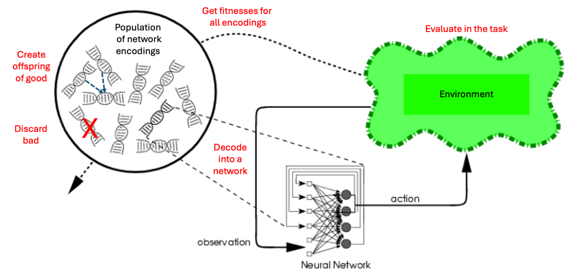
Fundamentals of Neuroevolution

Risto Miikkulainen

November 11, 2024

Neuroevolution vs. Gradient Descent

- Evolutionary Algorithms (EAs) optimize parameters without explicit gradients.
- Neural networks are powerful and flexible models.
- Backpropagation is key to deep learning but relies on differentiable functions.
- Many real-world problems lack well-behaved, differentiable objective functions.
- Neuroevolution combines EAs with neural networks to solve these challenges.



Neuroevolution vs. Reinforcement Learning (RL)

- RL algorithms require a reward signal at each timestep.
  - Lifelong learning, tracking changing environments
- EAs focus on the final cumulative reward after the agent's rollout.
  - Engineering; separate learning and performance phases
- EAs can be advantageous in tasks where only the final outcome matters.

NE Implementation: Fitness evaluation

- Each agent is evaluated in a separate rollout.
- General formulation:
  - In lifelong tasks: Cumulative reward is used as the fitness score. Can be delayed and sporadic
  - In engineering tasks, only one reward in the end.

```
def rollout(agent, env):
  obs = env.reset()
  done = False
  total_reward = 0
  while not done:
    a = agent.get_action(obs)
    obs, reward, done = env.step(a)
    total_reward += reward
  return total_reward
```

NE Implementation: Search

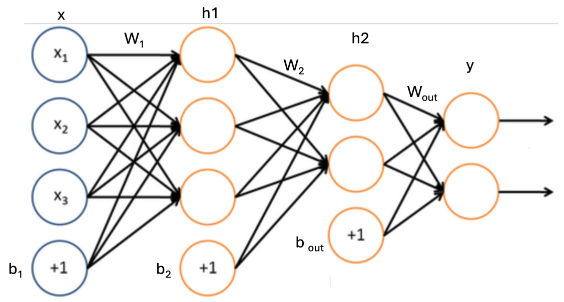
- E.g. OpenAI Gym environment
- The Evolution Strategy loop iterates until a solution is found that meets the requirements.
- The solver iteratively refines the model parameters.

```
env = gym.make('worlddomination-v0')
solver = EvolutionStrategy()
while True:
  solutions = solver.ask()
  fitlist = np.zeros(solver.popsize)
  for i in range(solver.popsize):
    agent = Agent(solutions[i])
    fitlist[i] = rollout(agent, env)
  solver.tell(fitness_list)
  bestsol, bestfit = solver.result()
  if bestfit > MY_REQUIREMENT:
    break
```

Neural Network for Policy Mapping

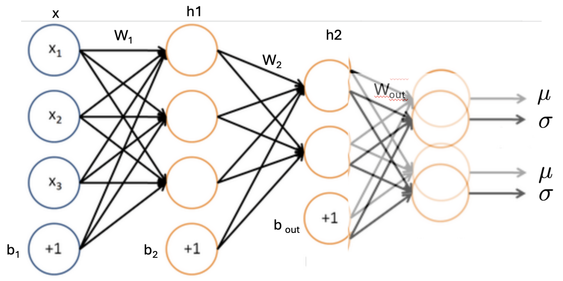
- The agent's observation is mapped to actions via a neural network.
- The network includes two hidden layers
- The connection weights and bias weights are evolved.

$$h_1 = f_h(W_1\, x + b_1), \quad (1)$$
$$h_2 = f_h(W_2\, h_1 + b_2), \quad (2)$$
$$y = f_{out}(W_{out}\, h_2 + b_{out}) \quad (3)$$



