Chapter 1) The role of self-organization (organizing behavior) for synthesis (synthesis of whatever [i.e. audio, video, behavior, etc.]) and understanding of behavioral systems

Basic idea of ER

- Initial random population (controllers)
- Each robot is then executed
- Performance is evaluated
- Fittest robots are allowed to reproduce
- Process is repeated for a number of generations

ER shares many characteristics with other approaches:

- **Behavior-based robotics**
Give a robot collection of *simple basic behaviors*. Complex behavior of a robot emerges from *interaction of simple behaviors and the environment* (Brooks).

Simple behaviors are encoded separately. Separate mechanism decides to *what degree* each simple behavior contributes to the total complex behavior. *Simple behaviors can interoperate in either: Competitive – only 1 simple behavior affects an output (i.e. Brook’s subsumption method) or Cooperative – different simple behaviors contribute to an output (with different strengths [i.e. Arkin’s behavioral fusion via vector summation]).*

Same as in ER, environment plays central role in determining the role of each simple behavior

Usually designed through trial-and-error (programmer modifies simple behaviors & increases # of simple behaviors) while observing complex behaviors

Breakdown of the desired behavior into simpler behaviors is done **intuitively by the designer** (in ER this is a result of self-organizing process)

---

**Figure 1.2**
In the behavior-based approach the desired behavior is broken down by the designer into a set of basic behaviors which are implemented into separate sub-parts (layers) of the robot’s control system (top box). In Evolutionary Robotics the designer does not need to decide how to divide the desired behavior into basic behaviors (bottom box). The way in which a desired behavior is broken down into modules is the result of a self-organization process.

---

**Robot learning**

Idea: *control system (ANN)* can be trained using incomplete data and then generalize the acquired knowledge to new situations (what I am trying to do now with controller in webots)
Learning algorithms: (impose constraints on type of architecture that can be used and on quality & quantity of supervision required from designer [i.e. if back-propagation is used, the designer must provide correct values for the outputs for the wheels of the robot during learning])

- back-propagation learning
- reinforcement learning
- classifier systems
- self-organized maps
- etc

Reinforcement learning is only used to evaluate whether robot is doing good or bad (different from back-propagation)

Artificial evolution may be seen as a way of learning

2 differences (ER vs Robot learning):

1. Amount of supervision in evolution is much lower (only general evaluation - “how well robot performs task he is asked to perform”)

2. Evolution does not put any constraints on what can be part of self-organization process (i.e. characteristics of sensors, shape of robot, etc. -> these can be included in evolutionary process)

---

**Figure 1.3**

Example of evolved architecture for a tripod robot. **Left:** The robot body. Black circles represent original cells that develop into sub-networks connecting sensory and motor neurons (white circles) **Right:** The corresponding developed locomotion network. Labels indicate sensory and motor neurons corresponding to the white circles in the figure on the left. (From Kodjabachian and Meyer 1998a. Reprinted by permission of Taylor and Francis Ltd, http://www.tandf.co.uk/journals.)

^^ result of evolutionary experiment

---

- **Artificial life**

Artificial life attempts to understand life phenomena through simulation.
In complex dynamical system’s properties emerge from simple elements at lower levels (complex dynamic properties cannot be predicted [eventhough emerging from simpler properties])

**ER shares these characteristics with artificial life**

When using real robots, several factors must be taken into account: (Brooks)
- Friction
- Inertia
- Light
- Noise
- etc

Only information available in the environment can be used for training.

Behavior is an emergent property of the interaction between the robot and the environment. As a consequence, simple robots can produce complex behavior. (Braitenberg)

Properties of emergent behavior cannot easily be predicted from knowledge of rules that govern interactions (Inversely, it is difficult to predict which rules will produce which behavior)

Divide-and-conquer approach was used in traditional robotics (breakdown into Perception, Planning, and Action) -> has been criticised by many and produced limited results. Brooks proposed different approach, in which the division is accomplished at the level of behavior (desired behavior is broken down into set of simpler behaviors).

Control system is built incrementally (layer-by-layer), where each layer is responsible for simple behavior by directly linking sensors to motors. (more successful than the traditional approach^^)

It was said that the structure of behavioral modules should emerge, not be pre-programmed.

Why it is difficult to break complex behavior into simple behaviors?
We should analyse complex behavior from 2 perspectives:
1) From human perspective
2) From robot perspective

Breakdown is accomplished intuitively by researcher.
The fact that complex behavior can be divided into a set of simpler behaviors DOES NOT imply that simple behaviors can be implemented into separate layers of agents control system.

