CURRICULUM

INF3490 - Fall 2013

Author:
JOAKIM MYRVOLL

User:
joakimmj

Due: 04.12.13

Contents

1 Man vs. Machine

Machines are good at:

- number crunching
- storing data
- specific tasks (e.g. control systems in manufacturing)

Humans are good at:

- sensing
- motion control (speaking, walking etc.)
- general thinking/reasoning

2 Turing Test (1956)

"A machine is intelligent when a human communicating with text is unable to distinguish the machine from a human"

Requirements:

- recognize and generate *natural language* to communicate as a human.
- store the information for *representing knowledge* it has received or are receiving.
- *reasoning* based on stored information and draw new conclusions.
- be able to learn to *adapt* to new circumstances and extract patterns.

Part I
Optimization And Search

3 Optimization

Continous Optimization is the mathematical disipline which is concerned with finding the maxima and minima of functions, possibly subject to constraints.
Examples:

Mechanics: optimized design of e.g. a new car or plane

Economics: computational finance

control engineering: process engineering, robotics etc.

Discrete Optimization = Search is the activity of looking thoroughly in order to find an item with specified properties among a collection of items.
Examples:

Chip design: routing tracks during chip layout design

Timetabling: find a course time table with the minimum clashes for registered students

Traveling Salesman Problem: optimize a travel route between a number of cities

3.1 Types of solutions

- A *solution* to an optimization problem specifies the values of the decision variables, and therefore also the value of the objective function.
- A *feasible solution* satisfies all constraints.
- A *near-original solution* is feasible and provides the best objective funtion value.

3.2 Gradient Descent

It is the optimization method that forms the basis of many machine learning algorithms. It is mainly used in *continous optimization problems*.
Gradient is defines as the vector of *partial derivatives*:

$$\Delta f(x) = \left(\frac{\delta f}{\delta X_1}, \frac{\delta f}{\delta X_2}, \ldots, \frac{\delta f}{\delta X_n}\right) \tag{1}$$

where

$$f(x) = (x_1, x_2, \ldots, x_n) \tag{2}$$

Gradient descent is a first-order optimization algorithm.

Local mimimum: take steps proportional to the **negative** of the *gradient* of the function at the current point routing tracks during chip layout design.
Known as: *steepest descent*

Local maximum: take steps proportional to the **positive** of the *gradient* of the function at the current point routing tracks during chip layout design.
Known as: *gradient ascent*

3.2.1 Algorithm

- Start with a point (guess)
- Loop until stopping criterion is satisfied
  - Determine a descent direction
  - Choose a step
  - Update current point

3.2.2 Problem

- **What if there is no gradient?**
  - Discrete problems.
  - Check all cases?
    - Guaranteed to find the global optimum but ...
    - Computations infeasible for any reasonable sized problem
- **Solution:** Search algorithms

3.3 Example: Traveling Salesman Problem (TSP)

Given the coordinates of *n* cities, find the **shortest closed tour** which visits each city **once and only once**.

Constraints:

- all cities must be visited once and only once.

Number of TSP tours

n	possible solutions (*n*-1)!
10	≈ 181000
20	$\approx 10^{16}$
50	$\approx 10^{62}$
101	$\approx 9 \cdot 10^{157}$

3.4 No Free Lunch

- There is no perfect search algorithm for every problem
- We always need to think about the problem
- We might have to design the encoding of the problem carefully

4 Search

- Three basic approaches (optimization without gradients)
  - Exhaustive search
  - Greedy search
  - Hill climbing

4.1 Exhaustive Search

- Also known as brute-force search or generate and test search
- Very general problem-solving technique
  - systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement.
- Trivial to implement
- For TSP, $O(N!)$ worse than $O(e^N)$

4.2 Greedy Search

- Follows the problem solving heuristic of making the locally optimal choice (the best local choice) at each stage with the hope of finding the global optimum.
- Greedy strategy applied to TSP: at each stage visit the neighbouring (unvisited) city that appears to be closest to goal (e.g. having the shortest straight-line distance to the goal).

4.2.1 Conlusion

- Positive:
  Computationally cheap: O(N).
  Will find a solution.
  Acceptable for simple problems.
- Negative:
  Inefficient for problems with large space.
  No guarantee of any kind of optimality.
  Cannot predict how good a solution is.

4.3 Hill Climbing

1. Starts with an arbitrary solution to a problem
2. Attempt to find a better solution by incrementally changing a single element of the solution.

- Local search around the current solution
- Hill Climbing in TSP: Start with a random initial solution and then swap pairs of cities in the tour to see if the total length of the tour decreases.

4.3.1 Disadvantages

Local Maximum: A state that is better than all of its neighbours, but not better than some other states far away.

Plateau: A flat area of the search space in which all neighbouring states have the same value.

Ridges: flat like a plateau, but with drop-offs to the sides; steps to the North, East, South and West may go down, but a step to the NW may go up.

4.4 Exploration and Exploitation

- Exploration
  - Try out new solutions.
  - Exhaustive search (global search)
- Exploitation
  - Try to improve your current best solution.
  - Hill climbing (local search).
- Methods need to contain both of these to find the global optima of the problem.

4.5 Simulated Annealing

- By allowing occasional ascent in the search process, we might be able to escape the trap of local minima.
- A variation of hill climbing in which, at the beginning of the process, some downhill moves may be made.
- Does exploration of the whole space early on, so that the final solution is relatively insensitive to the starting state.

- Lowering the chances of getting caught at a local maximum, or plateau, or a ridge.
- This is a stochastic algorithm (randomness). The outcome may be different at different trials.

4.5.1 Origin of Simulated Annealing

- Annealing in metallurgy.
- Annealing is a thermal process for obtaining low energy states of a solid in a heat bath.
- The process contains two steps:
  - Increase the temperature of the heat bath to a maximum value at which the solid melts.
  - Decrease carefully the temperature of the heat bath until the particles arrange themselves in the ground state of the solid. Ground state is a minimum energy state of the solid.
- The ground state of the solid is obtained only if the maximum temperature is high enough and the cooling is done slowly.

4.5.2 Analogy

- Metal $\leftrightarrow$ Problem
- Energy State $\leftrightarrow$ Cost Function
- Temperature $\leftrightarrow$ Control Parameter
- A completely ordered crystalline structure $\leftrightarrow$ The optimal solution for the problem.

5 Evolution

- Biological evolution:
  - Lifeforms adapt to a particular environment over successive generations.
  - Combinations of traits that are better adapted tend to increase representation in population.
  - Mechanisms: Selection+Crossover, Mutation and Survival of the fittest.
- Evolutionary Computing (EC):
  - Mimic the biological evolution to optimize solutions to a wide variety of complex problems.
  - In every new generation, a new set of solutions is created using bits and pieces of the fittest of the old.

			Representation	Strategy parameters	Parent Selection	Recombination
		GA	Any (bits)	No	Any	Repr.spesific
EC	EA	ES	$\mathbb{R}^n$	Yes	Random	Discrete/aritmetic
		EP	$\mathbb{R}^n$	Yes	1 parent $\rightarrow$ 1 offspring	No recombination
		GP	Trees	No	Any	Sub-node swap

			Mutation	Survival Selection	Typical λ (how many offspring compared to parents)
		GA	Repr.spesific	Any	$\leq \mu$
EC	EA	ES	Adaptive gaussian	Best Y	$\geq \mu$ (usually much greater)
		EP	Adaptive gaussian	Tournament	$= \mu$
		GP	Sub-node replacement	Any	Any

5.1 Evolutionary Algorithm (EA)

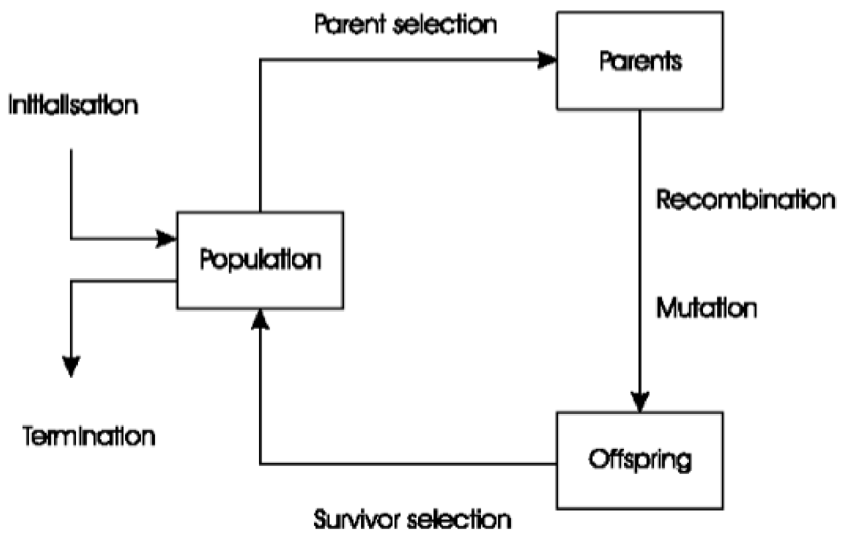


Figure 1: General Scheme of Evolutionary Algorithm

5.2 Evolutionary Operator

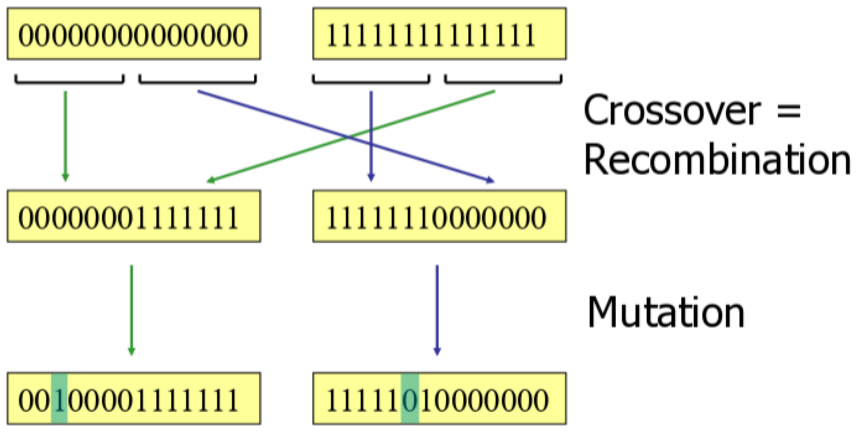


Figure 2: Cloning - Alternative to crossover where parents are copied to the offspring

5.3 Hill climbing in $\mathbb{R}^n$

- Hill climbing:
  - Randomly select one neighboring solution
  - Continuous space: need to define neighborhood

5.4 Random optimization

- The entire space is the neighborhood
    - Selection probabilities are normally distributed: Closer solutions are more likely to be selected

```
def random_opt():
  X = random_vector()
  while not_done():
    Y = X + normal(0,sigma)
    if (f(X) < f(Y)):
      X = Y
  return X
```

5.5 The (1+1) Evolution Strategy (ES)

When the current solution gets close to an optima the probability of a better solution being selected decreases.

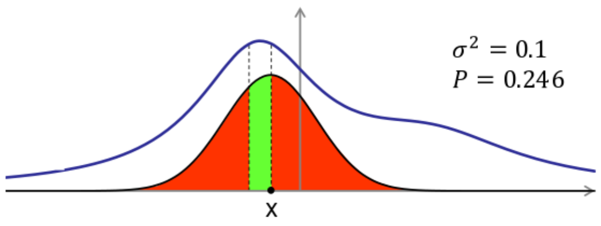


Figure 3:

- To compensate, we can reduce the spread of the distribution
- σ afftects our search strategy

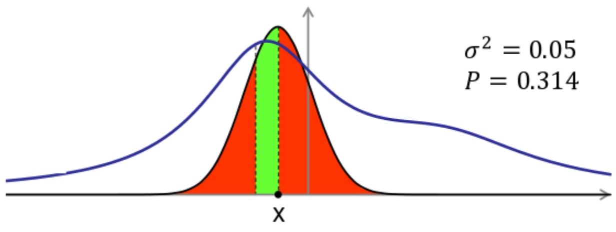


Figure 4:

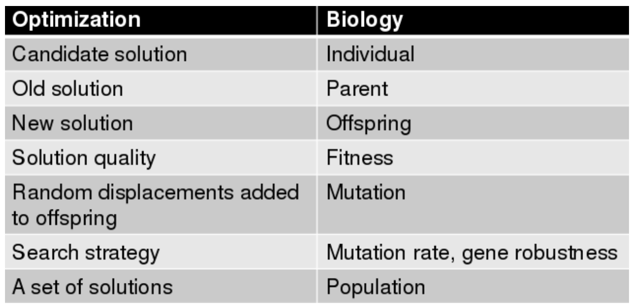
Add a strategy parameter σ to random optimization and you get the (1+1) ES:

```
def random_opt():
  X.x = random_vector()
  X.s = initial_sigma()
  while not_done():
    Y.s = X.s*exp(normal(0,tau))
    Y.x = X.x + normal(0,Y.s)
    if (f(X) < f(Y)):
    X = Y
  return X.x
```

5.5.1 Robustness

- The (1+1) ES is more efficient at finding accurate solutions, but it remains vulnerable to local optima
- Solution: Run multiple times?
  - Sometimes referred to as (1+1) reset
- Even better: Do multiple runs in parallel and make use of the extra information from having multiple solutions available at once

5.6 The evolution analogy



Optimization	Biology
Candidate solution	Individual
Old solution	Parent
New solution	Offspring
Solution quality	Fitness
Random displacements added to offspring	Mutation
Search strategy	Mutation rate, gene robustness
A set of solutions	Population

Figure 5:

5.7 Evolutionary algorithm outline

```
def evolve():
  P.x = initialize_population()
  P.fitness = evaluate(P.x)
  while not_done():
    Q.x = reproduce(P)
    Q.x = mutate(Q.x)
    Q.fitness = evaluate(Q.x)
    P = survival(P,Q)
  return best(P).x
```

- $initialize\_population()$
  - Generates a set of starting points
  - May be completely random solutions, or some hand-crafted selection
- $evaluate(P)$
  - Applies the objective function to all elements in *P*
  - Problem-dependent
- $reproduce(P)$
  - Creates a new population from *P*
- $mutate(X)$
  - Applies random changes to the individuals in *X*
- $survival(P, Q)$
  - Creates a new population from *P* and *Q*

5.8 Evolution Strategies (ES)

- Each individual is composed of n solution parameters and n_σ strategy parameters:

$$\langle x_1, \dots, x_n, \sigma_1, \dots, \sigma_{n_\sigma} \rangle \tag{3}$$

  - Usually n_σ is either 1 (all x_1 share one strategy) or n (each x_i have a separate search strategy)
  - Sometimes an additional set of parameters α_i is used to model correlations between strategies
- Recombination creates λ offspring
- Each one draws two parents at random and recombines them using intermediary or discrete recombination
- It is common to mix, i.e. use discrete for x_i and intermediate for σ_i

```
def reproduce(P):
  Q = []
  for i in range(1,lambda):
    parents = draw(2,P)
    offspring = recombine(parents[0], parents[1])
    Q.append(offspring)
  return Q
```

- Two survivor selection methods:
  - (μ, λ): from the offspring only.
  - $(\mu + \lambda)$: from both parents and offpsring.
- (μ, λ) is often preferred, for several reasons:
  - Better able to escape local optima
  - Able to adapt to changing fitness functions
  - Since solutions aren't evaluated for how good the strategy parameters are, bad strategies can linger in the population if parents can survive indefinitely

5.9 Discrete Recombination

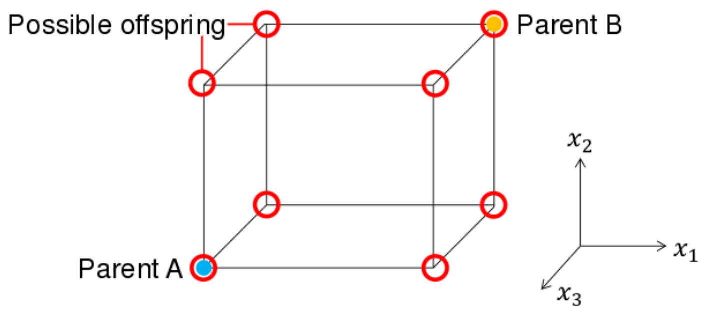


Figure 6: Each parameter is chosen from one of the parents at random.

5.10 Intermediary Recombination

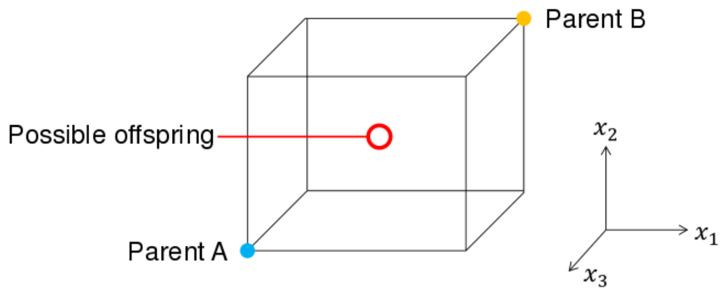


Figure 7: Each parameter is chosen as the average of from the parents

5.11 Evolutionary Programming (EP)

- Historically, evolutionary programming was mainly concerned with prediction problems
- More recently the field has diversified a lot, and is used for all kinds of different problems and with many different representations and mutation schemes
- Here we will focus on a variant for continuous optimization

5.11.1 Recombination

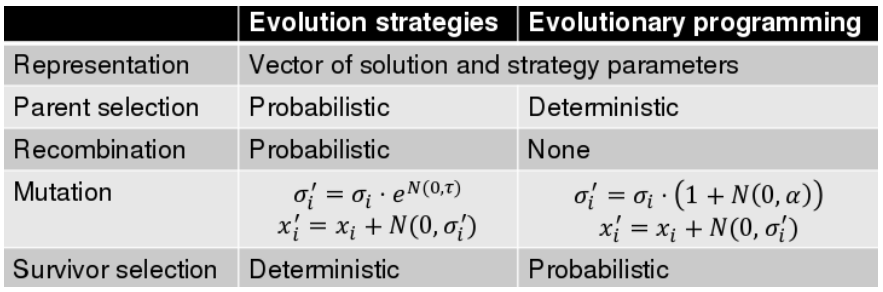
- In EP each solution is seen as a species instead of an individual
  - Recombination does not make sense!
  - Each solution gives rise to exactly one new solution each generation

5.11.2 Survivor Selection

- Survivor selection is done by tournaments
  - Each solution is compared to q other randomly selected solutions (q is typically about 10)
  - The best half, ranked by the number of "wins" survives

```
def survival(P,Q):
  PQ = [P,Q]
  for i in range(1, 2*mu):
    vs = draw(q, PQ)
    score = sum( PQ[i].fitness > vs.fitness )
  return best(mu, PQ, score)
```

5.12 Evolution strategies vs. Evolutionary programming



	Evolution strategies	Evolutionary programming
Representation	Vector of solution and strategy parameters	
Parent selection	Probabilistic	Deterministic
Recombination	Probabilistic	None
Mutation	$\sigma_i' = \sigma_i \cdot e^{N(0,\tau)}$ $x_i' = x_i + N(0, \sigma_i')$	$\sigma_i' = \sigma_i \cdot (1 + N(0,\alpha))$ $x_i' = x_i + N(0, \sigma_i')$
Survivor selection	Deterministic	Probabilistic

Figure 8:

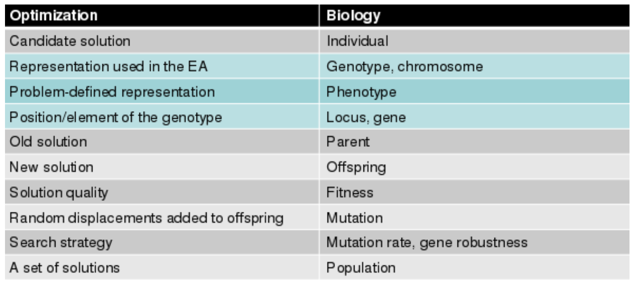
6 Representation

- Many optimization problems don't really have a continuous search space, or even a fixed dimensionality
  - Combinatorial problems (travelling salesman etc.)
  - Scheduling
  - Path planning
  - Evolvable hardware
- The central concepts in evolutionary algorithms are independent of representation
- Mutation and recombination must be tailored to the representation used

6.1 Indirect Representation

- Most problems will have a fixed solution representation associated with it
- However, sometimes it is beneficial to evolve solutions using a different representation and then transform them to do the evaluation

6.2 Expanding the evolution analogy



Optimization	Biology
Candidate solution	Individual
Representation used in the EA	Genotype, chromosome
Problem-defined representation	Phenotype
Position/element of the genotype	Locus, gene
Old solution	Parent
New solution	Offspring
Solution quality	Fitness
Random displacements added to offspring	Mutation
Search strategy	Mutation rate, gene robustness
A set of solutions	Population

Figure 9:

6.3 Binary Representation

- The representation used in the simple genetic algorithm (SGA)
  - Directly inspired by low level encoding in DNA
  - Uses a binary (0,1) coding instead of the quaternary (G,T,A,C) coding used in nature
  - Mutation and crossover (recombination) operations taken directly from biology

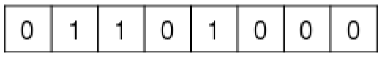


Figure 10:

6.3.1 Bit Flip Mutation

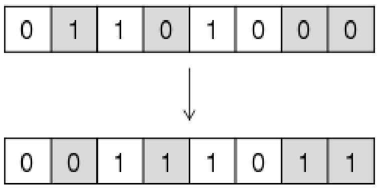


Figure 11: Each bit is inverted with a probability p_m

6.3.2 N-point Crossover

- *N* random points in the genotype is chosen
- At each point the source parent change

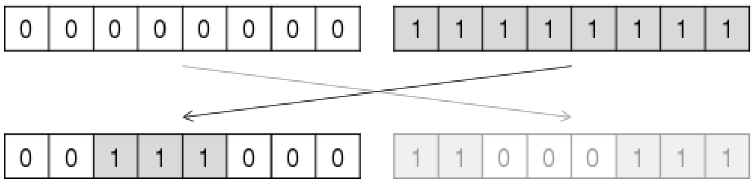


Figure 12:

6.3.3 Uniform Crossover

- Which parent to inherit from is chosen randomly for each position
- Identical to discrete recombination

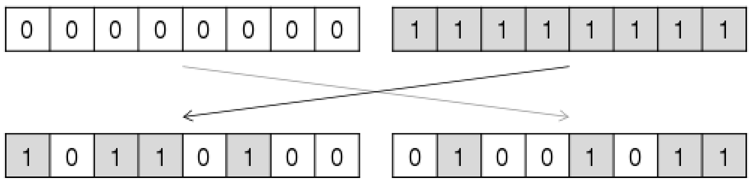


Figure 13:

6.3.4 Application - Knapsack Problem

The knapsack problem (rucksack problem) is a problem in combinatorial optimization: Given a set of items, each with a mass and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.

