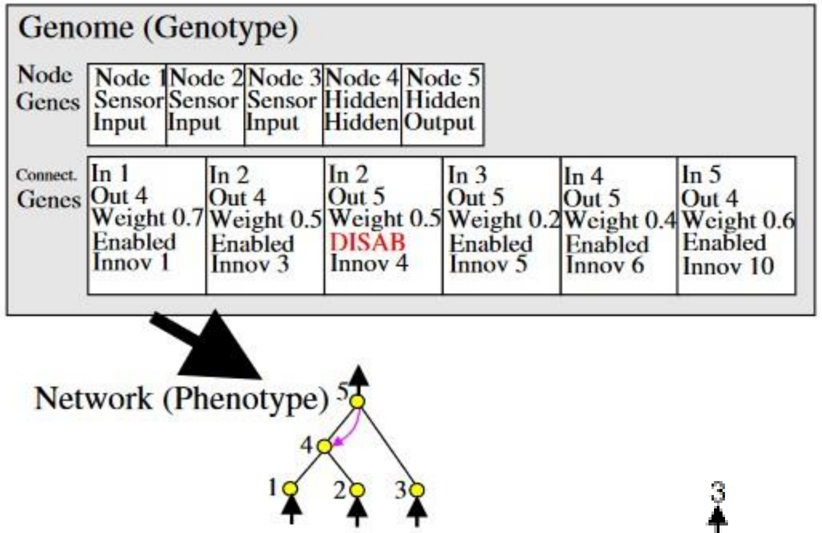
NEAT - Neuro-Evolution of Augmenting Topologies

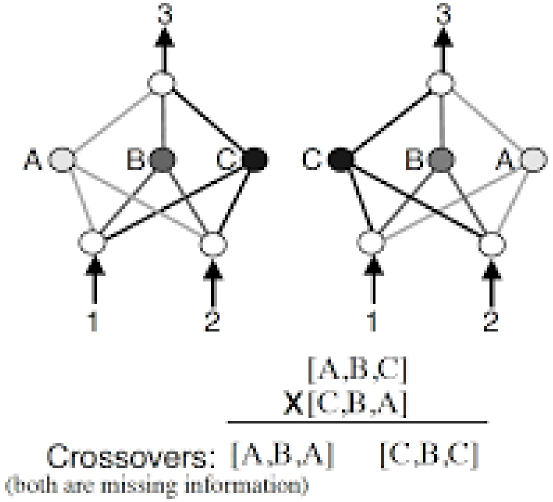
A genetic algorithm (GA) for the generation of evolving artificial neural networks.

Evolving Neural Networks through Augmenting Topologies
Kenneth O. Stanley, Risto Miikkulainen

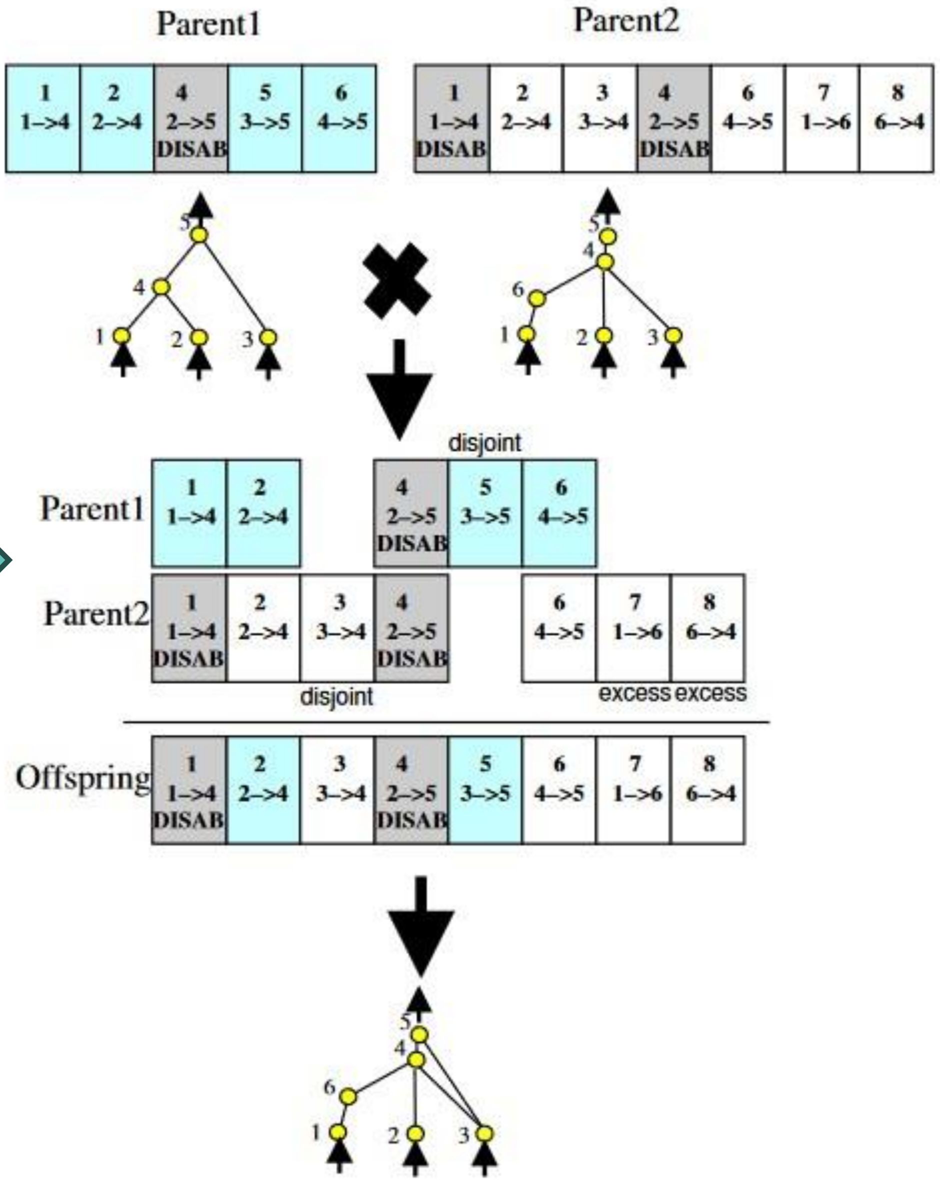
03

NEAT
Theory





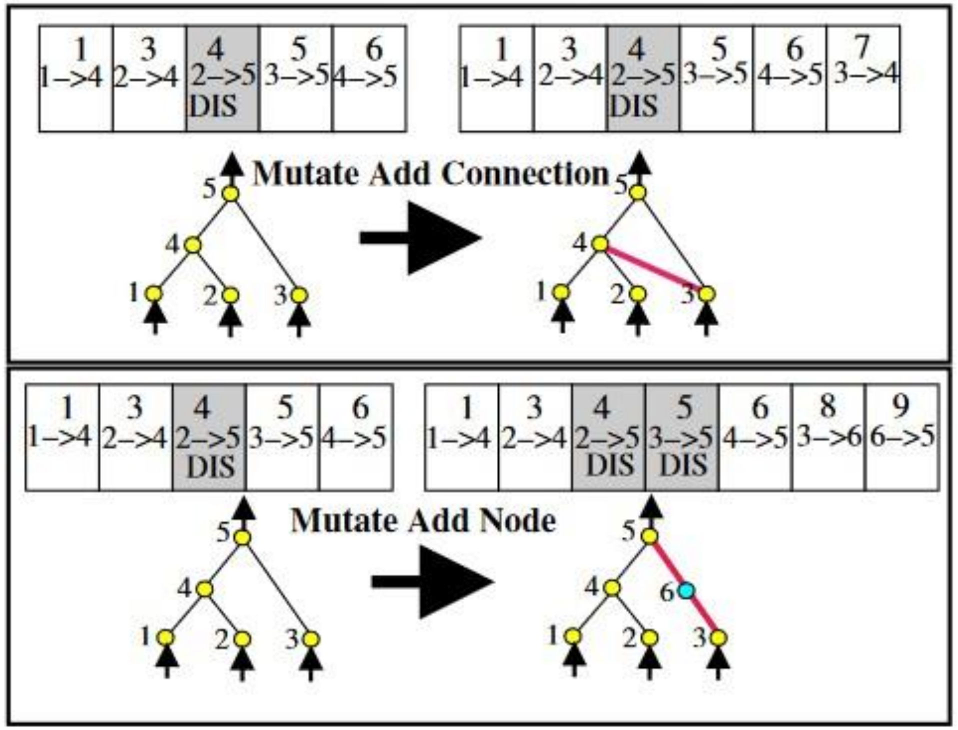
Equal fitness assumed



03

NEAT
Theory

Fig. 1. **A genotype to phenotype mapping example.** The third gene is disabled, so the connection that it specifies (between nodes 2 and 5) is not expressed in the phenotype.



