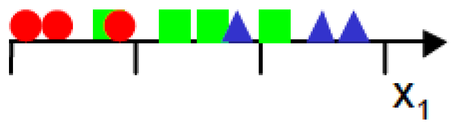
Computational Intelligence & Machine Learning

Dimensionality Reduction

Adopted from Ricardo Gutierrez-Osuna

Introduction

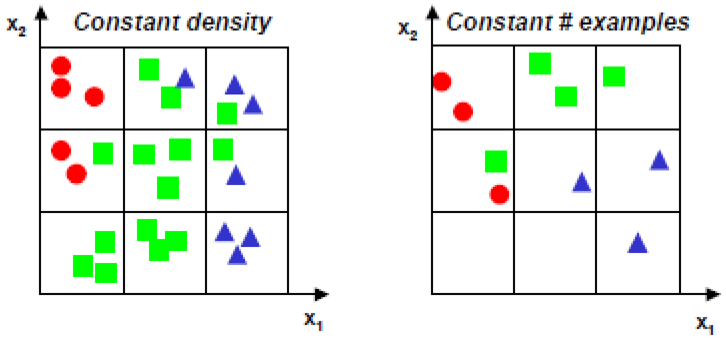
- **The “curse of dimensionality”**
  - Refers to the problems associated with multivariate data analysis as the dimensionality increases
- **Consider a 3-class pattern recognition problem**
  - Three types of objects have to be classified based on the value of a single feature:



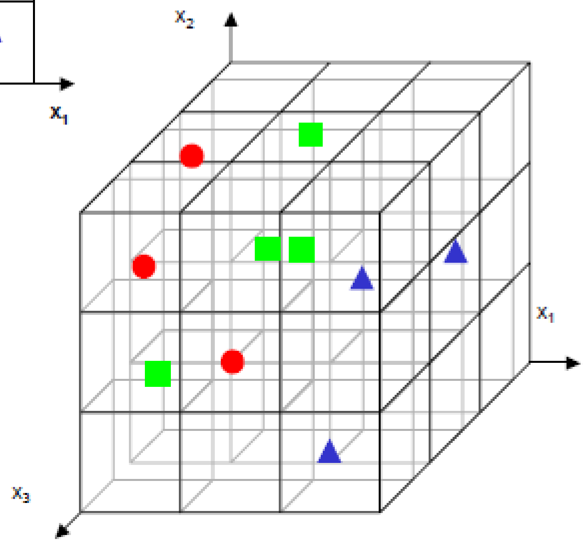
  - A simple procedure would be to
    - Divide the feature space into uniform bins
    - Compute the ratio of examples for each class at each bin and,
    - For a new example, find its bin and choose the predominant class in that bin
  - We decide to start with one feature and divide the real line into 3 bins
    - Notice that there exists a lot of overlap between classes ⇒ to improve discrimination, we decide to incorporate a second feature

Example

- **Moving to two dimensions increases the number of bins from 3 to 3^2=9**
  - QUESTION: Which should we maintain constant?
    - The density of examples per bin? This increases the number of examples from 9 to 27
    - The total number of examples? This results in a 2D scatter plot that is very sparse

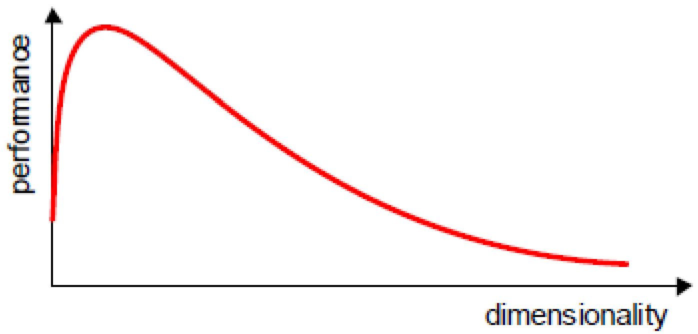


- **Moving to three features …**
  - The number of bins grows to 3^3=27
  - To maintain the initial density of examples, the number of required examples grows to 81
  - For the same number of examples the 3D scatter plot is almost empty



Implications

- **Implications of the curse of dimensionality**
  - Exponential growth with dimensionality in the number of examples required to accurately estimate a function
- **In practice, the curse of dimensionality means that**
  - For a given sample size, there is a maximum number of features above which the performance of our classifier will degrade rather than improve
    - In most cases, the information that was lost by discarding some features is compensated by a more accurate mapping in lower-dimensional space



- **How do we beat the curse of dimensionality?**
  - By incorporating prior knowledge
  - By providing increasing smoothness of the target function
  - By reducing the dimensionality

Solutions

- **Two approaches to perform dim. reduction $\Re^N \rightarrow \Re^M$ (M<N)**
  - **Feature selection**: choosing a subset of all the features

$$[x_1\ x_2 \ldots x_N] \xrightarrow{\text{feature selection}} [x_{i_1}\ x_{i_2} \ldots x_{i_M}]$$

  - **Feature extraction**: creating new features by combining existing ones

$$[x_1\ x_2 \ldots x_N] \xrightarrow{\text{feature extraction}} [y_1\ y_2 \ldots y_M] = f([x_{i_1}\ x_{i_2} \ldots x_{i_M}])$$

    - In either case, the goal is to find a low-dimensional representation of the data that preserves (most of) the information or structure in the data

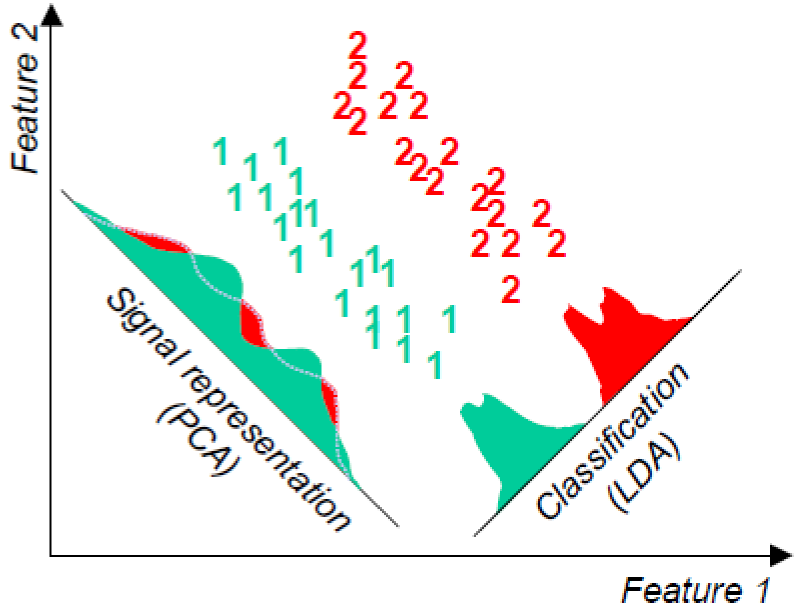
- **Linear feature extraction**
  - The "optimal" mapping y=f(x) is, in general, a non-linear function whose form is problem-dependent
    - Hence, feature extraction is commonly limited to linear projections **y=Wx**

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ \\ x_N \end{bmatrix} \xrightarrow{\text{linear feature extraction}} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_M \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1N} \\ w_{21} & w_{22} & \cdots & w_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ w_{M1} & w_{M2} & & w_{MN} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ \\ x_N \end{bmatrix}$$

Solutions

- **Two criteria can be used to find the "optimal" feature extraction mapping y=f(x)**
  - **Signal representation**: The goal of feature extraction is to represent the samples accurately in a lower-dimensional space
  - **Classification**: The goal of feature extraction is to enhance the class-discriminatory information in the lower-dimensional space

- **Within the realm of linear feature extraction, two techniques are commonly used**
  - Principal Components (PCA)
    - Based on signal representation
  - Fisher's Linear Discriminant (LDA)
    - Based on classification



Principal Components Analysis

Applications

- Data Visualization
- Data Compression
- Noise Reduction
- Data Classification
- Trend Analysis
- Factor Analysis

Example

- Given 53 blood and urine samples (features) from 65 people.

- How can we visualize the measurements?

Example

Matrix format (65x53)

	H-WBC	H-RBC	H-Hgb	H-Hct	H-MCV	H-MCH	H-MCHC
A1	8.0000	4.8200	14.1000	41.0000	85.0000	29.0000	34.0000
A2	7.3000	5.0200	14.7000	43.0000	86.0000	29.0000	34.0000
A3	4.3000	4.4800	14.1000	41.0000	91.0000	32.0000	35.0000
A4	7.5000	4.4700	14.9000	45.0000	101.0000	33.0000	33.0000
A5	7.3000	5.5200	15.4000	46.0000	84.0000	28.0000	33.0000
A6	6.9000	4.8600	16.0000	47.0000	97.0000	33.0000	34.0000
A7	7.8000	4.6800	14.7000	43.0000	92.0000	31.0000	34.0000
A8	8.6000	4.8200	15.8000	42.0000	88.0000	33.0000	37.0000
A9	5.1000	4.7100	14.0000	43.0000	92.0000	30.0000	32.0000

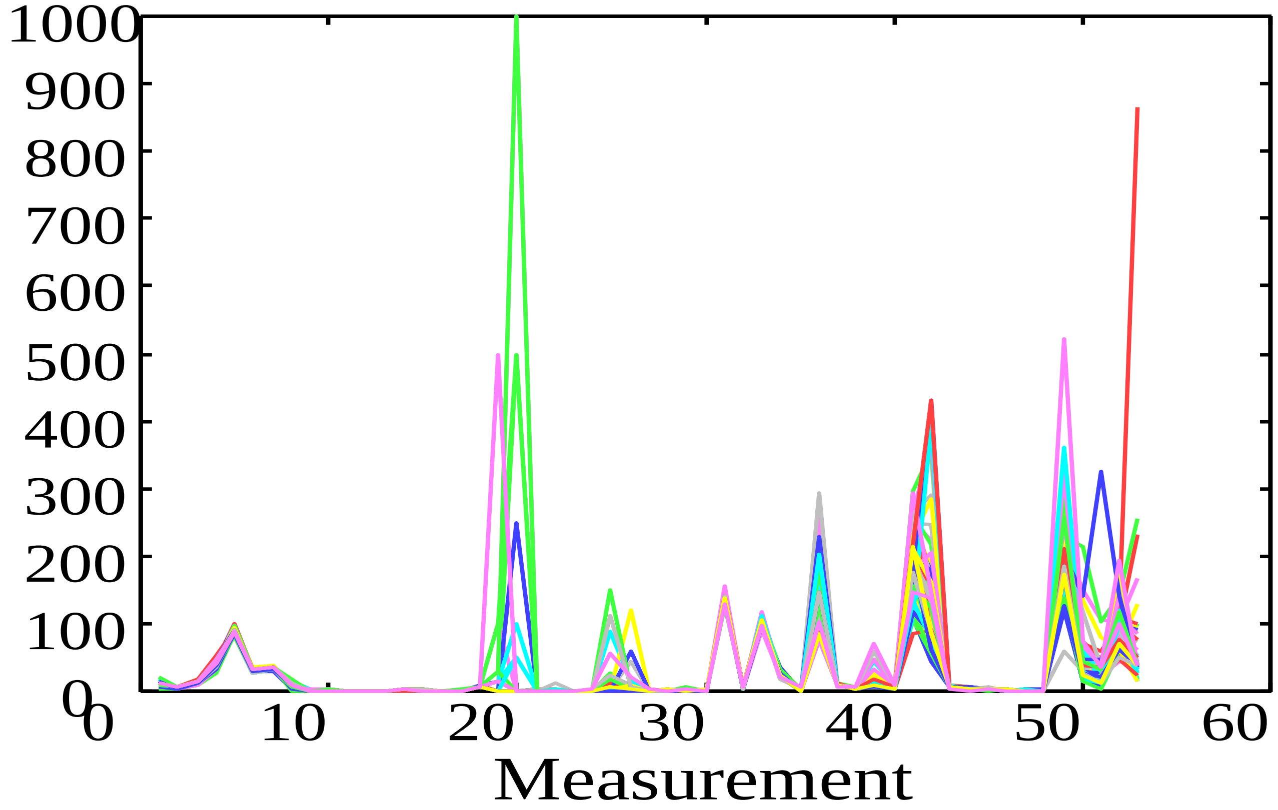
Instances

Features

Difficult to see the correlations between the features...

Example

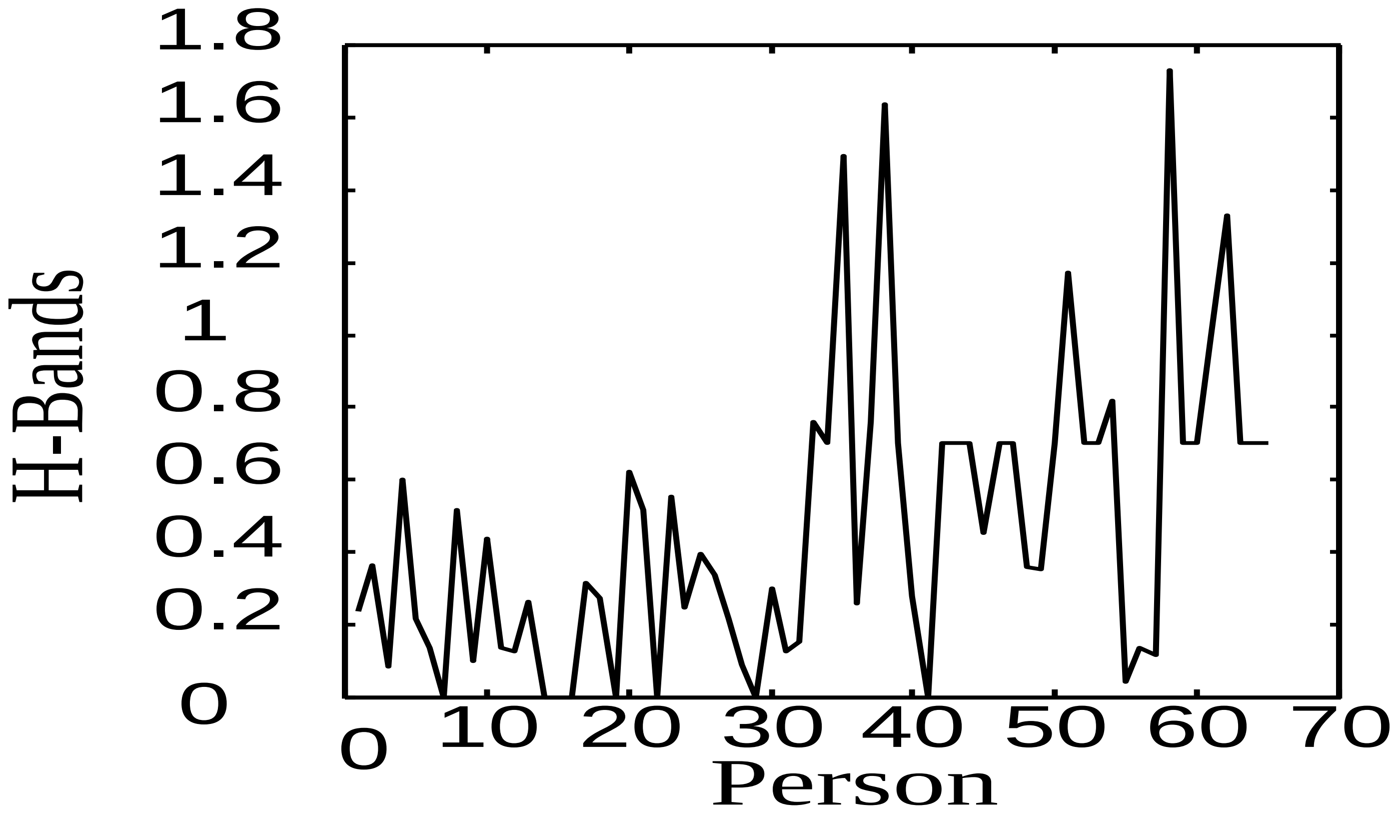
Spectral format (65 pictures, one for each person)



Difficult to compare the different patients...

Example

Spectral format (53 pictures, one for each feature)



Difficult to see the correlations between the features...

Solution

- Is there a representation better than the coordinate axes?

- Is it really necessary to show all the 53 dimensions?
  - … what if there are strong correlations between the features?

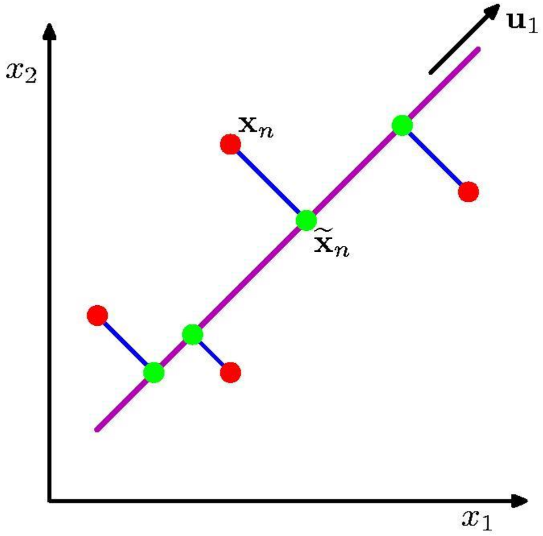
- How could we find the *smallest* subspace of the 53-D space that keeps the *most information* about the original data?