6.3.4.1 0-1 Knapsack problem

- Maximize gain given some budget and a set of binary decisions
- Can be used to decide on strategies, for example for advertising campaigns or investments

6.4 Binary coding of integers

- Encoding integers as blocks of a binary string has been quite common
  - Keeps the analogy to DNA clean
  - Problematic because mutations are not local
    - Small changes to the solution are not more probable
    - The result of flipping a single bit varies enormously with bit position and the value of all bits that encode the same integer

6.5 Integer Representation

- Each element is directly coded as an integer
  - Usually restricted to some pre-defined ranges
- Can use the same crossover operators as the binary representation

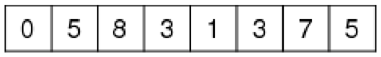


Figure 14:

6.5.1 Random Reset Mutation

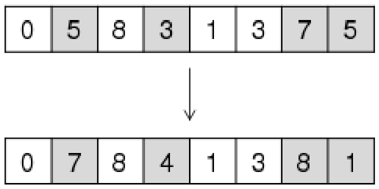


Figure 15: Each element is reset with probability p_m to a random number in the range

6.5.2 Creep Mutation

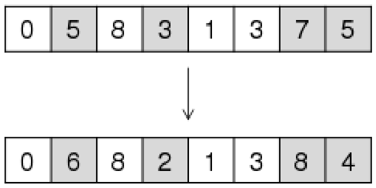
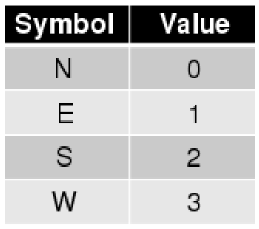


Figure 16: Adds a small value to each element with probability p_m

6.6 Integer coding of symbols

- Sometimes a vector of symbols with no clear order is the most reasonable representation choice
- In such cases, the symbols are usually enumerated and treated as integers, but without using the creep mutation



Symbol	Value
N	0
E	1
S	2
W	3

Figure 17:

6.7 Real-valued Representations

- Represents continuous solution spaces
- The solution parameters are often accompanied by strategy parameters for adaptive normal distribution-based mutation

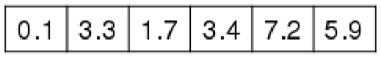


Figure 18:

6.7.1 Uniform Mutation

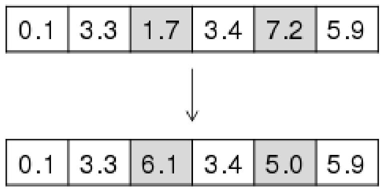


Figure 19: Each element has a probability p_m of being replaced with a number from some range

6.7.2 Arithmetic Recombination

- Makes a copy of one of the parents x and y
- Picks one or more random positions k and replaces those elements with the interpolation $\alpha x_k + (1-\alpha)y_k$, where α is either a fixed number or a random variable.
- Intermediate recombination: α is 0.5 for all k

Single Arithmetic Recombination

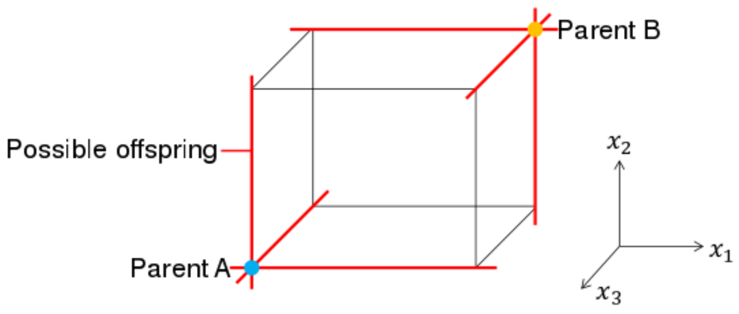


Figure 20: Arithmetic recombination is applied to only one k

Whole Arithmetic Recombination

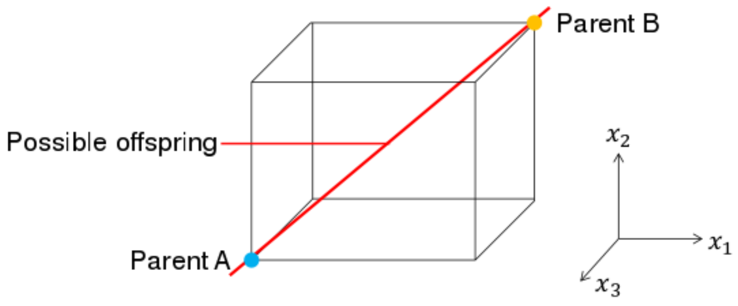


Figure 21: Arithmetic recombination is applied with the same α to all k

6.8 Permutation Representations

- Used to solve problems like the travelling salesman
  - Known set of actions (go to town X)
  - Want to optimize their sequence
- Special mutation/recombination operators
  - Each item should appear once and only once
  - Result should be "close" to the original solution(s)

6.9 Insert Mutation

- Two random elements are picked
- The second is placed right after the first

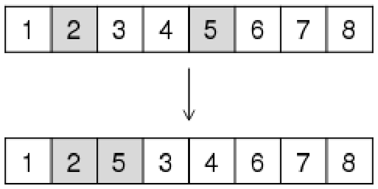


Figure 22:

6.10 Swap Mutation

- Two random elements are swapped
- In some variants neighbors are always chosen

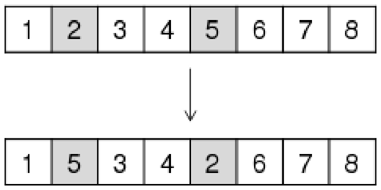


Figure 23:

6.11 Scramble & Invert Mutation

- Two random points are selected
- The order of the elements in between is scrambled (scramble mutation) or reversed (invert mutation)

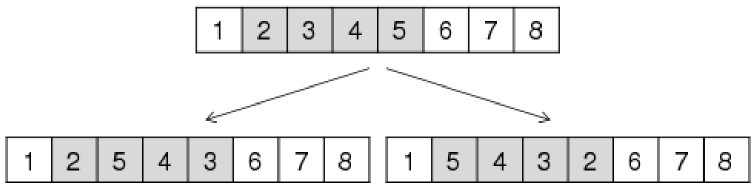


Figure 24:

6.12 Partially Mapped Crossover (PMX)

- Two random points are chosen
- All elements between the points in parent A are copied to the offspring

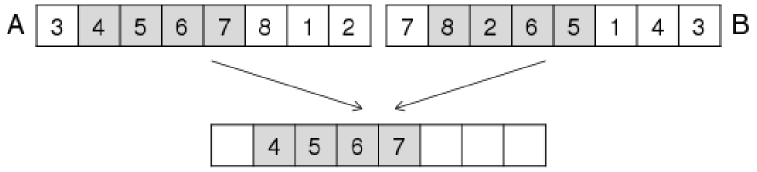


Figure 25:

- For each element x in parent B between those points that is not in parent A
  - Place it in the position in B of the element with the same position in A as x has in B

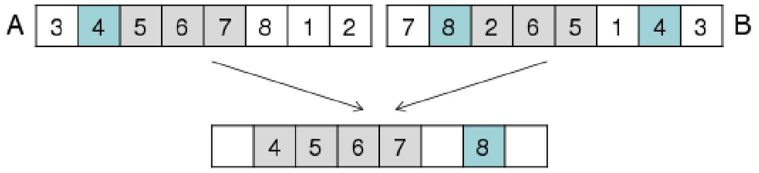


Figure 26:

- For each element x in parent B between those points that is not in parent A
  - Place it in the position in B of the element with the same position in A as x has in B
  - If that position is occupied, do one more redirection

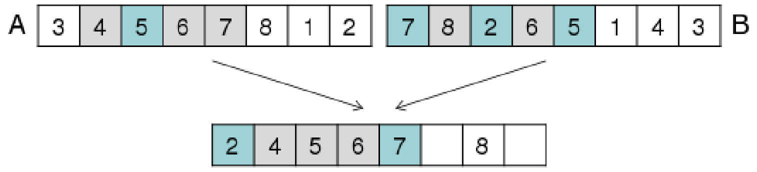


Figure 27:

- Finally, the missing elements are copied from their places in parent B

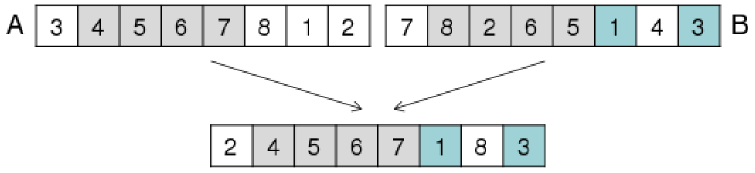


Figure 28:

6.13 Edge Crossover

- Heuristic to preserve as many edges as possible

```
def edge_xo(PA, PB, N):
  e = construct_edge_table()
  k = random(N)
  for I in range(1, N):
    X.append(k)
    e.remove(k)
    if e.empty(k): k=reverse(X)[-1]
    if e.empty(k): k = draw(1,e.remaining())
    else:
      k = e.pick_common(k) or draw(1, e.pick_shortest(k))
  return X
```

6.14 Order Crossover

- Two random points are chosen
- All elements between the points in parent A are copied to the offspring

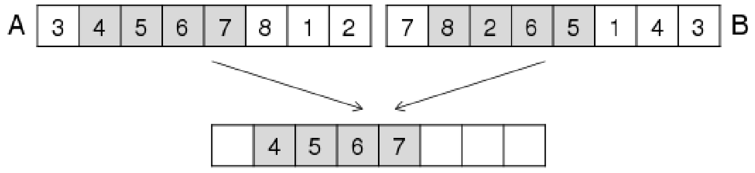


Figure 29:

- The rest of the elements are copied from parent B in the order starting from the second random point

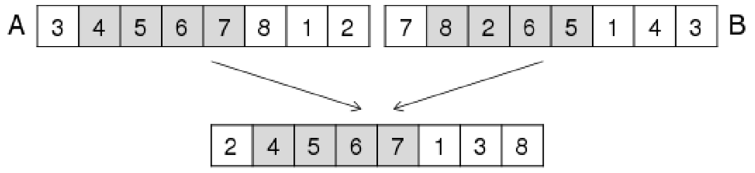


Figure 30:

6.15 Cycle Crossover

- Identify first cycle
- Copy from parent A and B to offspring A and B

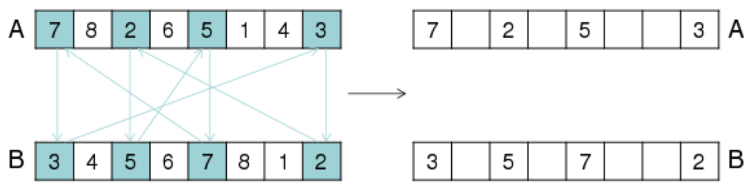


Figure 31:

- Identify next cycle
- Copy from parent A and B to offspring B and A

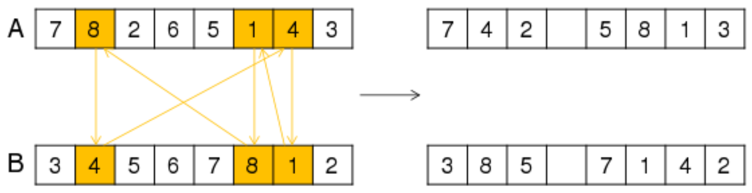


Figure 32:

- Identify last cycle
- Copy from parent A and B to offspring A and B

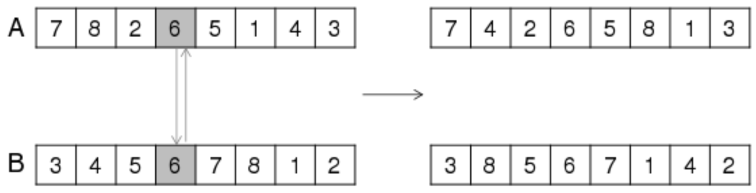


Figure 33:

6.16 Application - Transportation and logistics

- Arranging school bus routes
- Survey planning (geological surveys etc.)
- CNC machine / 3D printer programming

6.17 Tree Representation

- ree representations are used to represent are mainly used in genetic programming
  - Programs, mathematical or logical expressions

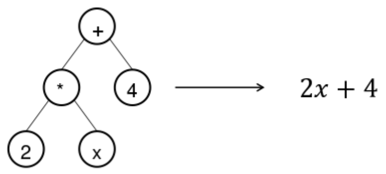


Figure 34:

6.18 Tree Mutation

- Take a random node and replace it by a new randomly generated subtree

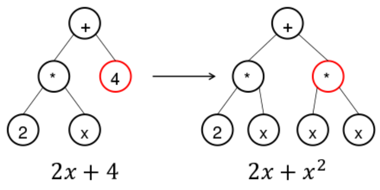


Figure 35:

6.19 Tree Crossover

- Take one random node from each parent and exchange them

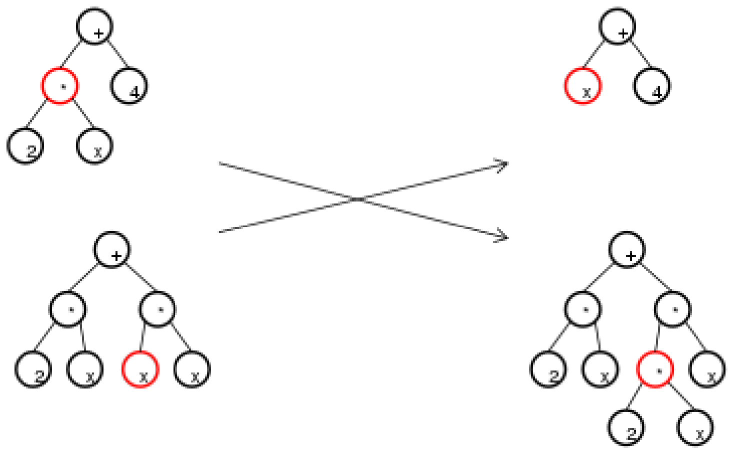


Figure 36:

6.20 Bloat In Tree Representation

- Larger trees will have greater fitness on average in most cases
- Without any active countermeasures the population will tend to grow indefinitely

6.21 Application

- Automatic program design and optimization
  - Everything from high-level languages (Lisp) to machine code can be evolved
- Architecture and industrial design
  - Many physical objects can be described as trees or some other kind of graph
- Decision trees
  - Generate simple programs to be used for classification of data

7 Selection

- Parent and survival selection operates independently of the representation
  - Can mix and match with ease
  - Most parent selection algorithms can also be used for survival selection by selecting μ times

7.1 Parent Selection - Fitness Proportional Selection

The probability of selecting an individual is proportional to its fitness

$$p_i = \frac{f_i}{\Sigma_{j=1}^{\mu} f_j} \tag{4}$$

7.1.1 Selection Pressure

- Proportional to what?
  - Selection pressure changes by adding different constant offsets to all fitneses

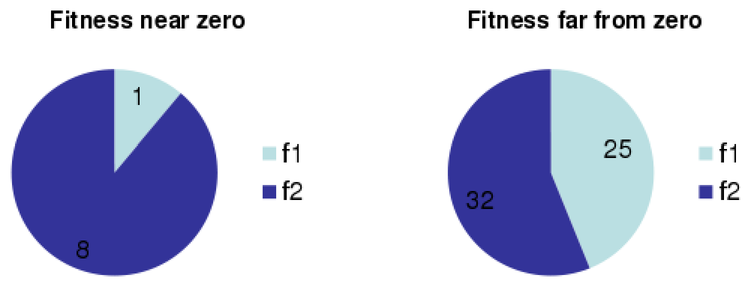


Figure 37:

7.2 Parent Selection - Ranking Selection

The probability of selecting an individual is proportional to its rank

$$p_i = \frac{2-s}{\mu} + \frac{2i(s-1)}{\mu(\mu-1)}, \qquad s \in (1,2] \tag{5}$$

7.3 Parent Selection - Tournament Selection

- As in evolutionary programming (EP)
  - For each parent needed, hold draw k contestants and pick the best one
  - Does not require any global information about the population
  - Gives results similar to ranking selection

7.4 Survivor Selection - Age-based Replacement

- Few parents compared to offspring ($\lambda \leq \mu$)
- Enough parents die for all offspring to survive
  - Dying parents selected either randomly or by age

7.5 Survivor Selection - Fitness-based Replacement

- As in evolution strategies
  - When ($\lambda \leq \mu$): replace worst
- Elitism:
  - The very best individuals can survive indefinitely
  - Either a fixed number of elites are kept, or the number is unbounded (e.g. ($\mu + \lambda$))

8 Multi-modal problems

Fitness functions usually have multiple local optima, or modes

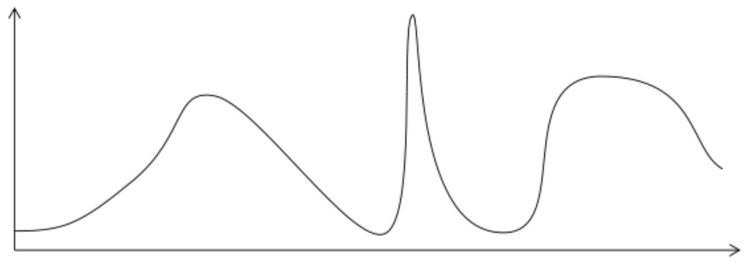


Figure 38:

Each of these modes will have a basin of attraction, an area around it where a local search would most likely lead to that mode.

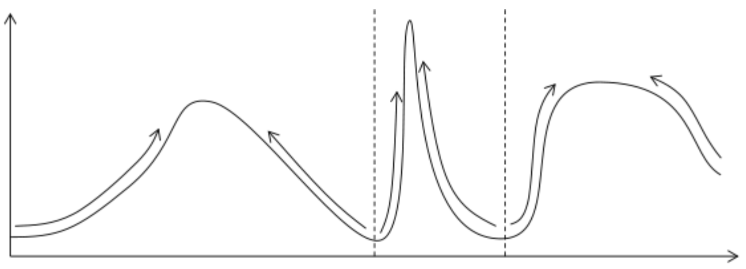


Figure 39:

If we're not careful with initialization, we might not get individuals in every basin of attraction

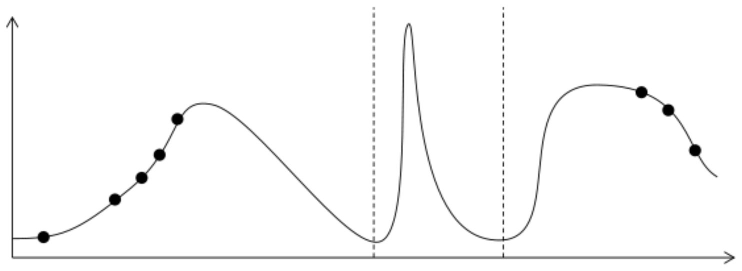


Figure 40:

Even when we have initialized really well there are many scenarios that can reduce the survival chances of the "right" individuals

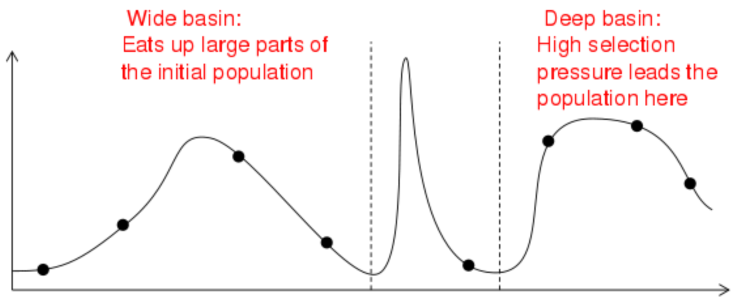

Wide basin:
Eats up large parts of the initial population
Deep basin:
High selection pressure leads the population here

Figure 41:

9 Diversity

- Maintain individuals in as many optima as possible
- Increases the chances of getting and keeping solutions in the basin of the global optima

9.1 The Island Model

- Divide the population into separate "islands"
- Allow only limited migration between islands (e.g. a couple of individuals every 10th generation)

9.2 Diffusion Model EAs

- Subpopulations have limited neighborhoods
  - Grids, rings, etc.
- Parent and survivor selection limited to the neighborhood

9.3 Fitness Sharing

- Decrease the fitness of individuals with neighbors closer than σ_{share}
  - Works best with fitness proportional selection
  - Need distance measure $d_{i,j}$

$$F_i' = \frac{F_i}{\Sigma_j sh(d_{i,j})} \leq F_i \tag{6}$$

$$sh(d) = \begin{cases} 1 - (\frac{d}{\sigma_{share}})^\alpha & d < \sigma_{share} \\ 0 & \text{else} \end{cases} \tag{7}$$

9.4 Crowding

- Two parents create a pair of offspring
- Each offspring competes with their nearest parent for survival
  - Need distance measure

9.5 Speciation

- Define subpopulations dynamically
  - Distance-based clustering
  - Genotype compatibility tags
- Restrict mating to within the subpopulations
- Subpopulation fitness sharing

10 Multi-objective Optimization

- Conflicting considerations
  - Quality
  - Speed
  - Cost
  - etc.

There is no longer only one optimal solution!