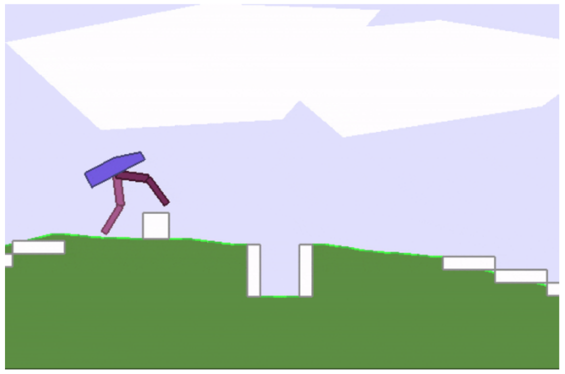
Deterministic vs. Stochastic Policies

- Deterministic policies map inputs to actions directly.
- Stochastic policies introduce randomness in action selection.
- Stochastic policies can prevent local optima and encourage exploration.
- Expand each output to two values: μ and σ
- Sample action values from $N(\mu, \sigma$



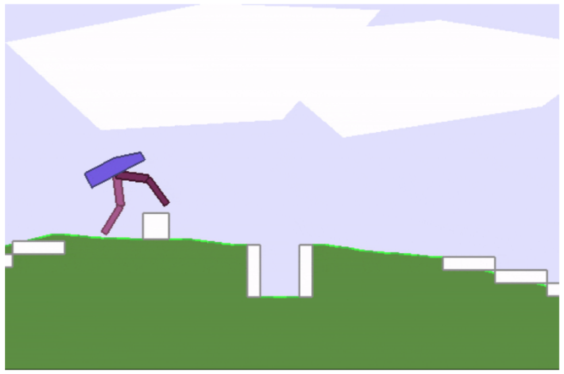
Example: Evolving a Bipedal Walker

- Neuroevolution (NE) is well-suited for evolving robust policies.
- Tradeoff between sample efficiency and policy robustness is critical.
- Example: Bipedal Walker environment in OpenAI Gym.



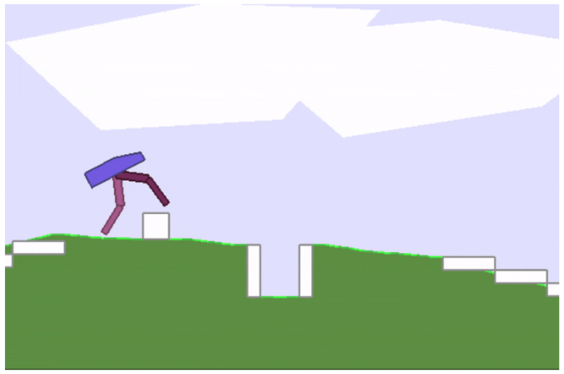
Bipedal Walker Environment Details

- The agent must navigate randomly generated terrain.
- 24 inputs: lidar sensors, angles, contacts (no absolute coordinates).
- 4 continuous outputs controlling motor torques.
- Reward based on distance, with penalties for excessive energy use.



Defining Task Success

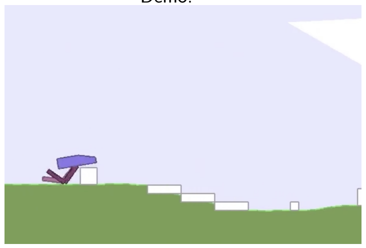
- Task success: average score of 300+ over 100 trials.
- Challenge: RL algorithms struggle with consistency and efficiency.
- NE can evolve policies that consistently meet the task's success criteria.



