University of Kent

AI SYSTEMS LECTURE 9

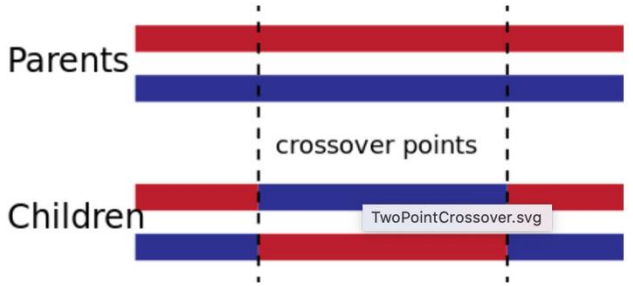
Genetic algorithms and Neuroevolution

OUTLINE

- Neuroevolution
  - Genetic Algorithms
  - NEAT
  - Extensions
- OpenGym

GENETIC ALGORITHMS

- Optimisation technique inspired by theory of evolution
- Population composed of N individuals (candidate solutions)
- Quality of each solution is measured by a *fitness* function (phenotype)
  - Each solution is composed by a set of genes
- Natural selection and replication
  - Good gene combinations ('building blocks') becomes more prevalent as they make reproduction more likely
  - Selection: individual with higher fitness are more likely to be selected for reproduction
    - Roulette-wheel selection: selection probability is proportional to fitness
    - Tournament-based selection selects best individual among k randomly selected ones
    - Truncation-based selection: top x percent of individuals are selected
  - Replication: pairs of selected individuals generate two offspring, then recombined with a given probability
    - One-point crossover: pick one point in the gene sequence, genes to the left are swapped in the two children
    - Multipoint crossover: k points are picked and children gets alternating intervals of genes from each parent
    - Uniform crossover: each gene is selected randomly from parents
- Mutation
  - With a given probability genes are perturbed in resulting individuals
  - This can (re-) introduce diversity that was lost due to selection pressure
- Elitism
  - Top k individuals are propagated to the next generation without modifications, especially useful if crossover and mutation probability are high

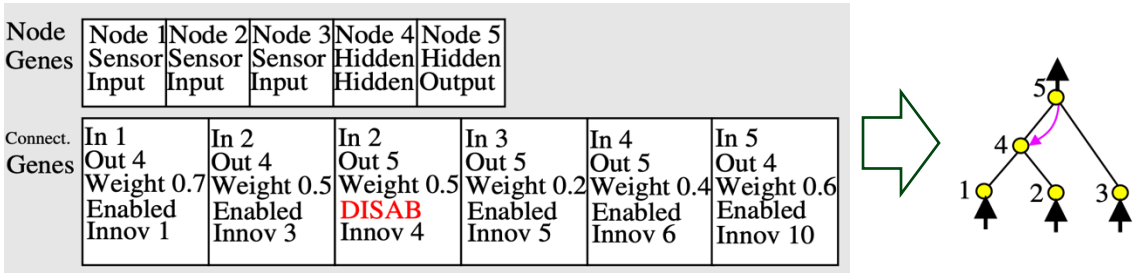


EVOLVING NN WITH GENETIC ALGORITHMS

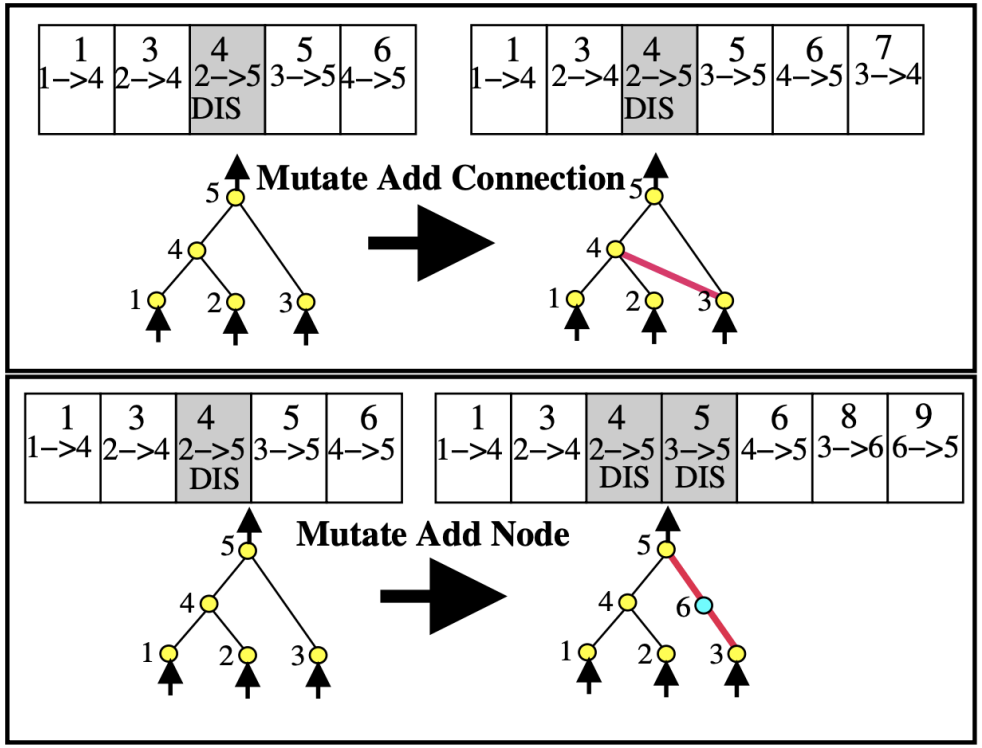
- GA can be applied to evolve neural networks weights
  - Define a network topology
  - Represent the weights of the network as genes (direct encoding)
  - Evolve weights using mutation and crossover
    - Crossover of network weights is problematic due to the problem of *competing conventions*: different networks can have concepts (e.g. hidden neurons) in different position of the network
- GA network evolution is an alternative to backpropagation based on gradients
  - Can be less efficient but can be used outside of supervised-learning, i.e. when direct input-output examples are not available
  - E.g. in reinforcement learning: action = network(observations), but direct action,observation pairings are not available (it is not known which is the best action given an observation), the quality (fitness) of a network is determined by the total *reward* obtained by the network over multiple actions, without knowing exactly which action was responsible for good reward, i.e. reward is delayed and sparse
- NeuroEvolution of Augmenting Topologies (NEAT)
  - Evolves the network topology in addition to the weights
  - Challenges
    - How to crossover network with different topologies?
    - How to manage the evolution of topologies and weights at the same time?
    - How to grow topologies incrementally, i.e. avoiding unecessarily complex topologies?