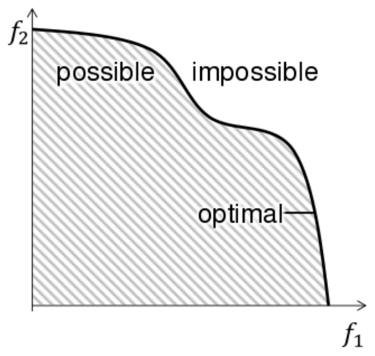


Figure 42:

10.1 Scalarization

- Use arithmetic to reduce to a single objective
- Objective priorities must be known

$$F = f_1 + af_2 + bf_3 \tag{8}$$

$$F = f_1 + f_2 \cdot f_3 \tag{9}$$

$$F = e^{f_1} + tanh f_2 + N(0, f_3) \tag{10}$$

10.1.1 Weighted Sum Scalarization

- Most common scalarization
- Weight w_i is chosen based on the importance of objective i

$$F = \sum_{i=1}^{M} w_i f_i \tag{11}$$

The weights define a gradient in objective space

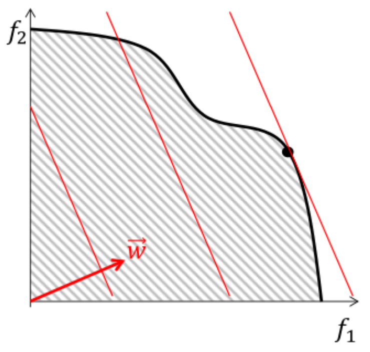


Figure 43:

10.1.2 "Artificial" Local Optima

- Reducing to a single objective function can create "artificial" local optima

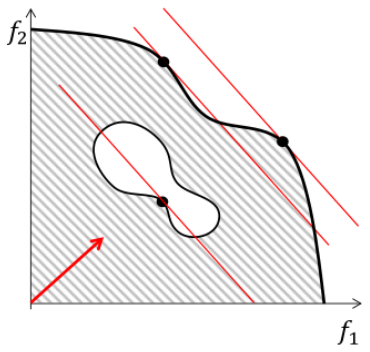
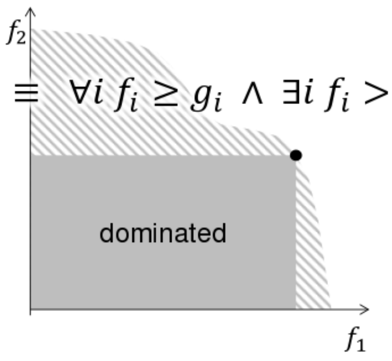


Figure 44:

10.2 Pareto Dominance

A solution dominates another if it is as good in every way and better in at least one



$$f \succcurlyeq g \;\equiv\; \forall i\; f_i \geq g_i \;\wedge\; \exists i\; f_i > g_i$$

Figure 45:

Undominated solutions are *Pareto optimal*

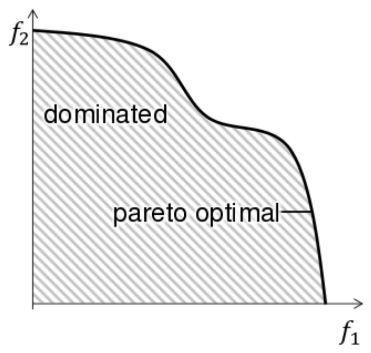


Figure 46:

10.2.1 Pareto dominance-based EAs

- Only selection operators are affected
- Pareto dominance replaces scalar comparison
  - Usually a secondary diversity measure is used for mutually non-dominated solutions:

```
def better_mo(a,b):
  if dominates(a,b): return true
  if dominates(b,a): return false
  return diversity(a) > diversity(b)

```

  - Tournament selection
  - Selection proportional to the number of solutions that are dominated in the population

10.3 Non-dominated Sorting

```
def nondominated_sort(P):
  Q = P
  P = []
  rank = 1
  while not Q.empty():
    F = Q.nondominated()
    Q.remove(F)
    F.assign_rank(rank)
    P.insert(F)
    rank = rank+1
  return P
```

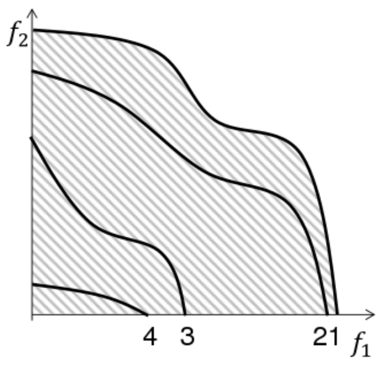


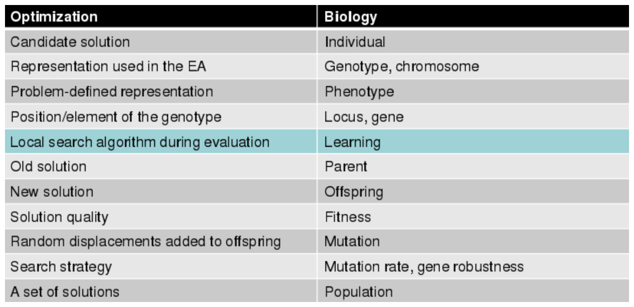
Figure 47: Sorts the population into layers by domination

10.4 Multi-objective Evolutionary Algorithms (MOEA)

- Most modern multi-objective evolutionary algorithms use some form of elitism
  - $(\mu + \lambda)$
  - Separate population (archive)

11 Hybrid EAs (aka. Memetic algorithms)

- In many cases, decent heuristics or local search algorithms exist
- These algorithms can often be integrated into an evolutionary algorithm either to speed up the process or improve solution quality



Optimization	Biology
Candidate solution	Individual
Representation used in the EA	Genotype, chromosome
Problem-defined representation	Phenotype
Position/element of the genotype	Locus, gene
Local search algorithm during evaluation	Learning
Old solution	Parent
New solution	Offspring
Solution quality	Fitness
Random displacements added to offspring	Mutation
Search strategy	Mutation rate, gene robustness
A set of solutions	Population

Figure 48: Evolution analogy

- Memetic algorithms is used as a collective term for algorithms making use of local search and problem-specific information
  - Meme: An evolving cultural entity/transmission
- Can be added at many stages
  - Clever initialization
  - Local search during evaluation
  - Problem-specific heuristics in mutation, crossover and selection

11.1 In Initialization

- Seeding
  - Known good solutions are added
- Selective initialization
  - Generate kN solutions, keep best N
- Refined start
  - Perform local search on initial population

11.2 Intelligent mutation and crossover

- Mutation bias
  - Mutation operator has bias towards certain changes
- Crossover hill-climber
  - Test all 1-point crossover results, choose best
- "Repair" mutation
  - Use heuristic to make infeasible solution feasible

11.3 Learning and evolution

- Do offspring inherit what their parents have learnt in life?
  - Yes $\rightarrow$ Lamarckian learning
    - Improved fitness and genotype
  - No $\rightarrow$ Baldwinian learning
    - Improved fitness only

12 Design

12.1 What do you want your EA to do?

12.1.1 Design problems

- Only need to be done once
- End result must be excellent
- Example:
  - Optimizing spending on improvements to national road network
    - Total cost: billions of Euro
    - Computing costs negligible
    - Six months on hundreds of computers
    - Many runs possible
    - Must produce very good result just once

12.1.2 Repetitive problems

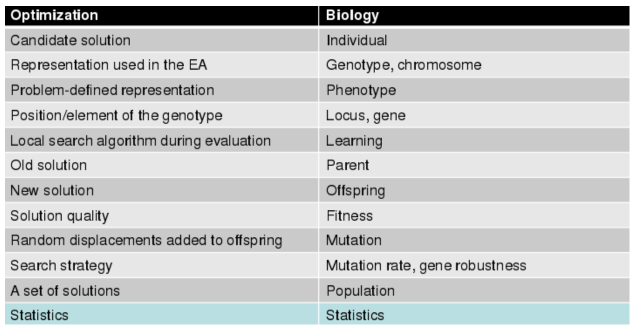
- Has to be run repeatedly
- Should produce OK results quickly and reliably
- Example:
  - Optimizing postal delivery routes
    - Different destinations each day
    - Limited time to run algorithm each day
    - Must always perform reasonably well in limited time

12.1.3 Research and development

- Must produce repeatable, unbiased results
- Academic publishing/engineering decisions
- Topics:
  - Show that an EA is applicable in some problem domain
  - Show that (your fancy new) EA outperforms benchmark EA/some traditional algorithm
  - Optimize or study impact of some parameters for an EA
  - Investigate algorithm behavior or performance

13 Statistics

- Evolutionary algorithms are stochastic
  - Result will vary from run to run
  - Many runs are needed to say anything concrete about the performance of the EA
  - Use statistics and statistical measures!
- Do proper science!
  - Same measures
  - Fair comparisons



Optimization	Biology
Candidate solution	Individual
Representation used in the EA	Genotype, chromosome
Problem-defined representation	Phenotype
Position/element of the genotype	Locus, gene
Local search algorithm during evaluation	Learning
Old solution	Parent
New solution	Offspring
Solution quality	Fitness
Random displacements added to offspring	Mutation
Search strategy	Mutation rate, gene robustness
A set of solutions	Population
Statistics	Statistics

Figure 49:

13.1 Things to measure

- Average result in given time
- Average time for given result
- Best result over N runs
- Proportion of X or amount of Y required to do Z under conditions W
- Etc.

13.2 Off-line performance measures

- Efficiency (speed)
  - Time
  - Average number of evaluations to solution (AES)
- Effectiveness (quality)
  - Successrate (SR)
  - Mean best fitness at termination (MBF)
    - Mean across runs, best of each run

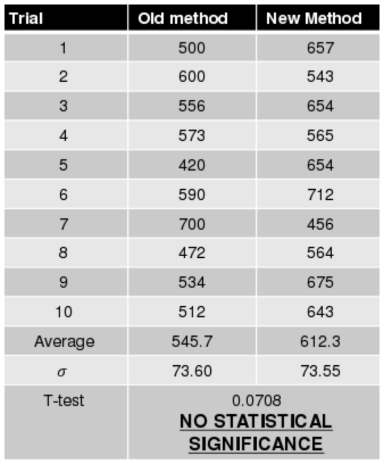
13.3 On-line performance measures

- Population distribution (genotypic)
- Fitness distribution (phenotypic)
- Improvements per time unit or per genetic operator

13.4 What time units do we use?

- Elapsed time?
  - Depends on computer, network, etc.
- CPU time?
  - Depends on skill of programmer, implementation, etc.
- Generations?
  - Incomparable when parameters like population size change
- Evaluations?
  - Evaluation time could be small compared to "overhead":
    - (Hybrid) selection, mutation and crossover
    - Genotype-phenotype translation

13.5 Note



Trial	Old method	New Method
1	500	657
2	600	543
3	556	654
4	573	565
5	420	654
6	590	712
7	700	456
8	472	564
9	534	675
10	512	643
Average	545.7	612.3
σ	73.60	73.55
T-test	0.0708 **NO STATISTICAL SIGNIFICANCE**	

Figure 50: Statistics

- Don't trust the averages
  - Extra important when the sample is small
- Always check the deviation
  - For normal distributions, use standard deviation
  - Interquartile range (1st quartile-3rd quartile) is also good
  - Range (max-min) is better than nothing
- T-tests, use them
  - Alternatively, the simpler Z-test is approximately equivalent for good sample sizes $(N \geq 100)$
  - Different tests also exist
    - Wilcoxon
    - F-test

14 Swarm Intelligence

14.1 Key Features

- Simple local rules
- Local interaction
- Decentralized control
- Complex global behavior
  - Difficult to predict from observing the local rules
  - Emergent behavior

14.2 Flocking model - "boids"

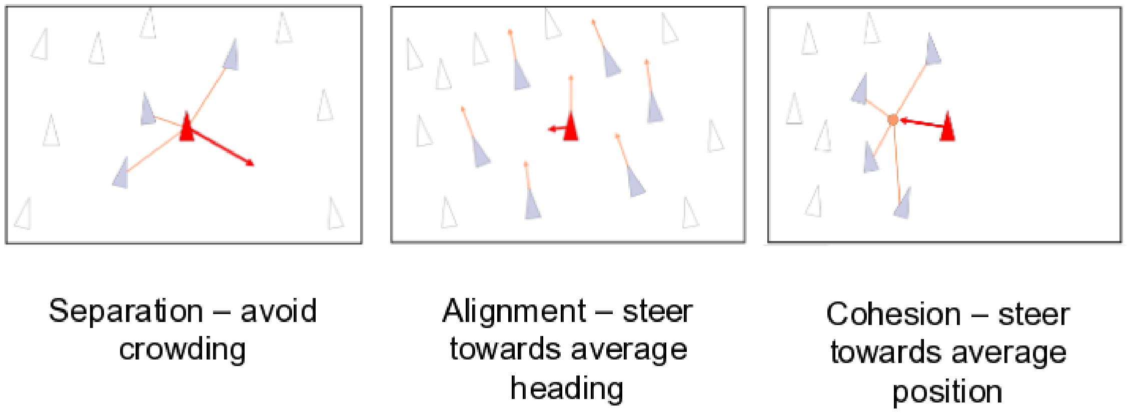

Separation – avoid crowding
Alignment – steer towards average heading
Cohesion – steer towards average position

Figure 51: Only considering the boid's neighborhood

14.3 Applications in bio-inspired computing

- Particle swarm optimization
  - Parameter optimization
- Ant colony optimization
  - Find shortest paths through graph by using artificial pheromones
- Artificial immune systems
  - Classification, anomaly detection
- Swarm robotics
  - Achieve complex behavior in robotic swarms through simple local rules

15 Particle Swarm Optimization (PSO)

- Optimizes a population of solutions
  - A *swarm of particles*

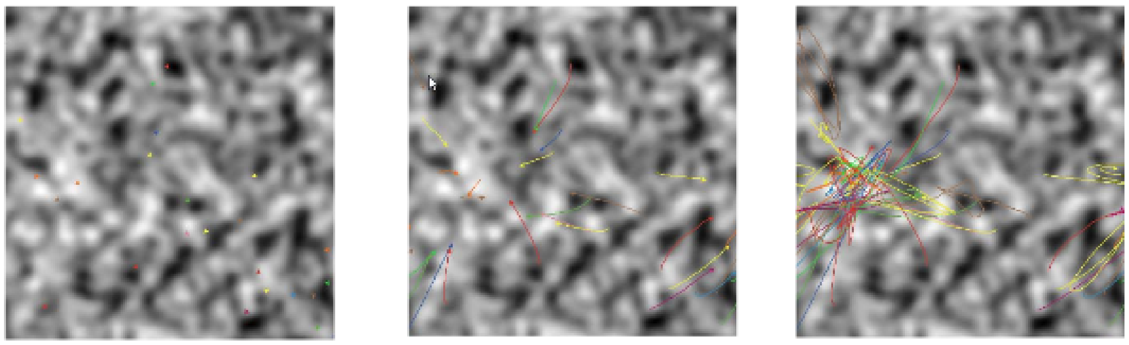


Figure 52:

15.1 Principle

- Evaluate your present position
- Compare it to your previous best and neighborhood best
- Imitate self and others

15.2 Simplified PSO algorithm

- For each particle i in the swarm
  - Calculate fitness
  - Update local best
  - Find neighborhood best
  - Update velocity
  - Update position

15.3 PSO update formulas

For each dimension d in particle i:

1. Velocity update

$$\begin{aligned} v_{id}^{(t+1)} &\leftarrow \text{inertia + random} \cdot \text{direction personal best + random} \cdot \text{direction neighbourhood best} \\ &\leftarrow \alpha v_{id}^{(t)} + U(0,\beta)(p_{id} - x_{id}^{(t)}) + U(0,\beta)(p_{gd} - x_{id}^{(t)}) \end{aligned} \tag{12}$$

2. Position update

$$x_{id}^{(t+1)} \leftarrow x_{id}^{(t)} + v_{id}^{(t+1)} \tag{13}$$

15.4 What happens?

- A particle circles around in a region centered between the best of itself and its neighbors
- The best are updated and the particles cluster around better regions in the search space
- The way good solutions are propagated, depends on how we define the neighborhood

15.5 Neighborhood Topologies

- *gbest*: all particles are connected
  - Every particle gets information about the global best value
  - Can converge (too) fast
- *lbest*: connected to K nearest neighbors in a wrapped population array
  - lower convergence, depending on K
  - More areas are searched in parallel
- Several other topologies exist

15.6 PSO Parameters

- Particle:
  - Usually a D-dimensional vector of real values
  - Binary variant exists
- Swarm size: usually 10 < N < 100
- Recommended α = 0.7298
- Recommended β = 1.4961

15.7 Parameter Experimentation

- NetLogo
  - Particle Swarm optimization model
- Model uses *gbest* neighborhood

15.8 Advantages of PSO

- Few parameters
- Gradient free
- Decentralized control (depends on variant.)
- Simple to understand basic principle
- Simple to implement

15.9 PSO vs. Evolutionary Algorithms

- Both are population based
- PSO: No selection - all particles survive
- Information exchange betwwen solutions:
  - PSO: neighbourhood best
  - GA: crossover (and selection)

15.10 Applications

- Similiar application areas as EAs
  - Most optimization problems
- Image and video analysis
- Electricity netwoork optimization
- Neural networks
- ...

16 Swarm Robotics

- Swarmbot project
- Kilobot project
- TERMES project
  - Termite-inspired swarm assembly robots

17 Evolvable Hardware (EHW)

- Hardware systems designed/modified automatically by EAs
- A string of symbols/bits is evolved by an EA and translated into a HW system
- Offline EHW
  - Solutions are simulated in a PC
- Online EHW
  - Solutions are tested on target HW
- FPGA
  - Reconfiurable hardware chip
  - Useful for online EHW
- On-chip evolution
  - EA running on the target chip, together with solutions
- Run-time adaptable EHW
  - Evolution can modify the system during operation

17.1 Applications

- Pattern recognition / classification circuits
- Digital image filters
- Evolution of analog circuits
- Cache mapping functions
- On-the-fly compression for printers
- Spacecraft antenna

18 Cartesian Genetic Programming (CGP)

- A type of Genetic Programming
- Allows restrictions compared to general GP:
  - Integer genome
  - Tree nodes are mapped to a grid
  - Connectivity can be restricted
- Popular in Evolvable Hardware applications
  - But can be used for many other things as well

18.1 Example structure: Digital circuit

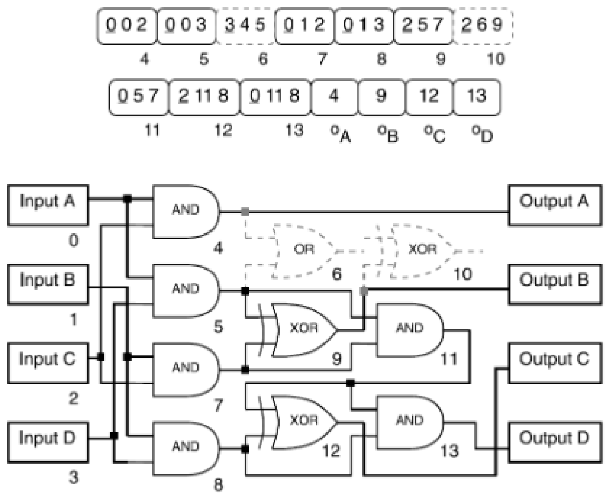


Figure 53:

18.2 CGP genome

- Internal node genes:
  - Node type: index to lookup table of functions
  - Inputs: index of other nodes
  - Optional: additional parameters
- Output node gene:
  - Internal node index

18.3 CGP parameters

- Columns: n_c
- Rows: n_l

- Levels-back: l
  - How many of the previous columns a node can connect to
- Columns x rows defines the maximum number of nodes in the graph

18.4 General structure

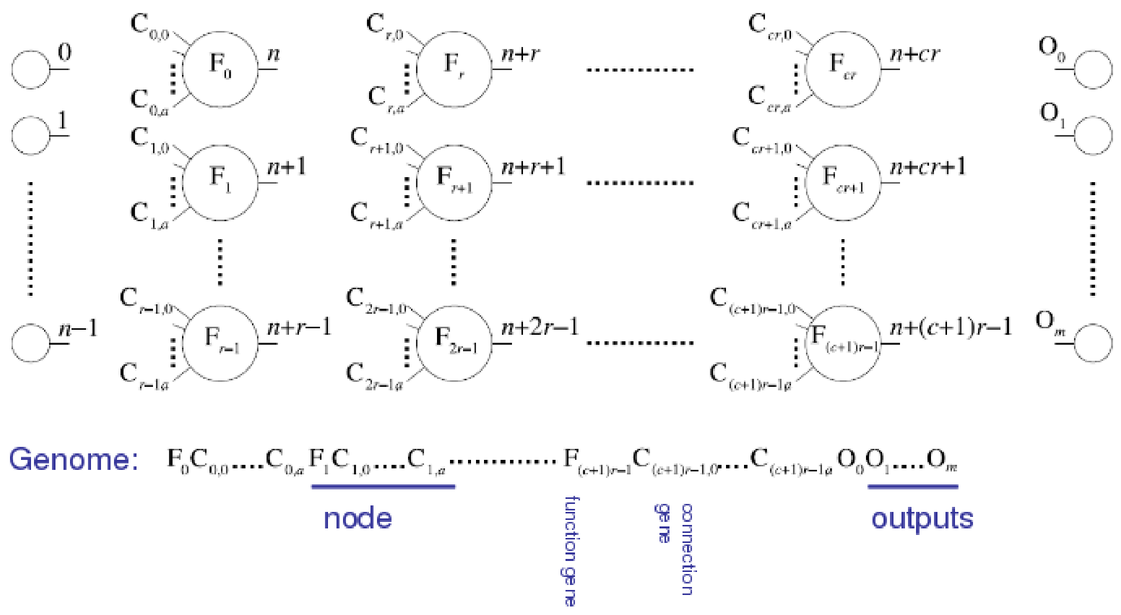


Figure 54:

18.5 Advantages of CGP

- Easy implementation
  - Fixed genome size and simple representation
  - Simple mutation and crossover
- Bloat is restricted
  - The number of nodes is restricted
- Regular structure suitable for e.g. hardware implementation
  - A grid structure with limited connectivity ideal for HW routing

18.6 Other features of CGP

- Reuse of parts of the tree is possible
- Allows multiple outputs
- Parts of the genome may be non-coding

- This has an analogy in biology, where only a fraction of the DNA is composed of exons ("coding" genes).
- The other part is called introns (non-coding genes, sometimes called "junk" DNA). It is however believed that these are useful for something.
- Likewise, the genetic redundancy (neutrality) in CGP is thought to be positive for the evolutionary search

18.7 Genetic Operations in CGP

- Mutation
  - Select randomly a number of genes to mutate
  - Change to new (valid) random values
- Crossover
  - One-point crossover or other variants directly on the genome
- Usually only mutations are used
  - Many applications find crossover to have a destructive effect - it disrupts the tree structure too much

18.8 Evolution in CGP

- The most popular is a variant of ES called (1+4)ES
- Choose children which have $\geq$ fitness than parent

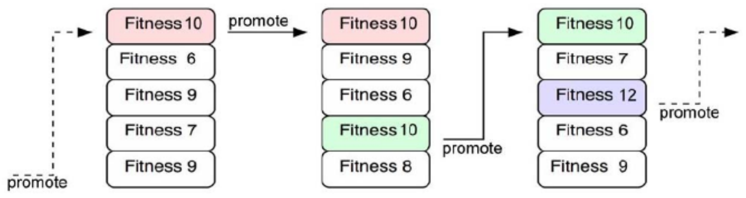


Figure 55:

18.9 CGP can code:

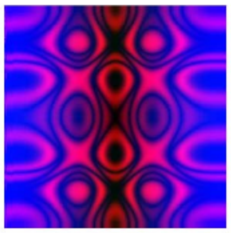
- Circuits
- Mathematical functions / equations
- Neural networks
- Programs
- Machine learning structures
- ...