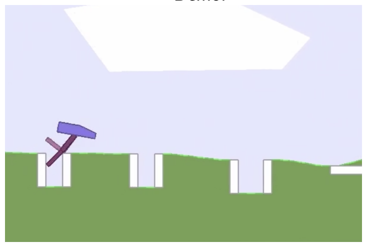
Initial Discovery of Walking

- Initially just have to learn to walk forward
- Then to get over obstacles
- RL often gets stuck in local minima
- NE can get unstuck and continue evolving

Demo:



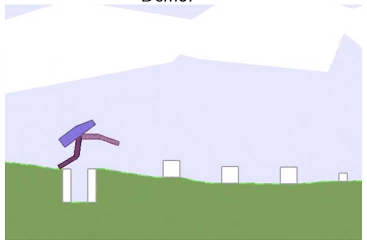
Demo:



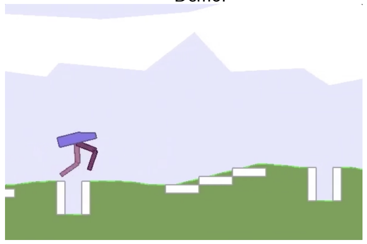
Eventually Robust Success

- NE learns several different strategies
- E.g. reach over the obstacle
- E.g. jump over obstacles
- Do they work on new terrain?

Demo:



Demo:



Handling Randomly Generated Terrains

- Random terrains introduce variability in task difficulty.
- Solution: Average over 16 random rollouts per agent.
- Fitness score based on the average cumulative reward.

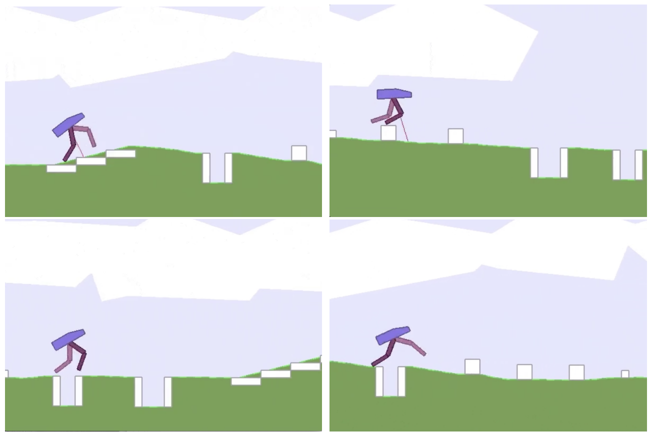
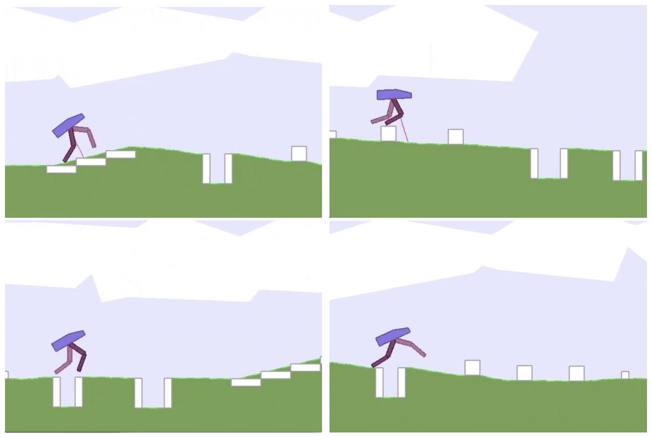


Figure: Averaging Rollouts on different terrains

Tradeoff: Data Efficiency vs. Robustness

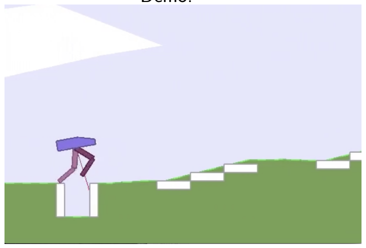
- Averaging rollouts increases robustness but decreases data efficiency.
- The final policy becomes more consistent across varied trials.
- Achieving an average score of 300+ over 100 trials demonstrates robustness.



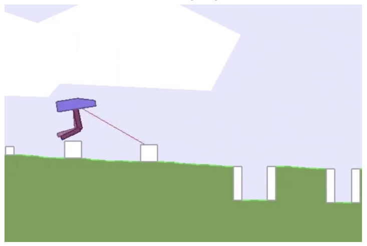
Importance of Robust Policies in the Real World

- Robust policies are essential for real-world applications.
- Engineers often need to satisfy Quality Assurance and safety factors.
- NE offers a way to evolve policies that meet these stringent requirements.

