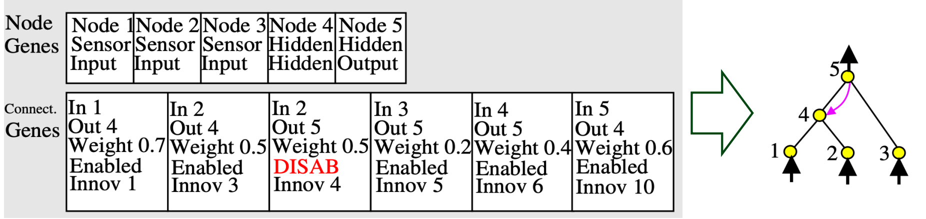
NEAT ENCODING

- Genome composed of
  - Node genes (can be input, hidden or output)
  - Connection genes
    - in_node, out_node, weight, enabled, innovation_ID (discussed later)

- Mutation
  - Weight mutation perturbs the weight
  - Add connection mutation
    - adds a new connection between two existing nodes (e.g. 3 and 4 in figure)
  - Add node mutation
    - Splits an existing connection by adding a node "in the middle"

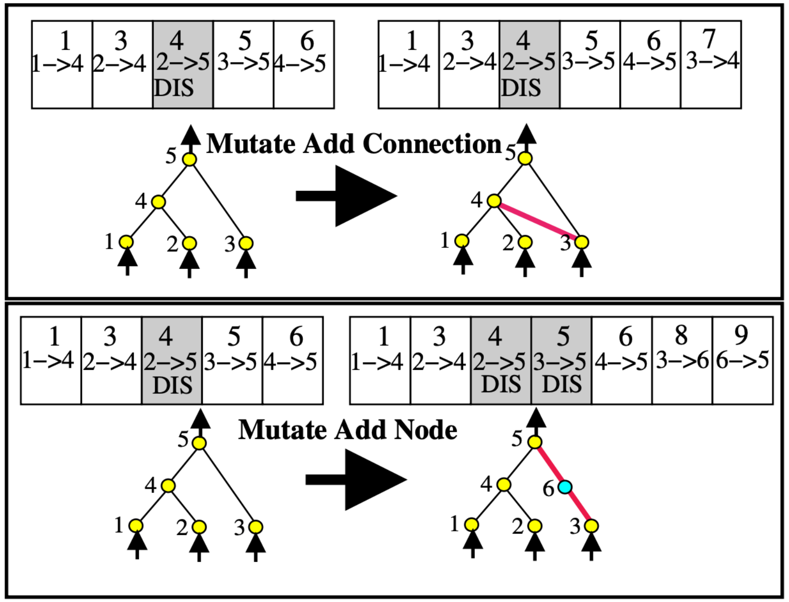


Node Genes

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

Connect. Genes

In 1	In 2	In 2	In 3	In 4	In 5
Out 4	Out 4	Out 5	Out 5	Out 5	Out 4
Weight 0.7	Weight 0.5	Weight 0.5	Weight 0.2	Weight 0.4	Weight 0.6
Enabled	Enabled	DISAB	Enabled	Enabled	Enabled
Innov 1	Innov 3	Innov 4	Innov 5	Innov 6	Innov 10