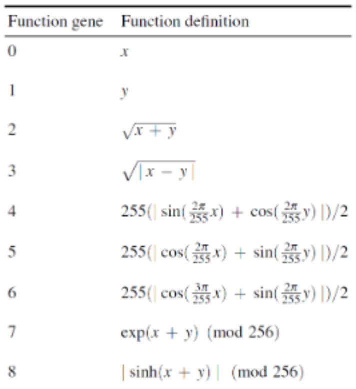
18.10 Examples

18.10.1 Art

- Inputs: image pixel position x,y
- Outputs: r,g,b intensities per pixel
  - Or single monochrome intensity

```
r = f1(x,y)
g = f2(x,y)
b = f3(x,y)
```





Function gene	Function definition
0	x
1	y
2	$\sqrt{x + y}$
3	$\sqrt{\lvert x - y \rvert}$
4	$255(\lvert \sin(\frac{2\pi}{255}x) + \cos(\frac{2\pi}{255}y) \rvert)/2$
5	$255(\lvert \cos(\frac{2\pi}{255}x) + \sin(\frac{2\pi}{255}y) \rvert)/2$
6	$255(\lvert \cos(\frac{3\pi}{255}x) + \sin(\frac{2\pi}{255}y) \rvert)/2$
7	$\exp(x + y) \pmod{256}$
8	$\lvert \sinh(x + y) \rvert \pmod{256}$

Figure 56:

18.10.2 Evolvable Hardware

- Evolution of digital image filters
- Input: distorted image
- Output: filtered image
- Fitness: distance between filtered and original image

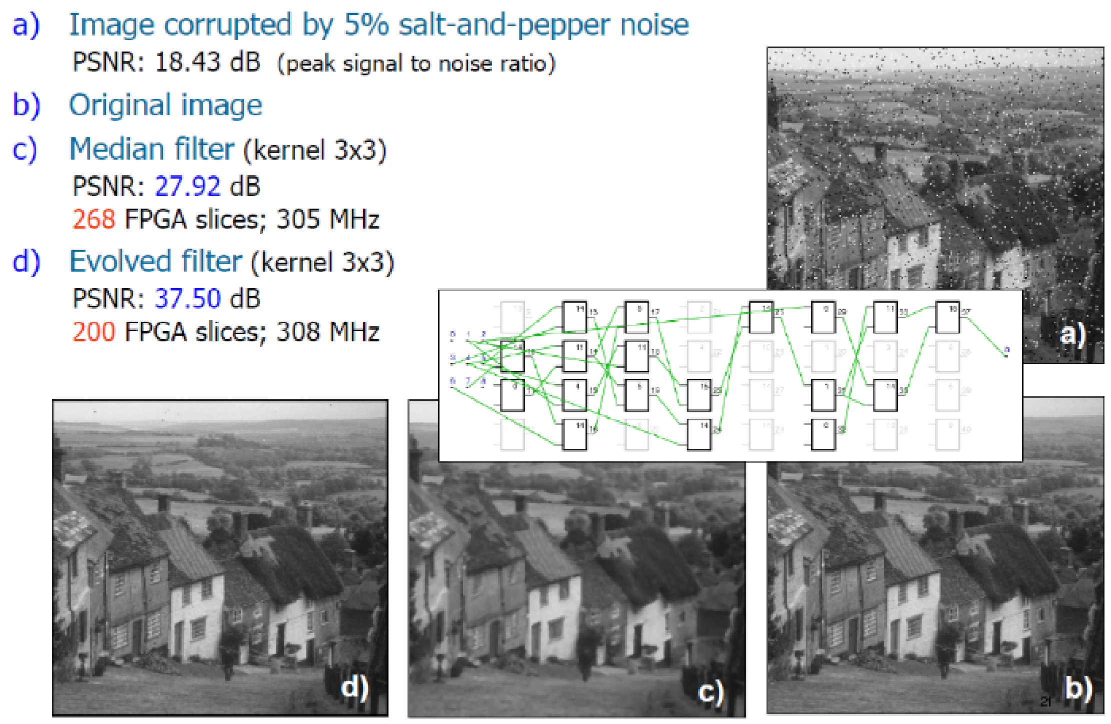

a) Image corrupted by 5% salt-and-pepper noise
PSNR: 18.43 dB (peak signal to noise ratio)
b) Original image
c) Median filter (kernel 3x3)
PSNR: 27.92 dB
268 FPGA slices; 305 MHz
d) Evolved filter (kernel 3x3)
PSNR: 37.50 dB
200 FPGA slices; 308 MHz
a)
d)
c)
b)

Figure 57:

18.11 Challenges of EHW

- Scalability – It's hard to evolve large systems!
  - General challenge in EC
  - Evolution of larger combinational circuits is difficult
    - ∗ Large and difficult search space
    - ∗ Time-consuming fitness function
    - ∗ 4x4 multiplier is hard
- On-chip evolution
  - Less flexibility offered by HW
  - Reconfiguration can be challenging

Part II

Machine Learning

- Self-learning/adaptive methods
- Learning by examples (rather than being programmed)
  - Modify its execution on the basis of newly acquired information
  - High-dimensional data – data that has more than three dimensions
- Machine learning is about automatically extracting relevant information from data and applying it to analyze new data
- Machine learning is *one* part of Artificial Intelligence (AI)

19 Learning

- Important parts of learning:
  - **Remembering**: Recognizing that last time we were in this situation, we tried out some particular action, and it worked.
  - **Adapting**: So, we will try it again, or it didn't work, so we will try something different.
  - **Generalising**: Recognizing similarity between different situations, so that things that applied in one place can be used in another

19.1 What is the Learning Problem?

- Learning = improving with experience
  - Improve over task T
  - with respect to performance measure P
  - based on experience E
- Defining the learning task, e.g.:
  - T: Recognizing hand-written words
  - P: Percentage of words correctly classified
  - E: Database of human-labeled images of handwritten words

20 Machine Learning (ML)

20.1 Some Characteristics of ML

- Typically used for classification tasks
- Learning from examples to analyze new data
- Provide generalization
- Iterative learning process
- Learning from scratch or adapt a previously learned system

20.2 Types of ML

Supervised learning : Training data *includes desired outputs*. Based on this training set, the algorithm generalises to respond correctly to all possible inputs.

Unsupervised learning : Training data *does not include desired outputs*, instead the algorithm tries to identify similarities between the inputs that have something in common are categorised together.

Reinforcement learning : The algorithm is told when the answer is wrong, but *does not get told how to correct it*. Algorithm must balance exploration of the unknown environment with exploitation of immediate rewards to maximize long-term rewards.

Evolutionary learning : Biological organisms adapt to improve their survival rates and chance of having offspring in their environment, using an idea of fitness (how good the current solution is).

20.3 Supervised Learning

- Training data provided as pairs:

$$\{(x_1, f(x_1)), (x_2, f(x_2)), \dots, (x_p, f(x_p))\} \tag{14}$$

- the goal is to predict an "output" y from an "input" x:

$$y = f(x) \tag{15}$$

- Output y for each input x is the "supervision" that is given to the learning algorithm.
  - Often obtained by manual annotation
  - Can be costly to do
- Most common examples
  - Classification
  - Regression

20.4 Classification

- Training data consists of "inputs", denoted x, and corresponding output "class labels", denoted as y.
- Goal is to correctly predict for a test data input the corresponding class label.
- Learn a "classifier" $f(x)$ from the input data that outputs the class label or a probability over the class labels.
- Example:
  - Input: image
  - Output: category label, eg "cat" vs. "no cat"
- Example:

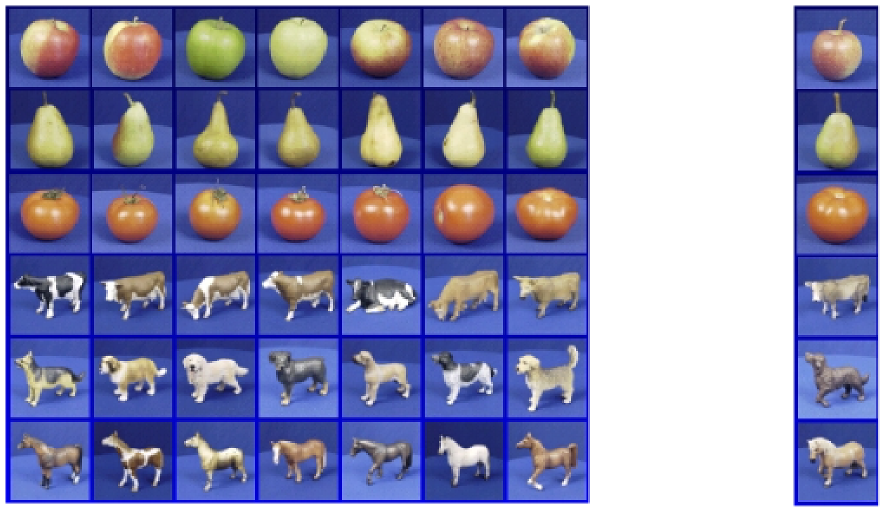


Figure 58: Given: training images and their categories. What are the categories of these test images?

- Two main phases:
  - **Training**: Learn the classification model from labeled data.
  - **Prediction**: Use the pre-built model to classify new instances.
- Classification can be binary (two classes), or over a larger number of classes (multi-class).
  - In binary classification we often refer to one class as "positive", and the other as "negative"
- Binary classifier creates boundaries in the input space between areas assigned to each class

20.4.1 Classification using Decision Boundaries

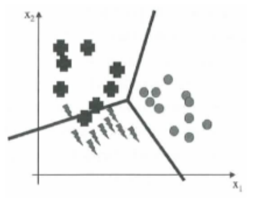


Figure 59: A set of straight line decision boundaries for a classification problem.

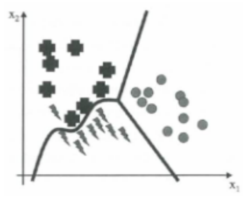
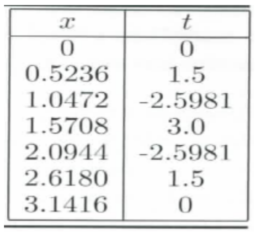


Figure 60: An alternative set of decision boundaries that separate the plusses from lightening strikes better, but it requires a line that isn't straight.

20.5 Regression

- Regression analysis is used to predict the value of one variable (the *dependent variable*) on the basis of other variables (the *independent variables*).
- Learn a continuous function.
- Given, the following data, can we find the value of the output when x = 0.44?



x	t
0	0
0.5236	1.5
1.0472	-2.5981
1.5708	3.0
2.0944	-2.5981
2.6180	1.5
3.1416	0

Figure 61:

- Goal is to predict for input x an output $f(x)$ that is close to the true y.
- It is generally a problem of function approximation, or interpolation, working out the value between values that we know.

20.5.1 Which line has the best "fit" to the data?

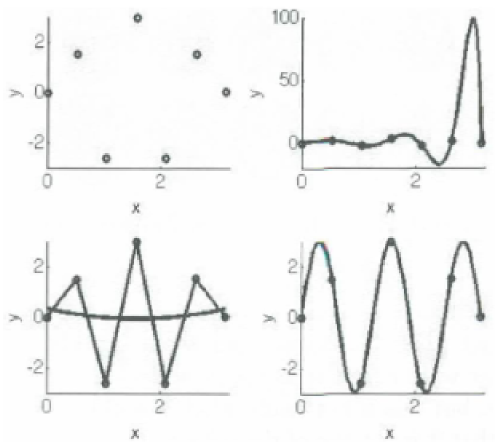


Figure 62:

- Top left: A few data points from a sample problem. Bottom left: Two possible ways to predict the values between the known data points: connecting the points with straight lines, or using a cubic approximation (which in this case misses all of the points). Top and bottom right: Two more complex approximators that passes through the points, although the lower one is rather better than the top.

21 Neural Networks

- Neurons are connected by synapses
- Signals "move" via electrochemical signals on a synapse
- The synapses release a chemical transmitter, enough of which can cause a threshold to be reached, causing the neuron to "fire"
- Synapses can be inhibitory or excitatory

21.1 Inspiration from Neurobiology

- A neuron: many-inputs / one-output unit.
- output can be *excited* or *not excited*.
- incoming signals from other neurons determine if the neuron shall *excite* ("fire")
- Output subject to attenuation in the *synapses*, which are junction parts of the neuron

21.2 Hebb's Rule

- Strength of a synaptic connection is proportional to the correlation of two connected neurons.
- If two neurons consistently fire simultaneously, synaptic connection is increased (if firing at different time, strength is reduced).
- "Cells that fire together, wire together."

21.3 McCulloch and Pitts Neurons

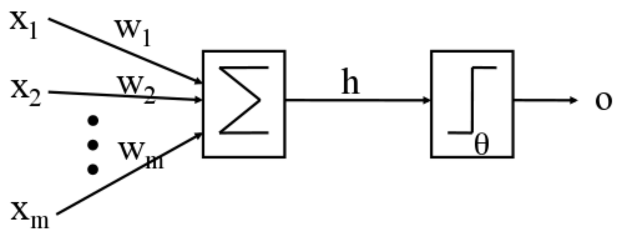


Figure 63:

- Greatly simplified biological neurons.
- Sum the weighted inputs
  - If total is greater than some threshold, neuron fires
  - Otherwise does not

$$h = \sum_{i=1}^{m} x_i w_i \ , \qquad o = \begin{cases} 1 & h \geq \theta \\ 0 & h < \theta \end{cases} , \qquad \text{for som threshold } \theta \tag{16}$$

- The weight w_j can be positive or negative
  - Inhibitory or exitatory.
- Use only a linear sum of inputs.
- Synchronous processing.
- No resting state following excitation.
- Scalar output instead of a pulse (spike train).

21.4 Limitations (McCulloch and Pitts Neurons Model)

- How realistic is this model? Not Very.
  - Real neurons are much more complicated.
  - Inputs to a real neuron are not necessary summed linearly.
  - Real neuron do not output a single output response, but a **spike train**.
  - Weights w_i can be positive or negative, whereas in biology connections are either *excitatory* OR *inhibitory*.
- We can put lots of McCulloch & Pitts neurons together. Connect them up in any way we like. In fact, assemblies of the neurons are capable of universal computation.
  - Can perform any computation that a normal computer can.
    Just have to solve for all the weights w_{ij}

22 The Perceptron

- Binary classifier function.
- Threshold activation function.

22.1 The Perceptron Network

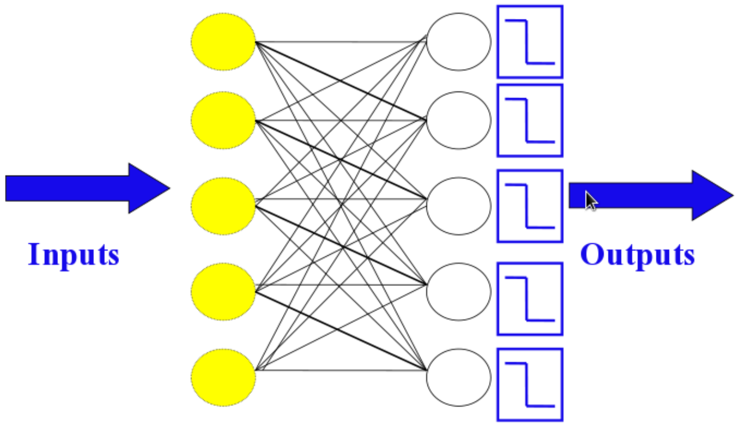


Figure 64:

22.2 Training Neurons

- Adapting the weights is learning
  - How does the network know it is right?
  - How do we adapt the weights to make the network right more often?
- Training set with target outputs.
- Learning rule.

22.3 A Simple Perceptron

- One unit (the loneliest network)
- Change the weights by an amount proportional to the difference between the desired output and the actual output.

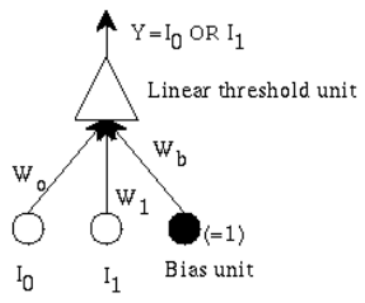


Figure 65: $w_{ij} \leftarrow w_{ij} + \Delta w_{ij}$

22.3.1 Updating the Weights

$$w_{ij} \leftarrow w_{ij} + \Delta w_{ij} \tag{17}$$

- Aim: minimize the error at the output
- If E = t-y, want E to be 0 (E=error)
- Use:

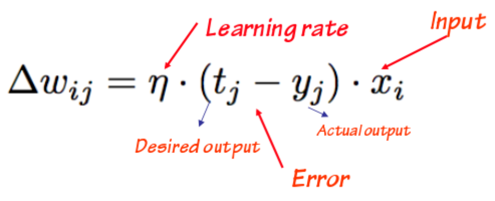


Figure 66:

22.3.2 The Learning Rate η

- η controls the size of the weight changes.
- Why not $\eta = 1$?
  - Weight change a lot, whenever the answer is wrong.
  - Makes the network unstable.
- Small η
  - Weights need to see the inputs more often before they change significantly.
  - Network takes longer to learn.
  - But, more stable network.

22.4 Bias Input

- What happens when all the inputs to a neuron are zero?
  - It doesn't matter what the weights are. The only way that we can control whether neuron fires or not is through the threshold.
- That's why threshold should be adjustable.
  - Changing the threshold requires an extra parameter that we need to write code for.
- We add to each neuron an extra input with a fixed value.

22.4.1 Biases Replace Threshold

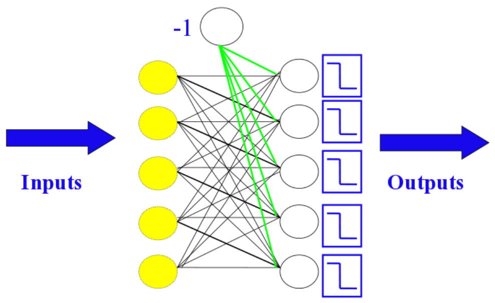
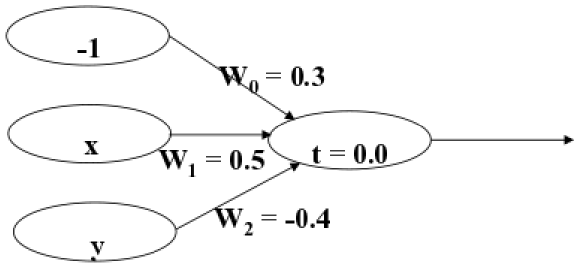


Figure 67:

22.5 Training a Perceptron

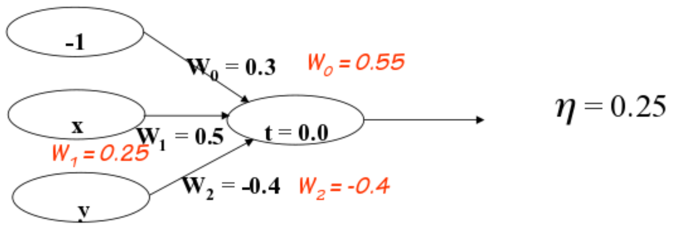
We are aiming for logical AND.

Input1	Input2	Output
0	0	0
0	1	0
1	0	0
1	1	1



I_1	I_2	I_3	Summation	Output
-1	0	0	(-1*0.3) + (0*0.5) + (0*-0.4) = -0.3	0
-1	0	1	(-1*0.3) + (0*0.5) + (1*-0.4) = -0.7	0
-1	1	0	(-1*0.3) + (1*0.5) + (0*-0.4) = 0.2	~~1~~
-1	1	1	(-1*0.3) + (1*0.5) + (1*-0.4) = -0.2	~~0~~

Figure 68:



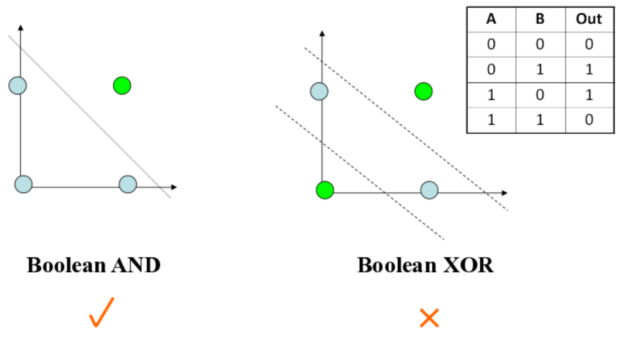
$W_0 = 0.3 + 0.25 * (0\text{-}1) * \text{-}1 = 0.55$
$W_1 = 0.5 + 0.25 * (0\text{-}1) * 1 = 0.25$
$W_2 = \text{-}0.4 + 0.25 * (0\text{-}1) * 0 = \text{-}0.4$

I_1	I_2	I_3	Summation	Output
-1	1	0	(-1*0.55) + (1*0.25) + (0*-0.4) = -0.3	~~1~~ 0

Figure 69:

22.6 Perceptron Limitations

- A single layer perceptron can only learn linearly separable problems.
  - Boolean AND function is linearly separable, whereas Boolean XOR function (and the parity problem in general) is not.



A	B	Out
0	0	0
0	1	1
1	0	1
1	1	0

Figure 70:

22.7 Multi-layer Perceptron (MLP)

- Multi-layer perceptron can solve this problem
- More than one layer of perceptrons (with a hardlimiting activation function) can learn any Boolean function
- A learning algorithm for multi-layer perceptrons was not developed until much later
  - backpropagation algorithm (replacing the hardlimiter with a sigmoid activation function)
- An extra layer of perceptrons will add another dimension.

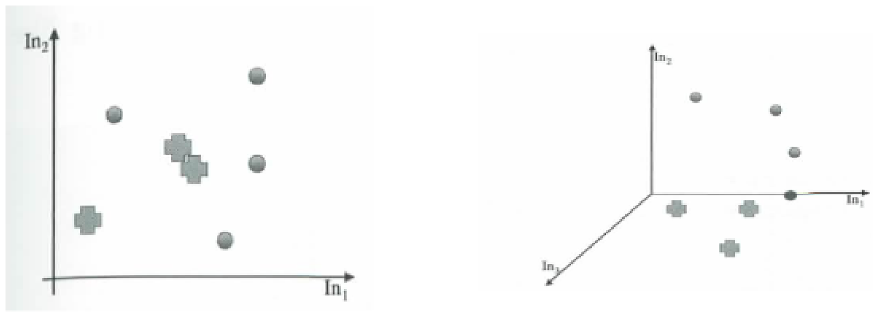


Figure 71: Left: Non-separable 2D dataset. Right: The same dataset with third coordinate x_1 x x_2 , which makes it separable.