- A solution: Principal Component Analysis

PCA

- Orthogonal projection of data onto lower-dimension linear space that...
  - maximizes variance of projected data (purple line)

  - minimizes mean squared distance between
    - data point and
    - projections (sum of blue lines)



PCA

- **Idea:**
  - Given data points in a d-dimensional space, project into lower dimensional space while preserving as much information as possible
    - Eg, find best planar approximation to 3D data
    - Eg, find best 12-D approximation to 10^4-D data
  - In particular, choose projection that minimizes *squared error* in reconstructing original data

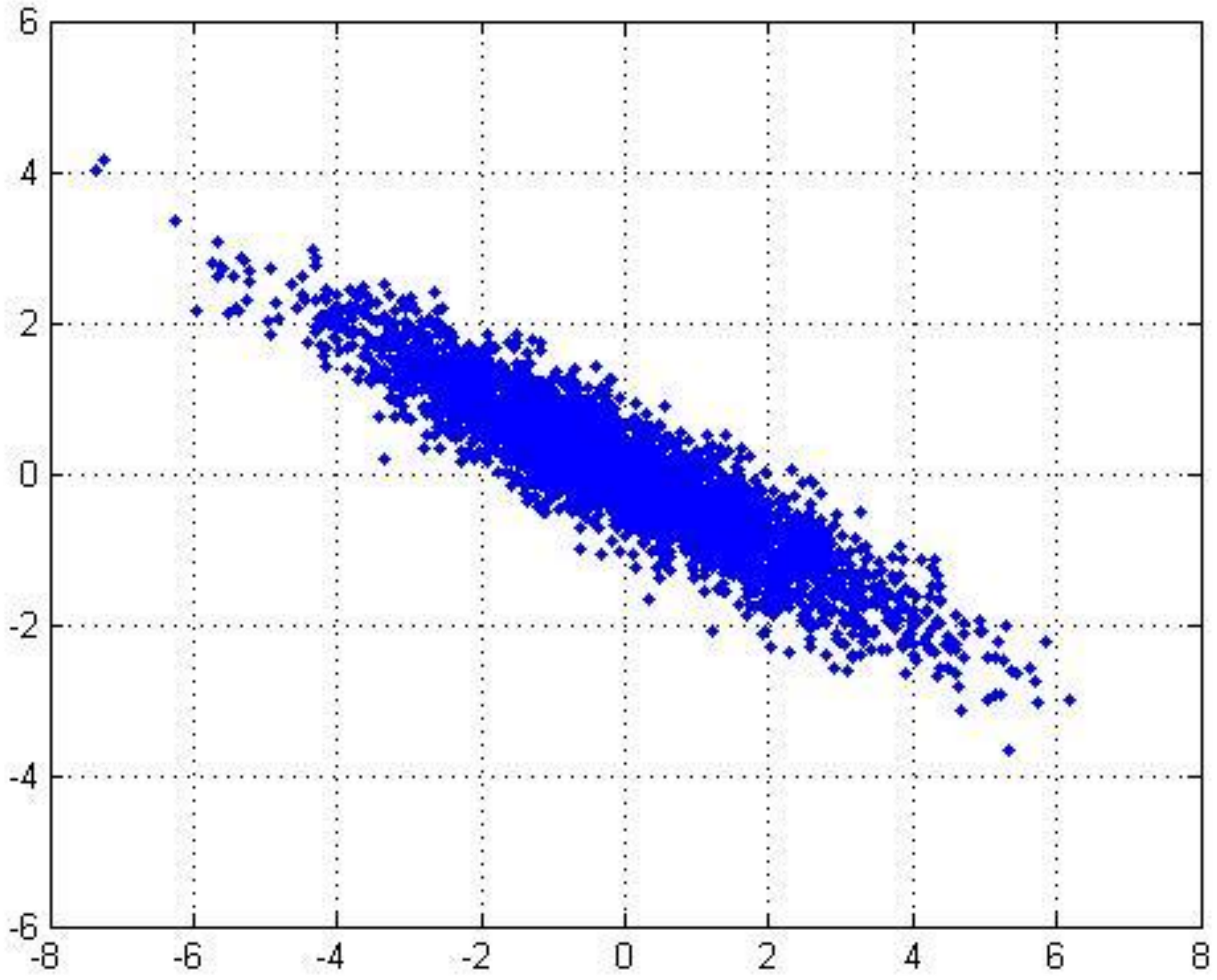
PCA

- **Vectors** originating from the center of mass

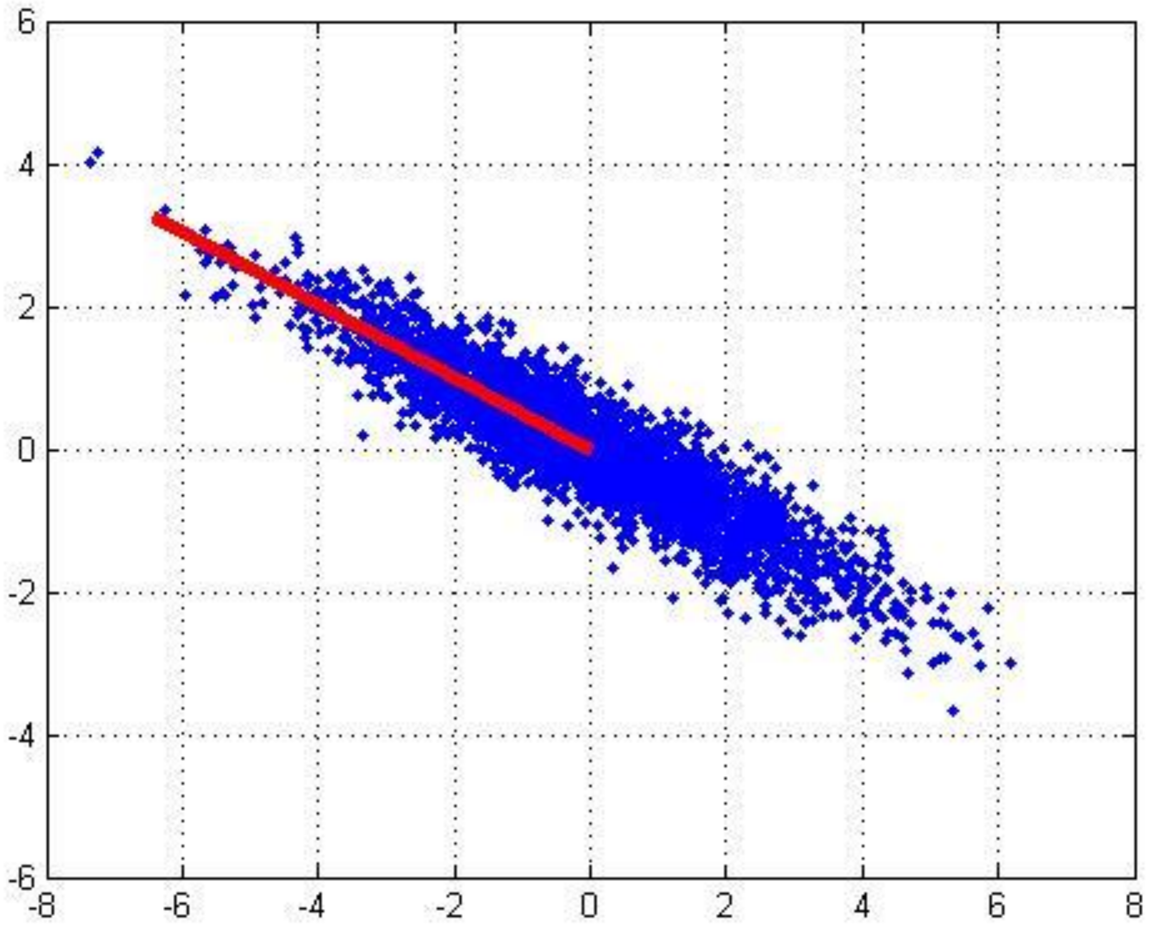
- Principal component #1 points in the direction of the **largest variance**.

- Each subsequent principal component…
    - is **orthogonal** to the previous ones, and
    - points in the directions of the **largest variance of the residual subspace**

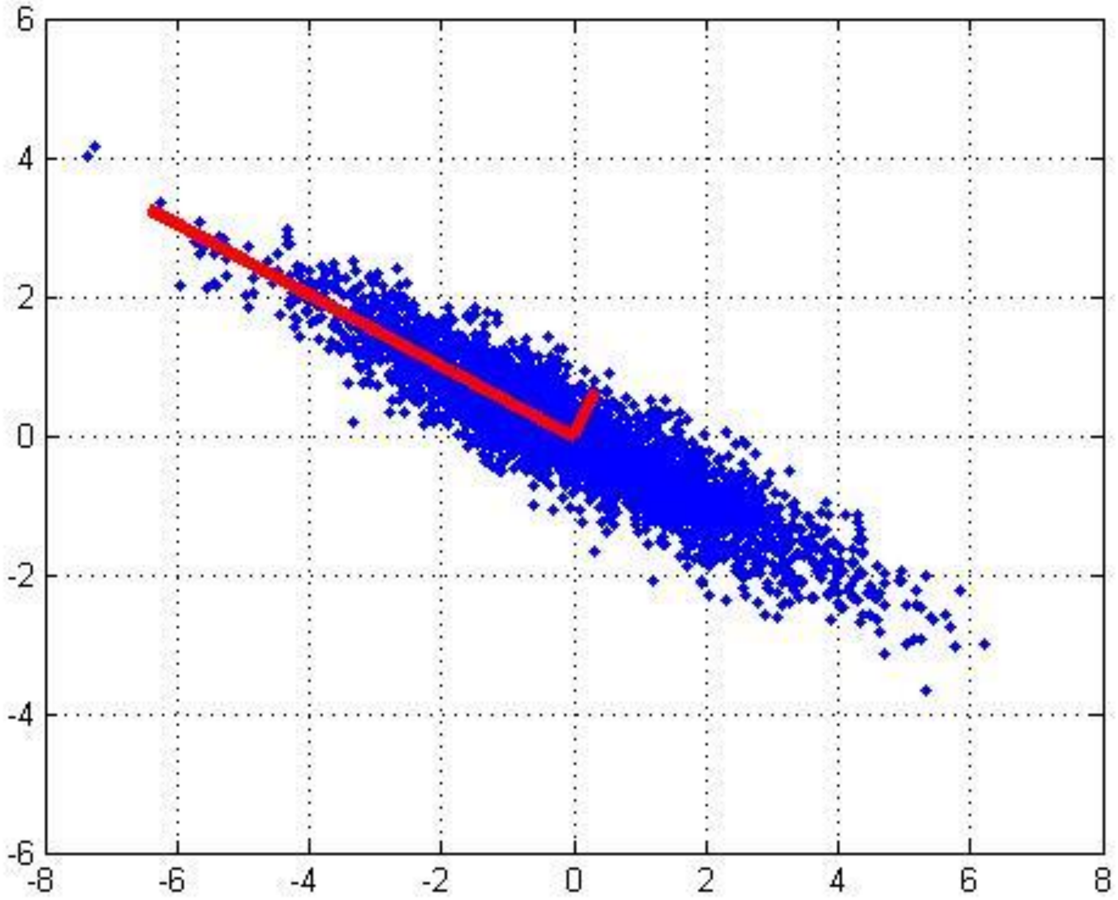
Calculations



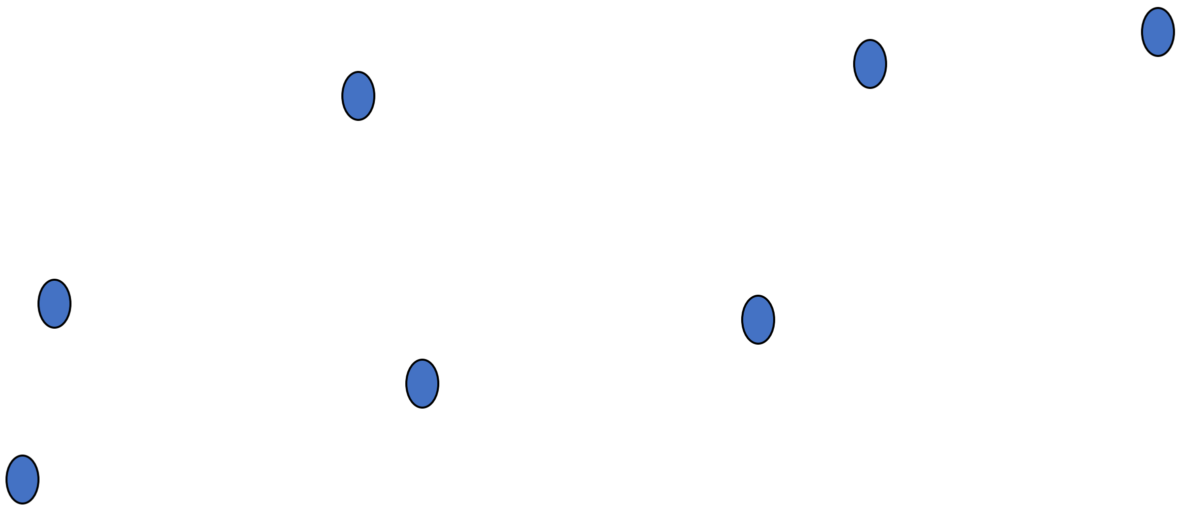
Calculations



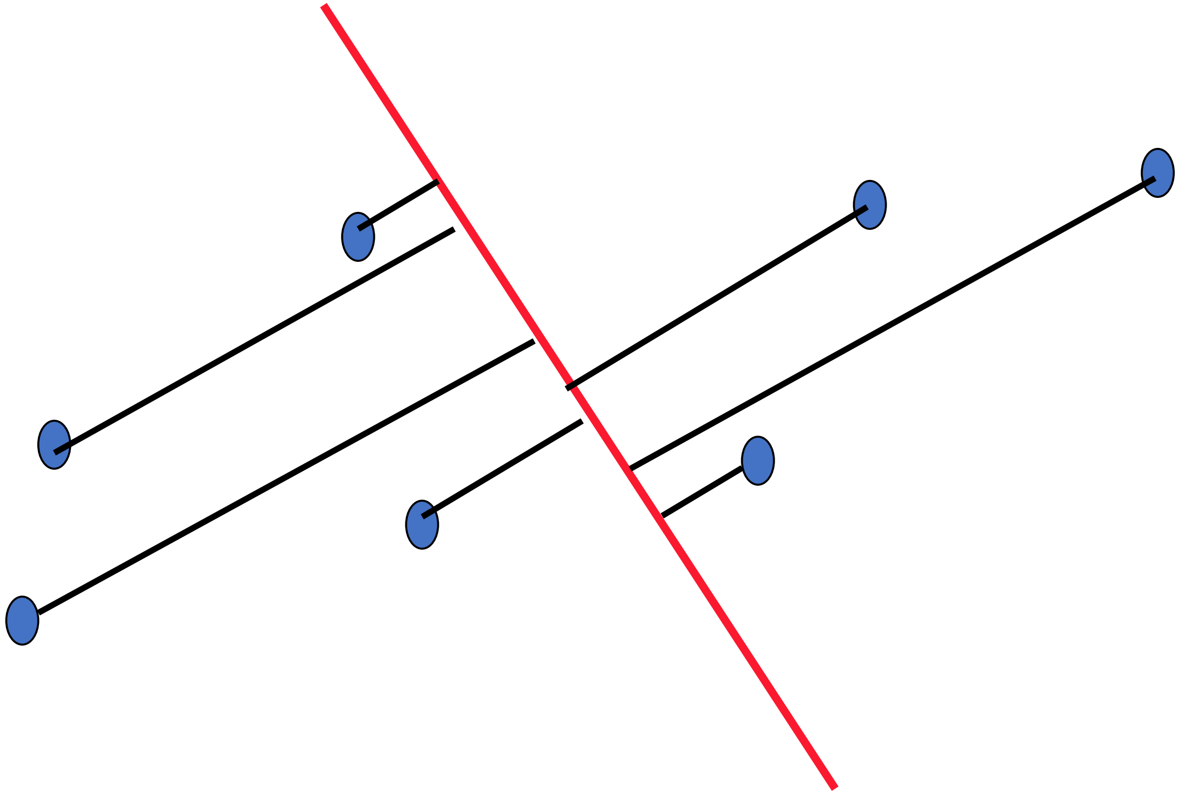
Calculations



Calculations

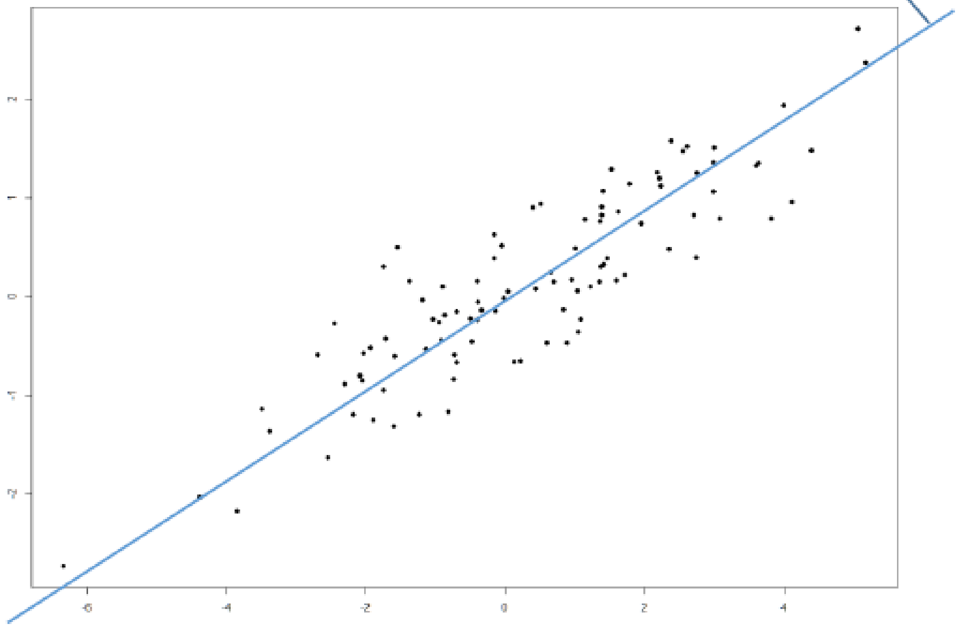


Calculations

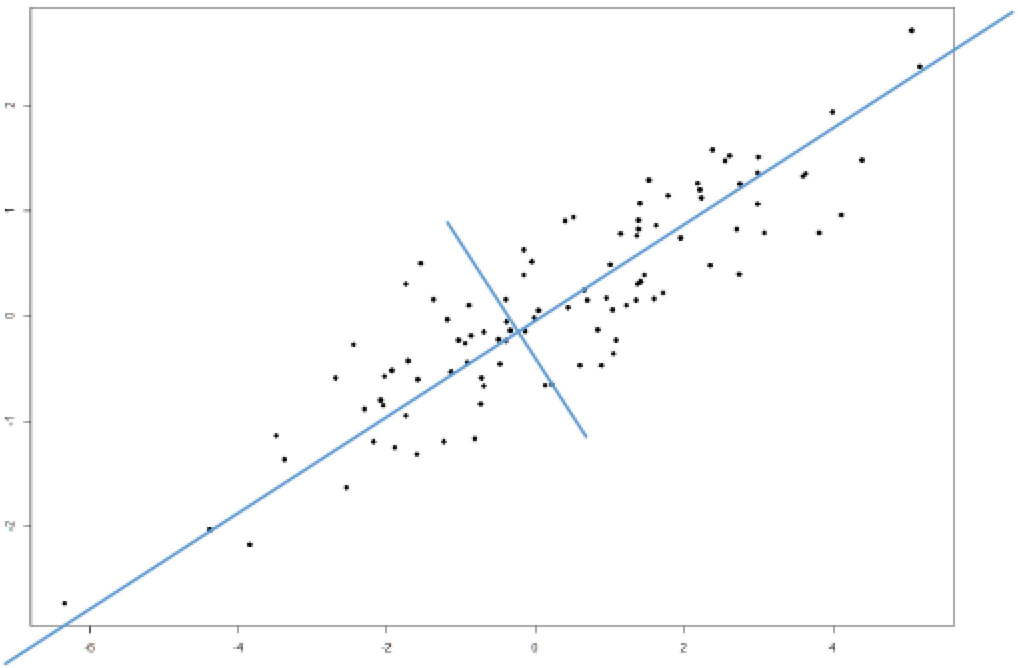


Calculations

If we project the data onto this line, we lose as little information as possible = we keep as much variance as possible.



Calculations



Calculations

Given a sample of *n* observations on a vector of *p* variables

$$\{x_1, x_2, \cdots, x_n\} \in \Re^d$$

define the first principal component of the sample by the linear transformation

$$z_1 = a_1^T x_j = \sum_{i=1}^{d} a_{i1} x_{ij}, \quad j = 1,2,\cdots,n.$$

where the vector

$$a_1 = (a_{11}, a_{21}, \cdots, a_{d1})$$
$$x_j = (x_{1j}, x_{2j}, \cdots, x_{dj})$$

is chosen such that $\mathrm{var}[z_1]$ is maximum.

Calculations

To find a_1 first note that

$$\text{var}[z_1] = E((z_1 - \overline{z_1})^2) = \frac{1}{n}\sum_{i=1}^{n}\left(a_1^T x_i - a_1^T \overline{x}\right)^2$$

$$= \frac{1}{n}\sum_{i=1}^{n} a_1^T \left(x_i - \overline{x}\right)\left(x_i - \overline{x}\right)^T a_1 = a_1^T S a_1$$

where $S = \frac{1}{n}\sum_{i=1}^{n}\left(x_i - \overline{x}\right)\left(x_i - \overline{x}\right)^T$

is the covariance matrix.

$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$ is the mean.

Calculations

To find a_1 that maximizes $\mathrm{var}[z_1]$ subject to $a_1^T a_1 = 1$

Let λ be a Lagrange multiplier

$$L = a_1^T S a_1 - \lambda(a_1^T a_1 - 1)$$

$$\frac{\partial}{\partial a_1} L = S a_1 - \lambda a_1 = 0$$

$$\Rightarrow (S - \lambda I_p) a_1 = 0$$

therefore a_1 is an eigenvector of S

corresponding to the largest eigenvalue $\lambda = \lambda_1$.

Calculations

We find that a_2 is also an eigenvector of S

whose eigenvalue $\lambda = \lambda_2$ is the second largest.

In general

$$\mathrm{var}[z_k] = a_k^T S a_k = \lambda_k$$

- The k^{th} largest eigenvalue of S is the variance of the k^{th} PC.
- The k^{th} PC z_k retains the k^{th} greatest fraction of the variation in the sample.