03

NEAT
Python

```
Genome ID: 1

Nodes:
  Node 0 - Type: INPUT, Bias: 0.0
  Node 1 - Type: INPUT, Bias: 0.0
  Node 2 - Type: OUTPUT, Bias: 0.5

Connections:
  (0, 2) -> Weight: -0.72, Enabled: True
  (1, 2) -> Weight: 1.34, Enabled: False

Fitness not yet assigned.
```

03

NEAT
Python

Config file..

```
[NEAT]
fitness_criterion     = max
fitness_threshold     = 10
pop_size              = 1000
reset_on_extinction   = False

[DefaultGenome]
# node activation options
activation_default      = sigmoid
activation_mutate_rate  = 0.05
activation_options      = sigmoid

# node aggregation options
aggregation_default     = sum
aggregation_mutate_rate = 0.0
aggregation_options     = sum
# node bias options
bias_init_mean          = 0.0
bias_init_stdev         = 1.0
bias_max_value          = 30.0
bias_min_value          = -30.0
bias_mutate_power       = 0.5
bias_mutate_rate        = 0.7
bias_replace_rate       = 0.1

# genome compatibility options
compatibility_disjoint_coefficient = 1.0
compatibility_weight_coefficient   = 0.5

# connection add/remove rates
conn_add_prob           = 0.5
conn_delete_prob        = 0.5

# connection enable options
enabled_default         = True
enabled_mutate_rate     = 0.01

feed_forward            = True
initial_connection      = full

# node add/remove rates
node_add_prob           = 0.3
node_delete_prob        = 0.15

# network parameters
num_hidden              = 1
num_inputs              = 49
num_outputs             = 9

# node response options
response_init_mean      = 1.0
response_init_stdev     = 0.0
response_max_value      = 30.0
response_min_value      = -30.0
response_mutate_power   = 0.0
response_mutate_rate    = 0.0
response_replace_rate   = 0.0

# connection weight options
weight_init_mean        = 0.0
weight_init_stdev       = 1.0
weight_max_value        = 30
weight_min_value        = -30
weight_mutate_power     = 0.5
weight_mutate_rate      = 0.8
weight_replace_rate     = 0.1

[DefaultSpeciesSet]
compatibility_threshold = 3.0

[DefaultStagnation]
species_fitness_func = max
max_stagnation       = 20
species_elitism      = 2

[DefaultReproduction]
elitism            = 2
survival_threshold = 0.2
```

04

NEAT
Sokoban problem

04

Definition

Sokoban

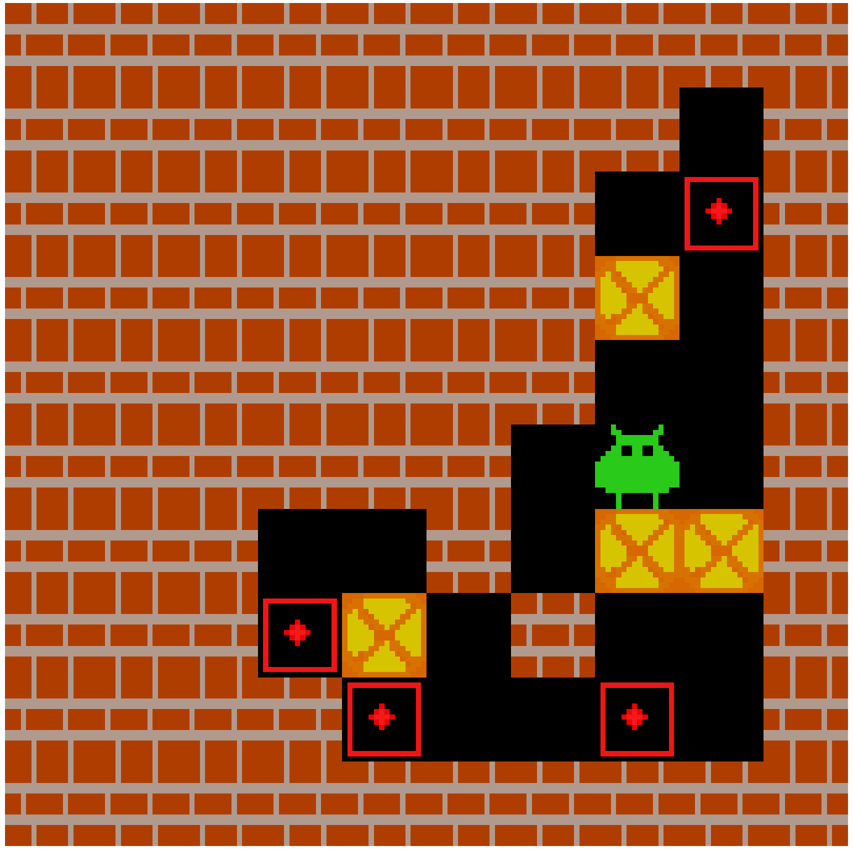
Sokoban is Japanese for 'warehouse keeper'.
This puzzle game was originally invented in
Japan in the early 80's.

You have to push crates to their proper locations
with a minimum number of moves.