1	3	4	5	6
1−>4	2−>4	2−>5 DIS	3−>5	4−>5

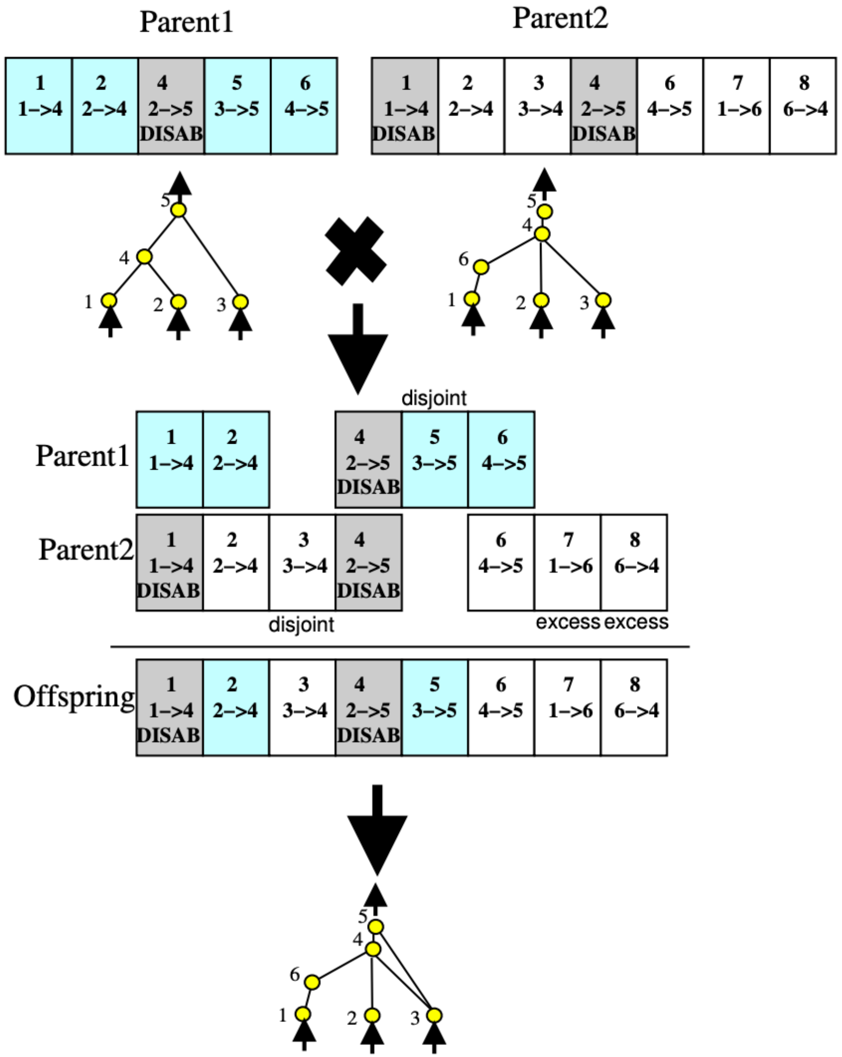
1	3	4	5	6	7
1−>4	2−>4	2−>5 DIS	3−>5	4−>5	3−>4

1	3	4	5	6
1−>4	2−>4	2−>5 DIS	3−>5	4−>5

1	3	4	5	6	8	9
1−>4	2−>4	2−>5 DIS	3−>5 DIS	4−>5	3−>6	6−>5

NEAT CROSSOVER

- How to crossover networks with different topologies?
- Crossover needs to identify "corresponding" parts of the network in the two parents to mix them effectively (avoiding competing conventions)
- Corresponding means that they were introduced in a common ancestor of the two parents
- How to know that? Simple idea:
  - Every connection is identified by an innovation ID, assigned incrementally every time a connection is created
- Crossover lines up genes by innovation ID and selects genes at random by either parents



Parent1

1 1->4	2 2->4	4 2->5 DISAB	5 3->5	6 4->5

Parent2

1 1->4 DISAB	2 2->4	3 3->4	4 2->5 DISAB	6 4->5	7 1->6	8 6->4

Parent1	1 1->4	2 2->4		4 2->5 DISAB	5 3->5	6 4->5		
Parent2	1 1->4 DISAB	2 2->4	3 3->4	4 2->5 DISAB		6 4->5	7 1->6	8 6->4
Offspring	1 1->4 DISAB	2 2->4	3 3->4	4 2->5 DISAB	5 3->5	6 4->5	7 1->6	8 6->4