Calculations

First PC is the linear combination

$$y_1 = a_1^T x = \sum_{i=1}^{p} a_{1i} x_i$$

where a_1 is chosen such that $\boldsymbol{var(y_1)}$ **is maximum**

subject to $\boldsymbol{a_1^T a_1 = 1}$

Second PC is the linear combination

$$y_2 = a_2^T x = \sum_{i=1}^{p} a_{2i} x_i$$

where a_k is chosen such that $\boldsymbol{var(y_2)}$ **is maximum**

subject to $\boldsymbol{a_2^T a_2 = 1}$ and $\boldsymbol{a_2^T a_1 = 0 = cov(a_k, a_l)}$

Generally, k-th PC is the linear combination

$$y_k = a_k^T x = \sum_{i=1}^{p} a_k x_i$$

where a_k is chosen such that $\boldsymbol{var(y_k)}$ **is maximum**

subject to $\boldsymbol{a_k^T a_k = 1}$ and $\forall \mathbf{l}, \mathbf{l} < \mathbf{k}$: $\boldsymbol{cov(a_k, a_l) = 0}$

Steps

- Main steps for computing PCs
  - Form the covariance matrix S.
  - Compute its eigenvectors: $\{a_i\}_{i=1}^{d}$
  - The first p eigenvectors $\{a_i\}_{i=1}^{p}$ form the p PCs.
  - The transformation G consists of the p PCs:

$$G \leftarrow [a_1, a_2, \cdots, a_p]$$

Python Example

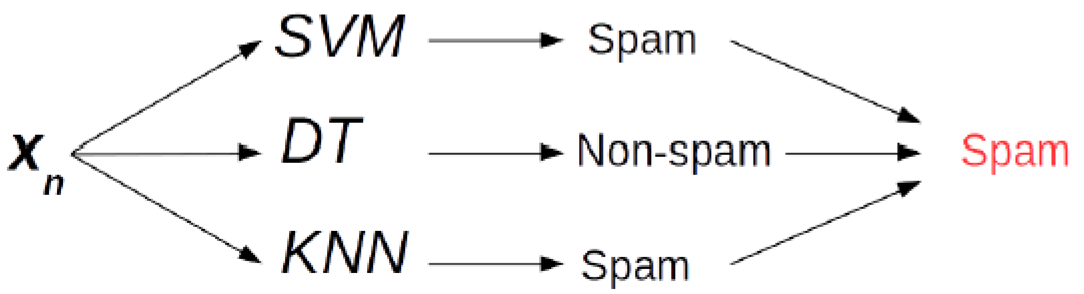
https://jakevdp.github.io/PythonDataScienceHandbook/05.09-principal-component-analysis.html

Ensemble Models

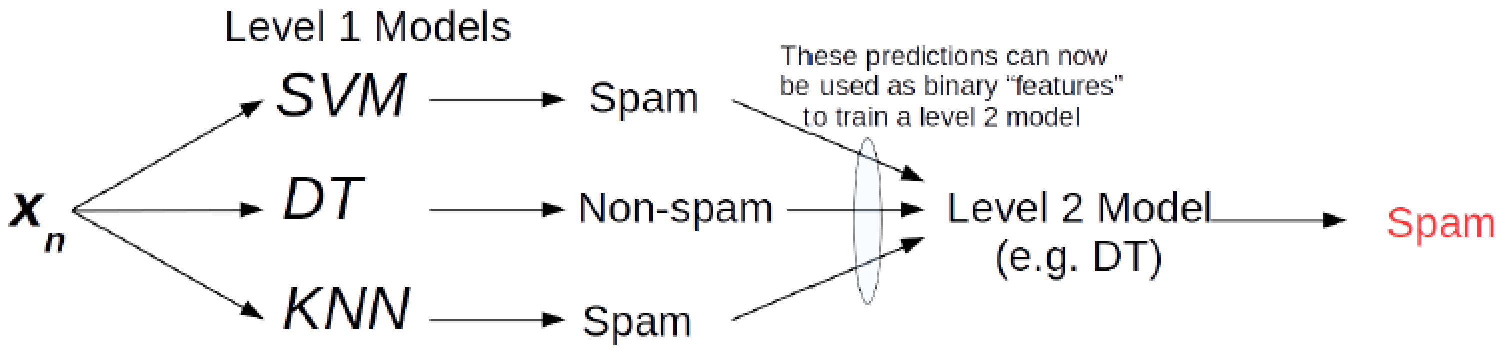
Adopted from Piyush Rai

Simple Models

- Voting or Averaging of predictions of multiple pre-trained models

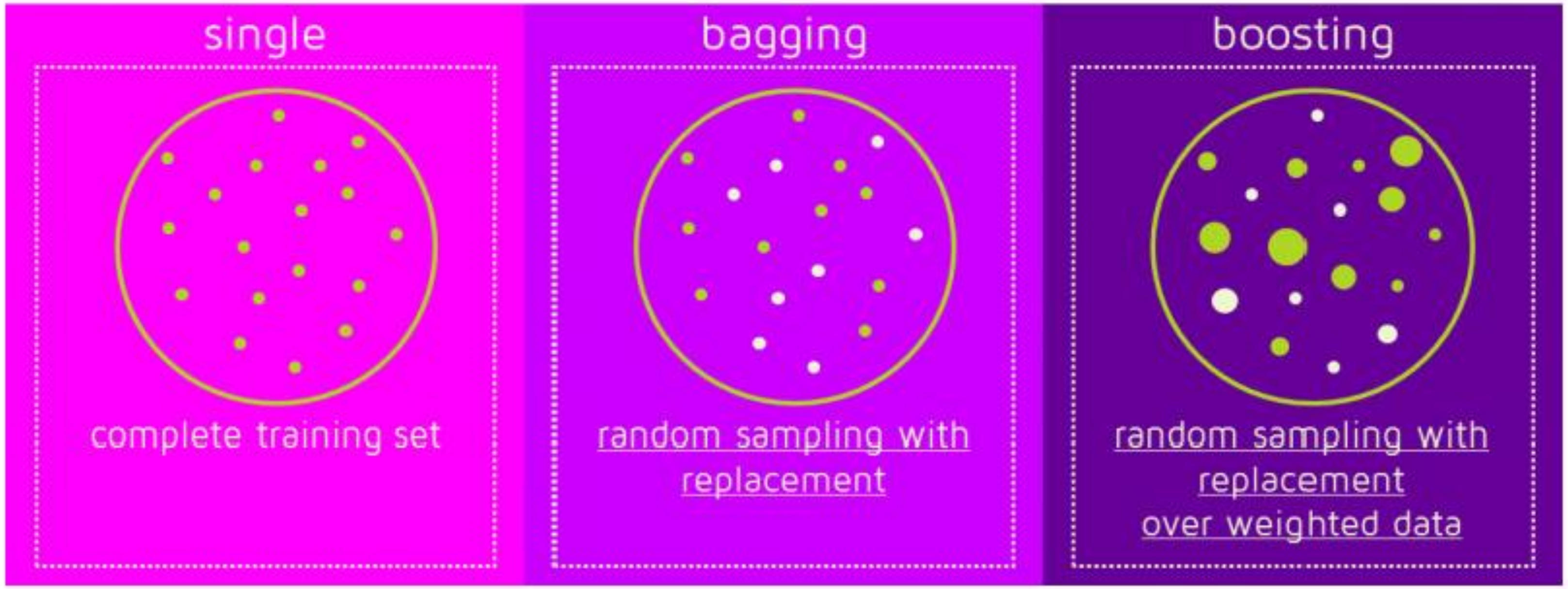


- "Stacking": Use predictions of multiple models as "features" to train a new model and use the new model to make predictions on test data



New Approach

- Instead of training different models on same data, train same model multiple times on different data sets, and "combine" these "different" models
- We can use some simple/weak model as the base model
- How do we get multiple training data sets (in practice, we only have one data set at training time)?

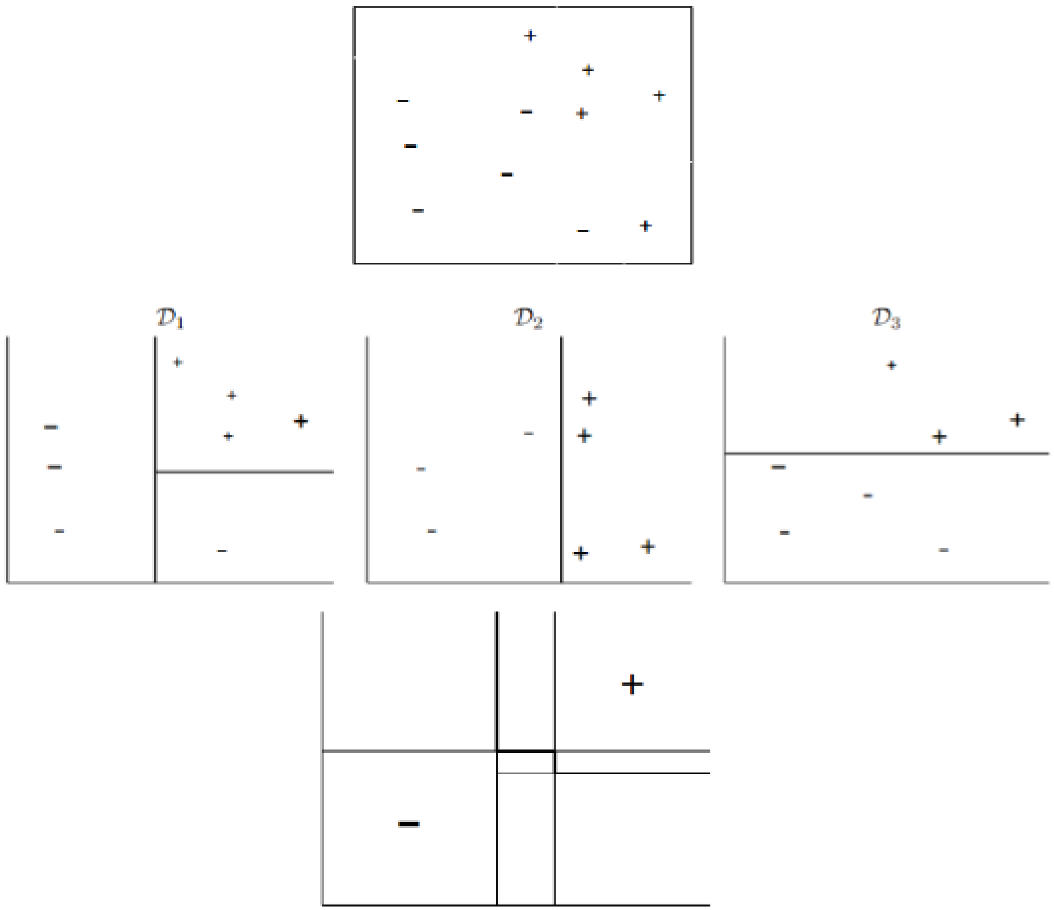


Bagging

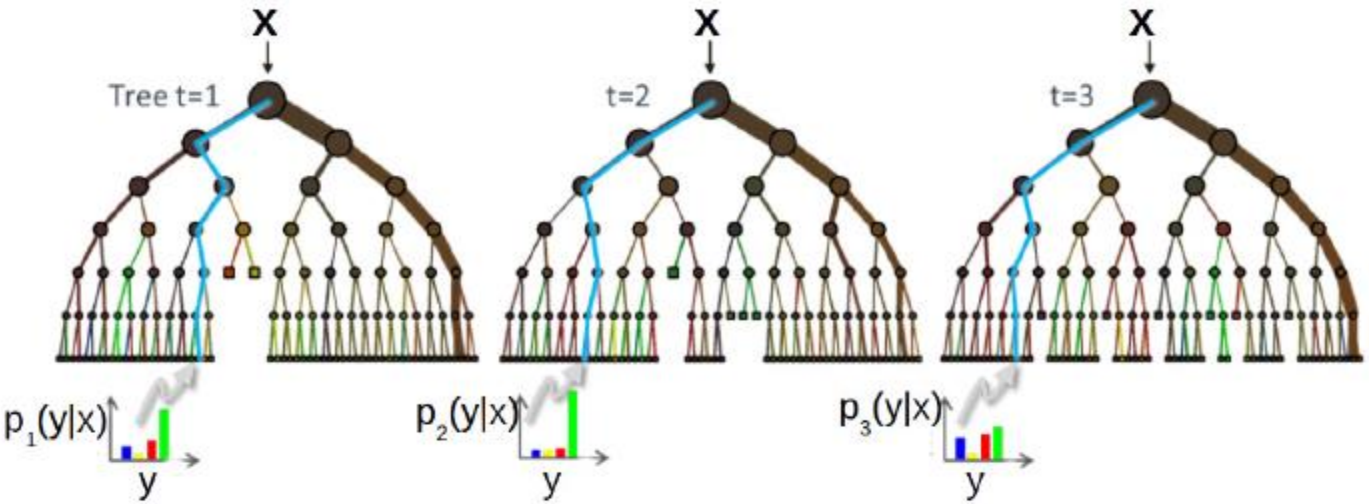
- Bagging stands for Bootstrap Aggregation
- Takes original data set D with N training examples
- Creates M copies $\{\tilde{D}_m\}_{m=1}^{M}$
  - Each $\tilde{D}_m$ is generated from D by sampling with replacement
  - Each data set $\tilde{D}_m$ has the same number of examples as in data set D
  - These data sets are reasonably different from each other (since only about 63% of the original examples appear in any of these data sets)
- Train models $h_1, \ldots, h_M$ using $\tilde{D}_1, \ldots, \tilde{D}_M$, respectively
- Use an averaged model $h = \frac{1}{M}\sum_{m=1}^{M} h_m$ as the final model
- Useful for models with high variance and noisy data

Bagging

Top: Original data, Middle: 3 models (from some model class) learned using three data sets chosen via bootstrapping, Bottom: averaged model



Random Forests



- An ensemble of decision tree (DT) classifiers
- Uses bagging on features (each DT will use a random set of features)
  - Given a total of D features, each DT uses $\sqrt{D}$ randomly chosen features
  - Randomly chosen features make the different trees uncorrelated
- All DTs usually have the same depth
- Each DT will split the training data differently at the leaves
- Prediction for a test example votes on/averages predictions from all the DTs

Boosting

- The basic idea
  - Take a weak learning algorithm
    - Only requirement: Should be slightly better than random
  - Turn it into an awesome one by making it focus on difficult cases
- Most boosting algorithms follow these steps:
  1. Train a weak model on some training data
  2. Compute the error of the model on each training example
  3. Give higher importance to examples on which the model made mistakes
  4. Re-train the model using “importance weighted” training examples
  5. Go back to step 2

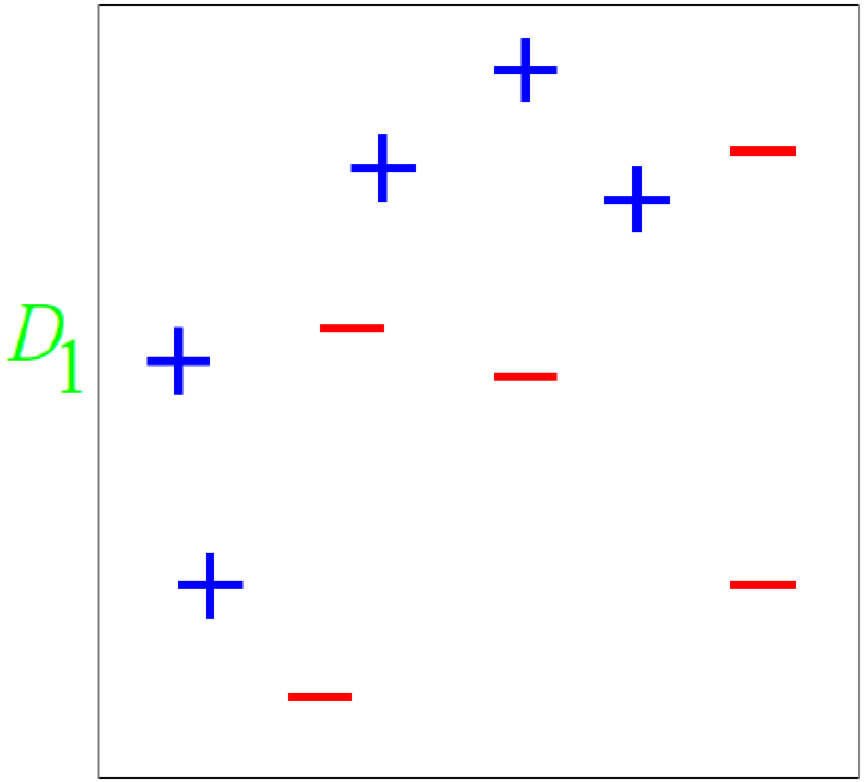
AdaBoost

- Given: Training data $(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_N, y_N)$ with $y_n \in \{-1, +1\}, \forall n$
- Initialize weight of each example $(\boldsymbol{x}_n, y_n)$: $D_1(n) = 1/N, \forall n$
- For round $t = 1 : T$
  - Learn a weak $h_t(\boldsymbol{x}) \rightarrow \{-1, +1\}$ using training data weighted as per D_t
  - Compute the weighted fraction of errors of h_t on this training data
$$\epsilon_t = \sum_{n=1}^{N} D_t(n) \mathbb{1}[h_t(\boldsymbol{x}_n) \neq y_n]$$
  - Set "importance" of h_t: $\alpha_t = \frac{1}{2} \log(\frac{1-\epsilon_t}{\epsilon_t})$(gets larger as ϵ_t gets smaller)
  - Update the weight of each example
$$\begin{aligned} D_{t+1}(n) &\propto \begin{cases} D_t(n) \times \exp(-\alpha_t) & \text{if } h_t(\boldsymbol{x}_n) = y_n \quad \text{(correct prediction: decrease weight)} \\ D_t(n) \times \exp(\alpha_t) & \text{if } h_t(\boldsymbol{x}_n) \neq y_n \quad \text{(incorrect prediction: increase weight)} \end{cases} \\ &= D_t(n) \exp(-\alpha_t y_n h_t(\boldsymbol{x}_n)) \end{aligned}$$
  - Normalize D_{t+1} so that it sums to 1: $D_{t+1}(n) = \frac{D_{t+1}(n)}{\sum_{m=1}^{N} D_{t+1}(m)}$
- Output the "boosted" final hypothesis $H(\boldsymbol{x}) = \text{sign}(\sum_{t=1}^{T} \alpha_t h_t(\boldsymbol{x}))$

AdaBoost Example

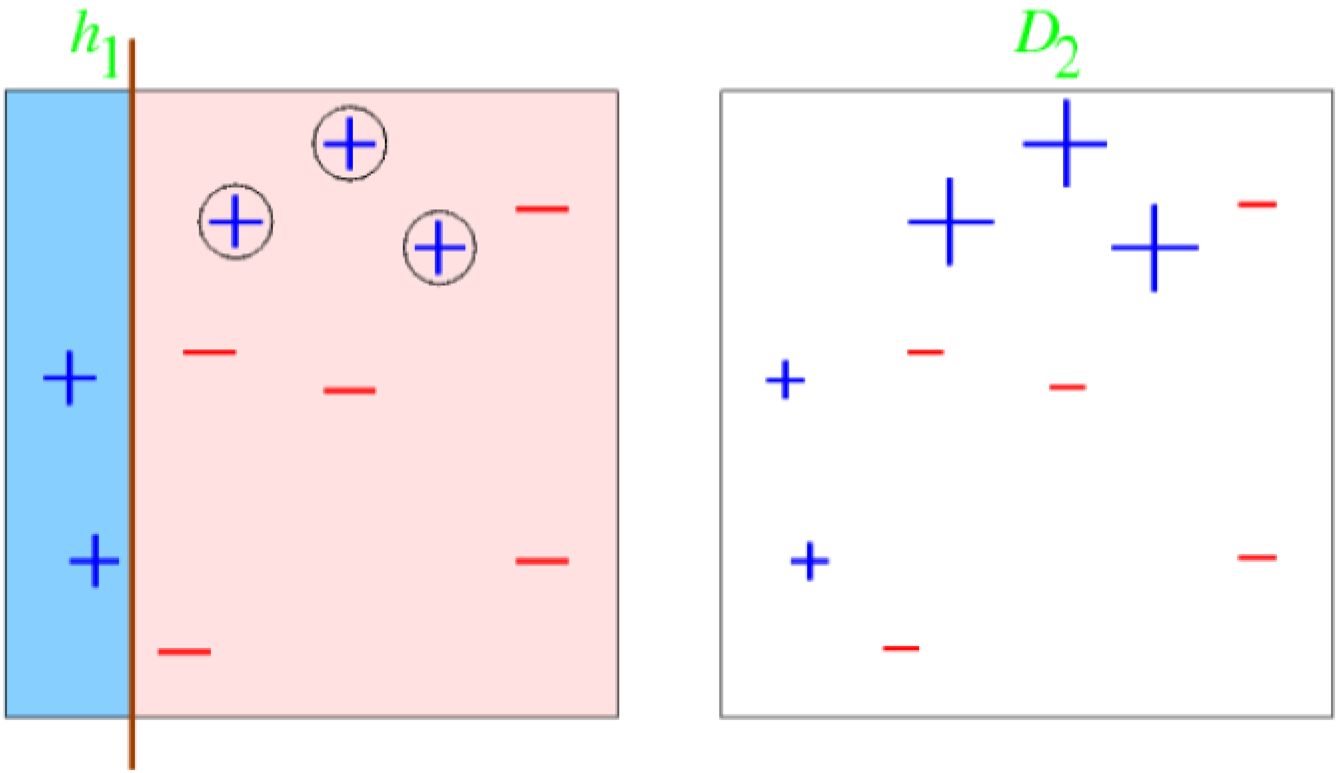
Consider binary classification with 10 training examples