Demo:



Demo:



Evolving Convolutional Neural Networks

- NE algorithms can be applied to find weights for convolutional neural networks (CNNs).
- Example: Evolving a simple 2-layer CNN to classify MNIST digits.
- Supervised learning tasks: a good match with gradient descent, but can be used to benchmark NE.

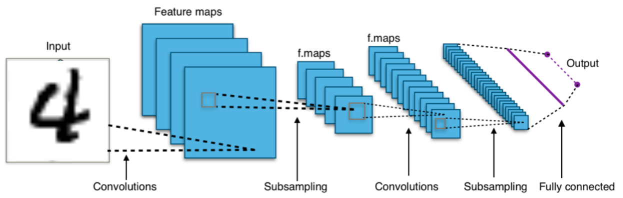
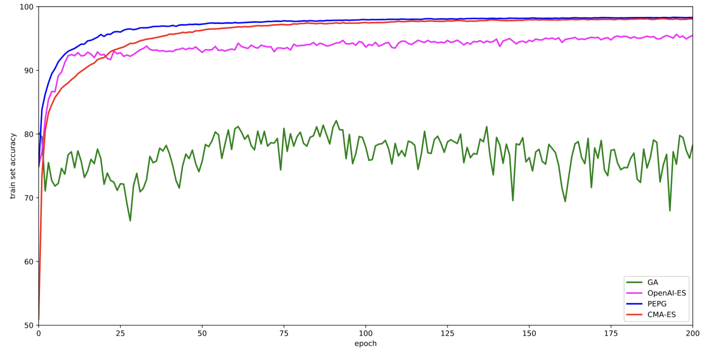


Figure: Simple 2-layer CNN for MNIST Classification

Comparing Evolutionary Methods on MNIST

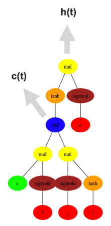
- CMA-ES is a very powerful method–comparable to backprop.
- Simple GA is a relatively weak baseline
- Scaling up NE to larger CNN requires indirect encoding

Method	Train Set	Test Set
Adam (BackProp) Baseline	99.8%	98.9%
CMA-ES	98.4%	98.1%
OpenAI-ES	96.0%	96.2%
Simple GA	82.1%	82.4%

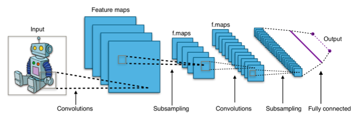


Topology and Weight Evolving Networks

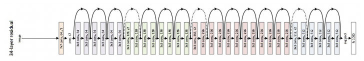
- Simple neuroevolution focuses on evolving weight parameters.
- It is also possible to evolve the architecture (morphology). Should we?
- Many neural network innovations have historically been hand-crafted:

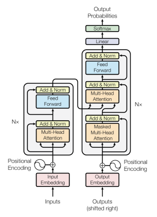


LSTM gates address the vanishing gradient problem in RNNs.



CNNs minimize connections in computer vision tasks.

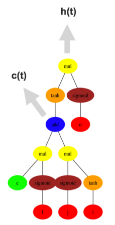

Residual Networks (ResNets) allow deep stacking of layers.



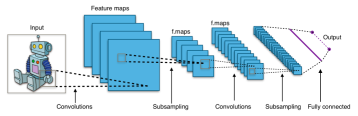
Transformer architectures enhance expressivity and trainability.

Automating Neural Network Discovery

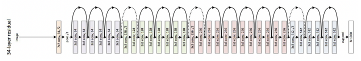
- Neuroevolution aims to automate the discovery of novel neural network architectures.
- Evolving both topology and weights can lead to highly optimized networks.
- It is also possible to evolve the architecture only, and backprop the weights.

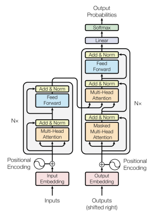


LSTM gates address the vanishing gradient problem in RNNs.



CNNs minimize connections in computer vision tasks.