04

NEAT

Sokoban problem

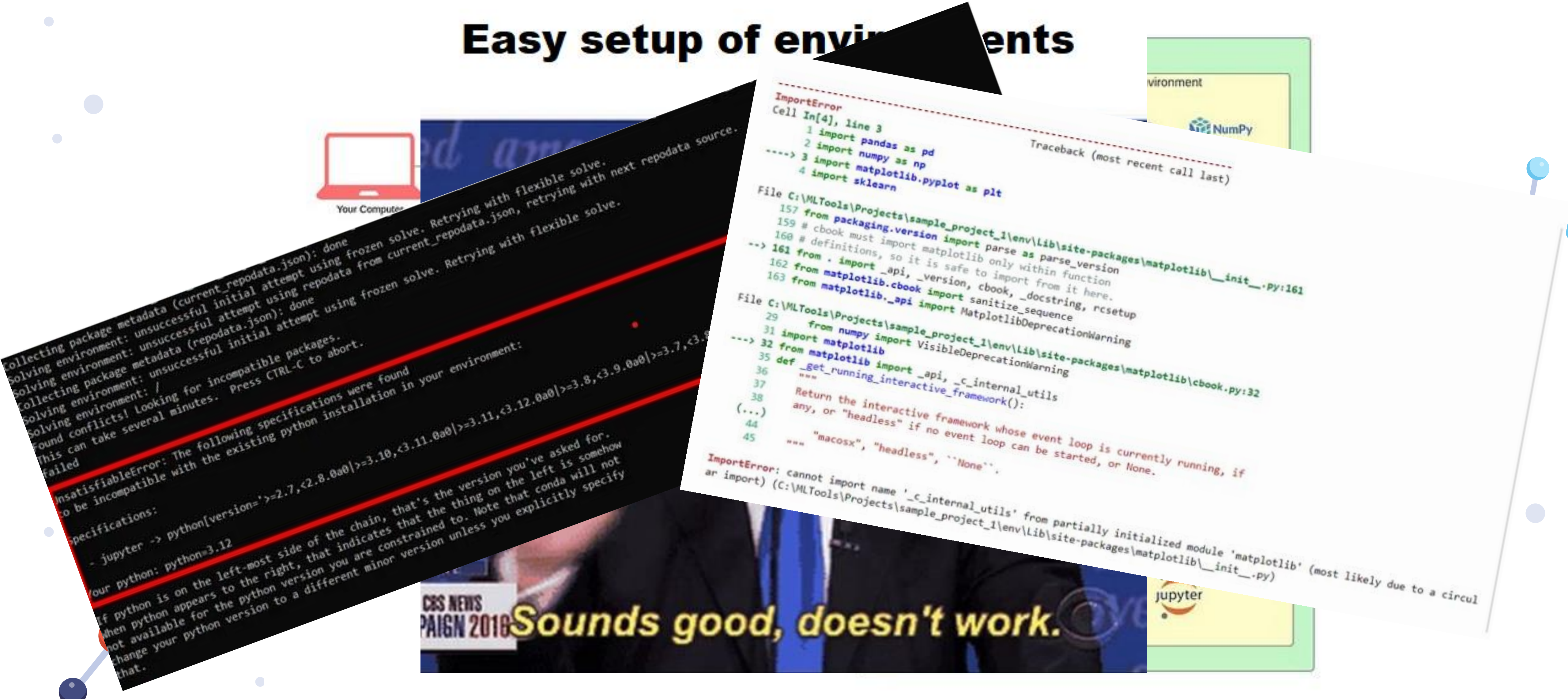


https://github.com/mpSchrader/gym-sokoban

04

NEAT

Sokoban problem–Environment setup



04

NEAT

Sokoban problem–Environment setup

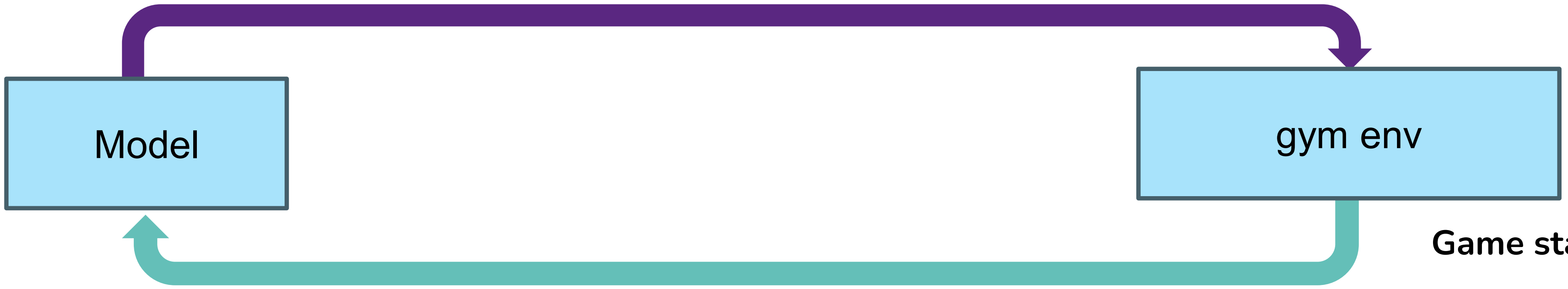
1 How to create the environment (replace neat_test4 with your env name):

- conda create -n neat_test4 python=3.10 gym ipykernel pyglet
- conda activate neat_test4
- pip install neat-python
- pip install -e "\gym-sokoban"
- python -m ipykernel install --user --name neat_test4 --display-name "Python (neat_test4)"
- pip install graphviz

04

NEAT
Sokoban problem – API setup

Input
Decision (move)



Game state

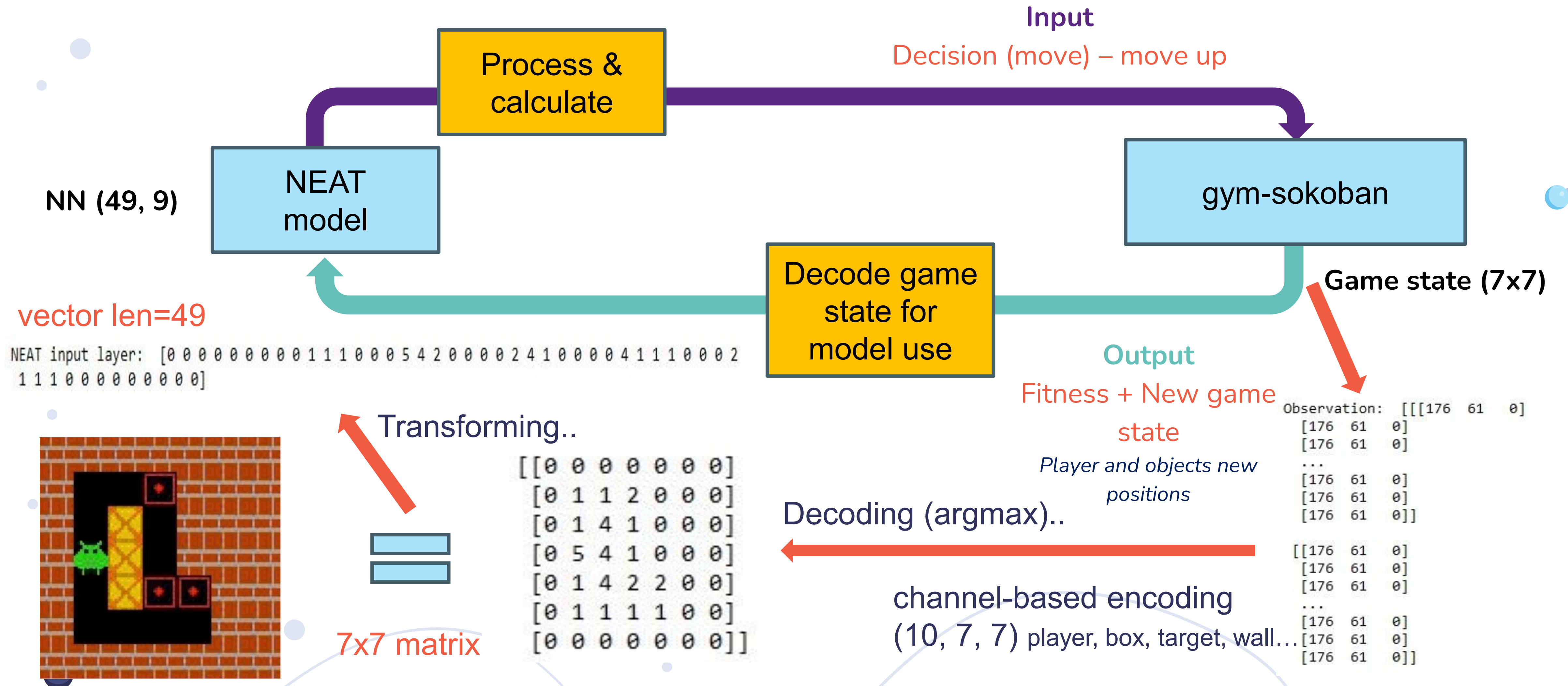
Output
Fitness + New game state
Player and objects new positions

Fitness in gym-sokoban:
- 0.1 for each move
+ 1 push box on target
- 1 push box off target
+ 10 for solving the task