Initial weight distribution D_1 is uniform (each point has equal weight $= 1/10$)



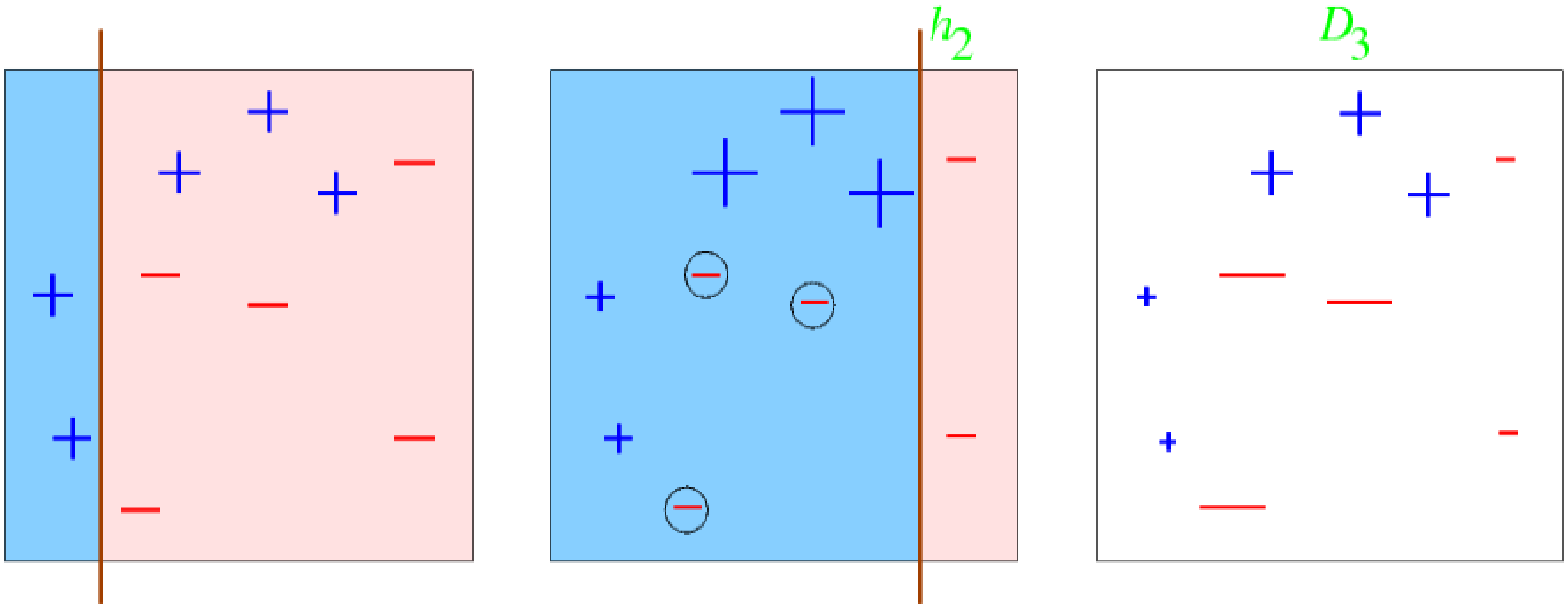
Each of our weak classifers will be an **axis-parallel linear classifier**

AdaBoost Example



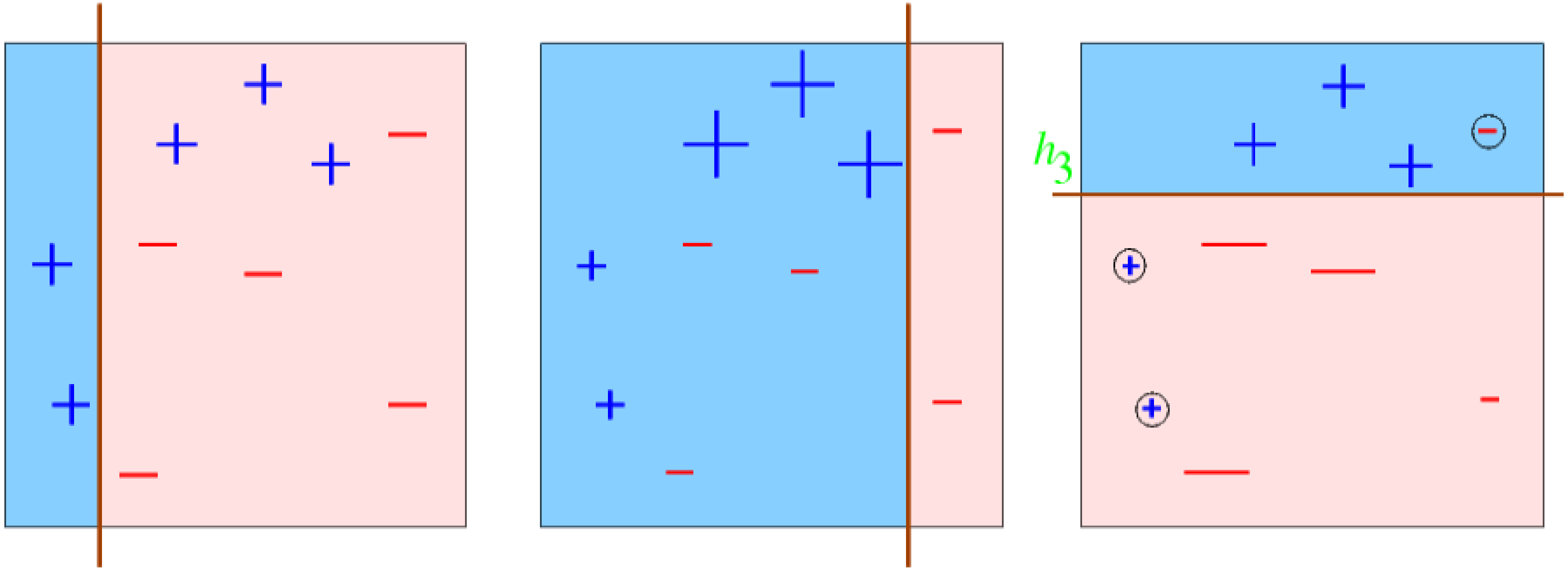
- Error rate of h_1: $\epsilon_1 = 0.3$; weight of h_1: $\alpha_1 = \frac{1}{2}\ln((1-\epsilon_1)/\epsilon_1) = 0.42$
- Each misclassified point upweighted (weight multiplied by exp(α_1)
- Each correctly classified point downweighted (weight multiplied by exp($-\alpha_1$)

AdaBoost Example



- Error rate of h_2: $\epsilon_2 = 0.21$; weight of h_2: $\alpha_2 = \frac{1}{2}\ln((1-\epsilon_2)/\epsilon_2) = 0.65$
- Each misclassified point upweighted (weight multiplied by $\exp(\alpha_2)$)
- Each correctly classified point downweighted (weight multiplied by $\exp(-\alpha_2)$)

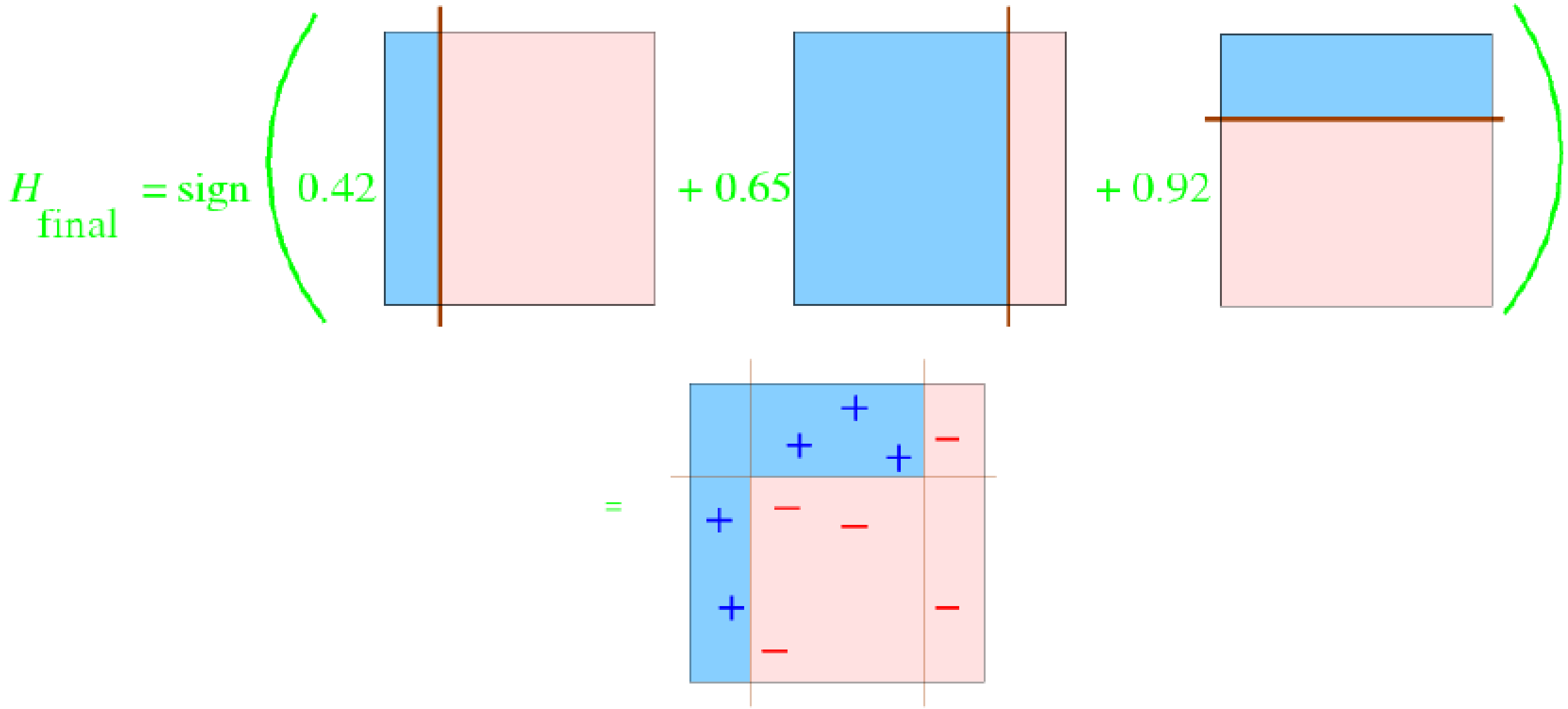
AdaBoost Example



- Error rate of h_3: $\epsilon_3 = 0.14$; weight of h_3: $\alpha_3 = \frac{1}{2}\ln((1-\epsilon_3)/\epsilon_3) = 0.92$
- Suppose we decide to stop after round 3
- Our ensemble now consists of 3 classifiers: h_1, h_2, h_3

AdaBoost Example

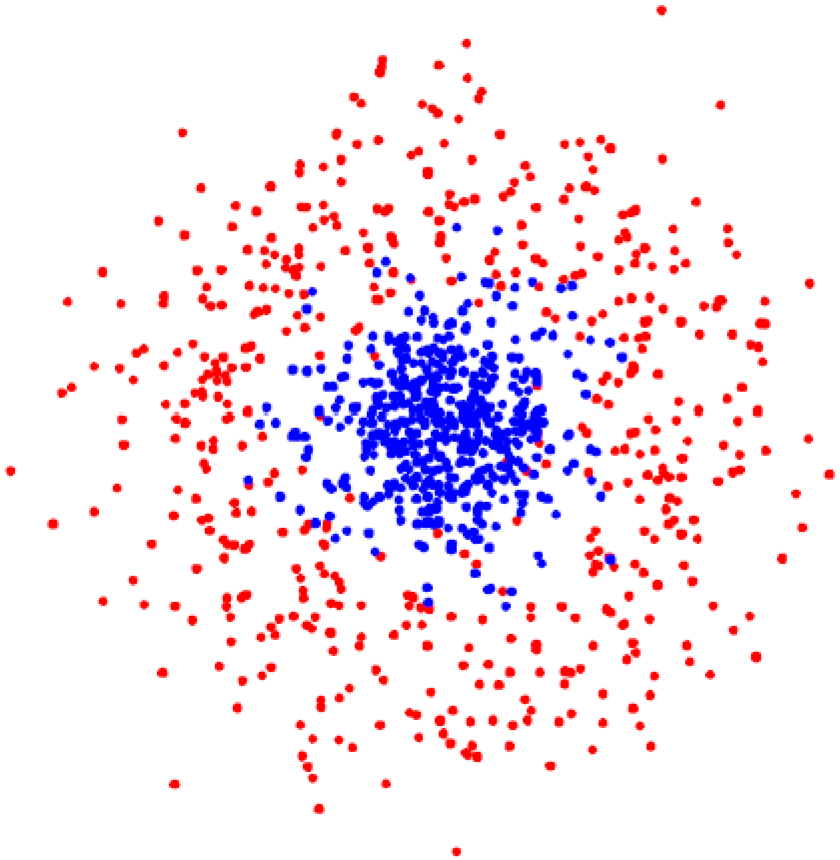
- Final classifier is a weighted linear combination of all the classifiers
- Classifier h_i gets a weight α_i



- Multiple weak, linear classifiers **combined** to give a strong, nonlinear classifier

Second Example

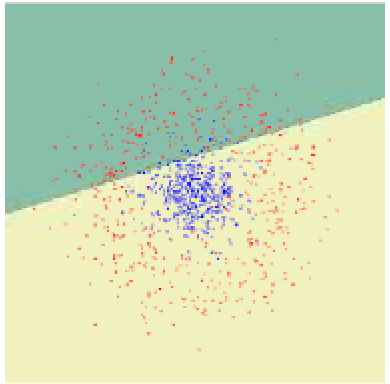
- Given: A nonlinearly separable dataset
- We want to use Perceptron (linear classifier) on this data

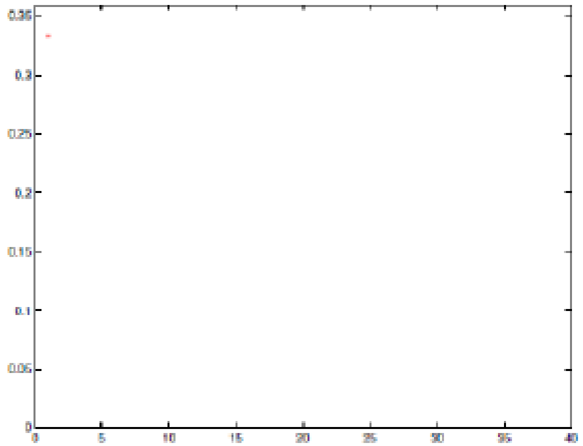


Second Example

- After round 1, our ensemble has 1 linear classifier (Perceptron)
- Bottom figure: X axis is number of rounds, Y axis is training error

$t = 1$

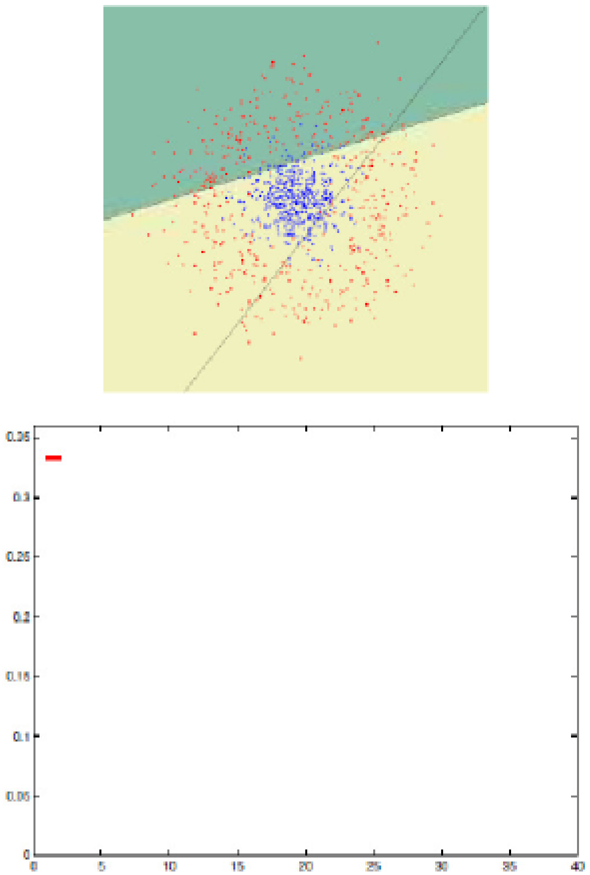




Second Example

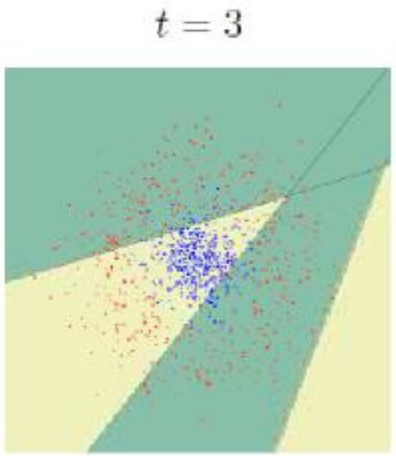
- After round 2, our ensemble has 2 linear classifiers (Perceptrons)
- Bottom figure: X axis is number of rounds, Y axis is training error

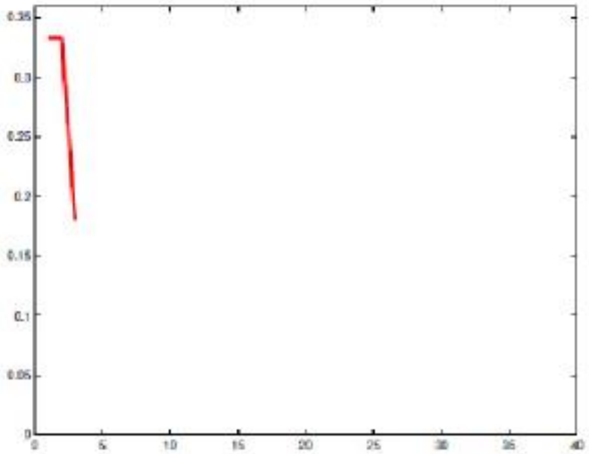
$t = 2$



Second Example

- After round 3, our ensemble has 3 linear classifiers (Perceptrons)
- Bottom figure: X axis is number of rounds, Y axis is training error

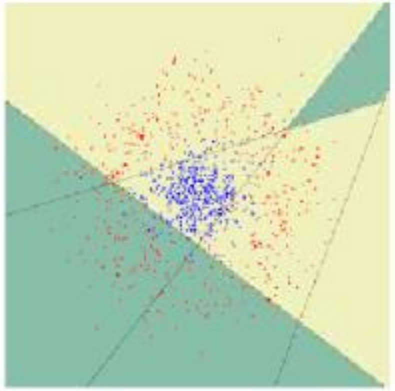


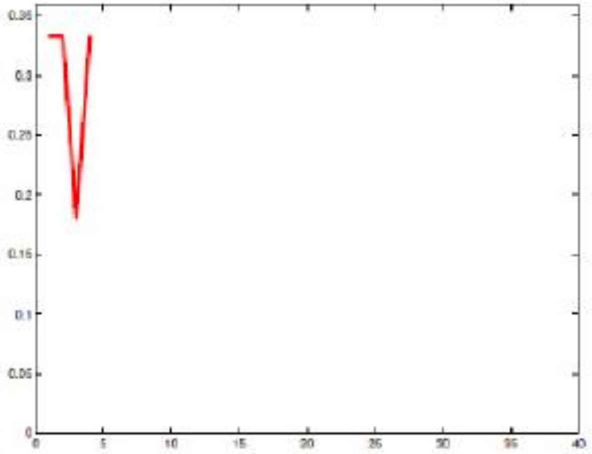


Second Example

- After round 4, our ensemble has 4 linear classifiers (Perceptrons)
- Bottom figure: X axis is number of rounds, Y axis is training error