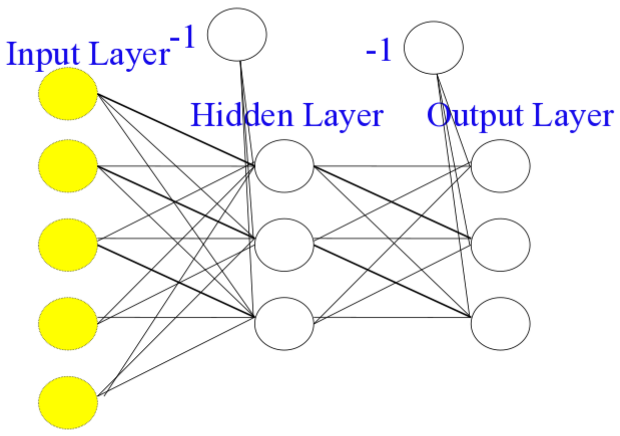


Figure 72:

22.8 MLP Decision Boundary

In contrast to perceptrons, multilayer networks can learn not only multiple decision boundaries, but the boundaries may be nonlinear.

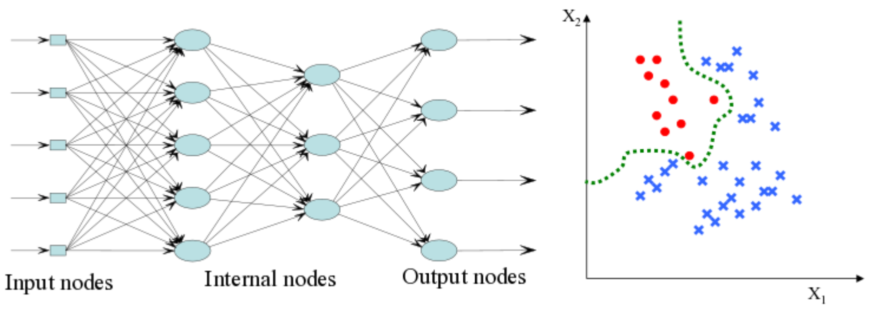


Figure 73:

22.9 Decision Boundries

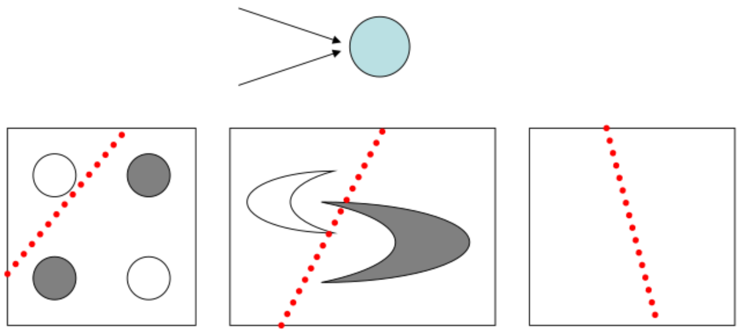


Figure 74: Straight lines (surfaces), linear separable, half plane bounded by hyperplane

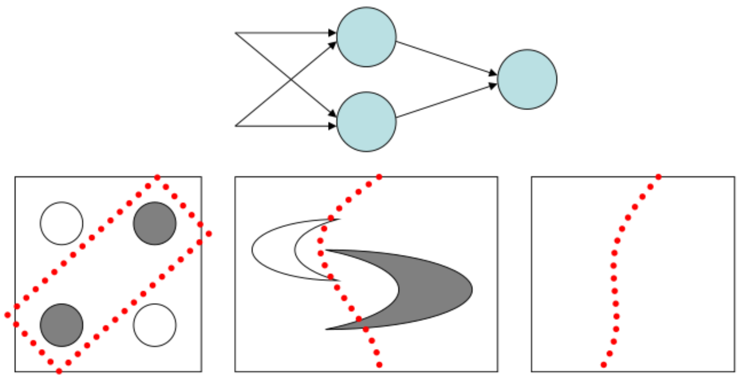


Figure 75: Convex areas (open or closed)

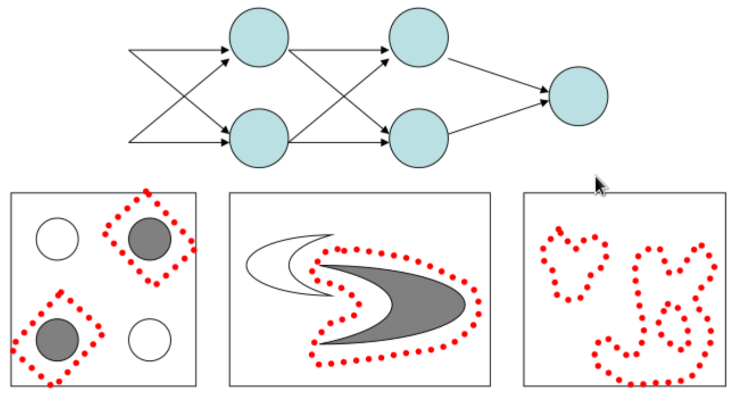


Figure 76: Combinations of convex areas

23 Multilayer Network

- A neural network with one or more layers of nodes between the input and the output nodes is called a *multilayer network*.
- The multilayer network structure, or architecture, or topology, consists of an *input layer*, *one or more hidden layers*, and *one output layer*.
- The input nodes pass values to the first hidden layer, its nodes to the second and so on till producing outputs.
  - A network with a layer of input units, a layer of hidden units and a layer of output units is a two-layer network.
  - A network with two layers of hidden units is a three-layer network, and so on.

23.1 Properties of architecture

- Layer $n-1$ is fully connected to layer n.
- No connections within a single layer.

- No direct connections between input and output layers.
- Fully connected; all nodes in one layer connect to all nodes in the next layer.
- Number of output units need not equal number of input units.
- Number of hidden units per layer can be more or less than input or output units.

23.2 What Do Each of The Layers Do?

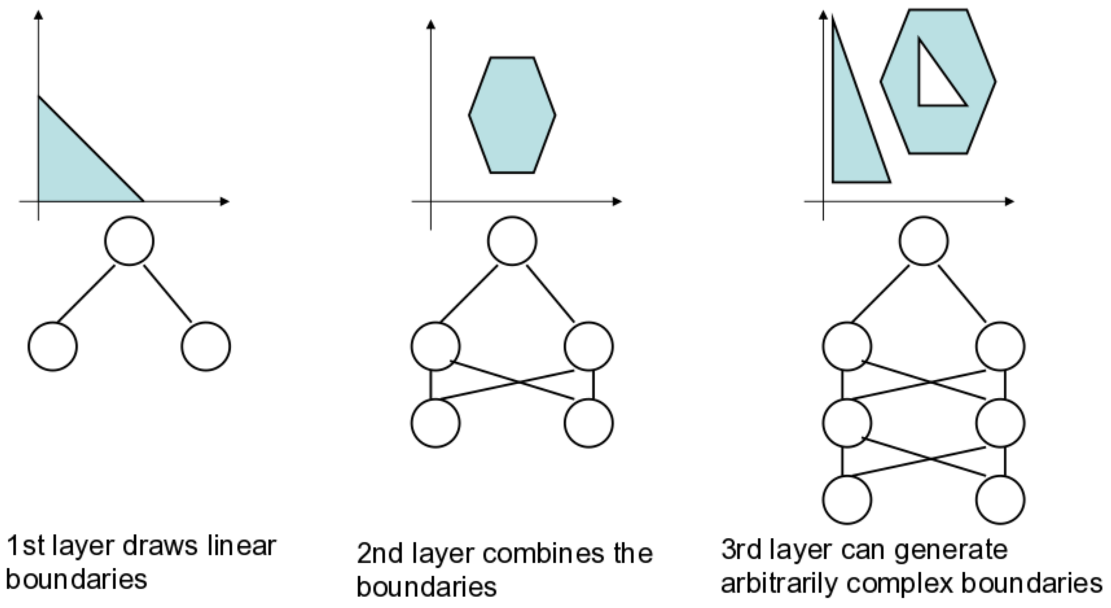

1st layer draws linear boundaries
2nd layer combines the boundaries
3rd layer can generate arbitrarily complex boundaries

Figure 77:

23.3 Solution for XOR : Add a Hidden Layer !!

Minsky & Papert (1969) offered solution to XOR problem by combining perceptron unit responses using a second layer of units.

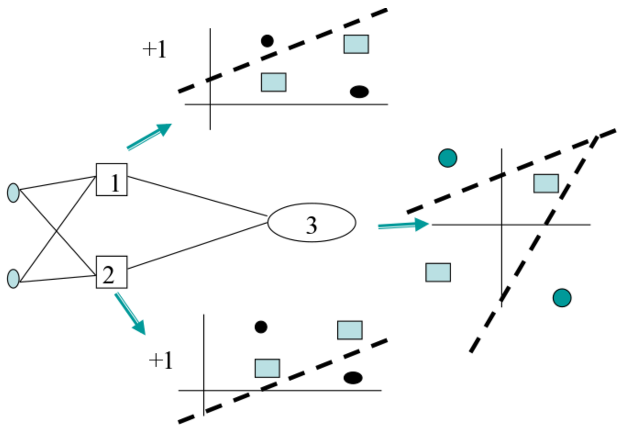
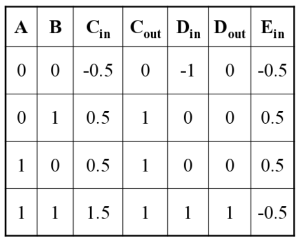


Figure 78:



A	B	C_{in}	C_{out}	D_{in}	D_{out}	E_{in}
0	0	-0.5	0	-1	0	-0.5
0	1	0.5	1	0	0	0.5
1	0	0.5	1	0	0	0.5
1	1	1.5	1	1	1	-0.5

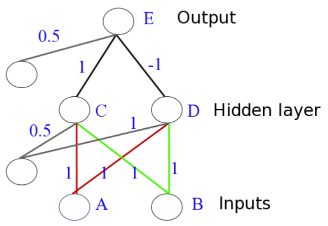


Figure 79:

23.4 How to Train MLP?

- How we can train the network, so that the weights are adapted to generate correct target/answer?

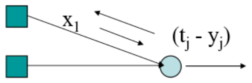


Figure 80:

- In Perceptron, errors are computed at the output.
- In MLP,
  - Don't know which weights are wrong
  - Don't know the correct activations for the neurons in the hidden layers
- Then ...
  - The **problem** is: How to learn Multi Layer Perceptrons??
  - **Solution**: Backpropagation Algorithm (Rumelhart and colleagues,1986)

24 Backpropagation

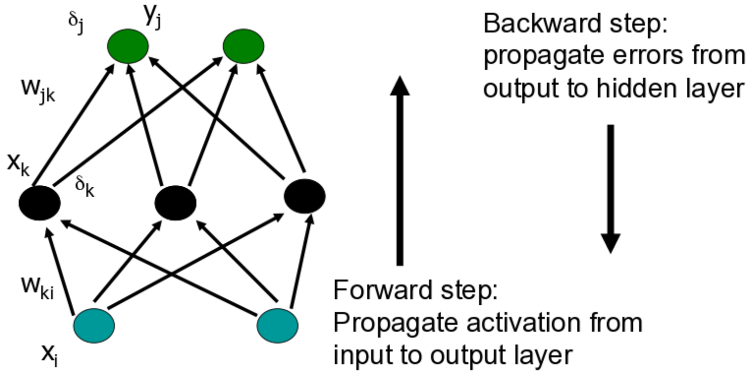

δ_j
y_j
w_{jk}
x_k
δ_k
w_{ki}
x_i
Backward step: propagate errors from output to hidden layer
Forward step: Propagate activation from input to output layer

Figure 81:

- Durign the backward pass the weight are adjusted on accordance with the *error correction rule*.

- Ther error is the *actual* output, and is subtracted from the *desired* output.
- The weights are adjusted to minimize this error.

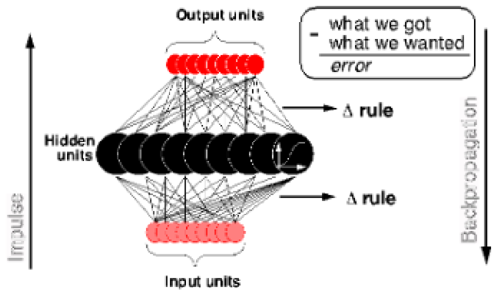


Figure 82:

24.1 Training MLPs

24.1.1 Forward Pass

- Put the input values in the input layer.
- Calculate the activations of the hidden nodes.
- Calculate the activations of the output nodes.

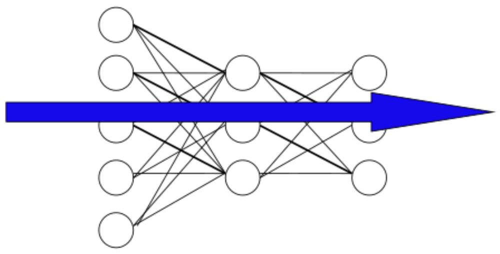


Figure 83:

24.1.2 Backward Pass

- Calculate the output errors
- Update last layer of weights.
- Propagate error backward, update hidden weights.
- Until first layer is reached

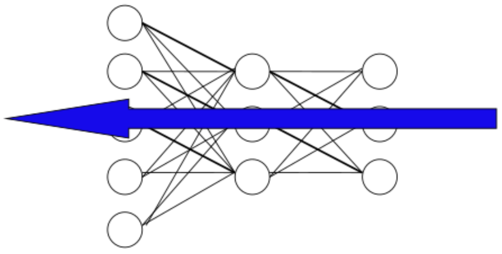


Figure 84:

24.2 Algorithm

- The backpropagation training algorithm uses the *gradient descent* technique to minimize the mean square difference between the desired and actual outputs.
- The network is trained initially selecting small random weights and then presenting all training data incrementally.
- Weights are adjusted after every trial until weights converge and the error is reduced to an acceptable value.

24.2.1 Gradient Descent Learning

- Target: Minimize error.
- Harder than Perceptron:
  - Many weights
  - Which ones are wrong; input-hidden ot hidden-output?

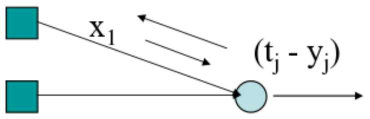


Figure 85:

- Use gradient descent learning
- Compute gradient → differentiate sum-of squares error function.

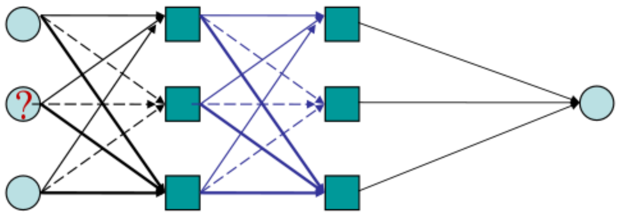


Figure 86:

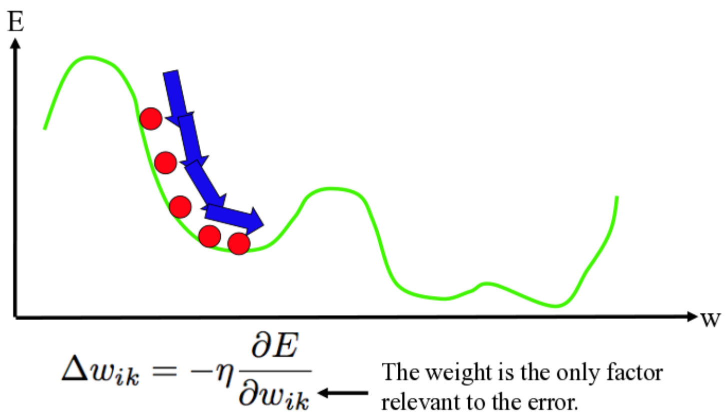

E
w
$\Delta w_{ik} = -\eta \frac{\partial E}{\partial w_{ik}}$
The weight is the only factor relevant to the error.

Figure 87:

24.2.1.1 Error Function

- Single scalar function for entire network.
- Parameterized by weights (objects of interest).
- Multiple errors of different signs should not cancel out.
- Sum-of-squares error:

$$E(w) = \frac{1}{2}\sum_k (t_k - y_k)^2 = \frac{1}{2}\sum_k \left(t_k - \sum_i w_{ik}x_i\right)^2 \tag{18}$$

24.2.1.2 Error Terms

- Need to differentiate the activation function
- Chain rule of differentiation.
- Gives us the following error terms (deltas)
  - For the outputs

$$\delta_k = (y_k - t_k)y_k(1 - y_k) \tag{19}$$

- - For the hidden nodes

$$\delta_j = a_j(1 - a_j)\sum_k w_{jk}\delta_k \tag{20}$$

24.2.1.3 Update Rules

- This gives us the necessary update rules
  - For the weights connected to the outputs:

$$w_{jk} \leftarrow w_{jk} - \eta \delta_k a_j^{hidden} \tag{21}$$

-
  - For the weights on the hidden nodes:

$$v_{ij} \leftarrow v_{ij} - \eta \delta_j x_i \tag{22}$$

24.3 Summary

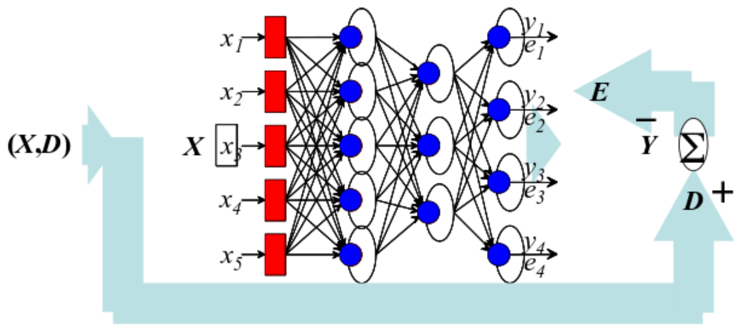


Figure 88:

- Introduce inputs.
- Feed values forward through network.
- Compute sum-of-squares error at outputs.
- Compute the delta terms at the output by differentiation.
- Use this to update the weights connecting the last hidden layer to the outputs
- Once these are correct, propagate deltas back to the neurons of the hidden layers.
- Compute the delta terms for these neurons.
- Use them to update the next set of weights.
- Repeat until the inputs are reached

24.4 Algorithm (sequential)

1. Apply an input vector and calculate all activations, a and u
2. Evaluate deltas for all output units:

$$\Delta_i = (d_i - y_i)g'(a_i) \tag{23}$$

3. Propagate deltas backwards to hidden layer deltas:

$$\delta_i = g'(u_i)\sum_k \Delta_k w_{ki} \tag{24}$$

4. Update weights:

$$v_{ij} \leftarrow v_{ij} + \eta\delta_i x_j \tag{25}$$

$$w_{ij} \leftarrow w_{ij} + \eta\Delta_i z_j \tag{26}$$

24.5 Example

1. Once weight changes are computed for all units, weights are updated at the same time (bias included as weights here). An example:

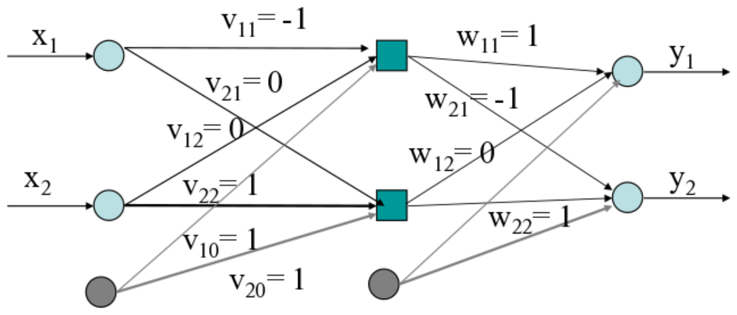


Figure 89:

Use identity activation function (i.e. $g(a) = a$) for simplicity of example.

2. All biases set to 1. Will not draw them for clarity.
Learning rate h = 0.1

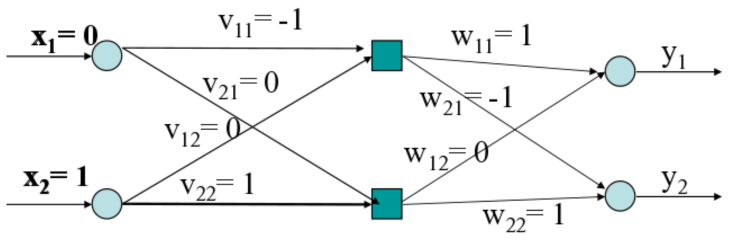
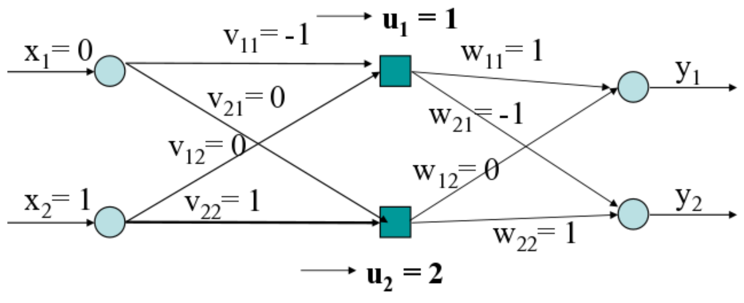


Figure 90:

Have input [0 1] with target [1 0].