**ER frees developer from deciding about how to break desired complex behavior into simple behaviors (by relying on evaluation)**

Example: If we divide complex behavior, like “following the line, not colliding with objects, coming back to line, etc.” into simple behaviors and try to train 4 separate ANNs to perform each basic behavior, the controller will not function properly. On the contrary, we can easily evolve a single network to produce 4 different simple behaviors if we select individuals for their ability to perform complex behavior. (Proposes a [fully
connected perceptron with many inputs [all sensors] and 2 outputs [speed of 2 wheels] can produce complex behavior
Better results are obtained if each module is result of self-organization process (not decision made by designer)

**Complex behavior cannot be explained by internal mechanisms of an agent only.**
Example: distinguish how big an object is by using only: (possible – agent can circle around objects)
  1) move forward;
  2) avoid objects;
  3) turn toward object;
Interaction with environment is unexplored (i.e. sensory-motor coordination)
For an agent that interacts with external environment, each motor action has 2 effects:
  1) It partially determines how well the agent performs with respect to a given task
  2) It partially determines the next sensory pattern the agent will receive from the environment (which may determine whether agent will be able to solve its task or not)

It is very difficult to determine which motor action the agent should perform each time, taking into account both 1) and 2) [+ each motor action can have long-term consequences]

ER (relying on self-organization) is not affected by neither 1) nor 2) -> therefore is an ideal framework for studying adaptive behavior.

Many evolved robots exploit active interaction with environment in order to maximize behavior selection criteria.

In biology, it is possible to develop complex behavior without increasing the amount of supervision by:

**1) By competitions between or within species**
  If we wish to select individuals able to solve a task that requires specific competence, the easiest thing to do is to select individuals for their ability to solve that specific task. -> design fitness function that scores individuals according to their ability to solve that task (can only work for simple tasks -> for complex tasks, everyone will score 0 [NULL] -> **BOOTSTRAP problem**)
Possible solutions:
  a) Increase amount of supervision
  b) Start evolutionary process with simplified version of the task and then increase the complexity by modifying the fitness.
Example: victims try to run away from predator

**2) By letting the system to extract supervision from the environment**
  Environment does not tell agent how it should act to attain given goal
Example: Using sensors, a robot can learn the consequence of its actions in different environments.

2 ways of adapting to environment:
In principle, any ability that can be acquired

1) **through lifetime learning** (how agent can interpret information whether it is doing good or bad? Or what the agent should do?)
can also be genetically acquired

2) **through evolution**

Example: An individual that is **born** to be able to produce a behavior that is effective in different environments is **equivalent** to another individual that can adapt to each environment through **lifetime learning** (both individuals are organized in different ways). [Individuals who **adapt** to each environment, should be able to **detect** the environment in which they are located and should be able to **modify** their strategy accordingly. Individuals that are **born** with such adoption capabilities, **do not need to change** – more effective -> do not have to undergo adaptation process throughout their lives]

Very little evidence in **ER** that **evolution + lifetime learning** can evolve more complex levels of competence than evolution alone.

3) **By including genotype-to-phenotype mapping in evolutionary process**

   Development and evolution of evolvability

   **Genotype-to-phenotype mapping problem** – for adaptation to happen, they both must have **evolvability** (i.e. ability of random variations to produce possible improvements)

   In nature, mapping is evolved. When designing, designer must make the mapping (we do not know yet how the mapping is done in nature)

**Chapter 2) Evolutionary and Neural techniques**

ER is based on evolutionary techniques for developing robot's systems – controllers.

In most cases, controllers in robots are ANN (sometimes just programs)

Not the only approach, but still:

Artificial evolution of controllers has been applied to:

   a) Controllers
   b) Parameters of a controller
   c) Programs

ER uses ANN as a controller

GA operates on population of artificial **chromosomes (genotype)** [that encode characteristics of an individual {**phenotype**}] by **selectively**
reproducing the chromosomes of individuals with higher performance and applying random changes. This procedure is repeated for several generations. (i.e. chromosome that encodes connection weights of an ANN {different types of encodings exist: binary/real values/grey/alphabets [ternary]})

Example: Chromosome - an array of connection weights of an ANN

Fitness f(x) evaluates performance of each phenotype (phenotype – generated complex appearance/behavior)

Genetic operators:
1) Selective reproduction
   Making copies of best individuals in the population
   (individuals with better fitness values tend to leave higher number of copies of their chromosome for the next generation)
   Often Roulette Wheel is used or Rank Selection

2) Crossover
3) Mutation

Should artificial evolution be implemented as a genetic algorithm?
⇒ relevant when we focus on self-organization (artificial evolution seen as spontaneous development) rather than system optimization (artificial evolution seen as search technique). As a result, computational efficiency, optimum became secondary. More important is emergence of complex abilities from a process of autonomous interaction between agent and its environment.