$t = 4$

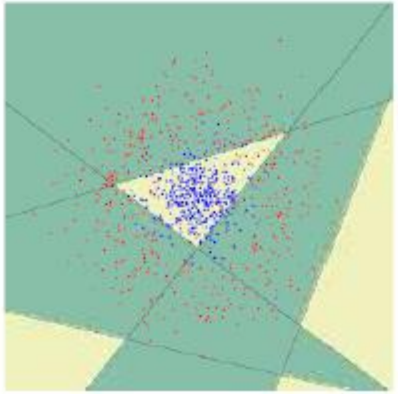


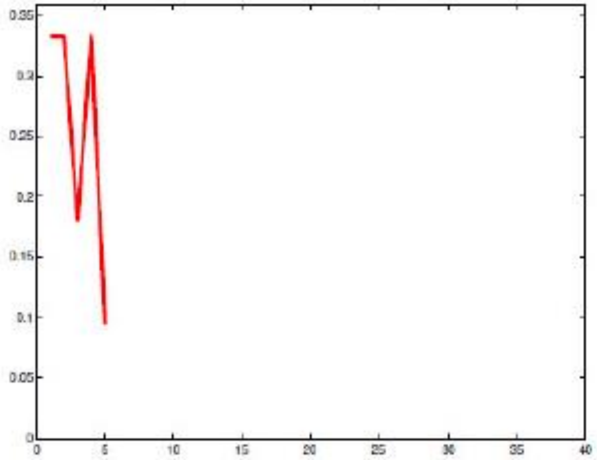


Second Example

- After round 5, our ensemble has 5 linear classifiers (Perceptrons)
- Bottom figure: X axis is number of rounds, Y axis is training error

$t = 5$

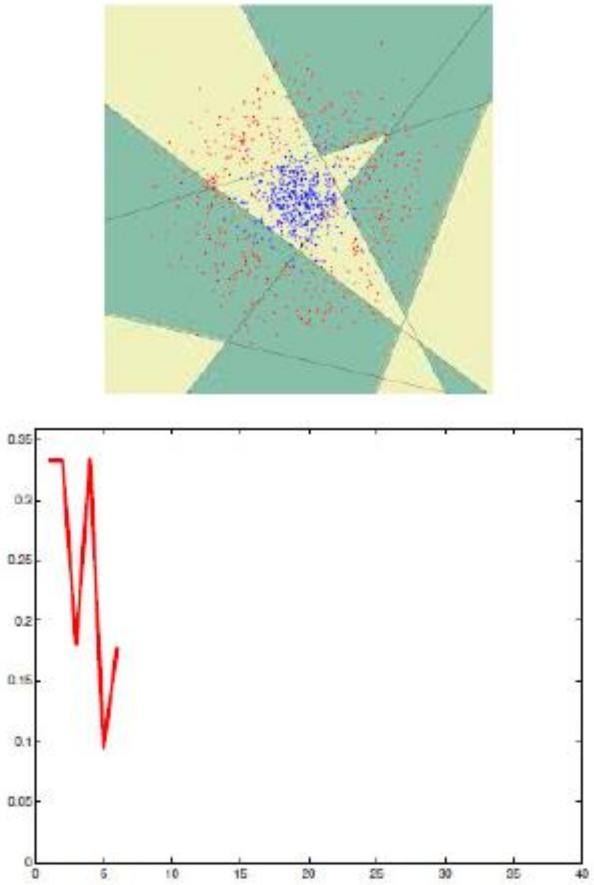




Second Example

- After round 6, our ensemble has 6 linear classifiers (Perceptrons)
- Bottom figure: X axis is number of rounds, Y axis is training error

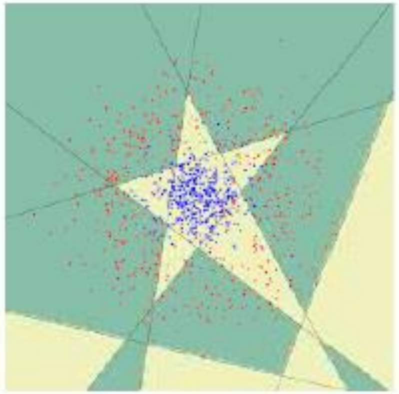
$t = 6$

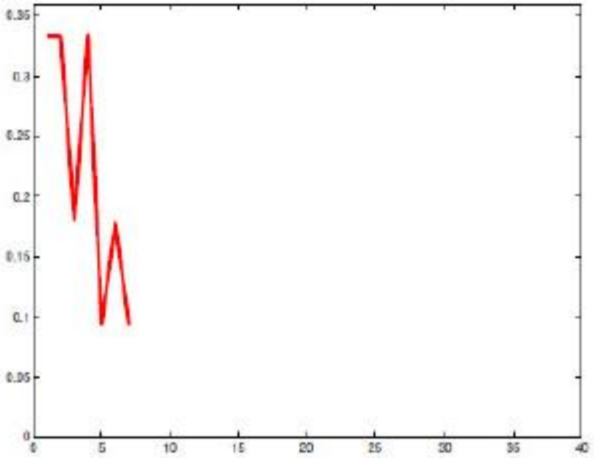


Second Example

- After round 7, our ensemble has 7 linear classifiers (Perceptrons)
- Bottom figure: X axis is number of rounds, Y axis is training error

$t = 7$

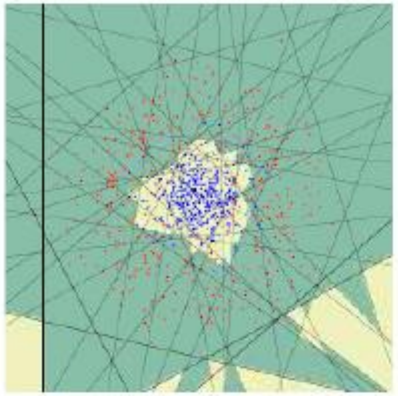


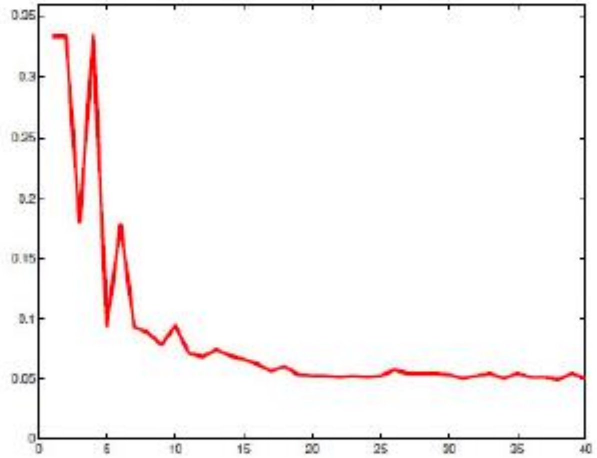


Second Example

- After round 40, our ensemble has 40 linear classifiers (Perceptrons)
- Bottom figure: X axis is number of rounds, Y axis is training error

$t = 40$





Comments

- For AdaBoost, given each model's error $\epsilon_t = 1/2 - \gamma_t$, the training error consistently gets better with rounds

$$\text{train-error}(H_{final}) \leq \exp(-2\sum_{t=1}^{T}\gamma_t^2)$$

- Boosting algorithms can be shown to be minimizing a loss function
  - E.g., AdaBoost has been shown to be minimizing an exponential loss

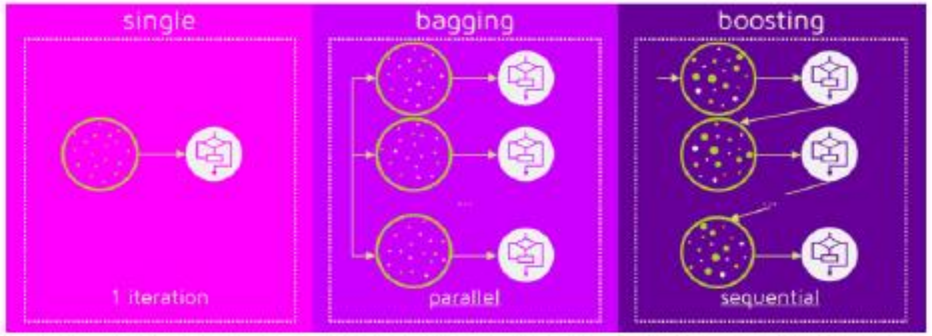
$$\mathcal{L} = \sum_{n=1}^{N}\exp\{-y_n H(\boldsymbol{x}_n)\}$$

where $H(\boldsymbol{x}) = \text{sign}(\sum_{t=1}^{T}\alpha_t h_t(\boldsymbol{x}))$, given weak base classifiers $h_1, \ldots, h_T$

- Boosting in general can perform badly if some examples are outliers

Comparison

- No clear winner; usually depends on the data
- Bagging is computationally more efficient than boosting (note that bagging can train the M models in parallel, boosting can't)



- Both reduce variance (and overfitting) by combining different models
  - The resulting model has higher stability as compared to the individual ones
- Bagging usually can't reduce the bias, boosting can (note that in boosting, the training error steadily decreases)
- Bagging usually performs better than boosting if we don't have a high bias and only want to reduce variance (i.e., if we are overfitting)

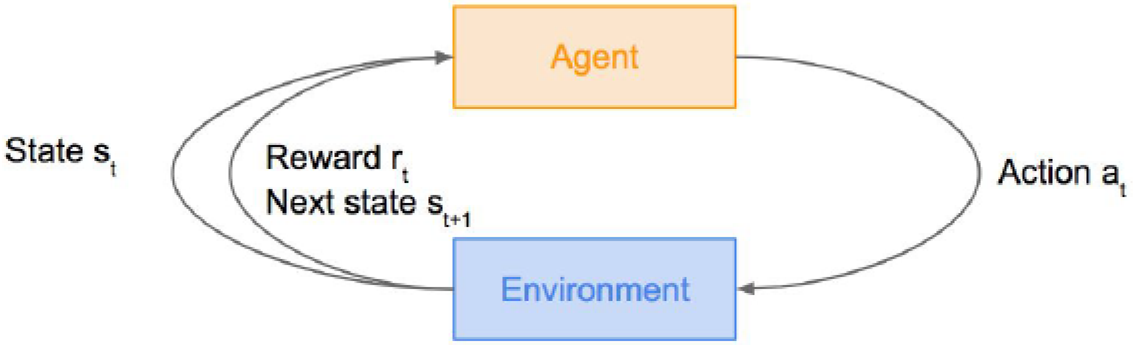
Reinforcement Learning

Adopted from Fei-Fei Li, Justin Johnson, Serena Yeung

Introduction

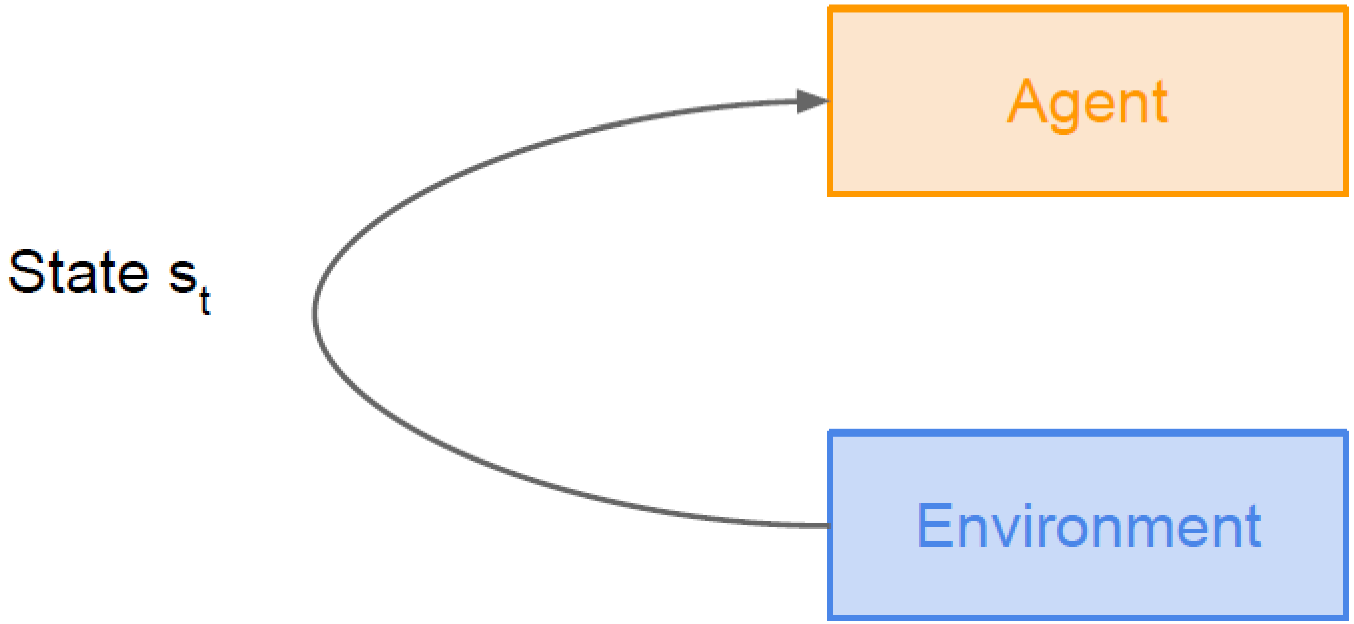
Problems involving an **agent** interacting with an **environment**, which provides numeric **reward** signals

Goal: Learn how to take actions in order to maximize reward

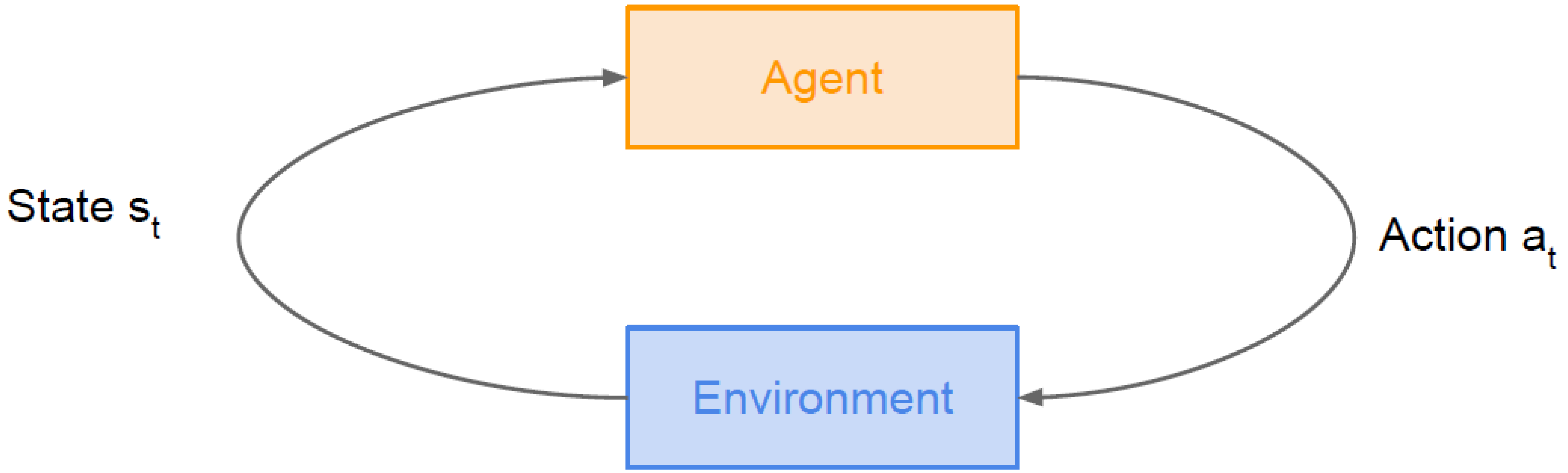


Introduction

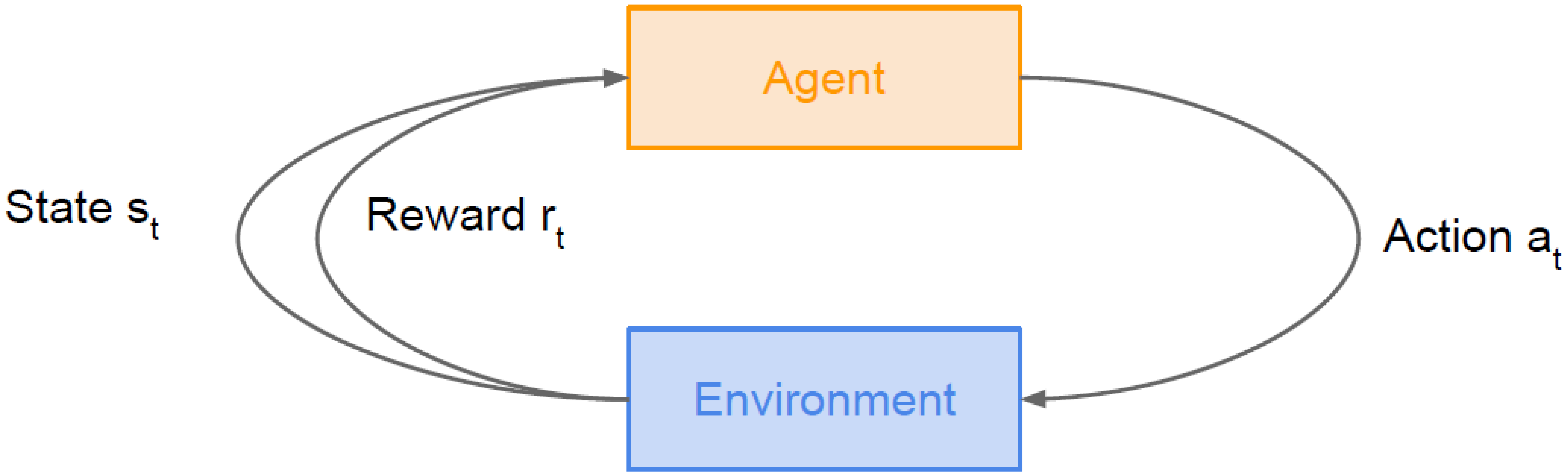
Introduction



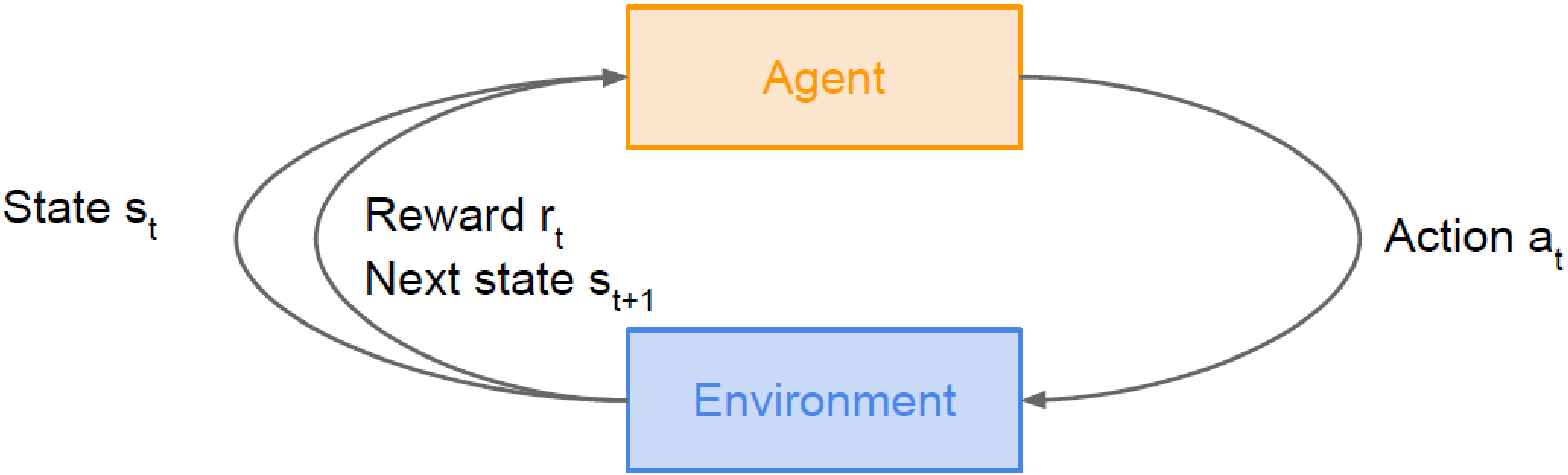
Introduction



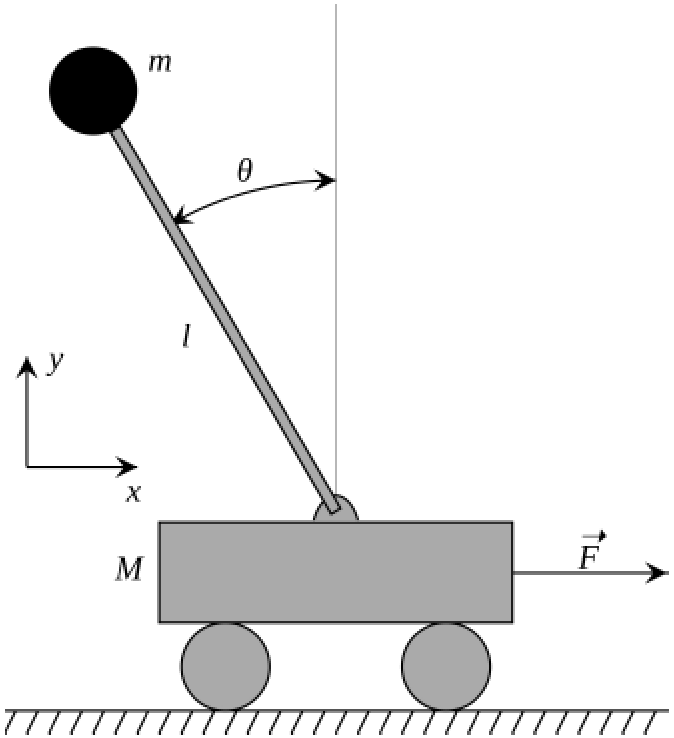
Introduction



Introduction



Introduction



Objective: Balance a pole on top of a movable cart

State: angle, angular speed, position, horizontal velocity
Action: horizontal force applied on the cart
Reward: 1 at each time step if the pole is upright