3. Forward pass. Calculate first layer activations:

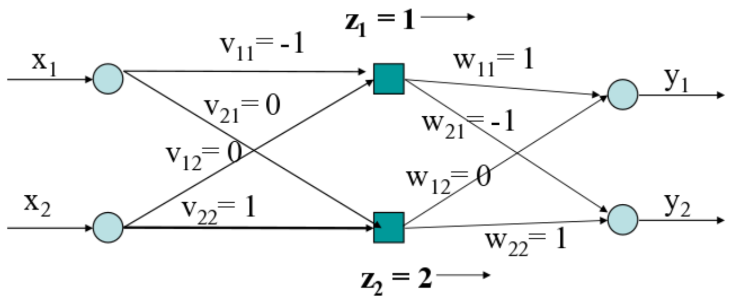


$$u_1 = -1 \times 0 + 0 \times 1 + 1 = 1$$

$$u_2 = 0 \times 0 + 1 \times 1 + 1 = 2$$

Figure 91:

4. Calculate first layer outputs by passing activations thru activation functions

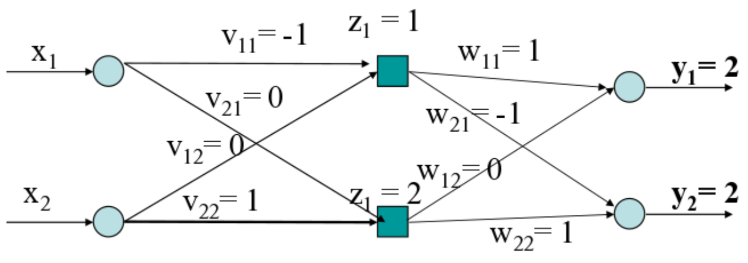


$$z_1 = g(u_1) = 1$$

$$z_2 = g(u_2) = 2$$

Figure 92:

5. Calculate second layer outputs (weighted sum through activation functions):



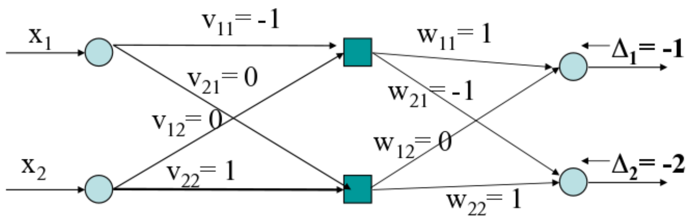
$$y_1 = a_1 = 1\text{x}1 + 0\text{x}2 + 1 = 2$$

$$y_2 = a_2 = -1\text{x}1 + 1\text{x}2 + 1 = 2$$

Figure 93:

6. Backward pass:

$$\Delta_i = (d_i - y_i)g'(a_i)$$



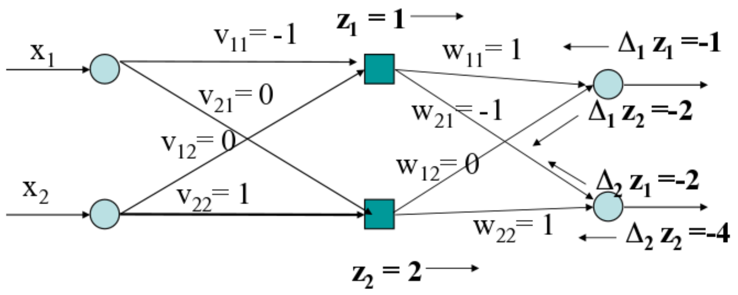
Target =[1, 0] so $d_1 = 1$ and $d_2 = 0$. So:

$\Delta_1 = (d_1 - y_1) = 1 - 2 = -1$

$\Delta_2 = (d_2 - y_2) = 0 - 2 = -2$

Figure 94:

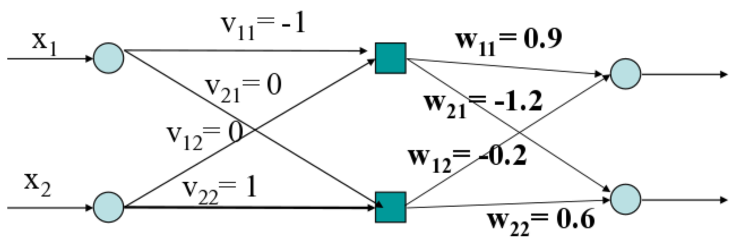
7. Calculate weight changes for first layer (cf perceptron learning):



$$w_{ij} \leftarrow w_{ij} + \eta\Delta_i z_j$$

Figure 95:

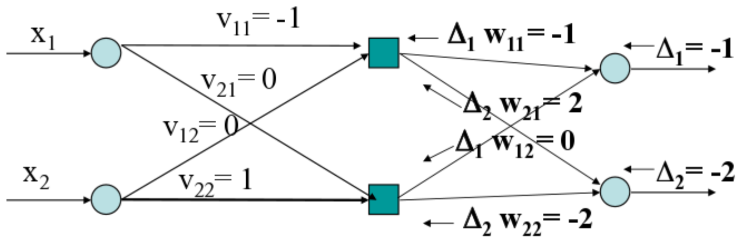
8. Weight changes will be:



$$w_{ij} \leftarrow w_{ij} + \eta \Delta_i z_j$$

Figure 96:

9. Calculate hidden layer deltas:

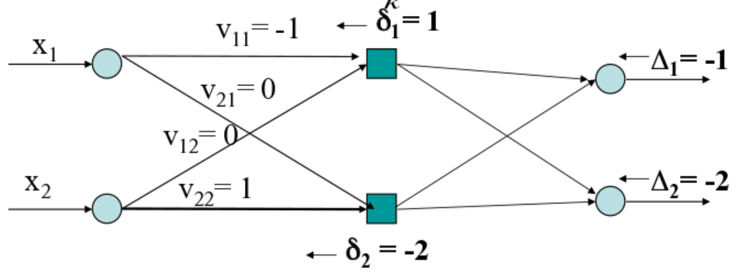


$$\delta_i = g'(u_i) \sum_k \Delta_k w_{ki}$$

Figure 97:

10. D's propagate back:

$$\delta_i = g'(u_i) \sum_k \Delta_k w_{ki}$$



$$\delta_1 = -1 + 2 = 1$$
$$\delta_2 = 0 - 2 = -2$$

Figure 98:

11. And are multiplied by inputs

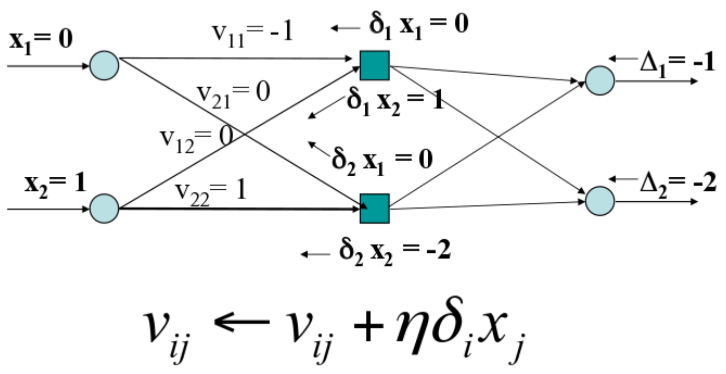


Figure 99:

12. Finally change weights:

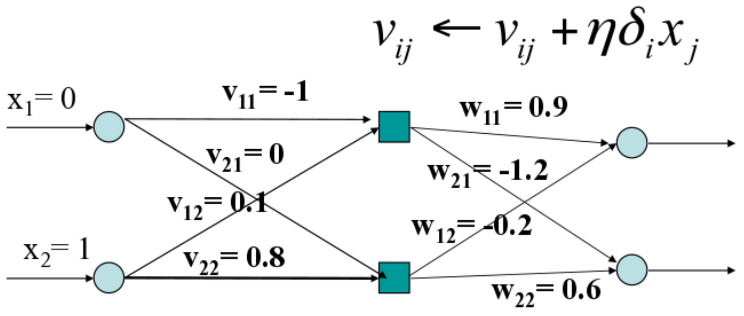


Figure 100:

Note that the weights multiplied by the zero input are unchanged as they do not contribute to the error. We have also changed biases (not shown)

13. Now go forward again (would normally use a new input vector):

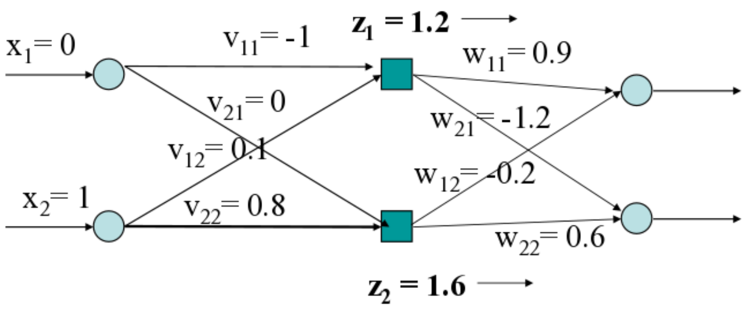


Figure 101:

14. Now go forward again (would normally use a new input vector):

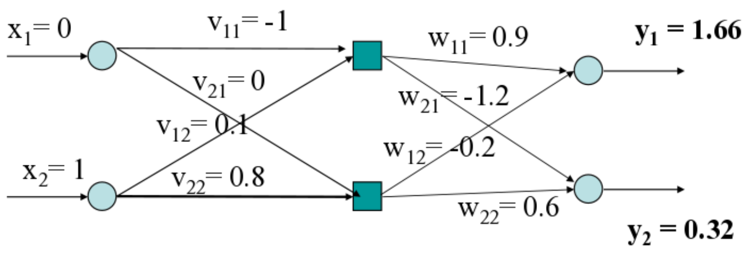


Figure 102:

Outputs now closer to target value [1, 0]

25 Activation Function

- We need to compute the derivative of activation function g
- What do we want in an activation function?
  - Differentiable
  - Nonlinear (more powerful)
  - Bounded range (for numerical stability)

25.1 Hard Limit Function

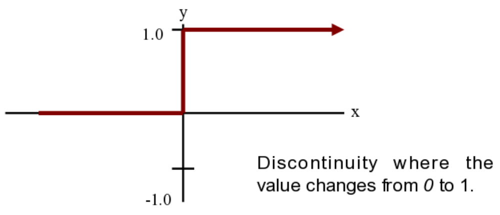


Figure 103:

25.2 A Quick Overview

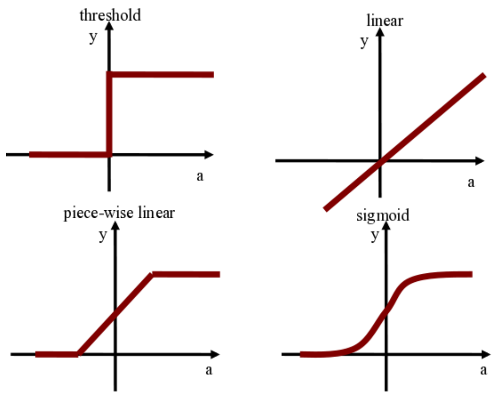


Figure 104:

25.3 Log Sigmoidal Function

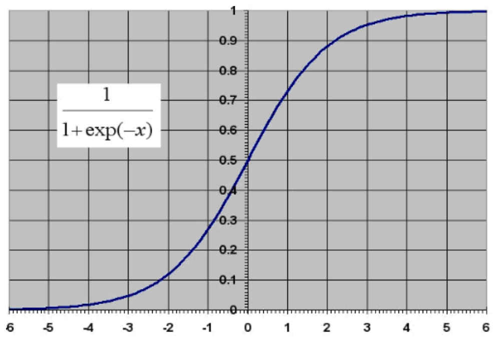


Figure 105:

25.3.1 Sigmoidal (Logistic) Function-Common in MLP

$$g(a_i) = \frac{1}{1+exp(-ka_i)} = \frac{1}{1+e^{-ka_i}} \tag{27}$$

- Where k is a positive constant.
- The sigmoidal function gives a value in range of 0 to 1.
- Alternatively can use $tanh(ka)$ which is same shape but in range -1 to 1.

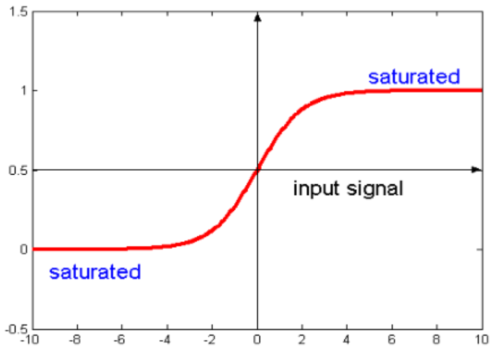


Figure 106: input-output function of a neuron (rate coding assumption)

Note: when net = 0, g = 0.5

25.3.2 Derivative of Sigmoidal Function

$$g'(a_i) = \frac{kexp(-ka_i)}{[1+kexp(-ka_i)]^2} = kg(a_i)[1-g(a_i)] \tag{28}$$

since: $y_i = g(a_i)$ we have: $g'(a_i) = ky_i(1-y_i)$

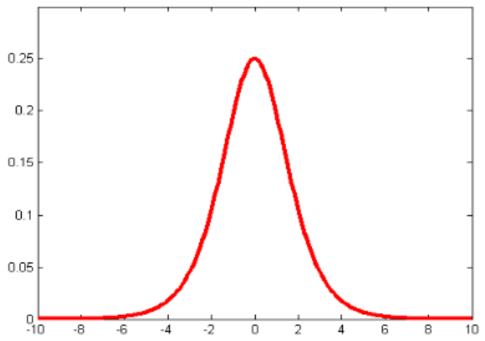


Figure 107:

Derivative of sigmoidal function has max at $a = 0$, $g = 0.5$, is symmetric about this point falling to zero as sigmoid approaches extreme values.

25.3.3 Learning Capacity

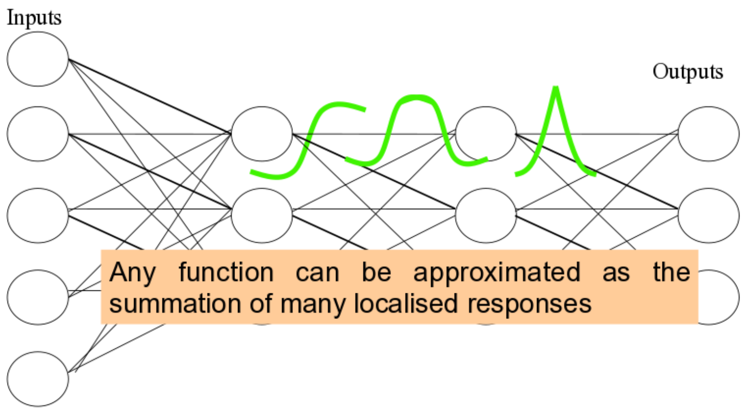

Inputs
Outputs
Any function can be approximated as the summation of many localised responses

Figure 108:

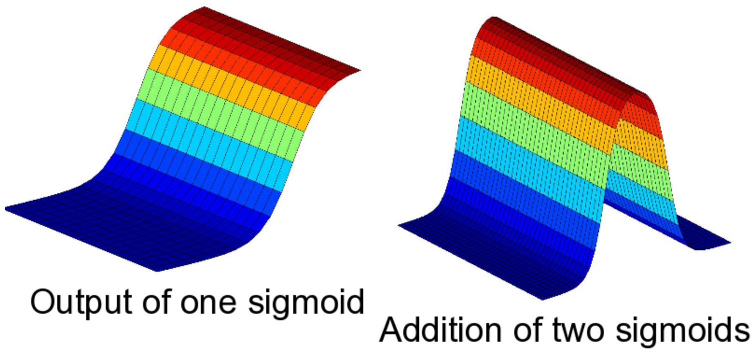


Figure 109:

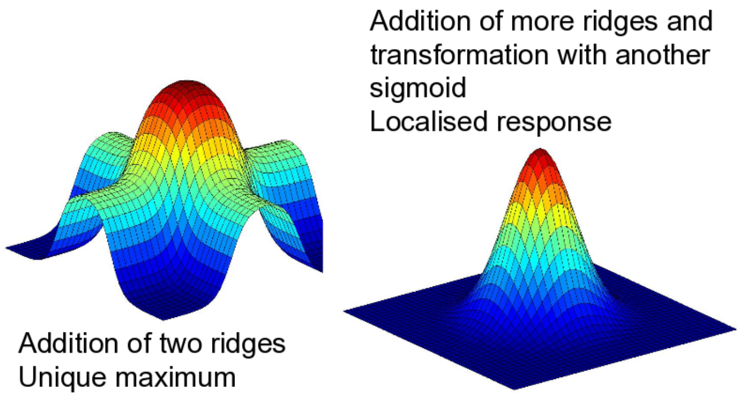


Figure 110:

26 Network Training

- Training set shown repeatedly until stopping criteria are met.
- Usual to randomize order of training patterns presented for each epoch in order to avoid correlation between consecutive training pairs being learnt (order effects).
- When should the weights be updated?
  - After all inputs seen (**batch**)
    - More accurate estimate of gradient
    - Converges to local minimum faster (Jim doesn't agree!)
  - After each input is seen (**sequential**)
    - Simpler to program
    - May escape from local minima (change order or presentation)
- Both ways, need many epochs - passes through the whole dataset

26.1 Selecting Initial Weight Values

- Choice of initial weight values is important as this decides starting position in weight space. That is, how far away from global minimum
- Aim is to select weight values which produce midrange function signals
- Select weight values randomly from uniform probability distribution
- Normalise weight values so number of weighted connections per unit produces midrange function signal

26.2 Network Topology

- How many layers?
- How many neurons per layer?
- No good answers
  - At most three weight layers, usually two
  - Test several different networks

26.3 Amount of Training

- How much training data is needed?
- How many epochs are needed?
- Data:
  - Count the weights
  - Rule of thumb: use 10 times more data than the number of weights

26.4 Generalisation

- Aim of neural network learning:
  - Generalise from training examples to all possible inputs.
- The objective of learning is to achieve good generalizaion to new cases; otherwise we would jsut use a look-up table.
- Under-training is bad
- Over-training is also bad
- Generalization can be viewed as a mathematical *interpolation* or *regression* over a set of training points:

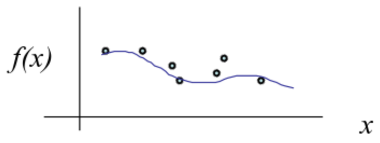


Figure 111:

26.4.1 A regression problem:

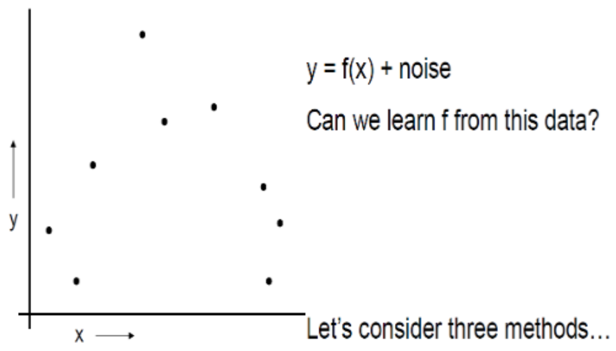


Figure 112:

- Linear Regression

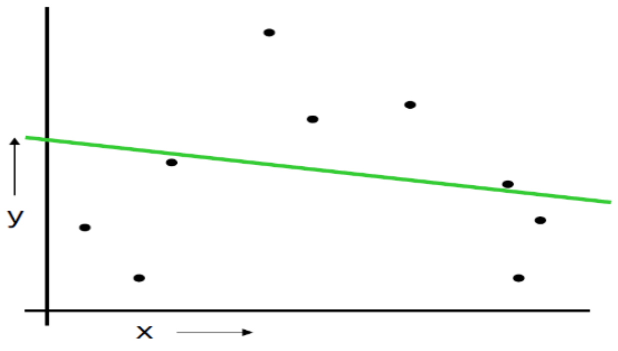


Figure 113:

- Quadric Regression

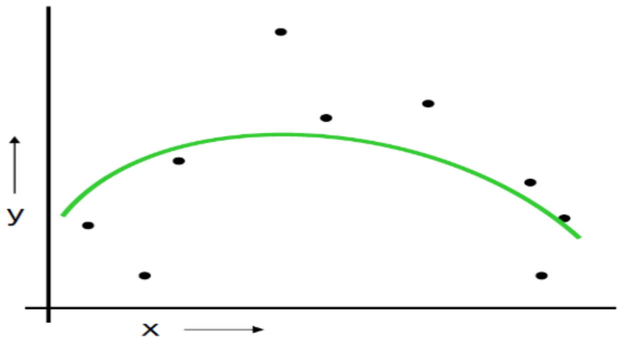


Figure 114:

- Join-the-dots

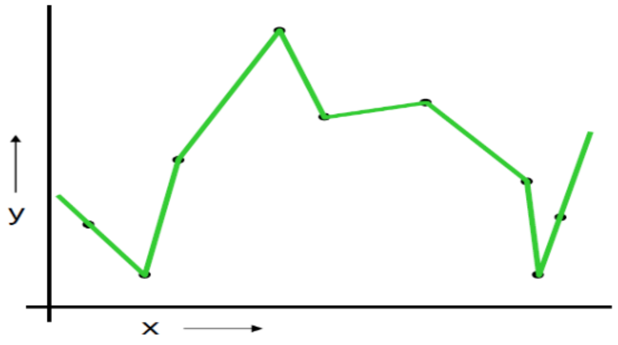


Figure 115:

26.5 Overfitting

- Overfitting occurs when a model begins to learn the bias of the training data rather than learning to generalize.
- Overfitting generally occurs when a model is excessively complex in relation to the amount of data available.
- A model which overfits the training data will generally have poor predictive performance, as it can exaggerate minor fluctuations in the data.

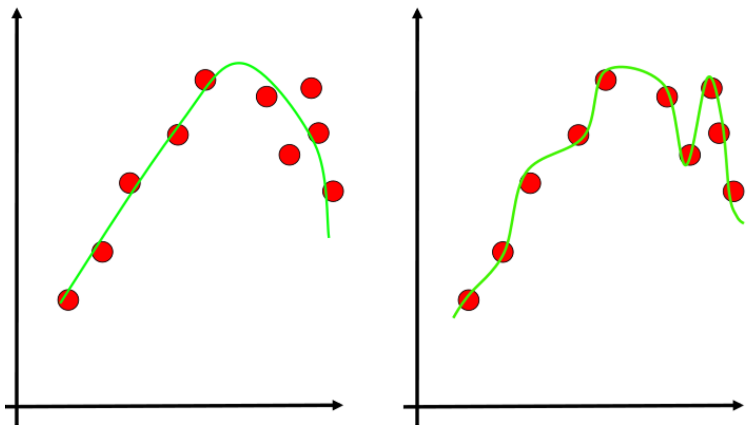


Figure 116:

- The training data contains information about the regularities in the mapping from input to output.
- Training data also contains bias:
  - There is *sampling bias*. There will be accidental regularities due to the finite size of the training set.
  - The target values may also be unreliable or *noisy*.
- When we fit the model, it cannot tell which regularities are relevant and which are caused by sampling error.
  - So it fits both kinds of regularity.
  - If the model is very flexible it can model the sampling error really well. **This is not what we want**.
- Approximation of the function $y = f(x)$:

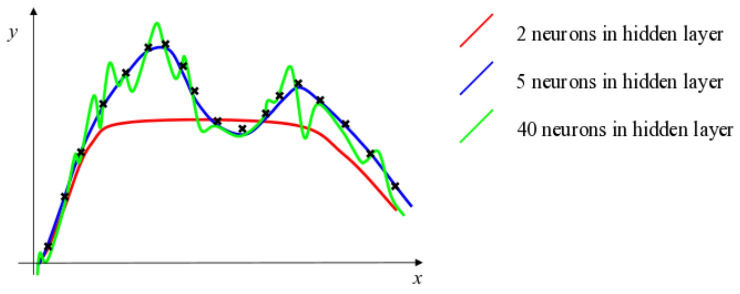


Figure 117:

26.5.1 The Solution: Cross-Validation

To maximize generalization and avoid overfitting, split data into three sets:

- Training set: Train the model.
- Validation set: Judge the model's generalization ability during training.
- Test set: Judge the model's generalization ability after training.