04

NEAT

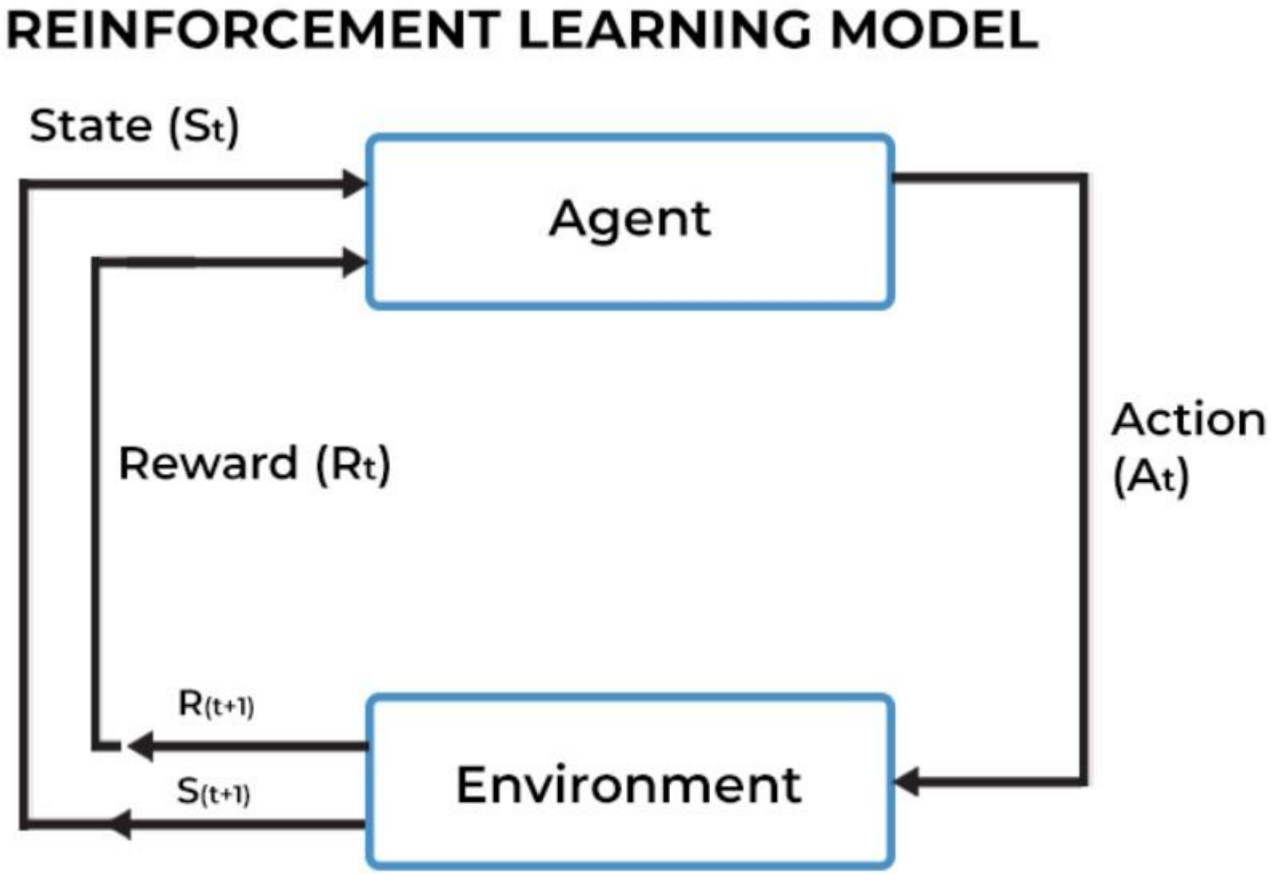
Sokoban problem – API setup



04

NEAT

Experiment with RL



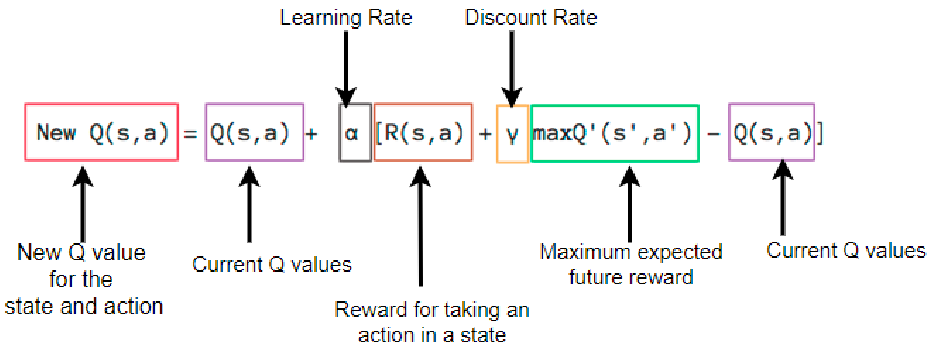
04

NEAT
Experiment with RL

Key concepts of Q-learning, DQN, PPO

Q learning

- A basic learning method that **uses a big table to remember the best moves after exploring the environment**
- Can't handle more complex situations or levels — **the table gets way too big.**



DQN (Deep Q-Network)

- A smarter version of Q-Learning that **uses a neural network** instead of a table to predict the Q-values

PPO (Proximal Policy Optimization)

- A policy gradient method: instead of using a value function like QL and DQN ((how good a state or an action is based on expected reward), **directly learns a policy** by mapping states to actions (probability of taking an action a in state s.)

NEAT

Experiment with RL

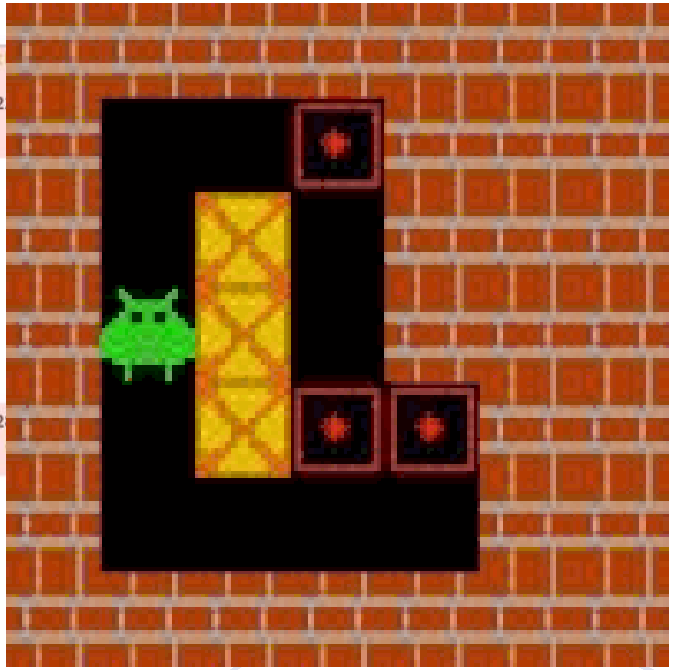
Test results - DQN, PPO, Q-learning

- PPO (Proximal Policy Optimization) – FAILED - struggles in sparse reward settings

DQN (Deep Q-Network) – FAILED - suffer from unstable training in environments with many dead ends

Q learning – SUCCEEDED! - relatively small search space, enough time to explore future moves

```
in old step API which returns one bool instead of two. It is recommended to rewrite t
  logger.deprecation(
/home/npenchev/.local/lib/python3.12/site-packages/gym/utils/passive_env_checker.py:2
for `np.bool_`.  (Deprecated NumPy 1.24)
  if not isinstance(done, (bool, np.bool8)):
Episode 100/1000 completed.
Episode 200/1000 completed.
Episode 300/1000 completed.
Episode 400/1000 completed.
Episode 500/1000 completed.
Episode 600/1000 completed.
Episode 700/1000 completed.
Episode 800/1000 completed.
Episode 900/1000 completed.
Episode 1000/1000 completed.
Training completed.
/home/npenchev/.local/lib/python3.12/site-packages/gym/utils/passive_env_checker.py:2
environment (env.metadata['render_fps'] is None or not defined), rendering may occur
  logger.warn(
Total reward during test: 11.0
Steps: 20
```



BUT...