Passive vs Active Learning

- Passive learning
  - The agent imply watches the world going by and tries to learn the utilities of being in various states
- Active learning
  - The agent not simply watches, but also acts

Passive Learning

```
function PASSIVE-RL-AGENT(e) returns an action
   static: U, a table of utility estimates
           N, a table of frequencies for states
           M, a table of transition probabilities from state to state
           percepts, a percept sequence (initially empty)

   add e to percepts
   increment N[STATE[e]]
   U ← UPDATE(U, e, percepts, M, N)
   if TERMINAL?[e] then percepts ← the empty sequence
   return the action Observe
```

Markov Decision Process

- Mathematical formulation of the RL problem
- **Markov property**: Current state completely characterises the state of the world

Defined by: $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathbb{P}, \gamma)$

$\mathcal{S}$: set of possible states
$\mathcal{A}$: set of possible actions
$\mathcal{R}$: distribution of reward given (state, action) pair
$\mathbb{P}$: transition probability i.e. distribution over next state given (state, action) pair
γ : discount factor

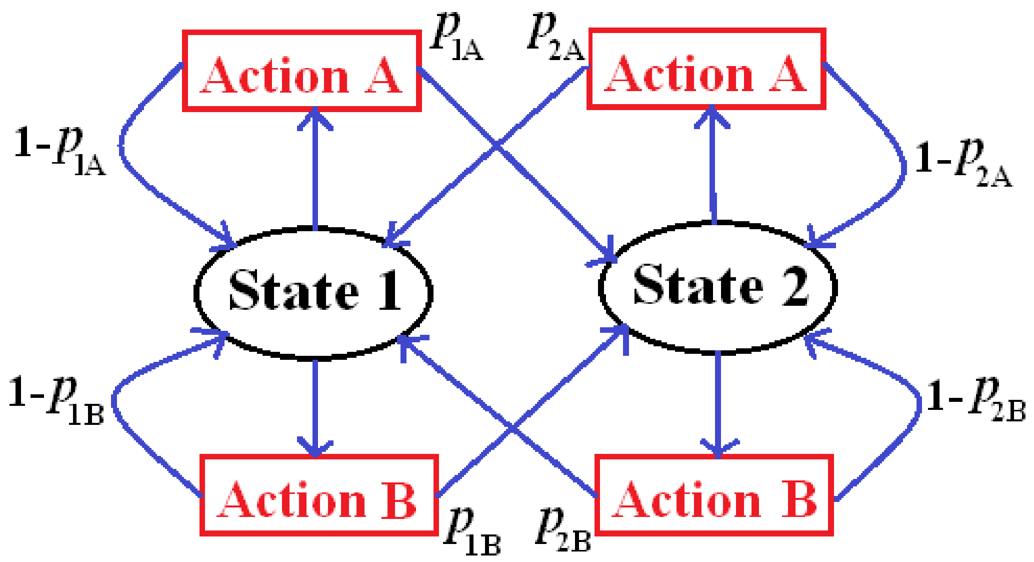
Markov Decision Process

- At time step t=0, environment samples initial state $s_0 \sim p(s_0)$
- Then, for t=0 until done:
  - Agent selects action a_t
  - Environment samples reward $r_t \sim R(. \mid s_t, a_t)$
  - Environment samples next state $s_{t+1} \sim P(. \mid s_t, a_t)$
  - Agent receives reward r_t and next state s_{t+1}

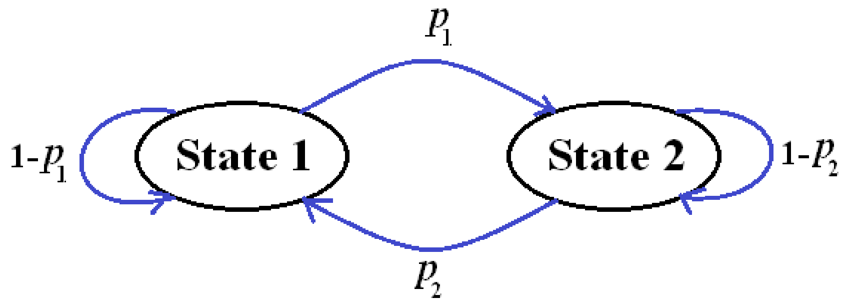
- A policy **π** is a function from S to A that specifies what action to take in each state
- **Objective**: find policy **π*** that maximizes cumulative discounted reward: $\sum_{t \geq 0} \gamma^t r_t$

Markov Decision Process

Markov Decision Process



Markov Chain



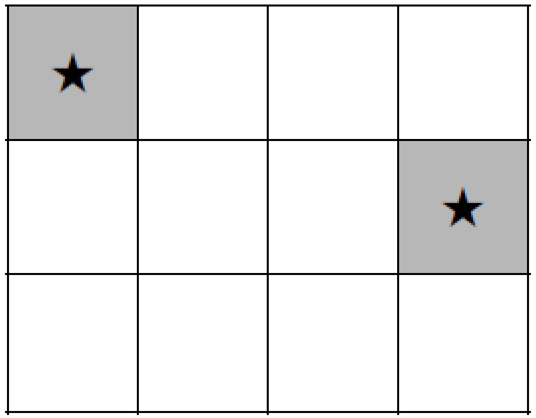
Example

actions = {

1. right
2. left
3. up
4. down

}

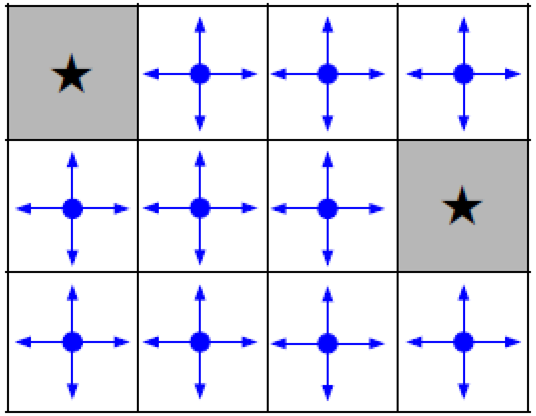
states



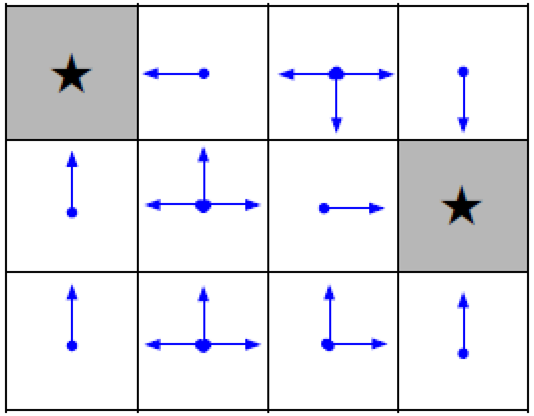
Set a negative "reward"
for each transition
(e.g. r = -1)

Objective: reach one of terminal states (greyed out) in least number of actions

Example



Random Policy



Optimal Policy

Optimal Policy

We want to find optimal policy **π*** that maximizes the sum of rewards.

How do we handle the randomness (initial state, transition probability…)?
Maximize the **expected sum of rewards!**

Formally: $\pi^* = \arg\max_{\pi} \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | \pi\right]$ with $s_0 \sim p(s_0), a_t \sim \pi(\cdot|s_t), s_{t+1} \sim p(\cdot|s_t, a_t)$

Value Function

Following a policy produces sample trajectories (or paths) $s_0, a_0, r_0, s_1, a_1, r_1, \ldots$

How good is a state?
The **value function** at state s, is the expected cumulative reward from following the policy from state s:

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | s_0 = s, \pi\right]$$

How good is a state-action pair?
The **Q-value function** at state s and action a, is the expected cumulative reward from taking action a in state s and then following the policy:

$$Q^{\pi}(s,a) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi\right]$$

Q-function

The optimal Q-value function Q* is the maximum expected cumulative reward achievable from a given (state, action) pair:

$$Q^*(s,a) = \max_\pi \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi\right]$$

Q* satisfies the following **Bellman equation**:

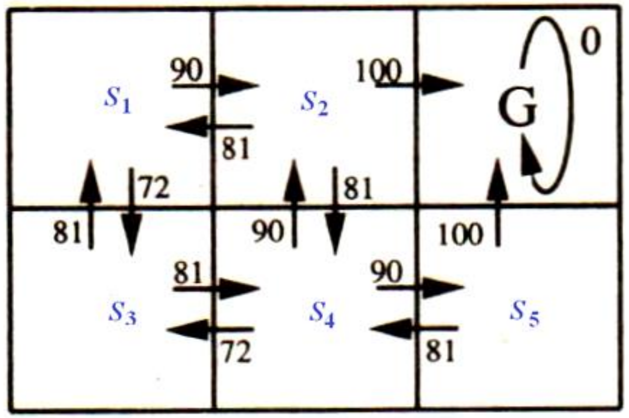
$$Q^*(s,a) = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s',a') | s, a\right]$$

Intuition: if the optimal state-action values for the next time-step Q*(s',a') are known, then the optimal strategy is to take the action that maximizes the expected value of $r + \gamma Q^*(s',a')$

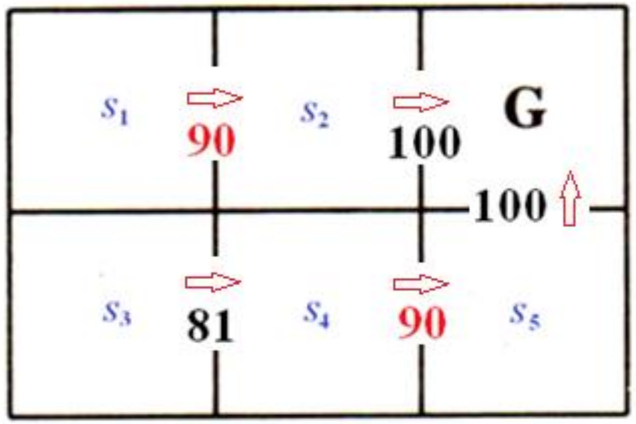
The optimal policy **π*** corresponds to taking the best action in any state as specified by Q*

Q-function

$Q(s,a)$	↑	↓	←	→
s_1	0	72	0	90
s_2	0	81	81	100
s_3	81	0	0	81
s_4	90	0	72	90
s_5	100	0	81	0
G	0	0	0	0



$Q(s, a)$ values



π^*

Solution

Value iteration algorithm: Use Bellman equation as an iterative update

$$Q_{i+1}(s,a) = \mathbb{E}\left[r + \gamma \max_{a'} Q_i(s',a')|s,a\right]$$

Q_i will converge to Q^* as i -> infinity

What's the problem with this?
Not scalable. Must compute Q(s,a) for every state-action pair. If state is e.g. current game state pixels, computationally infeasible to compute for entire state space!

Solution: use a function approximator to estimate Q(s,a). E.g. a neural network!

Policy Iteration Algorithm

Initialize a policy π' arbitrarily

Repeat

$\pi \leftarrow \pi'$

Compute the values using π by

solving the linear equations

$$V^{\pi}(s) = E[r|s, \pi(s)] + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, \pi(s)) V^{\pi}(s')$$

Improve the policy at each state

$$\pi'(s) \leftarrow \arg\max_a (E[r|s, a] + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) V^{\pi}(s'))$$

Until $\pi = \pi'$

Exploration - Exploitation

Exploration of unknown states and actions to gather new information

Exploitation of learned states and actions to maximize the cumulative reward

- ***ε*-greedy search:**

  Explore – with probability ε choose uniformly one action among all possible actions.

  Exploit – with probability 1-ε choose the best action.

  Start with a high ε and gradually decrease it in order initiate exploitation once enough exploration.

Probabilistic Search

Choose action *a* according to probability

$$P(a \mid s) = \frac{\exp Q(s,a)}{\sum_{b \in A} \exp Q(s,b)}$$

Move from exploration to exploitation using

$$P(a \mid s) = \frac{\exp\left[Q(s,a)/T\right]}{\sum_{b=1}^{A} \exp\left[Q(s,b)/T\right]}$$

Start with a large *T* and gradually decrease it.

T large, $P(a \mid s) \approx 1/A$ (constant) $\Rightarrow$ exploration

T small, better actions → exploitation.

Deep Q-learning

Q-learning: Use a function approximator to estimate the action-value function

$$Q(s, a; \theta) \approx Q^*(s, a)$$

function parameters (weights)

If the function approximator is a deep neural network => **deep q-learning**!

Deep Q-learning

Remember: want to find a Q-function that satisfies the Bellman Equation:

$$Q^*(s,a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q^*(s',a') | s, a \right]$$

Forward Pass

Loss function: $L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot)} \left[(y_i - Q(s,a;\theta_i))^2 \right]$

where $y_i = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) | s, a \right]$

Iteratively try to make the Q-value close to the target value (y_i) it should have, if Q-function corresponds to optimal Q* (and optimal policy $\boldsymbol{\pi}$*)

Backward Pass

Gradient update (with respect to Q-function parameters θ):

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot); s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i)) \nabla_{\theta_i} Q(s,a;\theta_i) \right]$$

Training

Learning from batches of consecutive samples is problematic:

- Samples are correlated => inefficient learning
- Current Q-network parameters determines next training samples (e.g. if maximizing action is to move left, training samples will be dominated by samples from left-hand size) => can lead to bad feedback loops

Address these problems using **experience replay**

- Continually update a **replay memory** table of transitions (s_t, a_t, r_t, s_{t+1}) as game (experience) episodes are played
- Train Q-network on random minibatches of transitions from the replay memory, instead of consecutive samples

Each transition can also contribute to multiple weight updates
=> greater data efficiency

Deep Q-learning

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
  end for
end for

Deep Q-learning

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
  end for
end for

Initialize replay memory, Q-network

Deep Q-learning

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do** ← Play M episodes (full games)
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
  end for
end for

Deep Q-learning

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
  end for
end for

Initialize state (starting game screen pixels) at the beginning of each episode

Deep Q-learning

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
  end for
end for

For each timestep t of the game

Deep Q-learning

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
  end for
end for

With small probability, select a random action (explore), otherwise select greedy action from current policy

Deep Q-learning

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
  end for
end for

Take the action (a_t), and observe the reward r_t and next state s_{t+1}

Deep Q-learning

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
  end for
end for

Store transition in replay memory

Deep Q-learning

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
  end for
end for

Experience Replay: Sample a random minibatch of transitions from replay memory and perform a gradient descent step

Swarm Intelligence

Adopted from Thiemo Krink

The need for new Computing Techniques

The computer revolution changed human societies:

- communication
- transportation
- industrial production
- administration, writing, and bookkeeping
- technological advances / science
- entertainment

However, some problems cannot be tackled with traditional hardware and software!

The need for new Computing Techniques

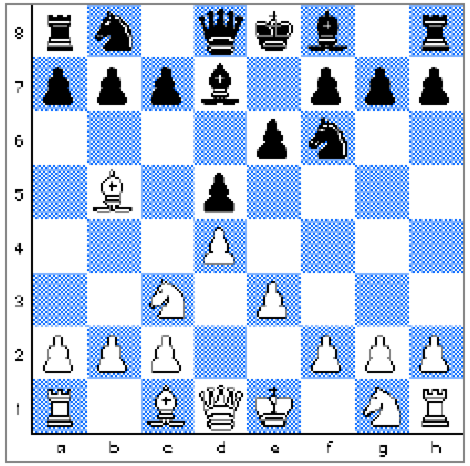
Computing tasks have to be

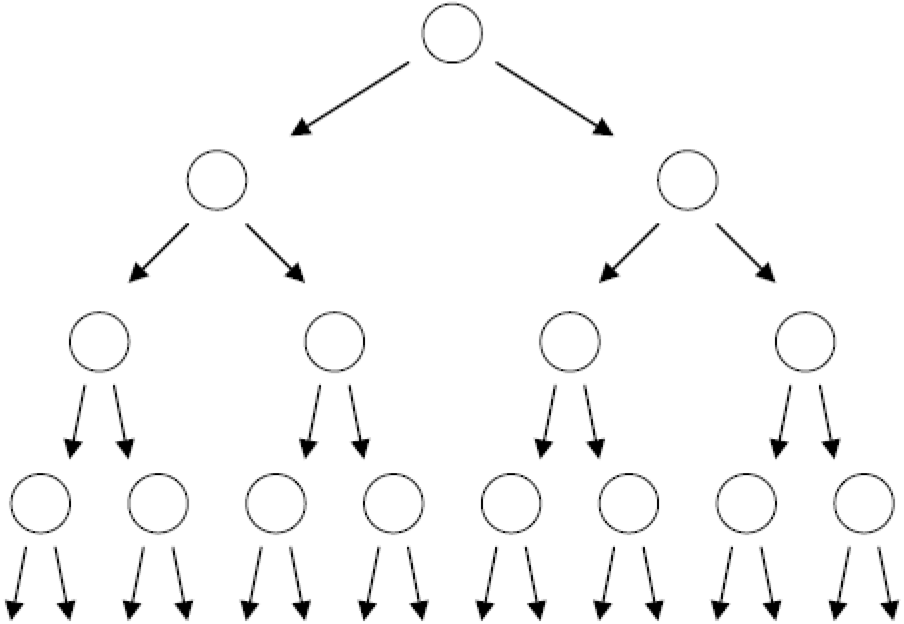
- well-defined
- fairly predictable
- computable in reasonable time with serial computers

Hard Problems

Well-defined, but computational hard problems

- NP hard problems (Travelling Salesman Problem)
- Action-response planning (Chess playing)