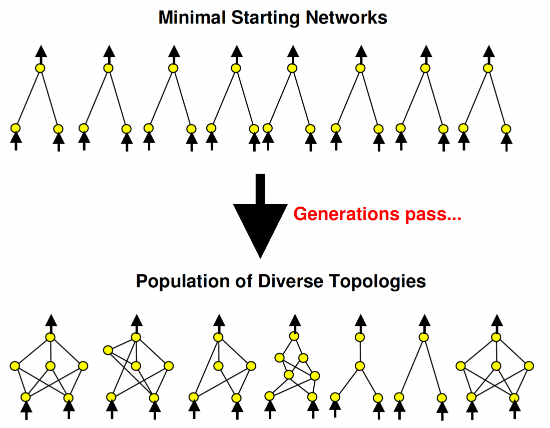

Residual Networks (ResNets) allow deep stacking of layers.



Transformer architectures enhance expressivity and trainability.

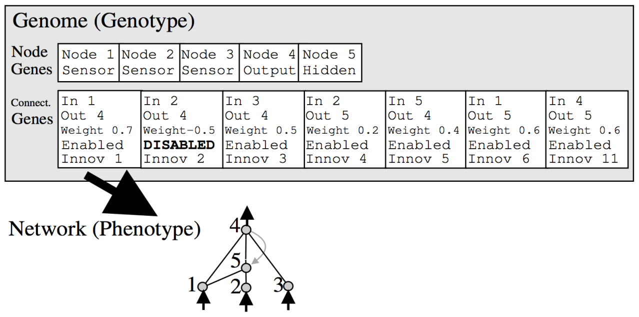
Neuroevolution of Augmenting Topologies (NEAT)

- NEAT is a popular method for evolving neural network topologies.
  - Developed in 2002 (by Ken Stanley at UT), there are now over 100 variations.
  - Often the first method to try on a new problem.
- It is best suited for evolving small recurrent networks, i.e. behavior.
- The main idea is *complexification*:



NEAT: Representation of Networks

- Each neuron and connection is assigned a unique historical marker.
- Networks are represented as a list of connections and weights.
- This allows NEAT to track the evolutionary history of each network.
- This allows representing arbitrary structures and matching them up for crossover.



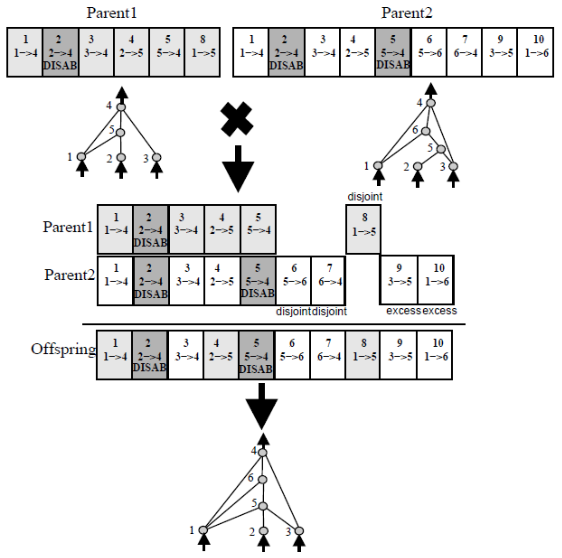
Node Genes				
Node 1 Sensor	Node 2 Sensor	Node 3 Sensor	Node 4 Output	Node 5 Hidden

Connect. Genes						
In 1	In 2	In 3	In 2	In 5	In 1	In 4
Out 4	Out 4	Out 4	Out 5	Out 4	Out 5	Out 5
Weight 0.7	Weight-0.5	Weight 0.5	Weight 0.2	Weight 0.4	Weight 0.6	Weight 0.6
Enabled	**DISABLED**	Enabled	Enabled	Enabled	Enabled	Enabled
Innov 1	Innov 2	Innov 3	Innov 4	Innov 5	Innov 6	Innov 11

Figure: Representation of Networks in NEAT

NEAT: Crossover Operation

- Crossover combines two parent networks to produce a new network.
- Matching genes are inherited randomly from either parent.
- Disjoint and excess genes are added (randomly, or all) to the offspring.