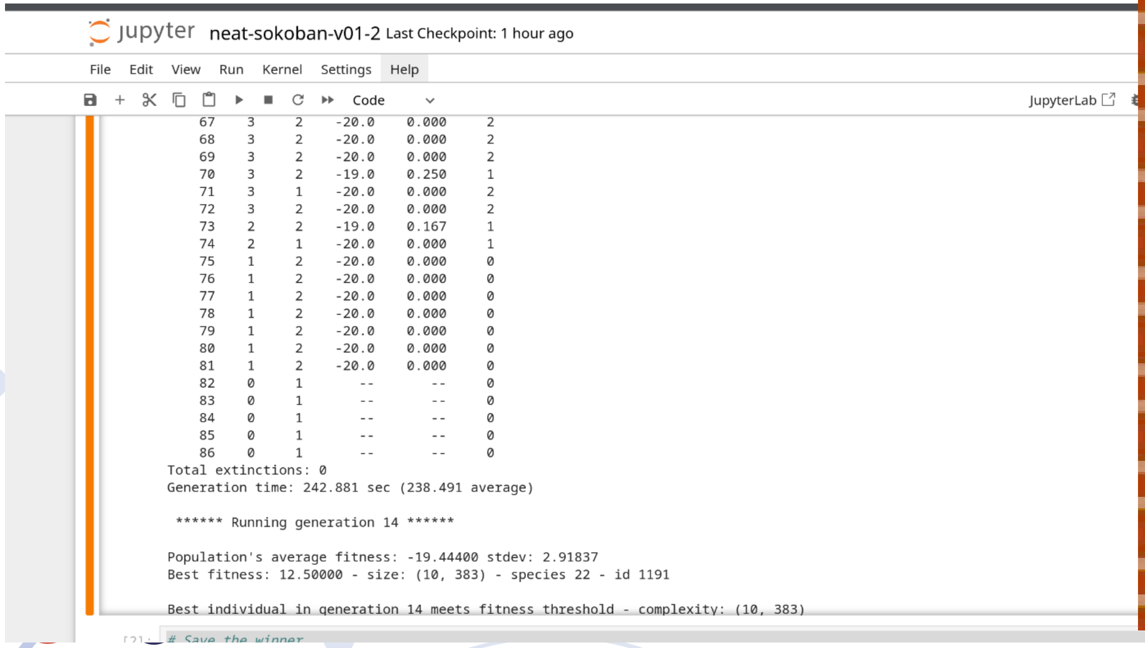
04

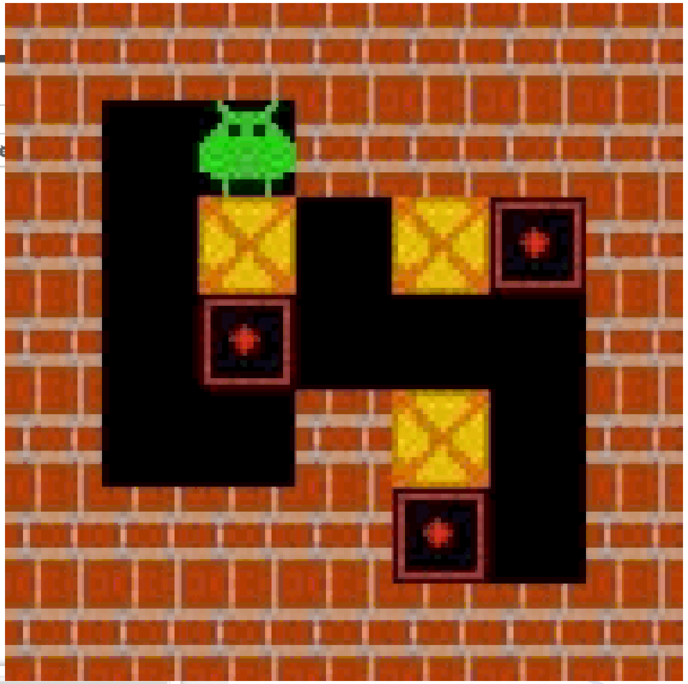
NEAT

Experiment with NEAT

Initial Test result

NEAT – SUCCEEDED too!





BUT…

And now..

The BIG BANG testing

04

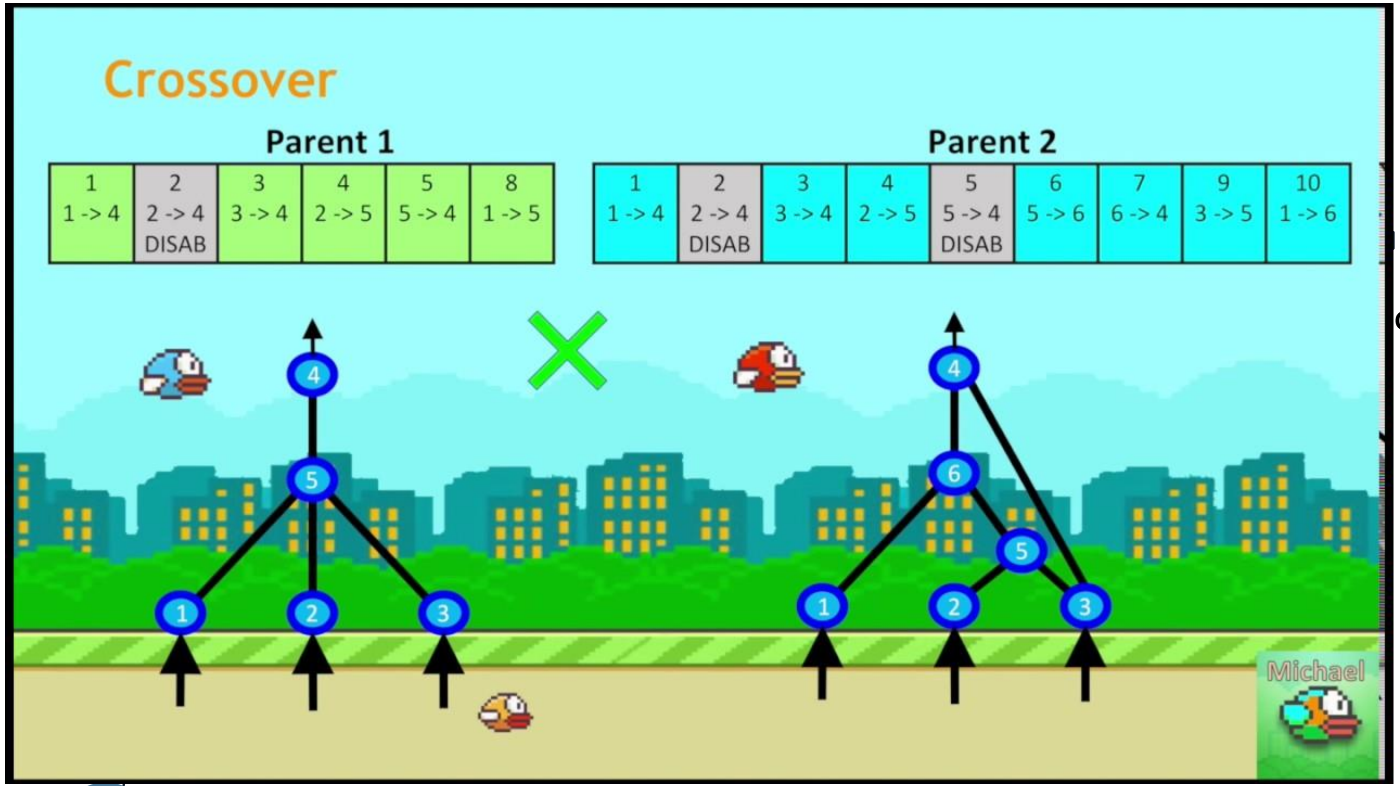
NEAT
The BIG BANG testing

Tweak configs

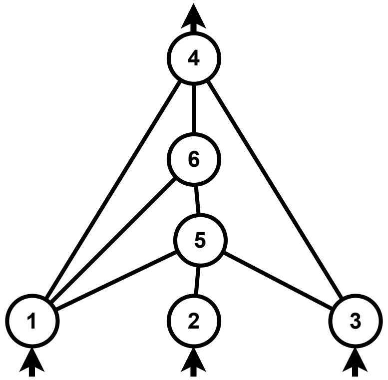
- Play with population size and num of generations
- Drastically increase mutation rates to promote diversification of NN’s
- Play with num of hidden layers (start with more complex NN’s)
- Play with Feedforward – in theory if False, it should allow NN’s to learn from past experience.

NEAT

The BIG BANG testing



	1	2	3	4	5	6	7	8	9	10
Parent 1	1→4	2→4	3→4	2→5	5→4			1→5		
Parent 2	1→4	2→4	3→4	2→5	5→4	5→6	6→4		3→5	5→6
Offspring	1→4	2→4	3→4	2→5	5→4	5→6	6→4	1→5	3→5	5→6



04

NEAT

The BIG BANG testing - results

04

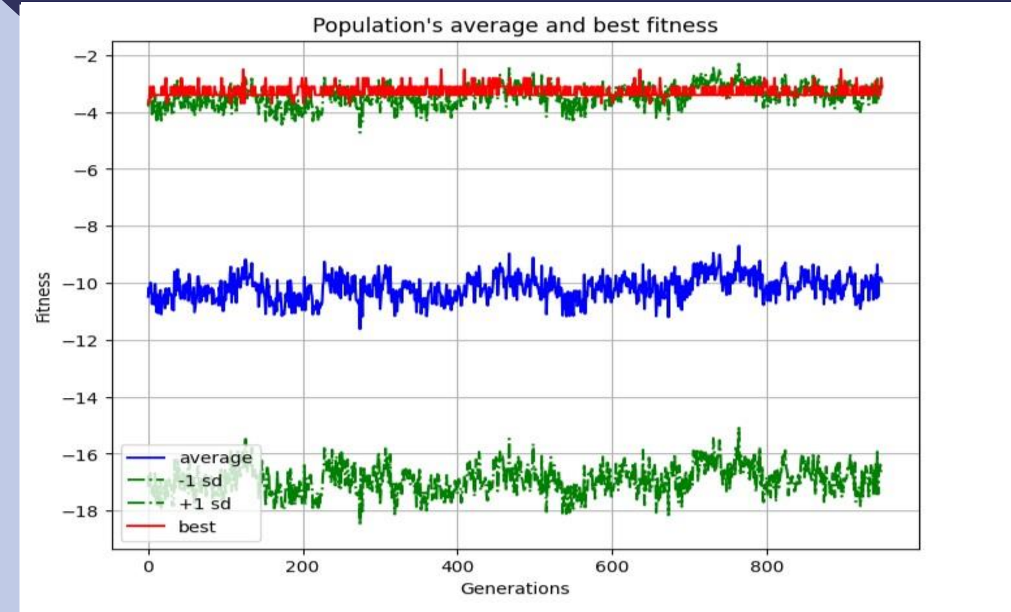
NEAT
The BIG BANG testing - results

Results

- Fitness stagnation..