26.5.1.1 Validation Set

- Data unseen by training algorithm – not used for backpropagation.
- Network is not trained on this data, so we can use it to measure generalization ability.
- Goal is to maximize generalization ability, so we should minimize the error on this data set.

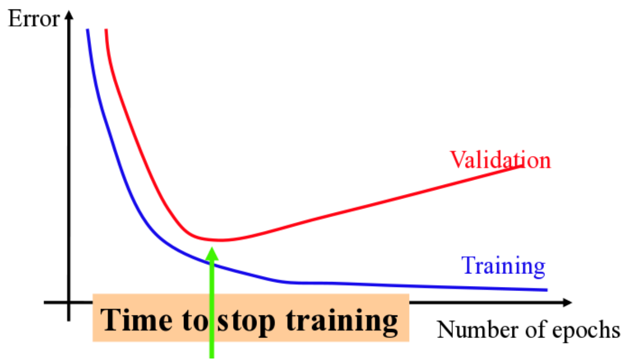


Figure 118: Early stopping

26.5.1.2 Testing Set

- Data unseen during training and validation.
- Has no influence on when to stop training.
- With early stopping, we've maximized the ability to generalize to *the validation set*.
- To judge the final result, we should measure its ability to generalize to completely unseen data.

26.5.2 k-fold Cross Validation

- Cross-validation leaves less training data.
- Generalization ability is still only measured on a small set (which will be biased).
- Solution: repeat over many different splits.
  - Divide all data into k sets (or folds).
  - For i = 1 . . . k:
    - Train on data[i], validate on data[i+1], test on rest.
  - Average the results.

26.5.3 Leave-one-out Cross Validation (LOOCV)

- For k=i to R
  1. Let (x_k, y_k) be the k^{th} record
  2. Temporarily remove (x_k, y_k) from the dataset
  3. Train on the remaining R-1 datapoints
  4. Note your error (x_k, y_k)

- When you've done all points, report mean error.

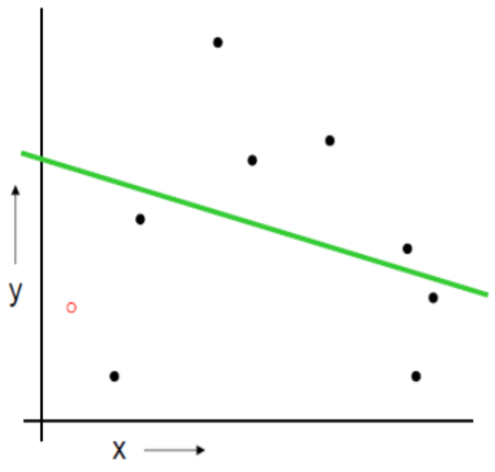


Figure 119:

26.5.4 Input Normalization

- Stops the weights from getting unnecessarily large.
- Treat each data dimension independently.
- Each input variable should be processed so that the mean value is close to zero or at least very small when compared to the standard deviation.

26.5.5 Learning Rate

The correct value of η depends on the application. Values between 0.1 and 0.9 have been used in many applications.

26.5.6 How Many Layers and Neurons

- The number of layers and of neurons depend on the specific task.
  - In practice this issue is solved by trial and error.
- Two types of adaptive algorithms can be used:
  - start from a large network and successively remove some neurons and links until network performance degrades.
  - begin with a small network and introduce new neurons until performance is satisfactory.

27 Support Vector Machines (SVM)

27.1 Optimal Separation

Choose the decision boundary with the best margin!

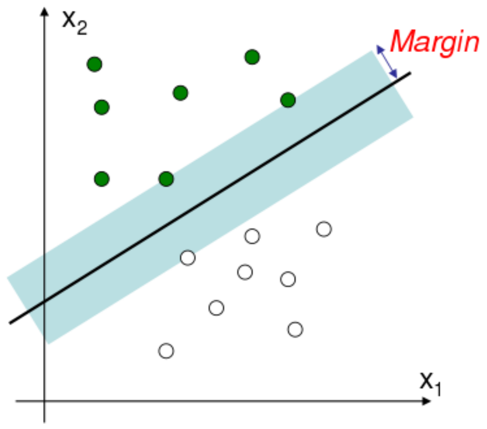


Figure 120:

Why?

- New data near the training data points will likely be of the same class

Support vectors:

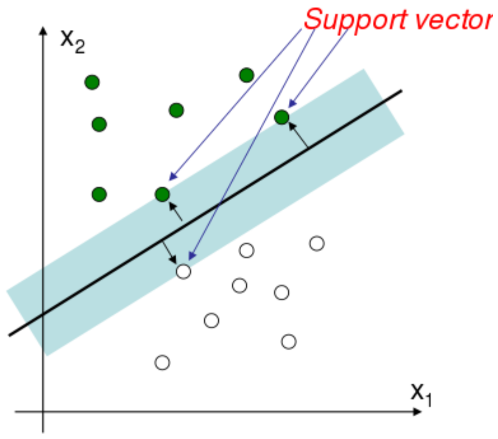


Figure 121:

- The training data defining the margin
- The rest of the data can be discarded when we are done learning
- Distance to hyperplane:

$$w \cdot x_i - b = \begin{cases} > 0 & \text{above plane – class A:} \quad y_i = 1 \\ < 0 & \text{below plane – class B:} \quad y_i = -1 \end{cases} \tag{29}$$

- If we require that $y_i(w \cdot x_i - b) \geq 1)$ then the margin is $M = 1/(2|w|)$
  - Maximizing the margin ⇔ minimizing $w \cdot w$
  - Exact solution can be found, along with a list of support vectors, using quadratic programming

27.2 Nonlinearity

- How to classify linearly inseparable data?
- Combine many linear SVMs?
  - Similar to multilayer neural networks
  - But what are the target outputs for the hidden layers?
- A different idea:
  - Map inputs into a higher-dimensional space
  - Hope that they are linearly separable there.

27.3 Increase Dimensionality

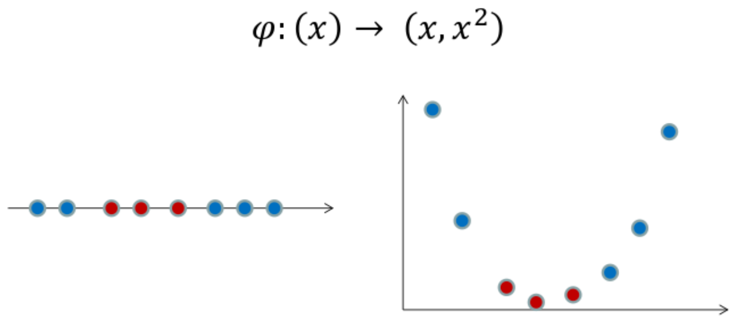


Figure 122:

27.3.1 High Dimensionality

- SVMs typically map to feature spaces of much higher dimension
  - With enough dimensions, it becomes very likely that the data becomes linearly separable

27.4 Kernels

- Finding the hyperplane only requires the dot product between vectors, not the actual vectors
  - Calculating $\varphi(x_i) \cdot \varphi(x_j)$ might be much easier than $\varphi(x_i)$!
- $K(x_i, x_j) = \varphi(x_i) \cdot \varphi(x_j)$ is called the kernel of φ
  - Common kernels include
    * None: $K(x_i, x_j) = x_i \cdot x_j$
    * Polynomial: $K(x_i, x_j) = (1 + x_i \cdot x_j)^p$
    * Sigmoid: $K(x_i, x_j) = tahn(kx_i \cdot x_j - \delta)$
    * Radial basis function: $K(x_i, x_j) = exp(-(x_i - x_j)^2/2\sigma^2)$

27.5 Overfitting

- Any data set is linearly separable in a feature space of sufficient complexity
- We have to be wary of overfitting: Use cross-validation and early stopping!
  - If there are noisy outliers (esp. mislabeled examples), we need to take stronger measures: soft margin.

27.6 Soft Margins

- Instead of perfectly separating all data, allow some misclassifications
- Introduce slack variables
  - Optimize tradeoff between maximum margin and misclassification penalty
  - Tradeoff is balanced by penalty factor C
- Useful when some error is tolerated, or when there are chances of mislabeled training data

27.7 Application

- Classification
  - Multi-class can be achieved via multiple outputs (1v1 or 1vMany)
- Regression
- Supervised and unsupervised learning
- Object detection & recognition
- Content-based image retrieval
- Text recognition
- Speech recognition
- Biometrics
- Etc.

27.8 Considerations

- Quite powerful
  - Must beware of overfitting
- Robust to some noise, if margin is managed properly
- Fast to apply
- Difficult to interpret
- How to pick kernel?
  - Start with Gaussian RBF or polynomial
  - May require domain-specific knowledge
  - Can combine kernels for heterogeneous data
  - Consult experts

28 Ensemble Learning

- "Decision by committee"
  - Train multiple classifiers to be slightly different
    - An "ensemble"
  - Make classifications based on the combined results of all of them
- Two common types of training differentiation
  - Boosting: change the importance of data points
  - Bagging: change the data sample

28.1 Boosting - AdaBoost

- Iteratively trains classifiers
- Each data point is assigned a weight
  - For the first classifier all the weights are equal
  - For the next classifier the weights of the data points that were misclassified previously is raised
  - This is continued until the combined error of the classifiers trained so far is sufficiently low
- Dependent on the classifier's ability to consider the weights in their training

28.2 Bagging

- Makes a random sample of the training data for each classifier – *bootstrap* samples
  - Same size as the training data
  - With replacement
  - Some data points will occur at least twice!
  - Variance will be reduced
  - Each classifier will have different views of the training data

28.3 Combining the Classifiers

- Which classifiers do we listen to when the ensemble is in disagreement?
  - Majority voting
    - The most "popular" class is chosen
  - Weighted voting
    - Some classifiers have greater influence than others
  - Mixture of experts
    - A meta-machine learning algorithm decides which classifiers are most likely to be correct

28.3.1 Majority voting

What to do when there is disagreement

- Refuse to classify?
- Classify only if more than half agree?
- Return the most common vote?

29 Dimensionality Reduction

Why reduce dimensionality?

- Reduces time complexity: Less computation
- Reduces space complexity: Less parameters
- Saves the cost of acquiring irrelevant features
- Simpler models are more robust
- Easier to interpret; simpler explanation
- Data visualization (structure, groups, outliers, etc.) if plotted in 2 or 3 dimensions

29.1 Principal Components

- The directions, along with the most variation, don't have to correspond to the coordinate axes

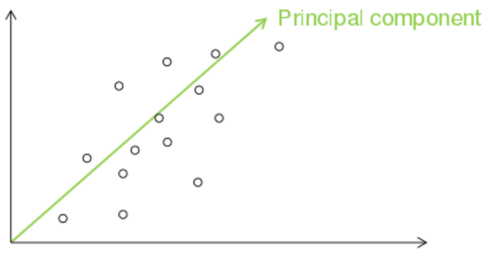


Figure 123:

29.1.1 Analysis (PCA)

- Rotate the axes to lie along the principal components
- Remove the axes with the least variation
  - Keep a certain number of dimensions
  - Or: keep a certain percentage of the variation

29.1.2 Calculating

- Calculate the covariance matrix of the data
- Calculate the eigenvalues and eigenvectors of the covariance matrix
- Transform the data with the eigenvectors for the largest eigenvalues as the new basis
- The variance of feature i:

$$\sigma_i^2 = \sigma_{ii} = \frac{1}{N}\sum_{k=1}^{N}(x_{ki} - \mu_i)^2 \tag{30}$$

- The covariance between feature i and j:

$$\sigma_{ij} = \frac{1}{N}\sum_{k=1}^{N}(x_{ki} - \mu_i)(x_{kj} - \mu_j) \tag{31}$$

- Calculating the covariance matrix
  - The covariance matrix is composed of the variances and covariances of every combination of feature:

$$\begin{bmatrix} \sigma_{11} & \sigma_{12} & \dots & \sigma_{1n} \\ \sigma_{21} & \sigma_{22} & \dots & \sigma_{2n} \\ \dots & \dots & \dots & \dots \\ \sigma_{n1} & \sigma_{n2} & \dots & \sigma_{nn} \end{bmatrix} \tag{32}$$

- The covariance eigenvectors
  - The eigenvectors v_i and eigenvalues λ_i are the n unique values of matrix C such that $\lambda_i v_= C v_i$
    - ∗ The eigenvectors of the covariance matrix describe the directions of the principal components
    - ∗ The eigenvalues tell us how large part of the total variation in the data that is accounted for by that principal component

29.1.3 Notes on PCA

- PCA is a linear transformation
  - Does not directly help with data that is not linearly separable
  - However, may make learning easier because of reduced complexity
- PCA removes some information from the data
  - Might just be noise
  - Might provide helpful nuances that may be of help to some classifiers

30 Supervised Learning

- Training data sets are available that consists of a collection of labelled *target* data.
- It enables us to show the algorithm the correct answer to sample data.
- *In many circumstances, it is difficult to obtain*.

31 Unsupervised Learning

- Suppose we don't have good training data
  - Hard and boring to generate targets
  - Don't always know targets
- Biologically implausible to have targets?
- No information about the correct outputs.
- The algorithm is left to spot some similarity between different inputs by itself.
- We can't hope to perform regression.
- Can we hope to do classification?
  - If the algorithms can exploit similarities between inputs in order to cluster inputs that are similar together, this might perform classification automatically.
- We have no external error information
  - *No task-specific error criterion*
- Usual method is to cluster data together according to activation of neurons:
  - *Competitive learning*

31.1 Competitive learning

- Set of neurons compete to fire
- Neuron that 'best matches' the input (has the highest activation) fires
  - Winner-take-all

32 Real-world Application of Unsupervised Clustering/Learning

Manufacturing : Groups people of similar sizes together to make "small", "medium" and "large" T-Shirts.

Marketing : Segment customers according to their similarities to do targeted marketing.

Data mining : E.g. given a collection of text documents, organize them according to their content similarities or find similarity between human motion and music being played.

33 The K-Means Algorithm

- Suppose that you know the number of clusters, but not what the clusters look like
- How do you assign each data point to a cluster?
  - Position k centres at random in the space
  - Assign each point to its nearest centre according to some chosen distance measure
  - Move the centre to the mean of the points that it represents
  - Iterate

33.1 Euclidean Distance

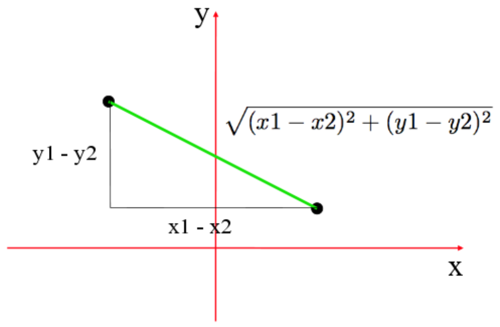


Figure 124:

33.2 K-Means Clustering

- The meaning of 'K-means'
  - Why it is called 'K-means' clustering: K points are used to represent the clustering result; each point corresponds to the centre (mean) of a cluster
- The number K, must be specified

33.3 Algorithm

1. Pick a number, k, of cluster centers (at random, do not have to be data items)

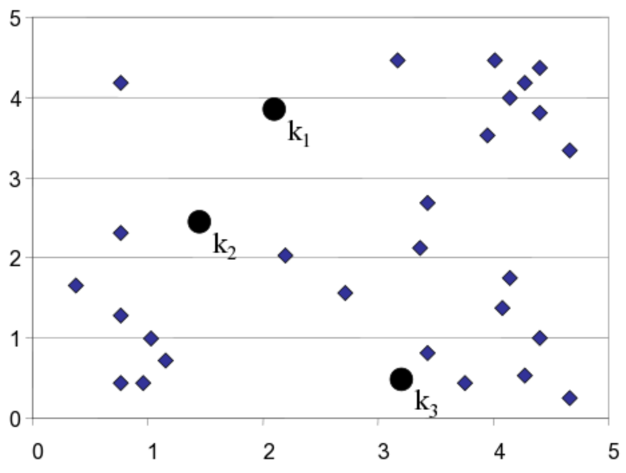


Figure 125: Algorithm: k-means, Distance Metric: Euclidean Distance

2. Assign every item to its nearest cluster center (e.g., using Euclidean distance)

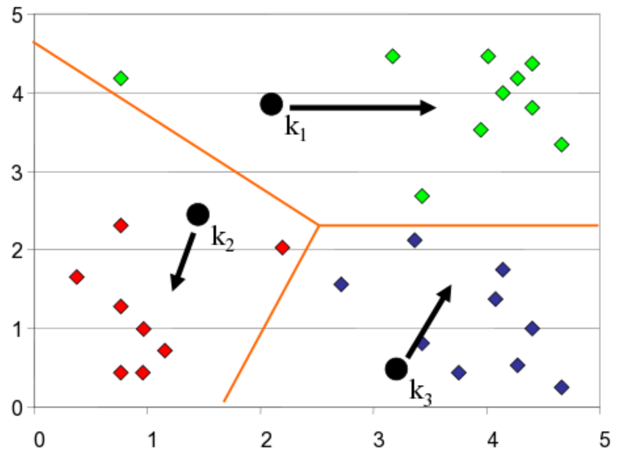


Figure 126: Algorithm: k-means, Distance Metric: Euclidean Distance

3. Move each cluster center to the mean of its assigned items

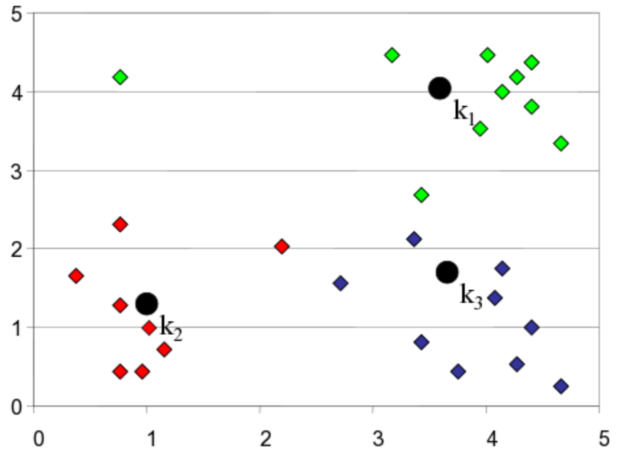


Figure 127: Algorithm: k-means, Distance Metric: Euclidean Distance

4. Repeat steps 2 and 3 until convergence (e.g., change in cluster assignments less than a threshold)

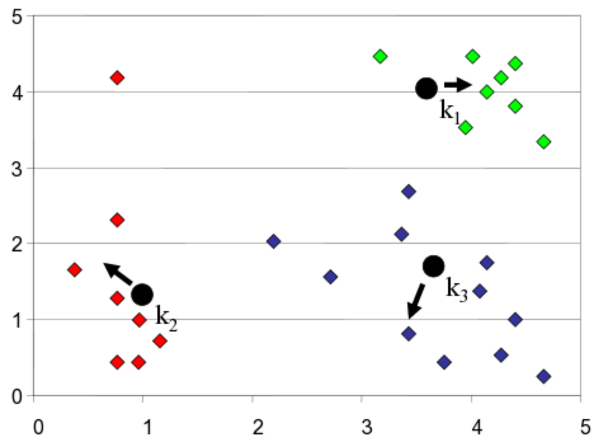


Figure 128: Algorithm: k-means, Distance Metric: Euclidean Distance

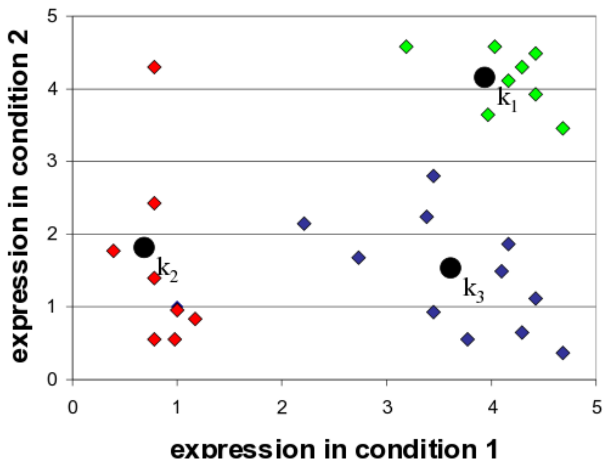


Figure 129: Algorithm: k-means, Distance Metric: Euclidean Distance

33.4 K-Means Discussion

- Result can vary significantly depending on initial choice of seeds.
- Can get trapped in local minimum
  - Example:

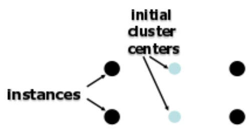


Figure 130:

  - Restart with different random seeds
- Does not handle outliers well

33.5 Algorithm

- One solution is to run the algorithm for many values of k
  - Pick the one with lowest error
  - Up to overfitting
- Run the algorithm from many starting points
  - Avoids local minima?
- What about noise?
  - Median instead of mean?

33.6 Dealing With Noise

- Mean avearge is very suspectible to *outliers* (i.e. very noisy environments).
- Replace the mean with the *median*.
  - Not affected by *outliers*.
- 1, 2, 1, 2, 100
- Mean = 21.2
  - Very much affected by outlier.
- Median= 2
  - Much better result.

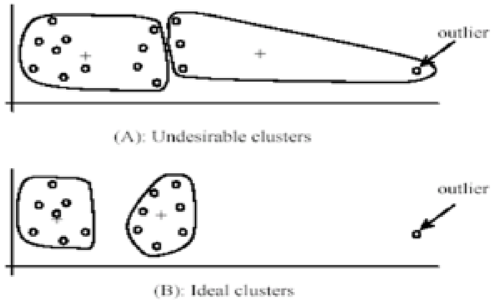


Figure 131:

33.7 Strengths

- Simple: easy to understand and to implement
- Efficient: Time complexity: $O(tkn)$,
  where n is the number of data points, k is the number of clusters, and t is the number of iterations.
- Since both k and t are small. k-means is considered a linear algorithm.