NEAT: Mutation Operations

- Mutation can adds new neurons and new connections between existing neurons.
- This allows the network to grow in complexity over generations.
- Motivated by biological complexification of genomes.

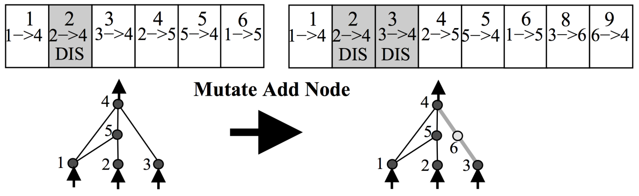


Figure: Mutation: Adding a New Neuron

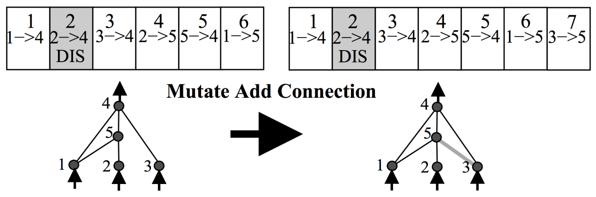
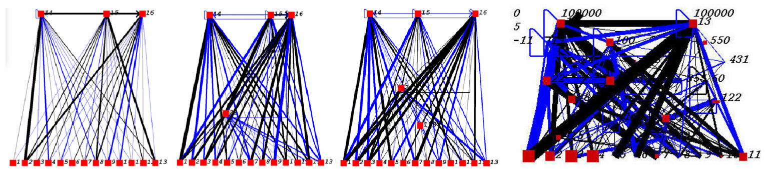


Figure: Mutation: Adding a New Connection

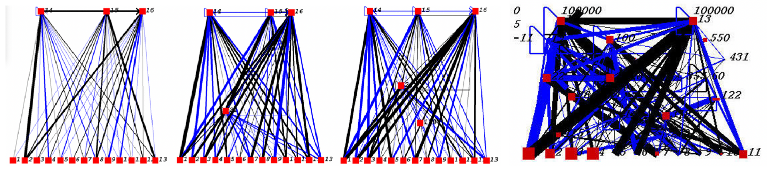
NEAT: Initial Population and Complexification

- NEAT begins with a simple initial population of networks.
- Networks start with minimal connections and no hidden layers.
- Mutation adds new neurons and connections.
- Note: No simplification is needed
  - Any structure added only stays if it is useful.
  - Sometimes adapted to new uses.
  - No need to ever discard.



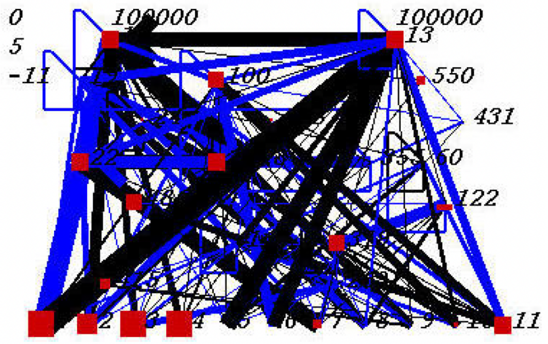
NEAT: Understanding the Solutions

- The resulting networks are parsimonious and interpretable:
- Complexification → elaboration of behavior
- Can analyze behavior at each step and identify what caused it.
- Can understand what each element is doing!



NEAT: Discovering Complexity

- Discovers complexity that otherwise would not be possible.
- E.g. in robotic foraging/pursuit/evasion, discovered a complex solution.
- Initializing population with it and evolving only weights doesn't work!
- It is only possible to discover through complexification.



NEAT: Speciation

- Speciation groups similar networks into species.
- It protects innovation: New structures have a chance to be optimized before they have to compete with others.
- It maintains diversity: Species are formed if they are diverse enough.