Questions

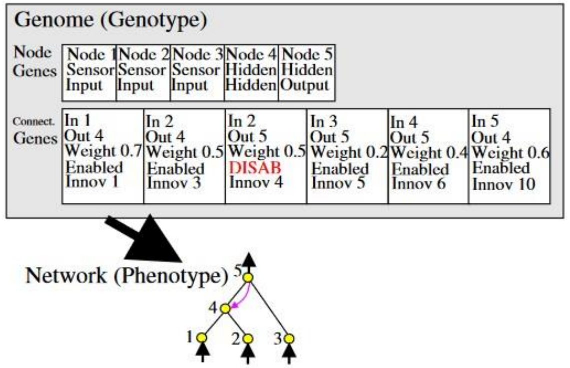
- How does neat-python actually perform selection, crossover & mutation?
- Are there issues with current way of Speciation?
- Are there other drawbacks on how the library operates?



05

Analyses, Issues, Lessons

05



NEAT Crossover

Issues

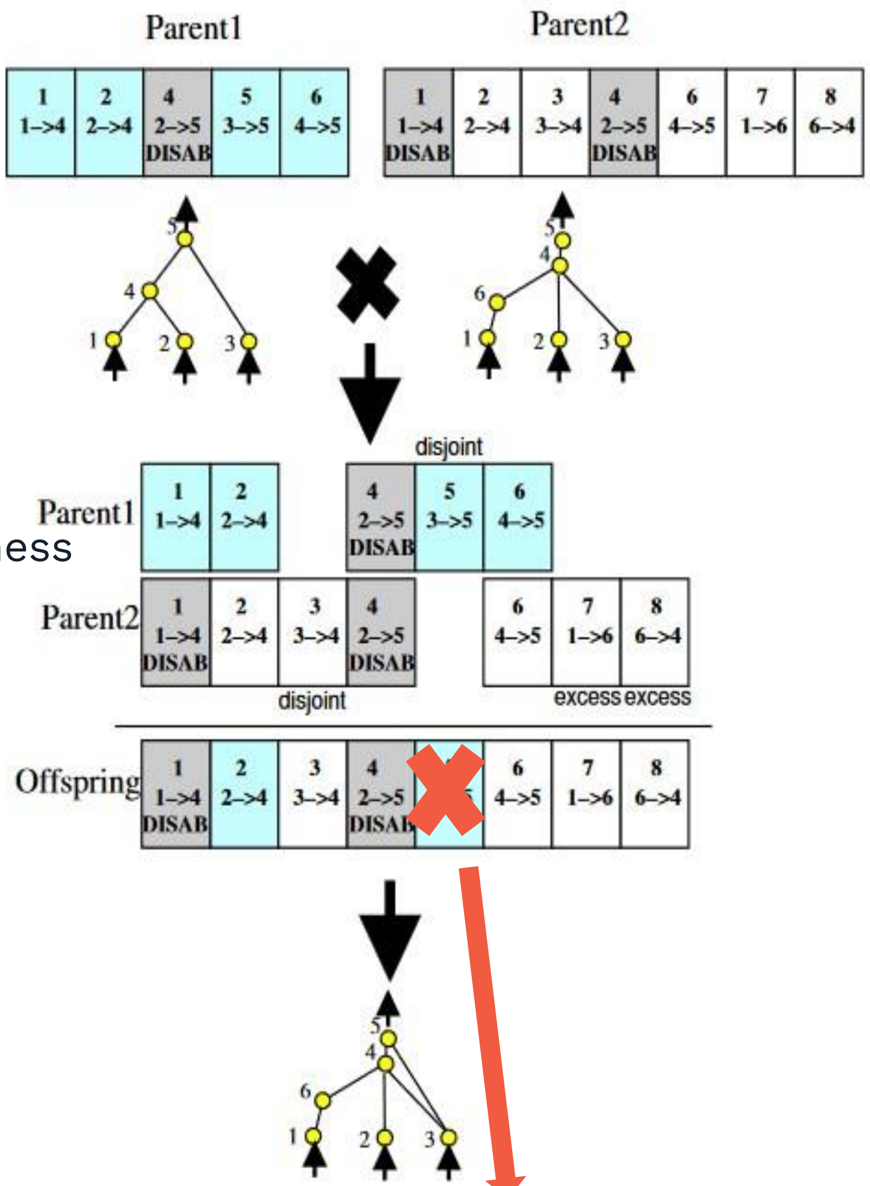
NEAT prefers genes from the more fit parent

- Reduced effectiveness of crossover after many generations
- As genomes grow (due to mutations) matching genes become rarer – as a result, the offspring usually inherits mostly from one parent

Weight incompatibility

- Connection weights of two parents could be way different and randomly picking one of the two can damage fine-tuned behaviors

Equal fitness assumed



this results in missing useful innovations from Parent 1

05

NEAT
Mutation

Issues

- **Doesn’t mutate outputs (steps in the game)**
- **Weight mutation could go wrong** (choosing a random weight that causes poor performance)
- **Leads to too many inactive connections** (when adding a new node or removing a connection)

05

NEAT
Speciation

How it works

- **Compatibility score** – determines whether a genome should be assigned to species X.

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \bar{W}$$

Where:

- E: number of excess genes,
- D: number of disjoint genes,
- $\bar{W}$: average weight difference of matching genes,
- N: normalization factor (typically number of genes in larger genome),
- c_1, c_2, c_3: coefficients tuning the importance of each term.

05

NEAT
Speciation

Species A (4 members):

Genome	Raw Fitness	Adjusted Fitness = Raw / 4
A1	4	1.0
A2	5	1.25
A3	6	1.5
A4	4	1.0

Total Adjusted Fitness (A) = 1.0 + 1.25 + 1.5

Species B (2 members):

Genome	Raw Fitness	Adjusted Fitness = Raw / 2
B1	10	5.0
B2	9	4.5

- Total Adjusted Fitness (All) = 4.75 (A) + 9.5 (B) = **14.25**

Allocation:

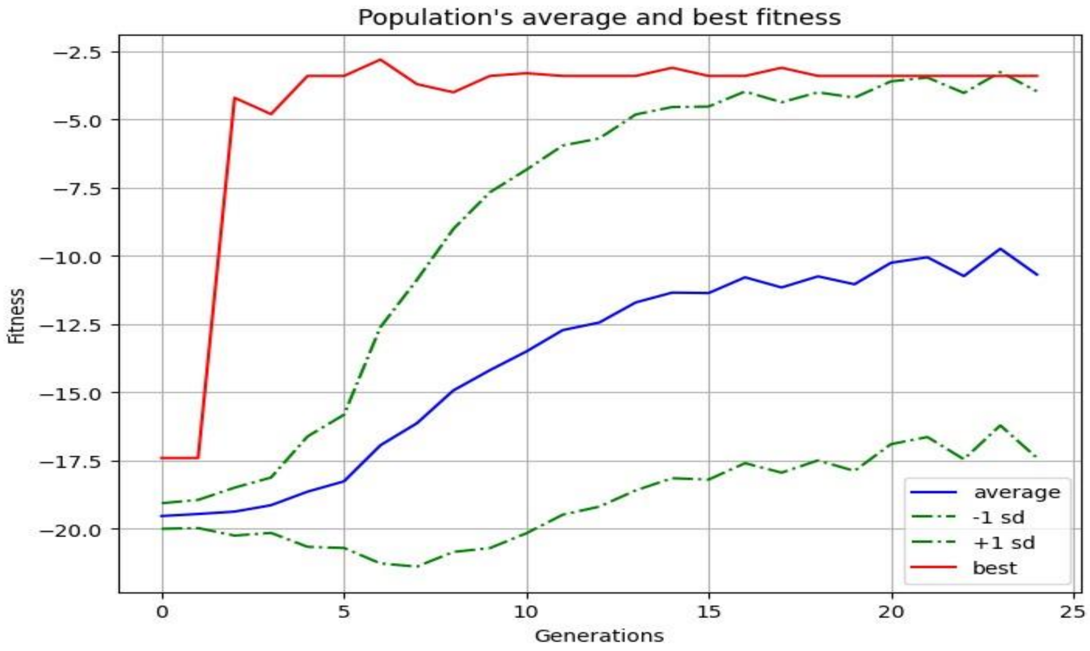
- Species A gets: $\frac{4.75}{14.25} \times 6 \approx 2$ offspring
- Species B gets: $\frac{9.5}{14.25} \times 6 \approx 4$ offspring