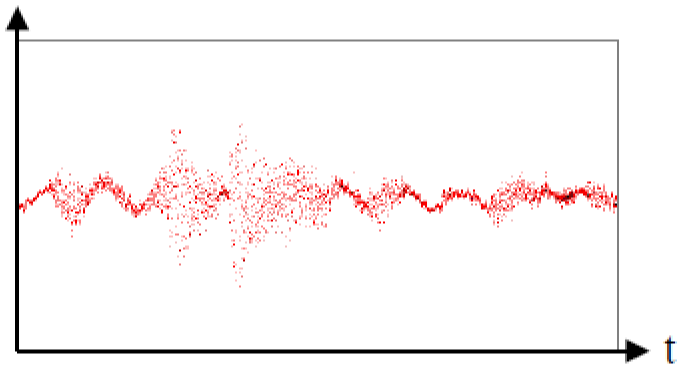


Hard Problems

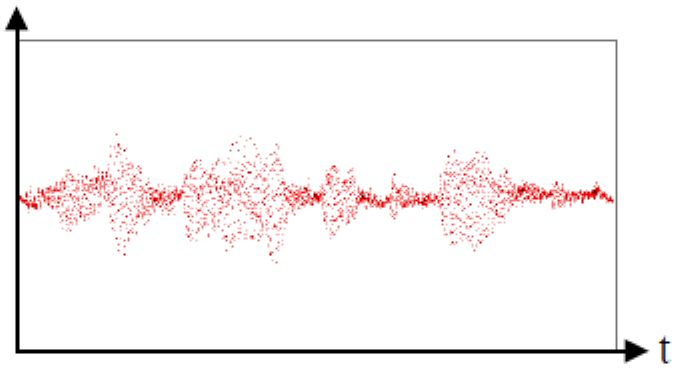
Fuzzy problems

- intelligent human-machine interaction
- natural language understanding

Example: Fuzziness in sound processing



“E-vo-lu-tio-na-ry Com-pu-ta-tion”



“E-vo-lu-tio-na-ry Com-pu-ta-tion”

Hard Problems

Hardly predictable and dynamic problems

- real-world autonomous robots
- management and business planning

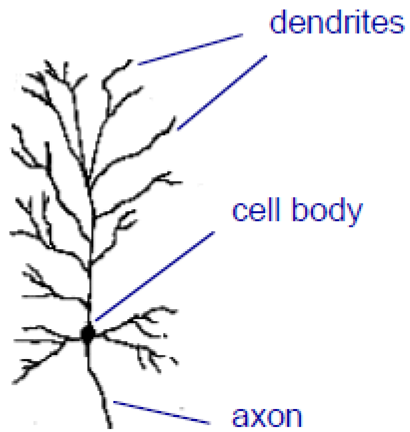
Japanese piano robot

Trade at the stock exchange

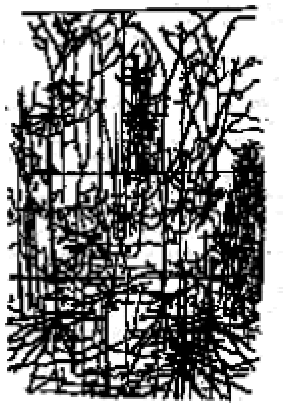
Alternatives

- DNA based computing (chemical computation)
- Quantum computing (quantum-physical computation)
- Bio-computing (simulation of biological mechanisms)

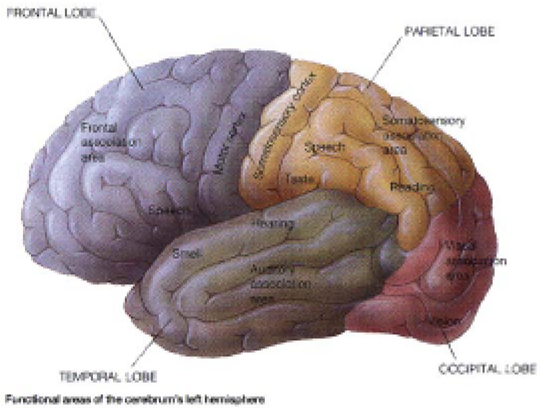
Artificial Networks



The basic unit - the neurone



Vertical cut through the neocortex of a cat

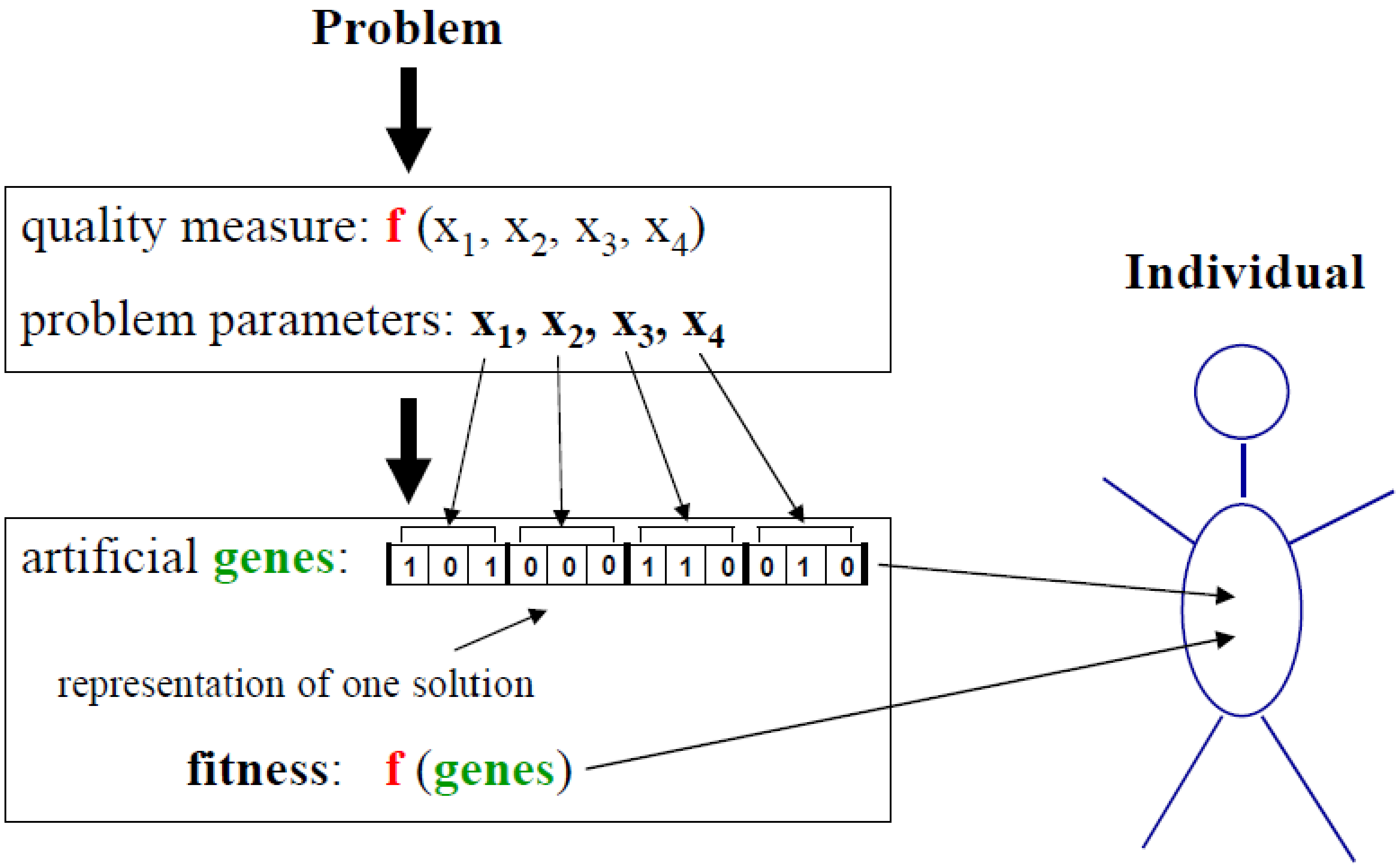


Functional units of the human brain

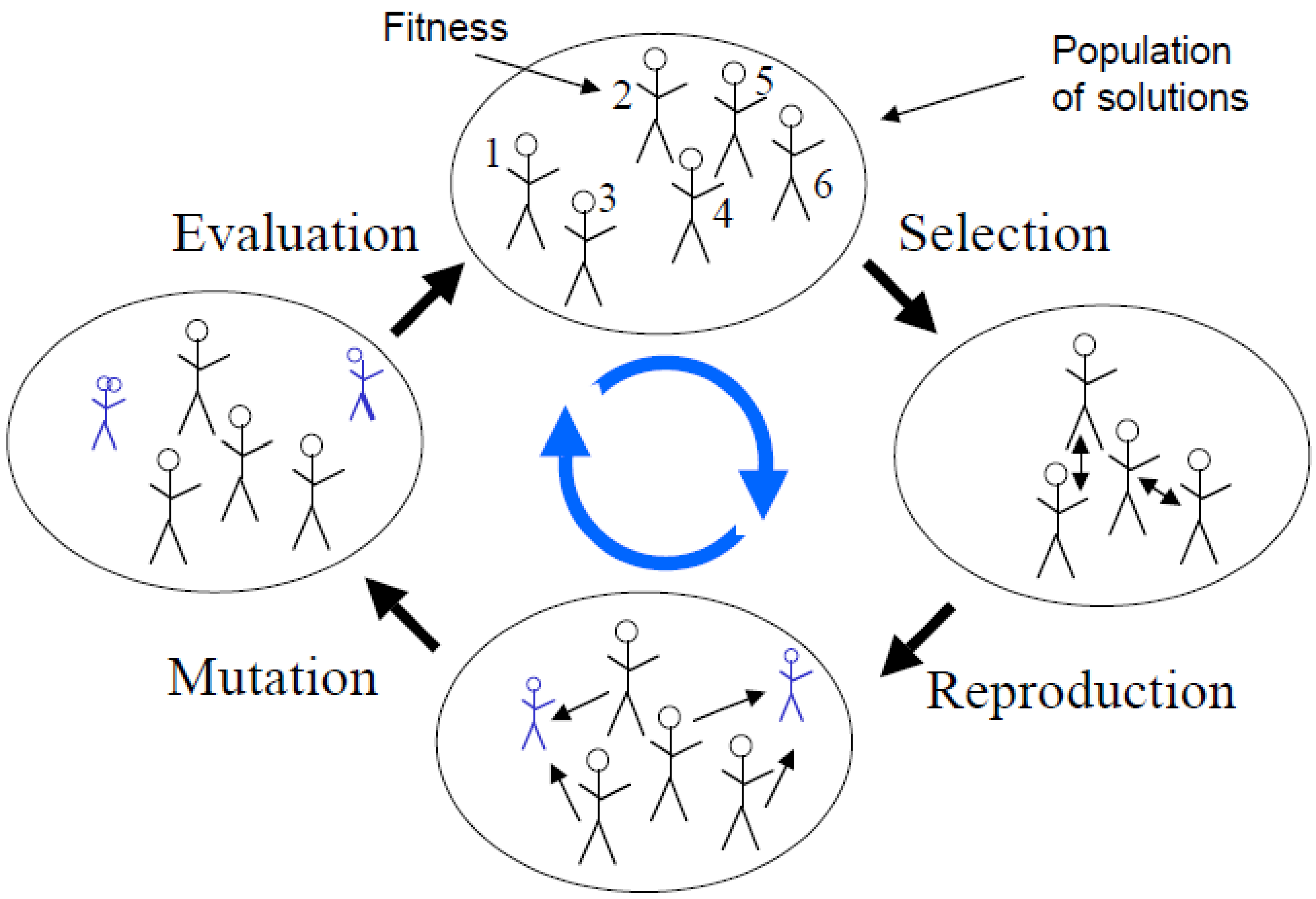
Properties of the brain

- holistic
- parallel
- associative
- learning
- redundancy
- self-organisation

Evolutionary Computation



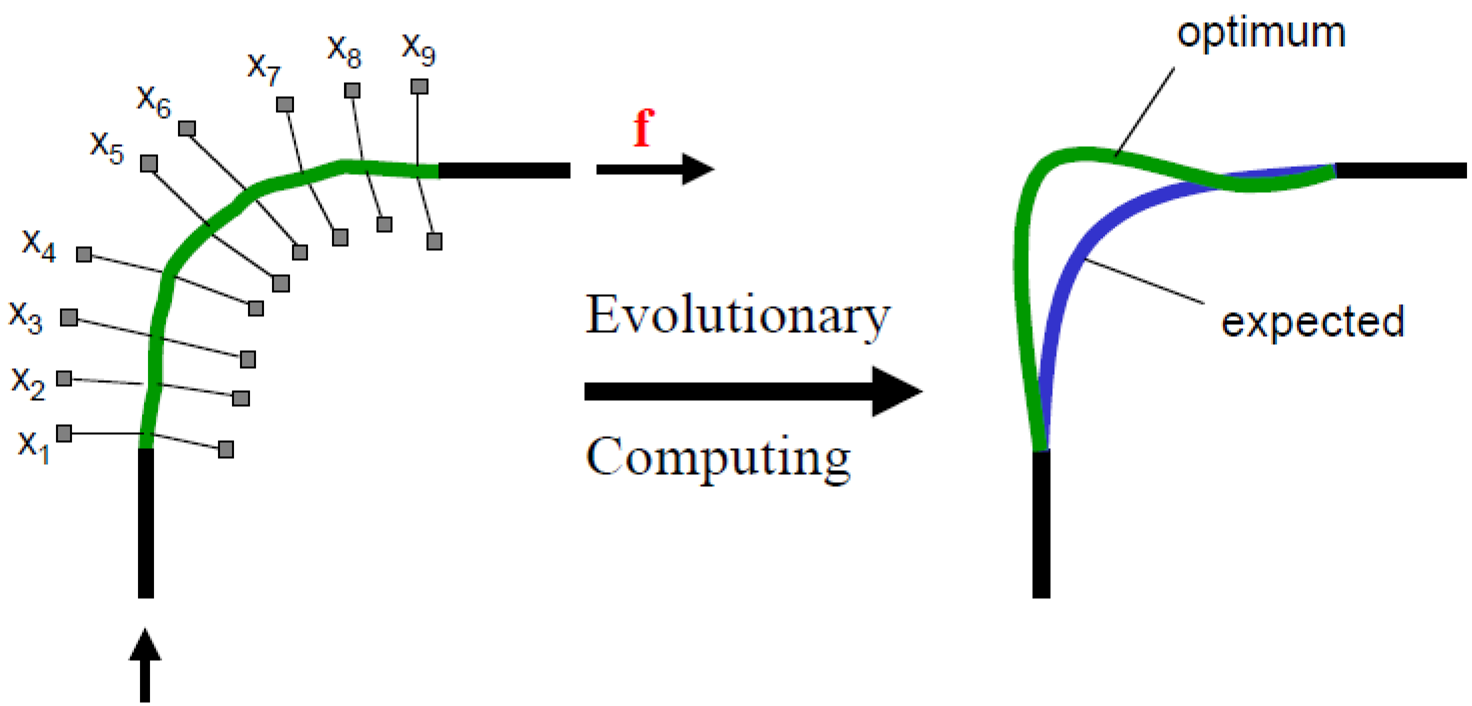
Evolutionary Computation



Evolutionary Computation

The task: Design a bent tube with a maximum flow

Goal: water flow $f(x_1, x_2, \ldots, x_9) = f_{max}$



Bio-Computing

	Inspiration	Identification	Application	Verification
Natural sciences	●	●		●
Complexity theory		●	●	
Adaptive algorithms			●	
Artificial Life	●		●	
Swarm Intelligence	●	●	●	

Applications

- Robotics / Artificial Intelligence
- Process optimisation / Staff scheduling
- Telecommunication companies
- Entertainment

Limitations

- biology makes compromises between different goals
- biology sometimes fails
- some natural mechanisms are not well understood
- well-defined problems can be solved by better means

Swarm Intelligence

"The emergent collective intelligence of groups of simple agents."

(Bonabeau et al, 1999)

Examples

- group foraging of social insects
- cooperative transportation
- division of labour
- nest-building of social insects
- collective sorting and clustering

Swarm Intelligence

Analogies in IT and social insects

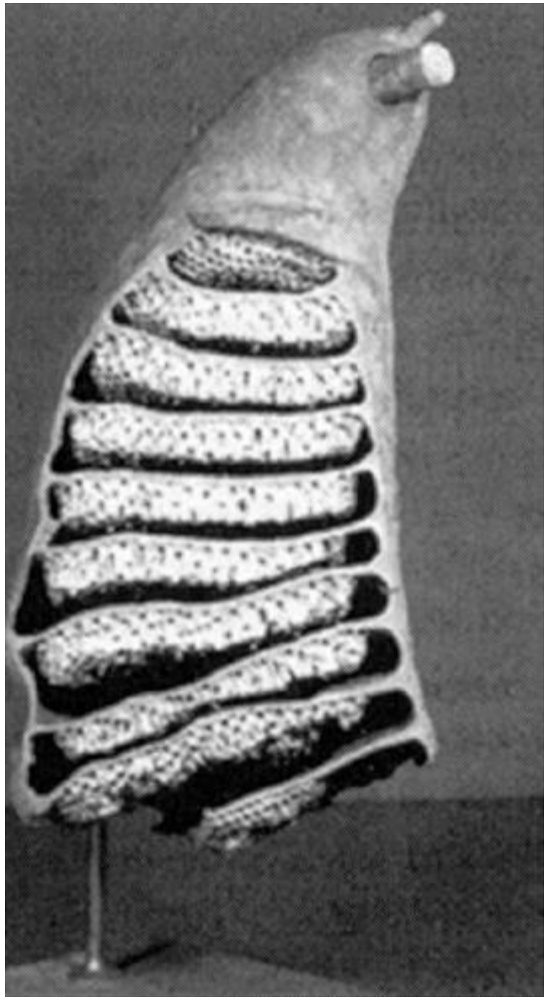
- distributed system of interacting autonomus agents
- goals: performance optimization and robustness
- self-organized control and cooperation (decentralized)
- division of labour and distributed task allocation
- indirect interactions

Swarm Intelligence

The 3 step process

- **identification of analogies**: in swarm biology and IT systems
- **understanding**: computer modelling of realistic swarm biology
- **engineering**: model simplification and tuning for IT applications

Model Examples



Model Examples

Ants

Why are ants interesting?

- ants solve complex tasks by simple local means
- ant productivity is better than the sum of their single activities
- ants are ‘grand masters’ in search and exploitation

Which mechanisms are important?

- cooperation and division of labour
- adaptive task allocation
- work stimulation by cultivation
- pheromones

Self-Organization

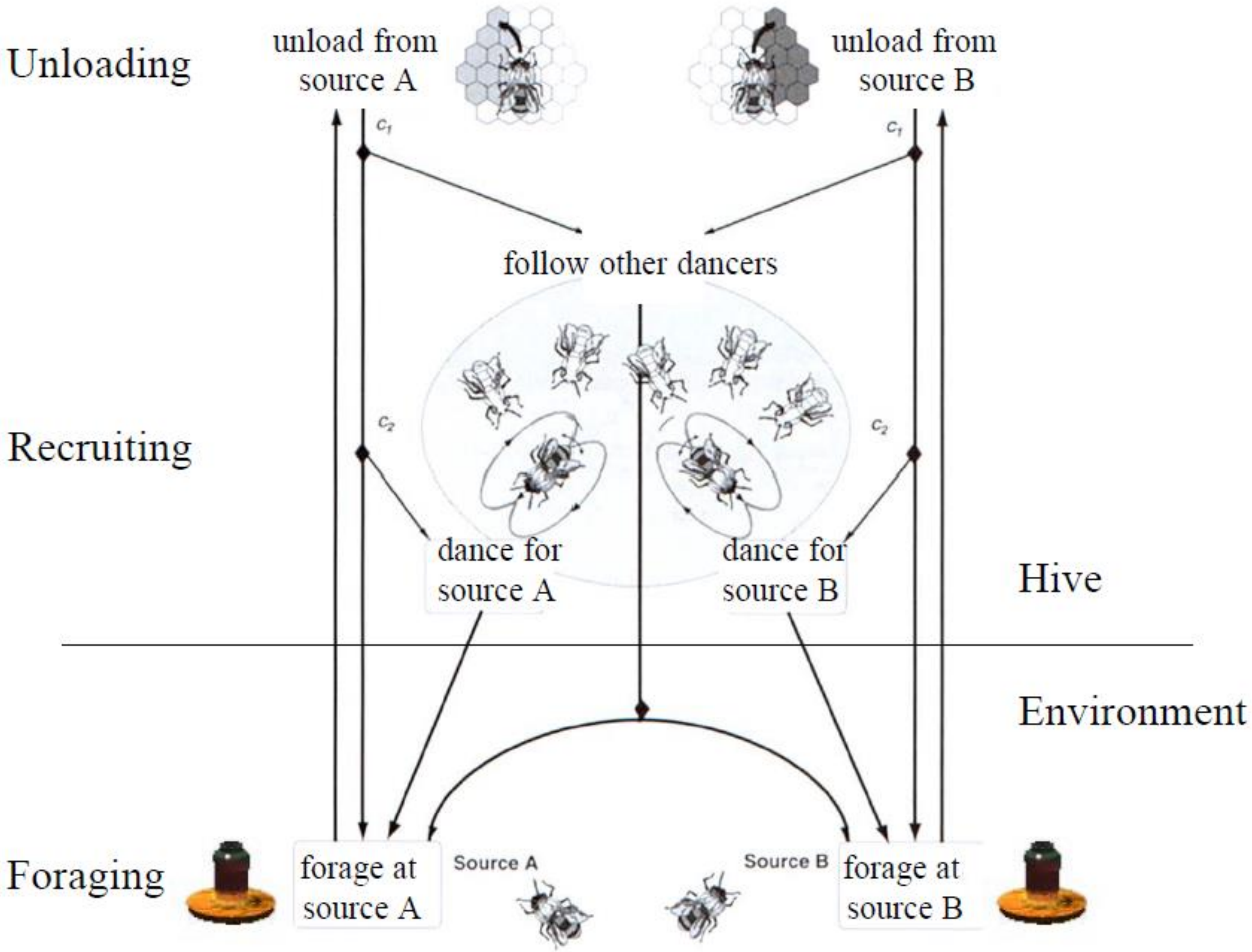
‘Self-organization is a set of dynamical mechanisms whereby structures appear at the global level of a system from interactions of its lower-level components.’