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)

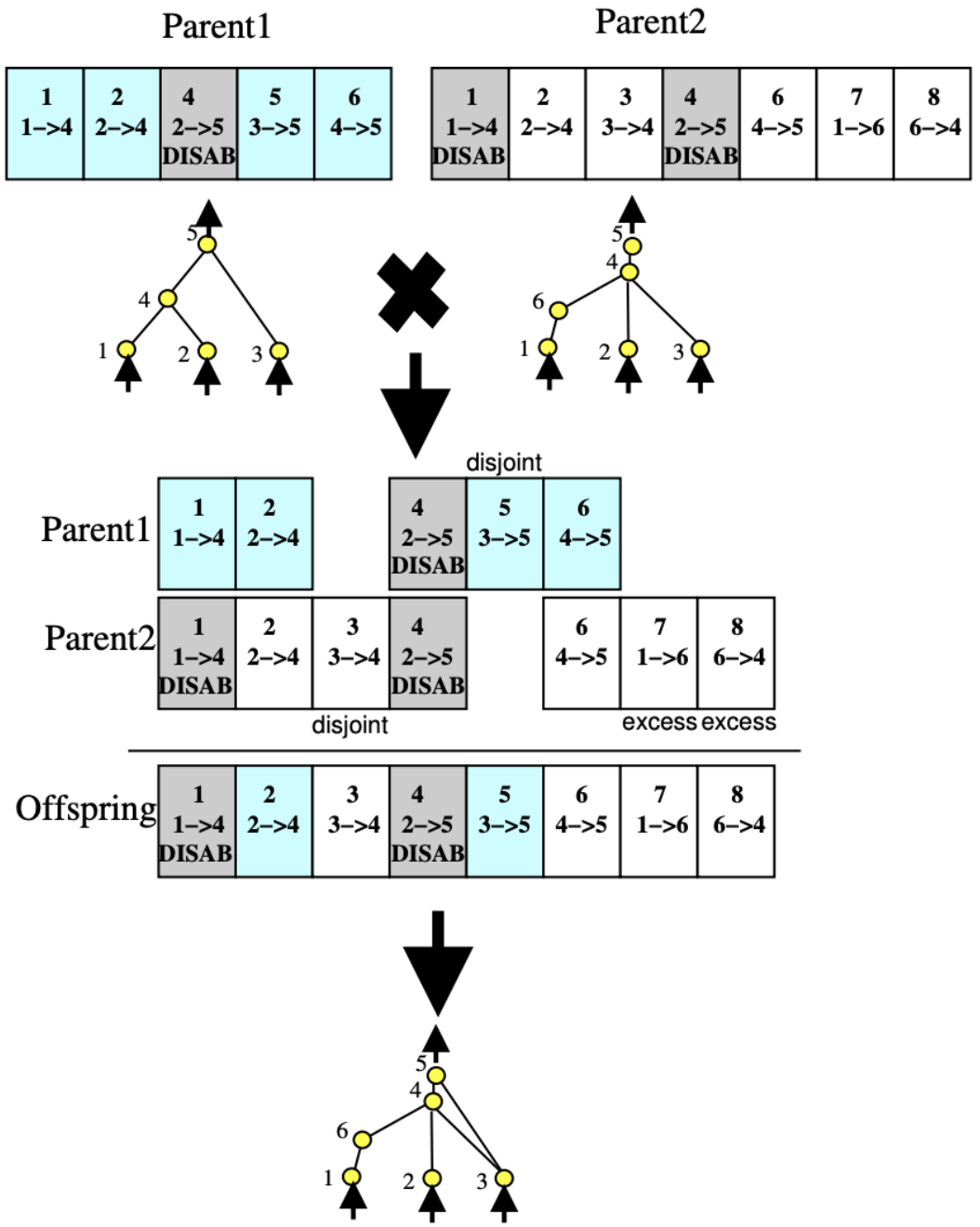


- Mutation
  - Weight mutation perturbs the weight
  - Add connection mutation
    - adds a new connection between two existing nodes (e.g. 3 and 4 in figure)
  - Add node mutation
    - Splits an existing connection by adding a node "in the middle"



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



NEAT NICHING

- How to manage the evolution of topologies and weights at the same time?
- When a new topology is introduced, by crossover or mutation, it is likely that it would not initially perform well because its weights are not optimised
- It is important to protect new topologies from competition from already established topologies
- Speciation: individual compete with networks that are "close" (Niching)
  - Define a similarity metric $\delta = c_1 \Delta G + c_2 \Delta W$ where ΔG is the average number of non-matching genes and ΔW is the average weight difference in matching genes
  - Speciation: place an individual in a species if its distance from the representative individual of the species is less than δ_t (compatibility threshold), if no compatible species exist create a new species with the individual as representative
- Reproduction
  - top k individuals of each species survive unmodified to the next generation (elitism)
  - top 60% reproduce replacing all other individuals in the species
  - the size of each species in the next generation is proportional to its average fitness
  - Species that do not improve for 20 generations become extinct

NEAT INITIALISATION

- How to grow topologies incrementally, i.e. avoiding unnecessarily complex topologies?
- Previous techniques start with a population of random topologies but this increases the size of the search space, potentially unnecessarily.