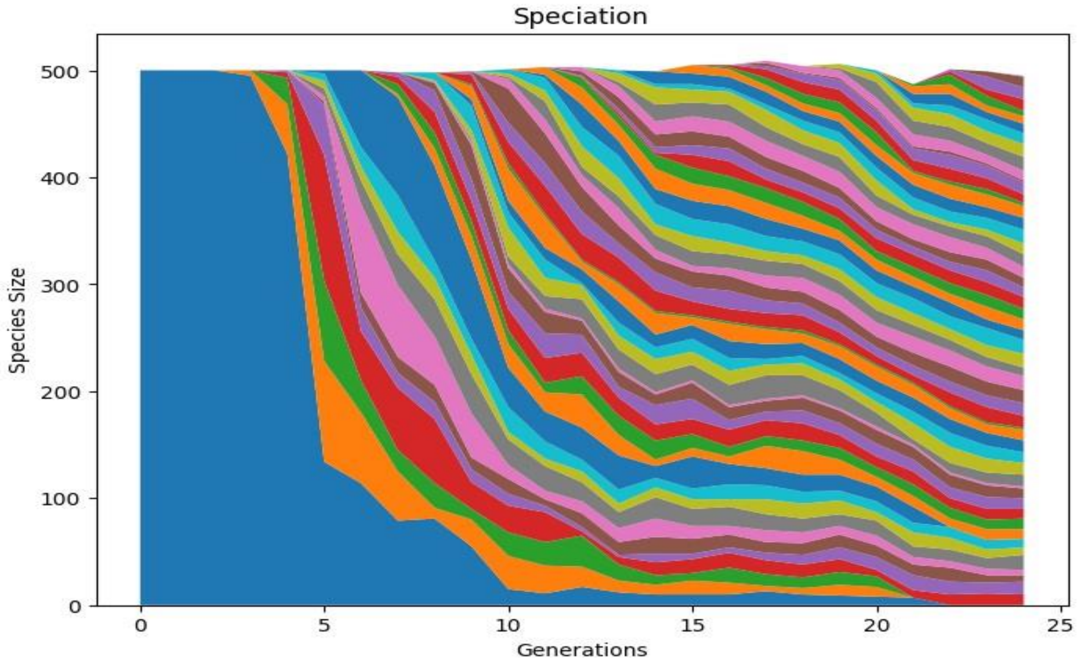
05

NEAT
Speciation

```
In [9]:  visualize.plot_stats(stats, ylog=False, view=True)
          visualize.plot_species(stats, view=True)
```

executed in 818ms, finished 23:40:40 2024-06-28





How number of Species could influence fitness?

Adjustments to the plots

- take the best fitness + k best fitnesses
- measure the fitness (avg) only of the n elements of the population (should equal the selection threshold number)

06.1

Future implementations

Phase 1 – “Incremental improvements”

06.1

Definition

Phase 1 – “Incremental improvements”

Improvements that could be made by simpler logic adjustments to neat-python.

06.1

NEAT

Incremental improvements

Crossover

Problem

Inheritance predominantly from the fitter parent.

Solution

Randomness in the crossover - traits from the less fit parent also have a chance to be inherited by the offspring/child (excess or disjoint genes) – using parameter that controls the chance of lesser fit parent to inherit a gene.

06.1

NEAT
Incremental improvements

Crossover

Problem

Weight incompatibility – connection weights of two parents could be way different and randomly picking one of the two for the next offspring can break fine-tuned behaviors in later generations

Solution

Weight averaging or some type of interpolation (random choice, blend, fitness_based)

06.1

NEAT
Incremental improvements

Speciation

Problem

Due to Stagnation we might delete the best solution OR a promising solution that couldn’t evolve further due to being stuck in stagnating species.

Solution

Species Rejuvenation
Don’t kill – revitalize instead (encourage exploration by increasing mutation rates drastically). Kill if no improvement is shown.

Currently controlled by few parameters:

- Max stagnation (n generations)
- Rejuvenation generations (n generations)
- Rejuvenation mutation multiplier – value + 0.5*time since rejuvenation (n generations)

06.1

NEAT

Incremental improvements

Species (Fitness sharing)

Problem

Currently doesn't capture novelty (or doesn't capture genomes that are still exploring different paths towards solving the task).

Doesn't account for: Behavioral diversity; Structural novelty; Potential long-term value.

Solution

Add behavior-based stagnation metric on a genome level (e.g. compare decisions vectors – present vs past).

Kill if both fitness and behavior are stagnant.

Solution 2

Add behavioral diversity (or novelty) as a factor for the adjusted fitness score for each genome.

$$\text{Adjusted fitness} = \left(\frac{\text{fitness}}{\text{species size}}\right) + \lambda \cdot \text{novelty score}$$

Where:

- `novelty score` = average behavior distance to other species or to historical archive
- `λ` is a tunable hyperparameter controlling novelty reward

This way a genome that has low fitness, but explores different behaviors gets the chance to produce more offsprings.

06.1

NEAT
Incremental improvements

Elitism

Problem

Elitism is only local to species. But if the species get deleted, so does the best genome.

Solution

Global Elitism – keep the best X (or X% of) genomes in the population.

06.1

NEAT
Incremental improvements

Network validation

Problem

Currently no network validation after mutation – due to mutation multiple networks have unconnected neurons (addition of nodes or disabled connections). Results in a waste of computational power.

Solution

In each network check for unconnected neurons and randomly associate at least one connection for them (input, output).

Force a random reconnection to the output neuron

06.1

NEAT
Incremental improvements

Network connections

Problem

Weak connections
(weight close to zero, e.g. 0.00005)

Solution

Prune them (delete these connections) or mutate them.

06.1

NEAT
Incremental improvements

Mutation

Problem

Currently no mutation parameter for mutating a decision, a step, or a move

Solution

Implement a mutation parameter that can change a decision. Could result in finding a better path into solving the problem.

06.2

Future implementations

Phase 2 – “Evolution”

06.2

Definition

Phase 2 – “Evolution”

A semi self-organizing NEAT involving automated definition of initial architectures; dynamic hyperparameter tuning; Scalable GPU-accelerated NEAT. Best NEAT traits combined with best traits of Traditional Optimization methods.

06.2

NEAT
Evolution

Weights optimization

Problem

Slow weight optimization (done only via crossover & mutation) - no backpropagation or gradient descent possible.

Solution

combine NEAT with a proven weight optimization algorithm (PSO, Nelder-Mead – gradient free optimization methods)

i. NEAT + PSO
ii. NEAT + Nelder-Mead
iii. NEAT + PSO + Nelder-Mead (PSO does global weight search, Nelder-Mead does the finetuning of weights)

06.2

NEAT
Evolution

Smarter Mutation

Problem

Mutation is a good mechanism for exploring diverse solutions, however because it's random it could create too many bad performers.

Solution

Steered mutation – define a score that measures whether a past mutation was good or not. If it was, steer future mutations in the same direction.

(like a RL mechanism for mutations)

06.2

NEAT
Evolution

Hyperparameters (initial configuration)

Problem

Too many hyperparameters to setup manually (includes both the initial architecture as well as NEAT specific hyperparameters)

Solution

Initial lightweight hyperparameter search (could be Grid Search, Bayes optimization, Random search, Evolutionary methods (e.g. a GA), etc.) that checks for promising configurations.

06.2

NEAT

Evolution

Static hyperparameters

(Species – speciation similarity threshold)
(Mutation – too little or too much)
(Selection & Crossover – using always the same approach)

Problem

Static hyperparameters often cannot address issues that the system faces in the mid and late stages.

Solution

Adaptive Hyperparameters – Dynamically adjustable hyperparameters based on system state (and metrics associated with it).

Example – introduce higher mutation rate for stagnating species few generations before automatically deleting them.

OR adjust the Speciation similarity threshold for few generations to allow for more diverse population.

06.2

NEAT
Evolution

Computation

Problem

NEAT-python is CPU-based by default (uses one CPU core). It evaluates each genome sequentially – very very slow.

Solution

Use parallel processing (per CPU core)
Use GPU acceleration

Slowest part – fitness evaluation
Offload NN evaluation to the GPU by using PyTorch or TensorFlow – represent NN's as Tensors and evaluate them in parallel.

OR use CUDA (Nvidia toolkit, dev env for GPU accelerated applications) or JAX (framework for high-performance numerical computing) – Allows massive parallelism for matrix computations.