Autonomous systems are expected to survive in unknown and unpredictable environments by finding solutions to challenges that may
arise. It is difficult to establish a priori what abilities will be necessary to score highest fitness

The more detailed and constrained fitness function is, the more evolution becomes a supervised learning -> less space is left to emergence and autonomy of evolving system

Hidden layers are called hidden because they are not in direct contact with the environment.

ANN is a parallel computational system, because signals travel independently

![ANN Diagram](image)

**Figure 2.5**
Left: An artificial feed-forward neural network. Hidden units are called so because they are not in direct contact with the external environment. Signals flow from input to output. Right: Computing the output of a neuron. Each input signal is multiplied by the corresponding connection strength and the results added together. The net input is then passed to a function whose output may take into consideration a threshold value.

![Output Functions](image)

**Figure 2.6**
Three common output functions. From left to right: step, linear, sigmoid. The x axis shows the weighted sum, the y axis the value returned by the function. The threshold of the neuron is always zero. The slope constant for the latter two function is always 1.

Architecture of ANN is defined by # of neurons and their interconnectivity

Layer – is a matrix of connections

Behavior of ANN is determined by values of connection weights

It is almost impossible to determine weights 'by hand' based on input and output that is required
2 types of learning of ANN:

1) **Supervised Learning**

   Weights are updated based on the error between desired output and the actual output for a given input

   a) **Reinforcement Learning**

   Based on global evaluation of the network response (not exact output)

2) **Unsupervised Learning**

   ANN updates weights based on input patterns only (used mainly for feature extraction, categorization)

Weights are modified either:

1) After every example (**online learning**)
2) After entire training data (**offline**)

Learning Algorithms: (concerned with calculation of $/_\backslash w_{ij}$)

1) **Hebbian Learning**

   Neurons that fire together, wire together
   After learning, ANN will give right output even if corrupted input is given
   Disadvantage: if neurons have only positive activations, hebbian learning can only strengthen the weights, but never decrease them -> in order to decrement weights, we can use output functions where neurons can have both negative and positive values
   a. Postsynaptic rule
   b. Presynaptic rule

2) **Supervised error based learning** (same as above – supervised learning) - can only be used when we know the output for a given input [not the case with autonomous robots]

   Modifies weights of ANN in order to reduce error between desired output and actual output
   Training pattern are presented several times in random order
   ‘Delta rule’ is applicable only to ANN with 1 layer of connections (**linearly separable**)
   Back-propagation-algorithm can be used to learn an arbitrary mapping between inputs and outputs
   Adding new layers to improve amount of patterns that can be learned is not necessarily a good solution

3) **Reinforcement Learning** (used when feedback from the environment is not as detailed [not 1 – good; 0 - bad])

   Attempts to **choose actions that maximize ’goodness’** over time
   Reinforcement learning tries to solve the same problems as Evolutionary Algorithms (in different ways)

   Core of the algorithm of reinforcement learning:
“Increase probability of producing actions that receive positive reinforcement and decrease producing actions that receive negative reinforcement”

4) **Learning in Recurrent Networks**
   Expand inputs
   Can be used to generate oscillatory behavior

5) Why ANN is a suitable structure for ER?
   ANN offer smooth search space (gradual changes to parameters [weights/structure] will result in gradual changes in behavior)
   ANN allow different levels of adaptation:
   a) Evolution (phylogenetic)
   b) Developmental (maturation)
   c) Ontogenetic (life learning)
   ANN provide direct mapping between sensors and motors
   ANNs are robust to noise (since output is a SUM of multiple weights) -> extreme values in some weights do not drastically affect the overall behavior (very useful for real robots with noisy sensors)
   It is smart to use crossover (much) <100% for ANN

3 ways to evolve ANN:

1) **Weights and learning parameters**
   GA finds much better ANN (and in many less computational cycles) than back-propagation (Montana&Davis, 1989)
   Back-propagation is very sensitive to initial weights

2) **Architecture**
   When fitness f(x) includes penalty for the number of connections, the best networks had very few connections
   When fitness f(x) includes penalty for the number of learning cycles, best networks learned very fast, but used many more connections
   Direct/Indirect encodings (i.e. encode as grammatical trees)

3) **Learning Rules**
   Fitness function
   GP – evolution of programs. Genotype does not encode a solution for a problem, but a program to solve that problem

**Chapter 3**