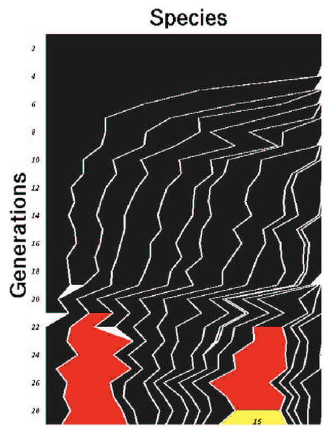


Figure: Speciation over time; white triangles indicate extinct species, red good solutions (1 stdev), yellow best solutions (2 stdev)

NEAT: Speciation

- Species are dynamically calculated at each generation
- They get larger if they perform well and shrink if poorly
- Species emerge and die out, similar to biological evolution.

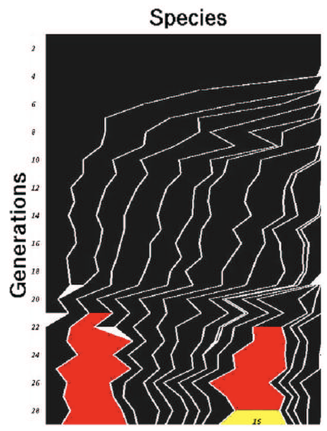


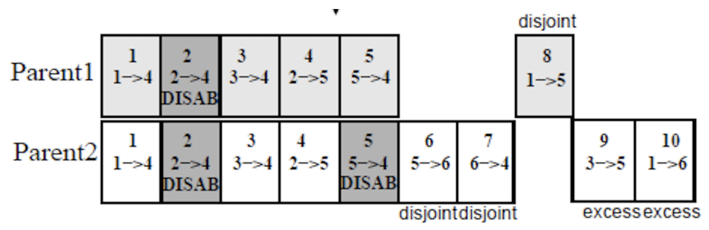
Figure: Speciation over time; white triangles indicate extinct species, red good solutions (1 stdev), yellow best solutions (2 stdev)

NEAT: How Speciation Works

- Speciation is based on a distance measure between networks:

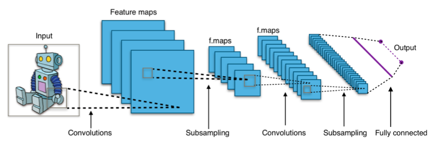
$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \overline{W}$$

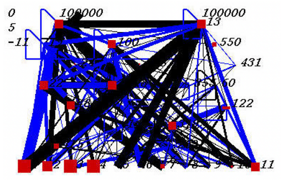
- Thus, δ is a linear combination of the number of excess (E) and disjoint (D) genes and the average weight differences of matching genes (W).
- Networks within a certain distance form a species or subpopulation.



Neuroevolution vs.' Deep Learning

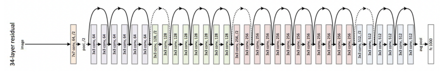
- Neuroevolutionary networks differ from those in deep learning.
- Focus is on AI-based decision making, not prediction from big data.
- Utilize neural computation when there are no targets, only fitness.

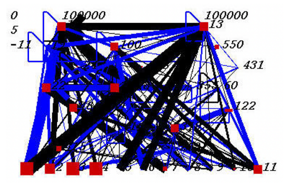




Neuroevolution vs.' Deep Learning Architectures

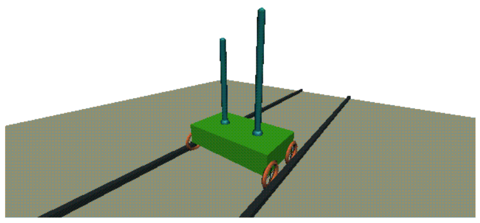
- Computational requirements and network designs are different.
- Deep learning often relies on overparameterization (e.g., ResNet modules).
- NEAT, by contrast, evolves networks with purpose-driven complexification.





Explainability of Neuroevolved Networks

- As a result, neuroevolved networks can be compact and explainable.
- Elements are constructed with specific functions, enhancing transparency.
- Example: A NEAT-evolved solution for the pole-balancing problem.
  - Using the recurrent connection to itself, the single hidden node determines whether the poles are falling away or towards each other.
  - This solution allows controlling the system without computing the velocities of each pole separately.