06.3

Future implementations

Phase 3 – “Revolution”

Introducing “RevNEAT”

06.3

Definition

Phase 3 – “Revolution”

Fully self-organizing NEAT which should help you solve a problem with very limited human intervention. In addition, a few paradigms will change (e.g. NN evolution – switch to modular approach; initial architecture definition – explore more than one type of NN).

NEAT
Revolution

Automatic decision on which type of NN is the most appropriate

Problem

Sometimes we don't know which type of NN would be the most promising for solving the task at hand.

Solution

Let NEAT test different types of NN's initially and see which ones are promising.

E.g. with Sokoban we could define three types of inputs: Flat representation, Image-like representation (7x7xC) (CNN's), Graph-based representation (GNN's).

06.3

NEAT
Revolution

📌 **Automatic decision on which type of NN is the most appropriate (Sokoban example)**

1. **Flat Representation (49 Nodes)**
   - **7×7 board as a 49-element vector**, where each element represents the type of object at that position (e.g., walls, player, boxes, targets).
   - This is a simple approach but **loses spatial relationships**.
2. **Image-like Representation (7×7×C Input - CNN Approach)**
   - **7×7 grid with multiple channels (C)** representing different objects (e.g., player, walls, boxes, targets).
   - Allows the use of **Convolutional Neural Networks (CNNs)**, which are great for grid-based games.
3. **Graph-based Representation (GNN Approach)**
   - Treat Sokoban as a **graph**, and design a **Graph Neural Network (GNN)** where nodes represent key game objects and edges represent relationships (e.g., adjacency, interactions).

06.3

NEAT
Revolution

Crossover on a modular level

Problem

Too simplistic crossover. Two parents. Offsprings inherit only single genes.

Solution

Modular crossover, allowing networks to combine entire sub-networks rather than individual neurons.

06.3

NEAT
Revolution

Crossover on a modular level

Problem

NEAT evolves one monolithic brain for solving a task.

Some problems (e.g., game-playing, robotics) benefit from modular networks—specialized sub-networks that handle different tasks.

NEAT-Python doesn't automatically discover modularity.

Solution

Modular evolution – a technique where instead of evolving a single monolithic network, NEAT evolves sub-networks (modules) that solve different aspects of the problem. The key idea is to promote the evolution of specialized parts that can later be combined into a more complex structure.

06.3

NEAT
Revolution

Modular evolution

Sub-network specialization

- Evolve independend sub-networks that perform different tasks. In Sokoban:
  - Pathfinding module – optimal path to a goal
  - Push Strategy module – when and how to push
  - Deadlock avoidance module
  - Goal prioritization module - Deciding which box to push first
- Each module is a separate species, evolving its own structure.

Combine modules into a Meta-Agent

- Use a higher-level controller that decides which module to activate based on the game state.
- The controller itself evolves, learning how to switch between modules.

Effect: The network doesn't need to be trained to solve **everything at once**—each sub-task is handled **separately**, and evolution finds how to combine them.

06.3

NEAT
Revolution

Modular evolution (Meta-Agent)

How Does It Decide Which Module to Activate?

1. State Features: The agent **extracts key features** from the Sokoban board, such as:

1. Player position
2. Box positions
3. Goal positions
4. Walls and obstacles

2. Feature-Based Activation: The **meta-controller** activates the most relevant module:

1. If no boxes are nearby → **Use Pathfinding**
2. If the agent is near a pushable box → **Use Push Strategy**
3. If a move results in a deadlock → **Use Deadlock Avoidance**
4. If multiple moves are possible → **Use Goal Prioritization**

06.3

NEAT
Revolution

And finally.. The cherry on the top of the cake:
Close to Fully self-organizing NEAT using ChatGPT prompting

Problem

Too many things to configure for a human being:
Hyperparameters (NEAT architecture, NN type to use, NEAT hyperparameters, Fitness function, etc.)

Solution

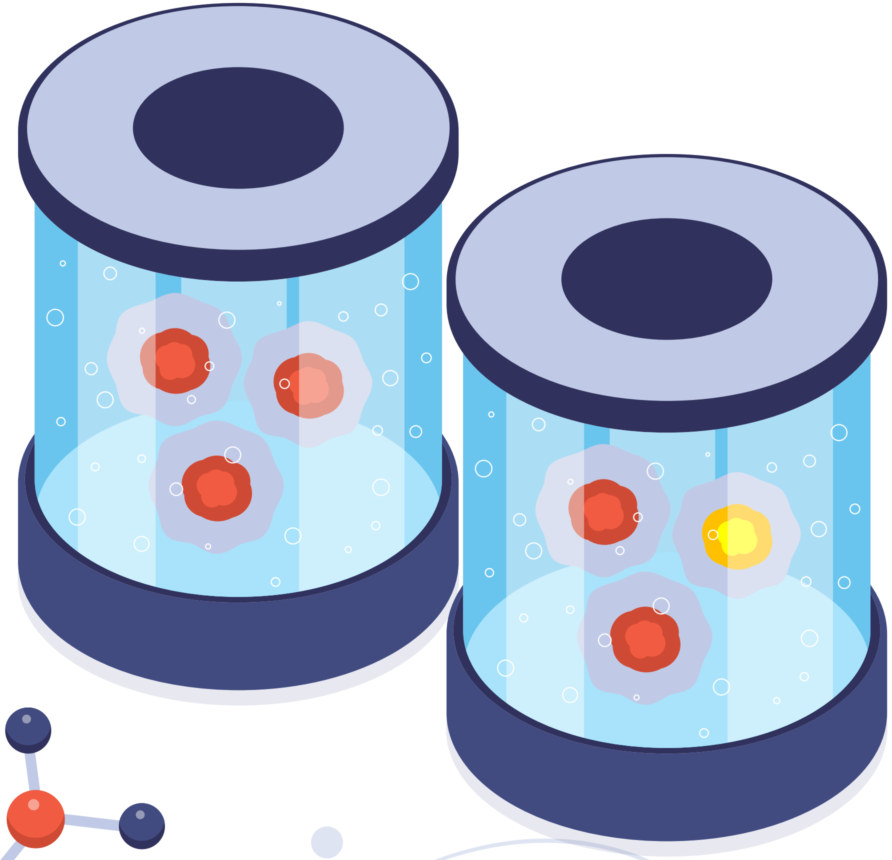
AI Agent helps the user to define:

- The fitness function based on task description
- The search space for best NN type
- The most promising NN architectures & Hyperparameters config (based on preliminary tests)

06.3

NEAT
Revolution

Fully self-organizing NEAT using ChatGPT prompting



Example prompting

User:
- "I want to optimize warehouse routing for delivery trucks".

Chat:
interprets the **key objectives** (e.g., minimize distance traveled, balance truckloads) & **defines a fitness function** that aligns with these goals

- "Are you optimizing for speed, efficiency, or cost?"

User:
- "Cost is most important, then efficiency."

Chat:
creates a weighted fitness function balancing cost & efficiency

06.3

Dream Scenario

Phase 3 – “Revolution”

Essentially the goal (the dream) is to create a single pipeline that receives a problem description (e.g. Solve Sokoban OR Optimize PI planning, Warehouse Operations, Seating plans, etc.) and solves the task with limited human assistance.

06.3

Speal


Спри! Него не!!!
Той винаги казваше "благодаря"
10 години по-рано
ChatGPT
Супер отговор, благодаря.

06.3

Some resources..

- **Efficient Evolution of Neural Network Topologies** (Kenneth O. Stanley and Risto Miikkulainen) - https://nn.cs.utexas.edu/downloads/papers/stanley.cec02.pdf

- **Testing the NEAT Algorithm on a PSPACE-Complete Problem** (Angel Marchev, Dimitar Lyubchev, Nikolay Penchev) - https://github.com/Marchev-Science/AIMSA2024-paper

- **Applying Genetic Algorithms for Optimizing Program Increment Planning in Software Development Teams** – (Dimitar Lyubchev, Angel Marchev) - https://www.researchgate.net/publication/384794878_Applying_Genetic_Algorithms_for_Optimizing_Program_Increment_Planning_in_Software_Development_Teams

- **Neat-python library** - https://neat-python.readthedocs.io/en/latest/neat_overview.html

THANKS!

DO YOU HAVE ANY QUESTIONS?

dimitar.lyubchev@gmail.com

d.lyubchev@unwe.bg

Dimitar Lyubchev

d.lyubchev