- Neat instead starts with topologies which are the simplest possible topologies (no hidden nodes) and grows them as incrementally as they are found useful, protecting new topologies through speciation.
- Thanks to the reduction in the search space NEAT can evolve good networks quickly and these are more likely to be minimal, which also gives an advantage during inference.

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

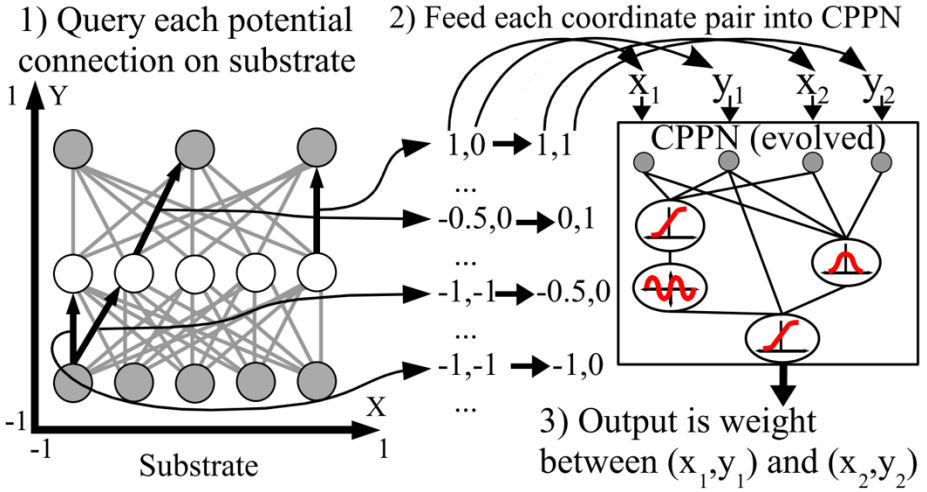
- A number of hyperparameters are used to define

- 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

HYPER-NEAT AND ES-HYPERNEAT

- Standard NEAT works well for control problems where the number of inputs is rather limited
- When the number of input is high (such as images) a direct network encoding in terms of neurons and connection is too big to be handled efficiently by neuroevolution
- Indirect encoding: Instead of evolving a NN directly we evolve a more compact (less dimensions) representation that can be translated into a NN. This representation is actually a special type of neural network called compositional pattern-producing network (CPPN)



- CPPN can create patterns (of weights) with regularities and repetitions (https://nbenko1.github.io/#/) similarly to how e.g. CNN operate, they are much more compact (a change in a single CPPN-weight can affect multiple NN-weights)
- Location of hidden nodes can be manual or sampled automatically from CPNN (ES-HyperNeat)

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))
```

REFERENCES

- ES-HyperNEAT and HyperNEAT http://eplex.cs.ucf.edu/papers/risi_alife12.pdf
- Neuroevolution on Atari games https://www.cs.utexas.edu/~mhauskn/papers/atari.pdf
- Deep Neuroevolution at UBER https://eng.uber.com/deep-neuroevolution/