NEAT EXAMPLE

```
"""
2-input XOR example -- this is most likely the
simplest possible example.
"""

import os
import neat


# 2-input XOR inputs and expected outputs.
xor_inputs = [(0.0, 0.0), (0.0, 1.0), (1.0, 0.0),
(1.0, 1.0)]
xor_outputs = [ (0.0,), (1.0,), (1.0,), (0.0,)]


def eval_genomes(genomes, config):
  for genome_id, genome in genomes:
    genome.fitness = 4.0
    net = neat.nn.FeedForwardNetwork.create(genome,
config)
    for xi, xo in zip(xor_inputs, xor_outputs):
      output = net.activate(xi)
      genome.fitness -= (output[0] - xo[0]) ** 2


def run(config_file):
  # Load configuration.
  config = neat.Config(neat.DefaultGenome,
neat.DefaultReproduction,
             neat.DefaultSpeciesSet,
neat.DefaultStagnation,
             config_file)

  # Create the population, which is the top-level
object for a NEAT run.
  p = neat.Population(config)

  # Add a stdout reporter to show progress in the
terminal.
  p.add_reporter(neat.StdOutReporter(True))
  stats = neat.StatisticsReporter()
  p.add_reporter(stats)
  p.add_reporter(neat.Checkpointer(5))

  # Run for up to 300 generations.
  winner = p.run(eval_genomes, 300)

  # Display the winning genome.
  print('\nBest genome:\n{!s}'.format(winner))

  # Show output of the most fit genome against
training data.
  print('\nOutput:')
  winner_net =
neat.nn.FeedForwardNetwork.create(winner, config)
  for xi, xo in zip(xor_inputs, xor_outputs):
    output = winner_net.activate(xi)
    print("input {!r}, expected output {!r}, got
{!r}".format(xi, xo, output))


run('config-feedforward')
```

NEAT HYPER-PARAMETES

- A number of hyperparameters are used to define

- Parameters in original NEAT paper:
  - Population: 150-100
  - Similarity: c_1 = 1, $c_2 = 0.4$, $\delta_t = 3 - 4$
  - Species becomes extinct after 15-20 generations without improvements
  - Elitism: 1 (if species has at least 5 members) top 60% reproduce
  - 80% weight mutation probability, 75% crossover probability
  - Add node probability 0.03, Add link probability 0.05 up to 0.3 for big populations

- Default parameters for python-neat XOR example
  - Population 150
  - Similarity c_1 = 1, $c_2 = 0.5$, $\delta_t = 3$
  - Extinction at 20 generations, elitism 2, top 20% reproduce
  - 80% mutation probability, 100% crossover probability
  - Add node probability 0.2, Add link probability 0.5 (quite high) but has similar remove probabilities
  - initial_connection can be full or partial 0.5

OPENAI-GYM

- Environments
  - Key class of Gym used to simulate the world experienced by the agent
  - Environments are created by using the make method, and initialised using the reset method

    ```
    import gym
    env = gym.make('MountainCar-v0')
    ```

  - Environments evolve step by step
  - agents observe the state of the environment to decide which action to apply to the world and receive a reward, typical loop:

    ```
    obs, info = env.reset() #reset env to random state, returns initial obs
    done = False; total_reward = 0
    while (not terminated and not truncated):
        action = agent(obs)
        obs, reward, terminated, truncated, info = env.step(agent)
        total_reward +=reward
    ```

- Observation and Action Spaces can be continuous (Box) or discrete
  - E.g. MountainCar observation is continuous (position, velocity); action is discrete 0=left, 1=nothing, 2=right

    ```
    print(env.observation_space)
    print(env.action_space)
    Box([-1.2 -0.07], [0.6 0.07], (2,), float32)
    Discrete(3)
    ```

GYM EXAMPLE (JUPYTER)

```
from pyvirtualdisplay import Display
virtual_display = Display(visible=0,
size=(1400, 900))
virtual_display.start()
import matplotlib.pyplot as plt
%matplotlib inline
from IPython import display
def jrender(env, step=None, info=""):
    plt.figure(3,(5,5))
    plt.clf()
    plt.imshow(env.render())
    if (step!=None):

info="step:{}|{}".format(step,info)
    plt.title("%s | %s"%(env.spec.id,
info))
    plt.axis('off')

    display.clear_output(wait=True)
    display.display(plt.gcf())
    plt.close()
```

```
import gym
env = gym.make('MountainCar-v0',
render_mode= 'rgb_array')
print(env.observation_space)
print(env.observation_space)
print(env.observation_space.low)
print(env.observation_space.high)

obs, info = env.reset()
done = False; total_reward = 0; step = 0;
while (not done):
    action = 2 if obs[1]>0 else 0
    obs, reward, terminated, truncated,
info = env.step(action)
    done = terminated or truncated
    total_reward += reward
    step +=1

jrender(env,step,"Reward:{}".format(total
_reward))
```