33.8 Weaknesses

- Weak in handling noisy data and outliers
- Very large or very small values could skew the mean
- Not suitable to discover clusters with non-convex shapes
- Need to specify k, the $number$ of clusters, in advance

34 Self-Organizing Feature Map (SOM)

- SOMs is a data visualization technique which *reduce the dimensions of data* through the use of self-organizing neural networks.
- The problem that data visualization attempts to solve is that humans simply cannot visualize high dimensional data.
- The way SOMs go about reducing dimensions is by producing a map of usually 1 or 2 dimensions which plot the similarities of the data by grouping similar data items together.
- So SOMs accomplish two things,
  - they reduce dimensions, and
  - display similarities.
- It is a type of artificial neural network (ANN) that is trained using unsupervised learning.
  - No human intervention is needed during the learning and little needs to be known about the characteristics of the input data.
- In addition, it creates a network that stores information in such a way that any topological relationships (relative ordering preservation) within the training set are maintained.
  - Neurons that are close together represent inputs that are close together, while neurons that are far apparat represent inputs that are far apart.
- SOM uses *competitive learning* algorithm.
  - Only one output neuron activated at any one time.
  - Winner-takes-all neuron or winning neuron.

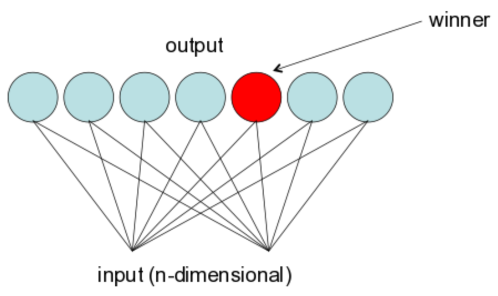


Figure 132:

- The goal of learning in the SOM is to cause different parts of the network to respond input patterns being far apart.
  - This is partly motivated by how visual, auditory or other sensory information is handled in separate parts of the cerebral cortex in the human brain.
- The model was first described as an ANN by the Finnish professor Teuvo Kohonen, and is sometimes called a *Kohonen map*.

34.1 Feature Maps

- Sounds that are similar ('close together') excite neurons that are near to each other
- Sounds that are very different excite neurons that are a long way off
- This is known as *topology preservation*
- The ordering of the inputs is preserved (if possible perfectly topology-preserving)

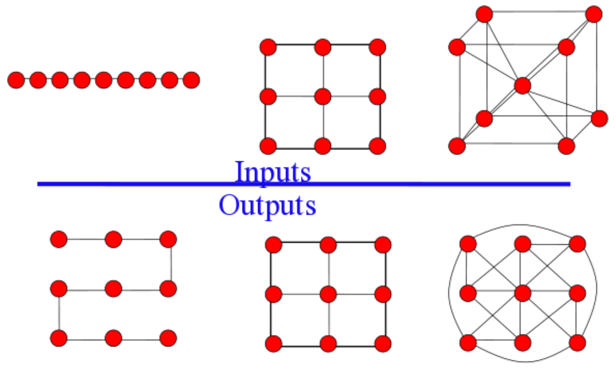


Figure 133: Topology Preservation

34.2 SOM Structure

- Like most ANNs, SOMs operate in two modes:
  - Training, and
  - Mapping (apply the network for e.g. classification)
- Training builds the map using input examples. It is a *competitive process*.
- Mapping automatically classifies a new input vector.
- SOM consists of components called nodes or neurons.
- Associated with each node is a weight vector of the same dimension as the input data vectors and a position in the map space.
- Topology-conserving mapping can be achieved by SOMs:
  - Two layers: input layer and output (map) layer.
  - Input and output layers are completely connected.
  - Output neurons are interconnected within a defined neighborhood.
  - A topology (neighborhood relation) is defined on the output layer.

34.3 SOM Architecture

- Neurons ('nodes') in the 2D map respond to set of input signals
- Responses compared; 'winning' neuron selected
- Selected neuron activated together with 'neighbourhood' neurons
- Adaptive process changes weights to more closely resemble inputs

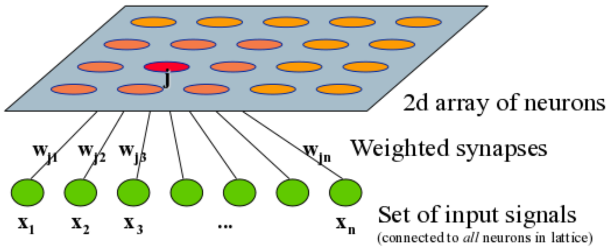


Figure 134:

- The network is created from a 2D lattice of 'nodes', each of which is fully connected to the input layer.

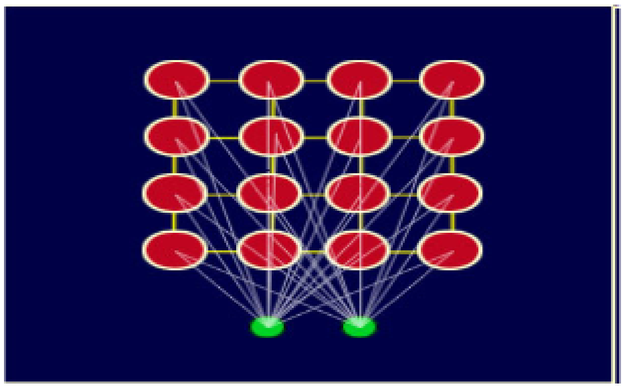


Figure 135:

- Figure shows a very small Kohonen network of 4 X 4 nodes connected to the input layer (shown in green) representing a 2D vector.
- Each node has a specific topological position (an x, y coordinate in the lattice) and contains a vector of weights of the same dimension as the input vectors.

34.4 SOM (Kohonen Maps)

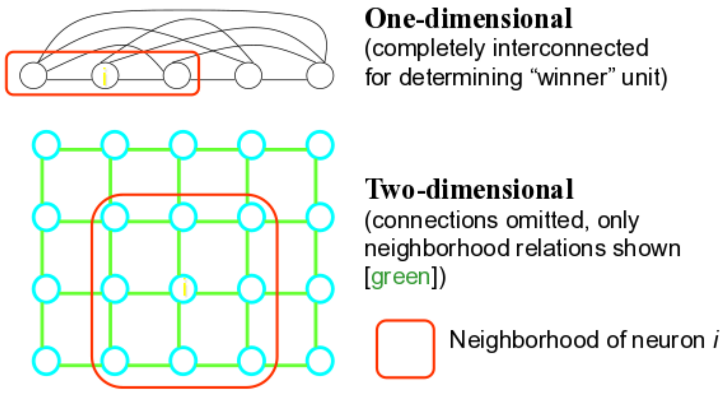

One-dimensional
(completely interconnected for determining "winner" unit)
i
Two-dimensional
(connections omitted, only neighborhood relations shown [green])
Neighborhood of neuron *i*

Figure 136: Common output-layer structures

34.4.1 The Goal

- We have to find values for the weight vectors of the links from the input layer to the nodes of the lattice, in such a way that adjacent neurons will have similar weight vectors.
- For an input, the output of the neural network will be the neuron whose weight vector is most similar (with respect to Euclidean distance) to that input.
- In this way, each (weight vector of a) neuron is the center of a cluster containing all the input examples which are mapped to that neuron.

34.4.2 The Learning Process

1. - An informal description:
     - Given: an input pattern x
     - Find: the neuron i which has closest weight vector by competition ($w_i^T x$ will be the highest).
     - For each neuron j in the neighbourhood $N(i)$ of the winning neuron i:
       update the weight vector of j.
2. - Neurons which are not in the neighbourhood are left unchanged.
   - The SOM algorithm:
     - Starts with large neighbourhood size and gradually reduces it.
     - Gradually reduces the learning rate η.
3. - Upon repeated presentations of the training examples, the weight vectors tend to follow the distribution of the examples.
   - This results in a topological ordering of the neurons, where neurons adjacent to each other tend to have similar weights.
4. - There are basically three essential processes:
     - competition
     - cooperation

  - weight adaption

5.
- Each neuron in an SOM is assigned a weight vector with the same dimensionality N as the input space.
- Any given input pattern is compared to the weight vector of each neuron and the closest neuron is declared the winner.
- The Euclidean norm is commonly used to measure distance.

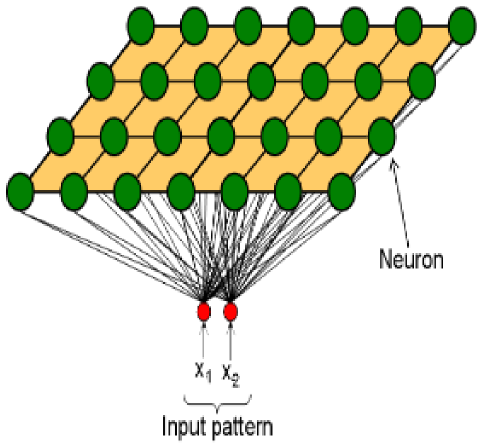


Figure 137:

6.
- The activation of the winning neuron is spread to neurons in its immediate neighborhood. This allows topologically close neurons to become sensitive to similar patterns.
- The winning neuron locates the center of a topological neighbourhood of cooperating neurons.
- The size of the neighborhood is initially large, but shrinks over time.
  - An initially large neighborhood promotes a topology-preserving mapping. Smaller neighborhoods allow neurons to specialize in the latter stages of training.
- The point of the topographic neighbourhood is that not only the winning neuron gets its weights updated, but its neighbours will have their weights updated as well, although by not as much as the winner itself.
- The effect of each learning weight update is to move the weight vectors w_i of the winning neuron and its neighbours towards the input vector x.
- Repeated presentations of the training data thus leads to topological ordering.

34.4.3 Network Size

- We have to predetermine the network size
- Big network
  - Each neuron represents exact feature
  - Not much generalisation
- Small network
  - Too much generalisation

  - No differentiation

- Try different sizes and pick the best

34.4.4 Advantages

- Better visualization and interpretation.
- Excellent for classification problems.
- Can greatly reduce computational complexity.
- High sensitivity to frequent inputs.
- New ways of associating related data.
- No need of supervised learning rules.

34.4.5 Disadvantages

- System is a black box
- Many problems can't be effectively represented by a SOFM
- A large training set may be required
- For large classification problems, training can be lengthy
- Can be difficult to come up with the input vector.
- Associations developed by SOFM not always easily understood by people.

35 Reinforcement Learning

A child learning to walk

- Tries out many different strategies
- Some do not work
  - The child falls
  - Strategy discarded
- Some seem to work
  - The child stays up longer than before
  - Strategy kept until replaced by something better

35.1 Reinforcement Learning Loop

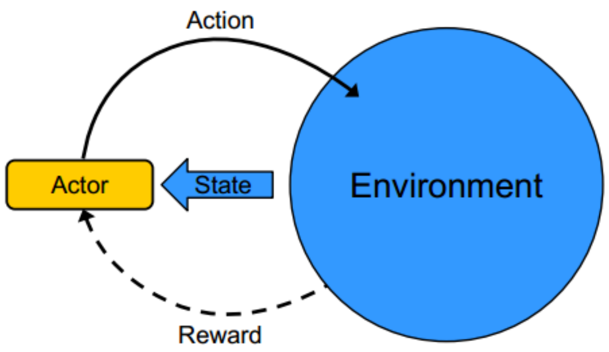


Figure 138:

35.2 Toy problem

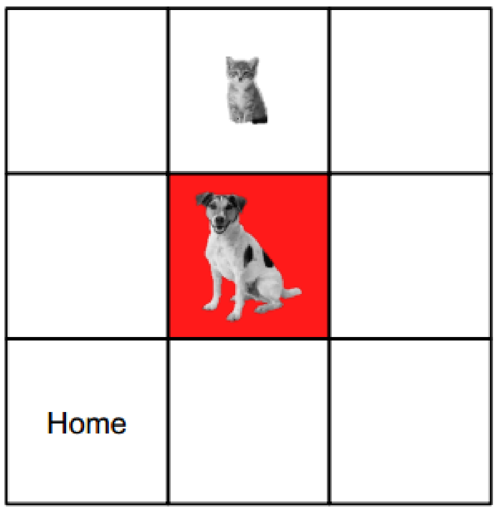


Figure 139:

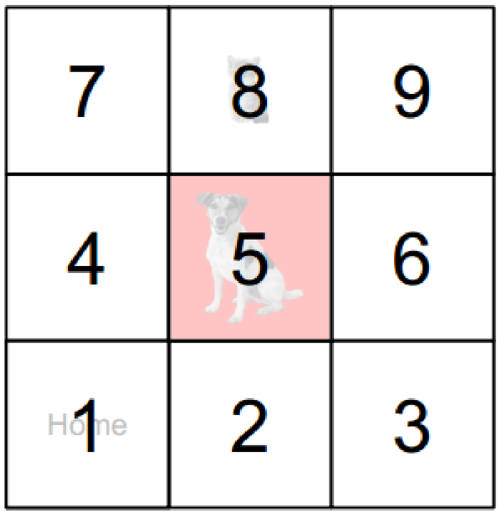


Figure 140: States

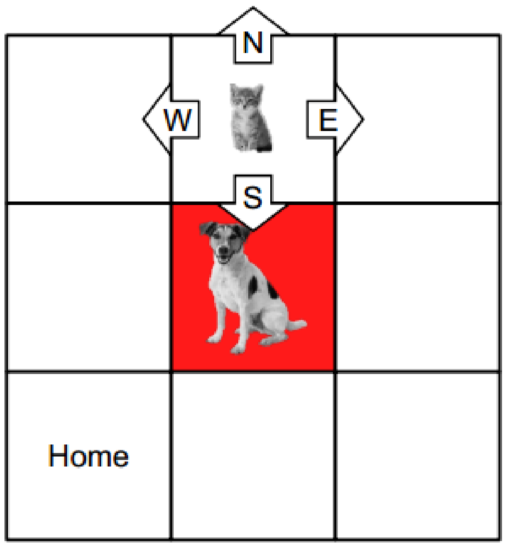


Figure 141: Actions

35.2.1 State and action spaces

- The size of these spaces can be really large
  - 5 integer values between 1 and 100
    Size of space: 100 x 100 x 100 x 100 x 100
- Specifying the spaces well is crucial
  - Quantize the integers to two classes
    Size of space: 2 x 2 x 2 x 2 x 2
- Toy problem:
  - 9 states and 4 actions - 36 possible combinations

35.2.2 Rewards

Taking an action in some state results in an *immediate reward* (can be negative)

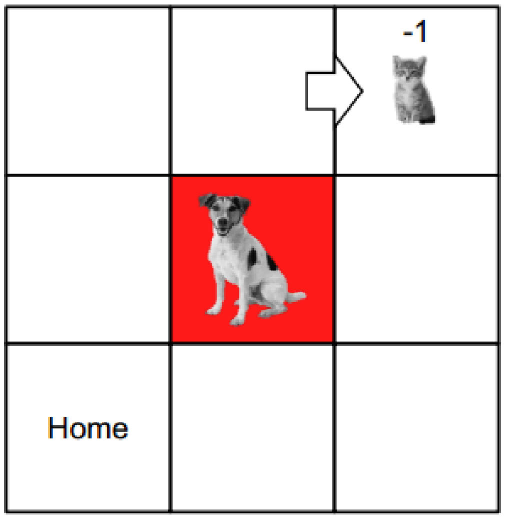


Figure 142: Rewards

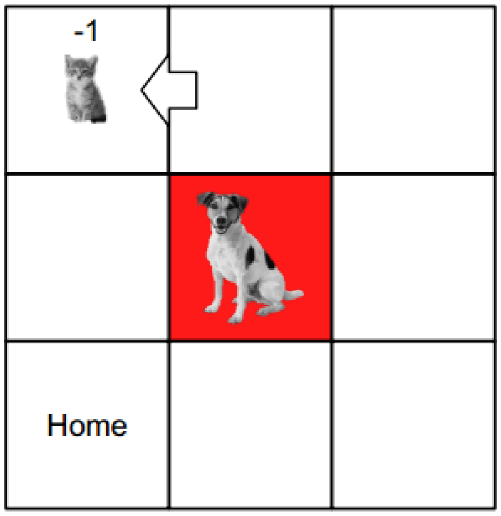


Figure 143: Rewards

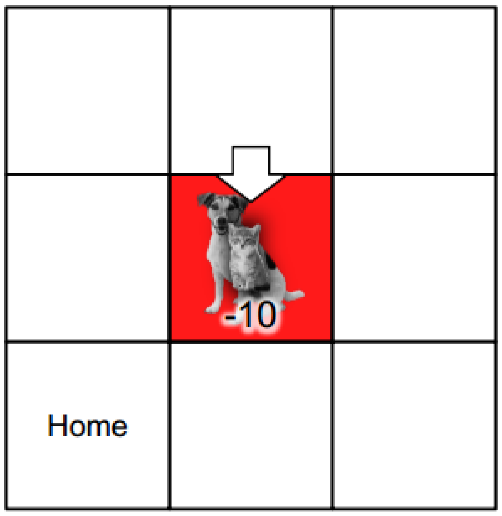


Figure 144: Rewards

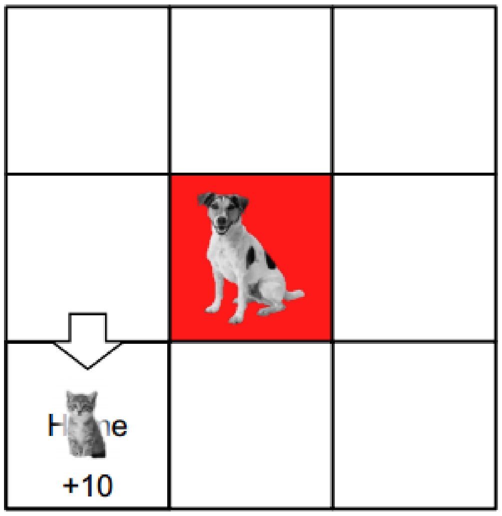


Figure 145: Rewards

35.3 Reward System

Should tell the agent what to achieve (not how to achieve it)
The agent should choose the action that expected to give maximum long term reward.

35.4 Long Term Reward

But future rewards are uncertain!

- Multiply by a discount factor γ for each action

$$R_n = \sum_{i}^{n} \gamma^i r_i \tag{33}$$

- γ is between 0 and 1 so that $\lim_{k\to\infty}\gamma^k = 0$
  - γ = 0 : very shortsighted
  - γ = 1 : very farsighted

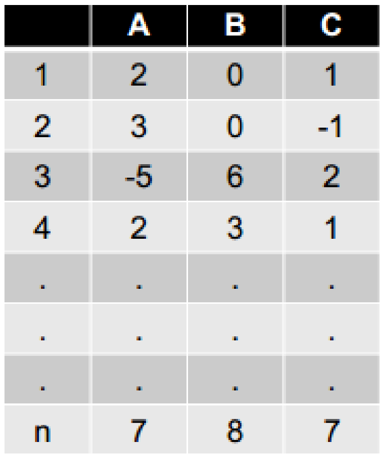
35.5 Reward estimation

These expected rewards are not known to the agent beforehand!

- The agent has to act somehow
  - How should the agent act?
  - How can the agent estimate future rewards?
- Keep a value for each possible action in each possible state!
  - This value will estimate the expected reward of carrying out that action when in that state
- How do we get these values? Estimate the rewards while acting!

35.6 The Q table and policies

- Estimated rewards are kept in a table $Q(s,a)$
  - The agent selects actions using these values based on a policy
  - The policy $\pi(s,a)$ gives the probability of choosing an action given a state



	A	B	C
1	2	0	1
2	3	0	-1
3	-5	6	2
4	2	3	1
.	.	.	.
.	.	.	.
.	.	.	.
n	7	8	7

Figure 146:

35.7 Common policies

- **Greedy** – always choos best action

$$\pi(s,a) = \begin{cases} 1 & a = argmax_a\{Q(s,a)\} \\ 0 & m\text{otherwise} \end{cases} \tag{34}$$

- ϵ**-greedy** – choose randomly $\frac{1}{\epsilon}$ of the time

$$\pi(s,a) = \begin{cases} 1 - \frac{1}{\epsilon} & a = argmax_a\{Q(s,a)\} \\ \frac{1}{\epsilon n_a} & \text{otherwise} \end{cases} \tag{35}$$

- **Soft-max** – the probability is related to reward

$$\pi(s,a) = \frac{e^{Q(s,a)}}{\sum_i e^{Q(s,a_i)}} \tag{36}$$

35.8 Reward estimation

- After taking action a_i in state s_i we get the reward r_i, and the state changes to s_{i+1}
  - We can use the experienced reward r_i and the best estimated reward $Q(s_{i+1}, a_{i+1}$ to make a new estimate $Q(s_i, a_i) = r_i + \gamma Q(s_{i+1}, a_{i+1})$ easd
  - We don't want to change the estimates too quickly, so we "blend" the old and the new estimates with the learning rate, μ:

$$Q(s_i, a_i) \leftarrow Q(s_i, a_i) + \mu(r_i + \gamma Q(s_{i+1}, a_{i+1}) - Q(s_i, a_i)) \tag{37}$$

If we rewrite just a little:

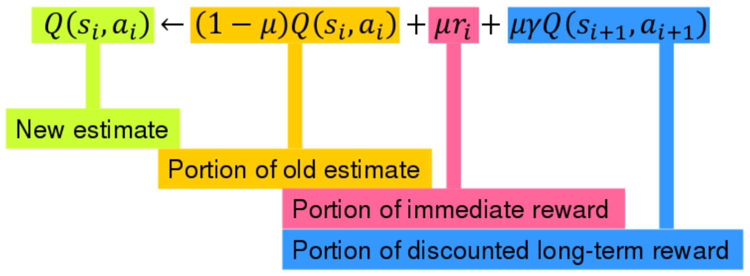

$Q(s_i, a_i) \leftarrow (1-\mu)Q(s_i, a_i) + \mu r_i + \mu\gamma Q(s_{i+1}, a_{i+1})$
New estimate
Portion of old estimate
Portion of immediate reward
Portion of discounted long-term reward

Figure 147:

35.9 Q-learning

```
def q−learn(states, actions, policy):
  Q = small_random_matrix(states, actions)
  while (not done):
    s0 = start()
    while (not episode_done):
      a = policy(Q, s0)
      s1 = take_action(a)
      update(Q, s0, a, s1)
      s0 = s1
```

35.10 On-policy vs. off-policy

- The update formula used so far isn't affected by the policy used
  (off-policy learning)
- If we use the policy to decide what the next action should be, we can use the complete tuple $(s_i, a_i, r_i, s_{i+1}, a_{i+1})$ to update Q
  (on-policy learning)

35.11 SARSA

```
def sarsa(states, actions, policy):
  Q = small_random_matrix(states, actions)
  while (not done):
    s0 = start()
    a0 = policy(Q,s0)
    while (not episode_done):
      s1 = take_action(a0)
      a1 = policy(Q, s1)
      update(Q, s0, a0, s1, a1)
      s0 = s1
      a0 = a1
```

Index