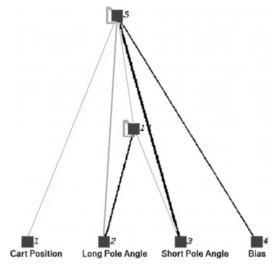
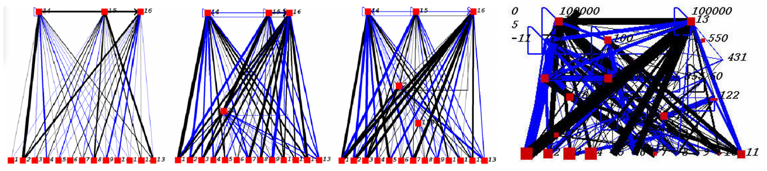


Figure: NEAT Solution for Pole-Balancing Problem

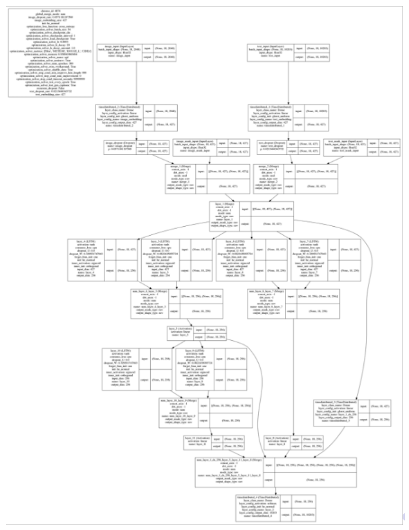
Regularization and Overfitting

- Neuroevolved networks tend to be more regularized, avoiding overfitting.
- Compact networks generally lead to better regularization.
- This is particularly useful in applications with small datasets.



Extension: Combining Neuroevolution with Backpropagation

- Neuroevolution is excellent for finding network architectures.
- Backpropagation can be used to optimize weights within the discovered architecture.
- Combining both methods leverages the strengths of each approach.



Extension: Neuroevolution for Neuromorphic Hardware

- Deep learning depends on large-scale hardware and lots of energy.
- Neuroevolution offers an alternative for edge devices with limited resources.
- Evolved networks can be optimized for the given hardware constraints.
- This flexibility is crucial for neuromorphic and other emerging hardware.

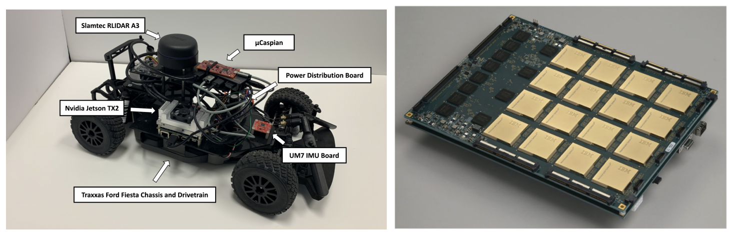
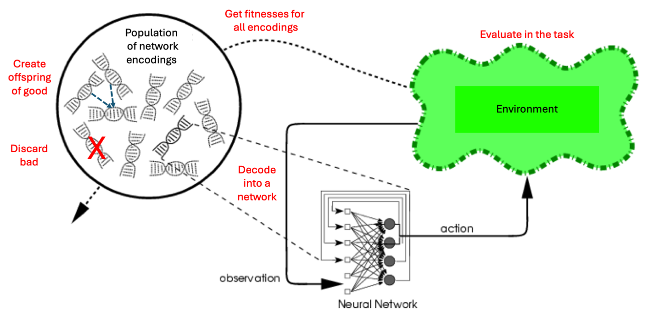


Figure: Efficiency of Neuroevolution on Minimal Hardware

Conclusion

- Neuroevolution is a useful tool in the machine learning / AI toolbox.
- It makes it possible to discover behavior when optimal targets are not know.
- It finds creative solutions that other methods are likely to miss.
- It applies to a broad range of problems in the real world
- It can be used to enhance other methods, like Deep learning, reinforcement learning, hardware, LLMs.
- It may allow us to gain insight into biology and cognition


Get fitnesses for all encodings
Population of network encodings
Create offspring of good
Discard bad
Evaluate in the task
Environment
Decode into a network
action
observation
Neural Network