(Bonabeau et al, in Swarm Intelligence, 1999)

Self-Organization

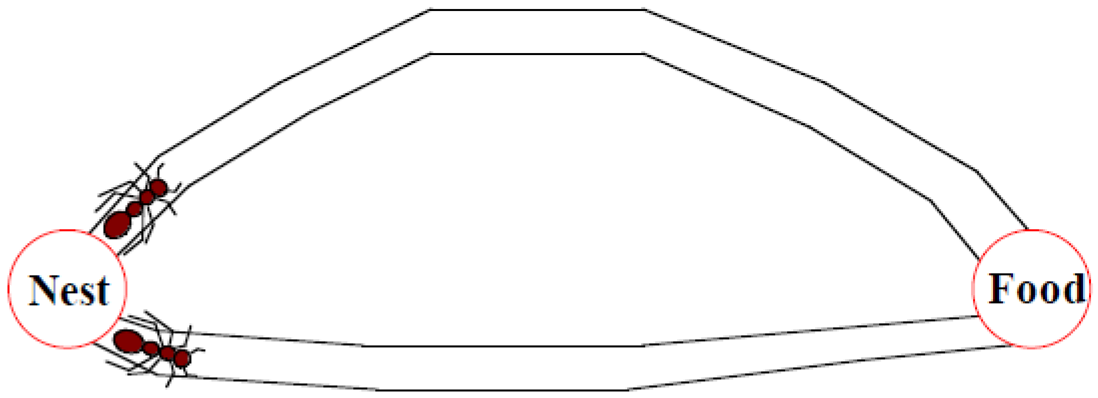
- positive feedback (amplification)
- negative feedback (for counter-balance and stabilization)
- amplification of fluctuations (randomness, errors, random walks)
- multiple interactions

Self-Organization



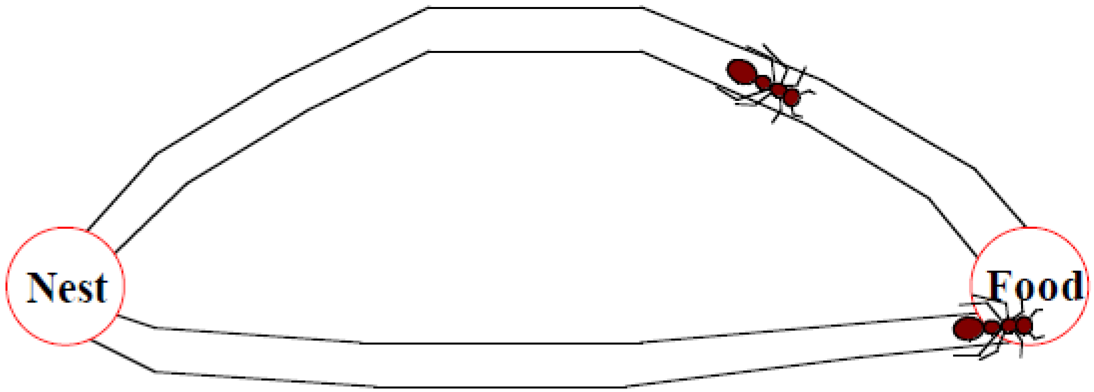
Ant Foraging

Cooperative search by pheromone trails

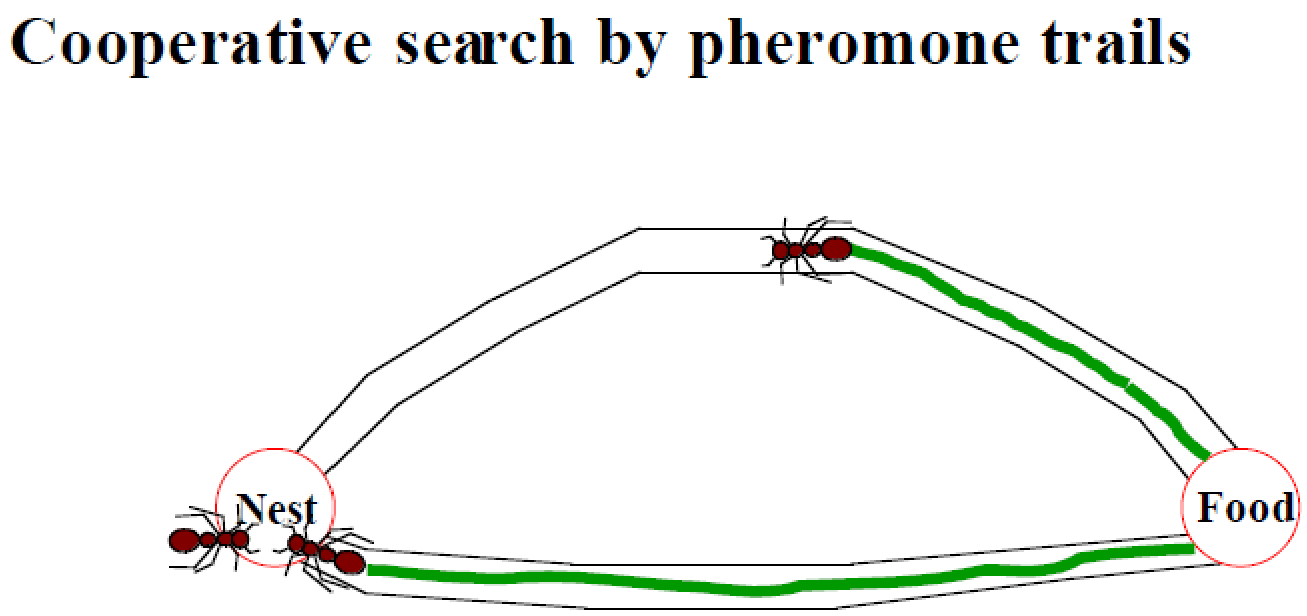


Ant Foraging

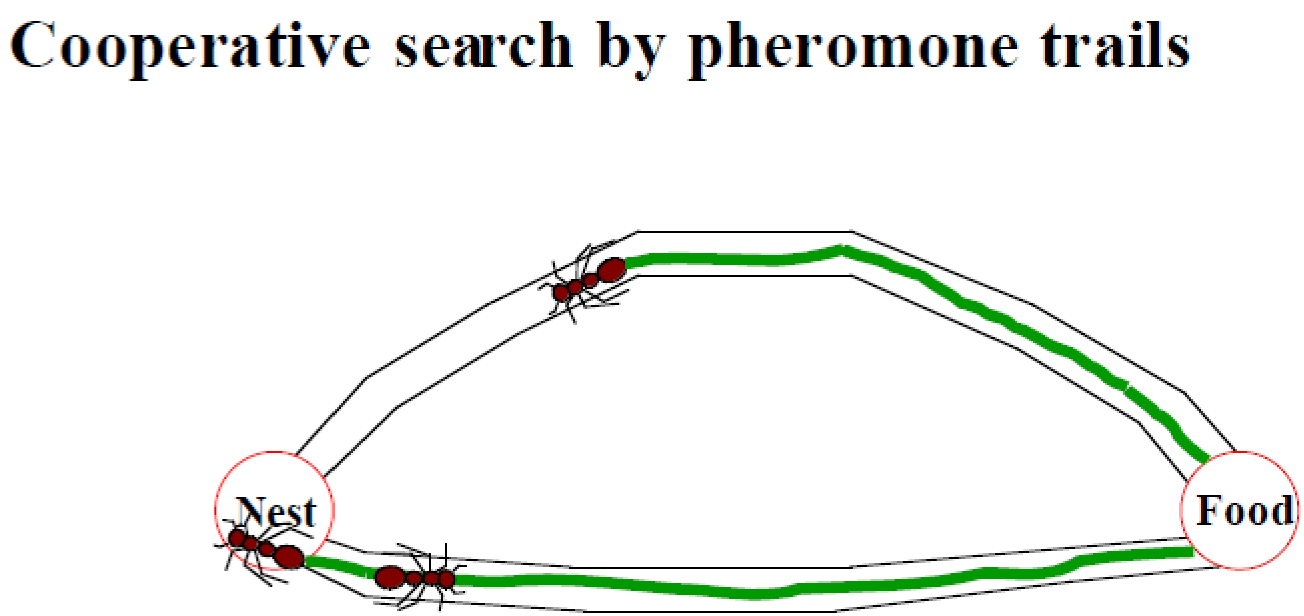
Cooperative search by pheromone trails



Ant Foraging

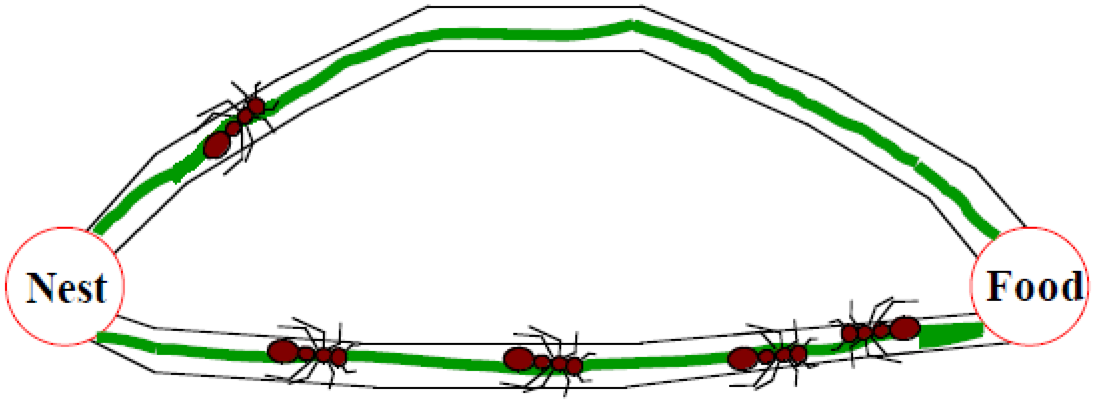


Ant Foraging



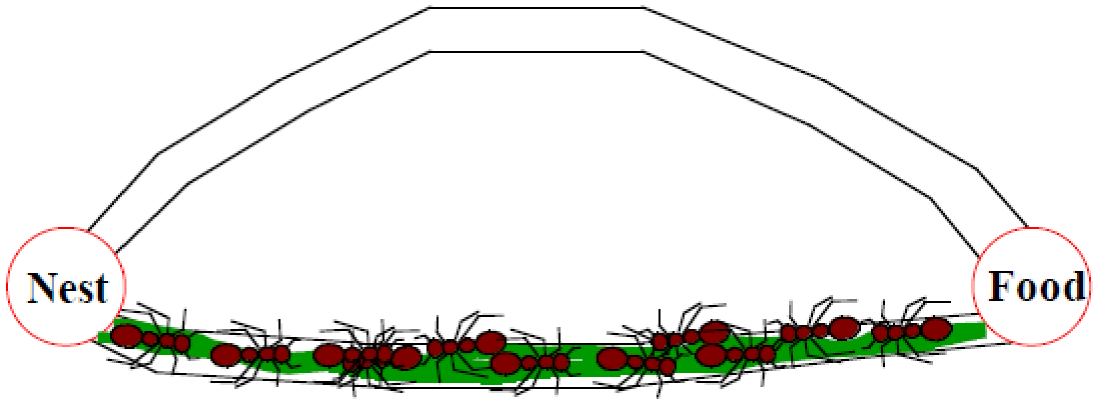
Ant Foraging

Cooperative search by pheromone trails



Ant Foraging

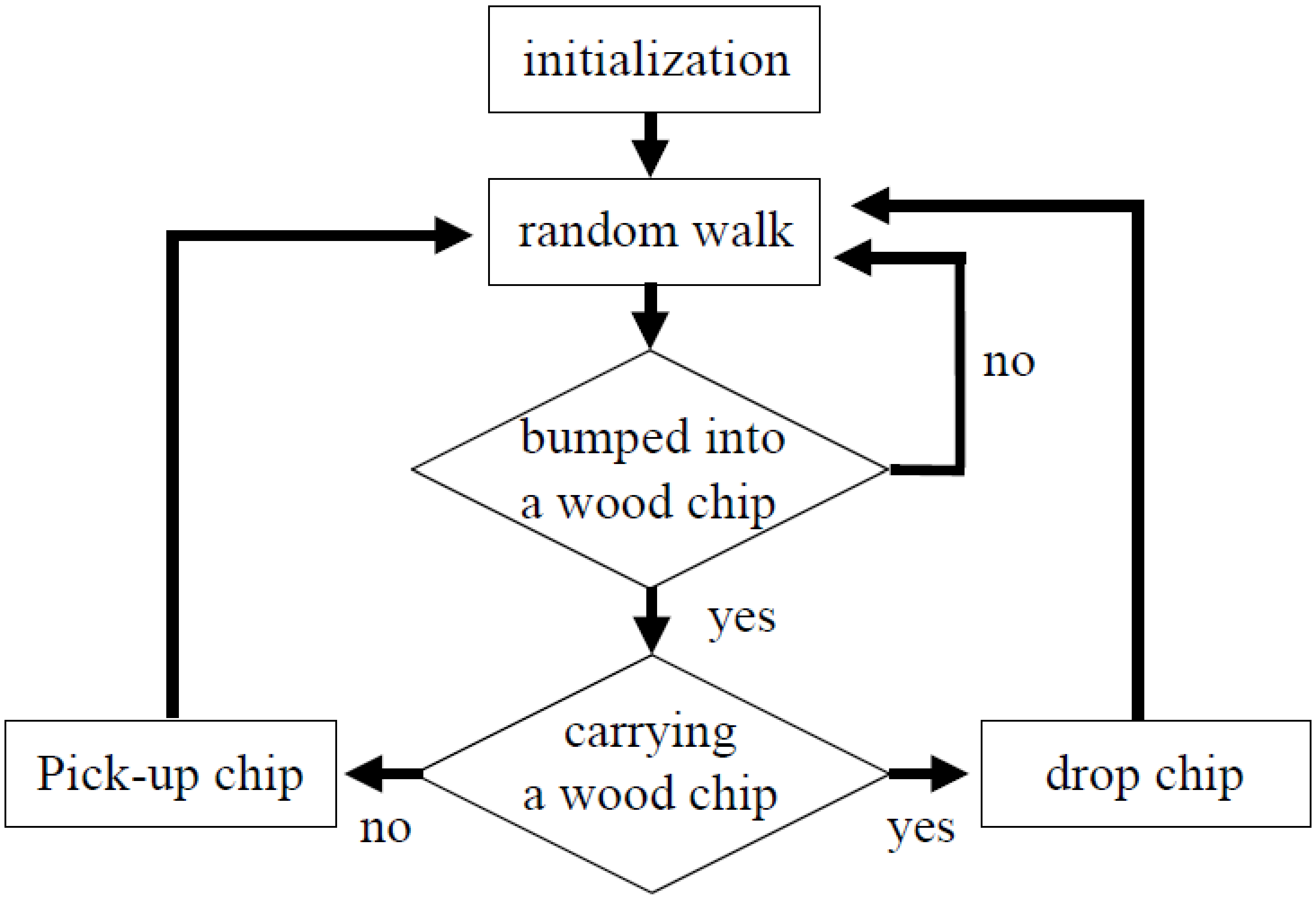
Cooperative search by pheromone trails



Characteristics of Self-Organization

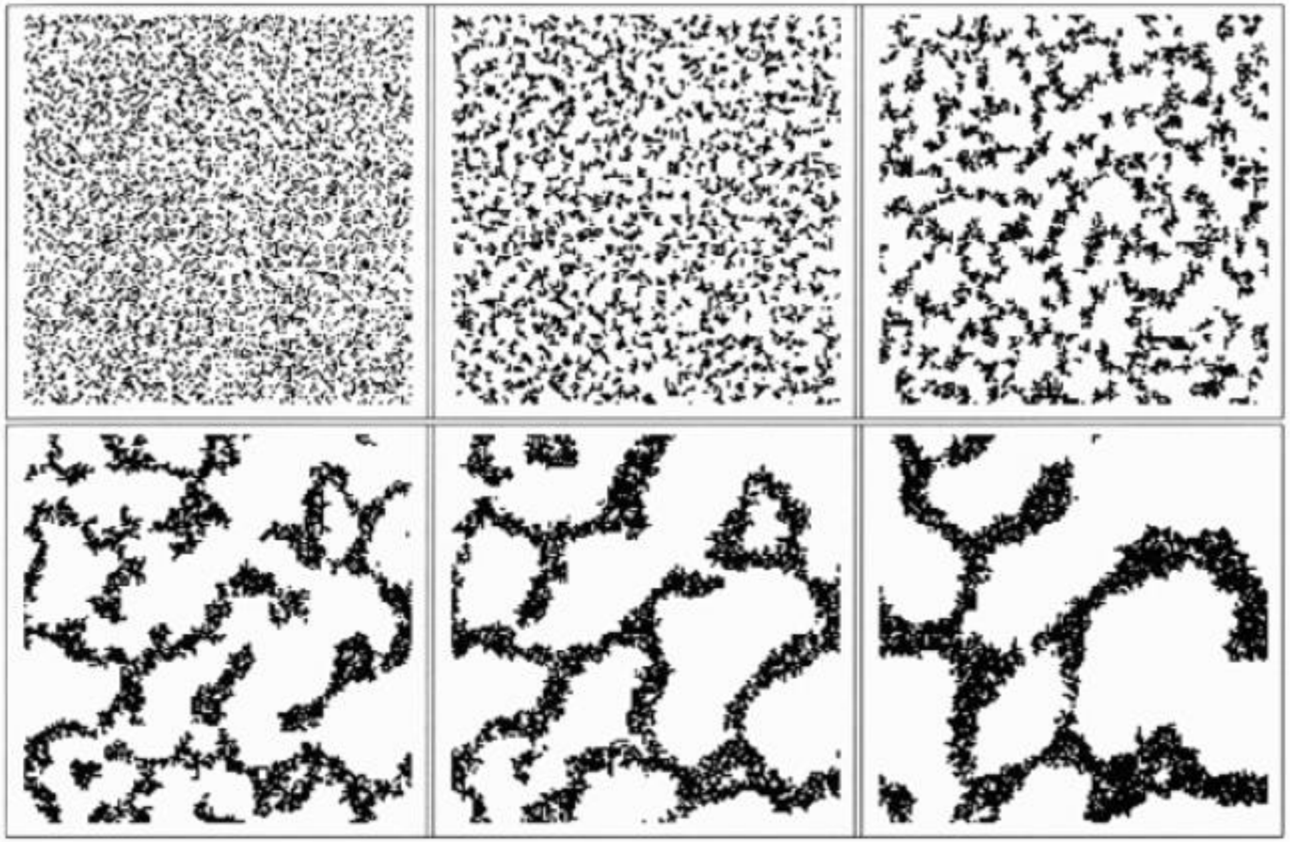
- structure emerging from a homogeneous startup state
- multistability - coexistence of many stable states
- state transitions with a dramatical change of the system behaviour

Termites Simulation



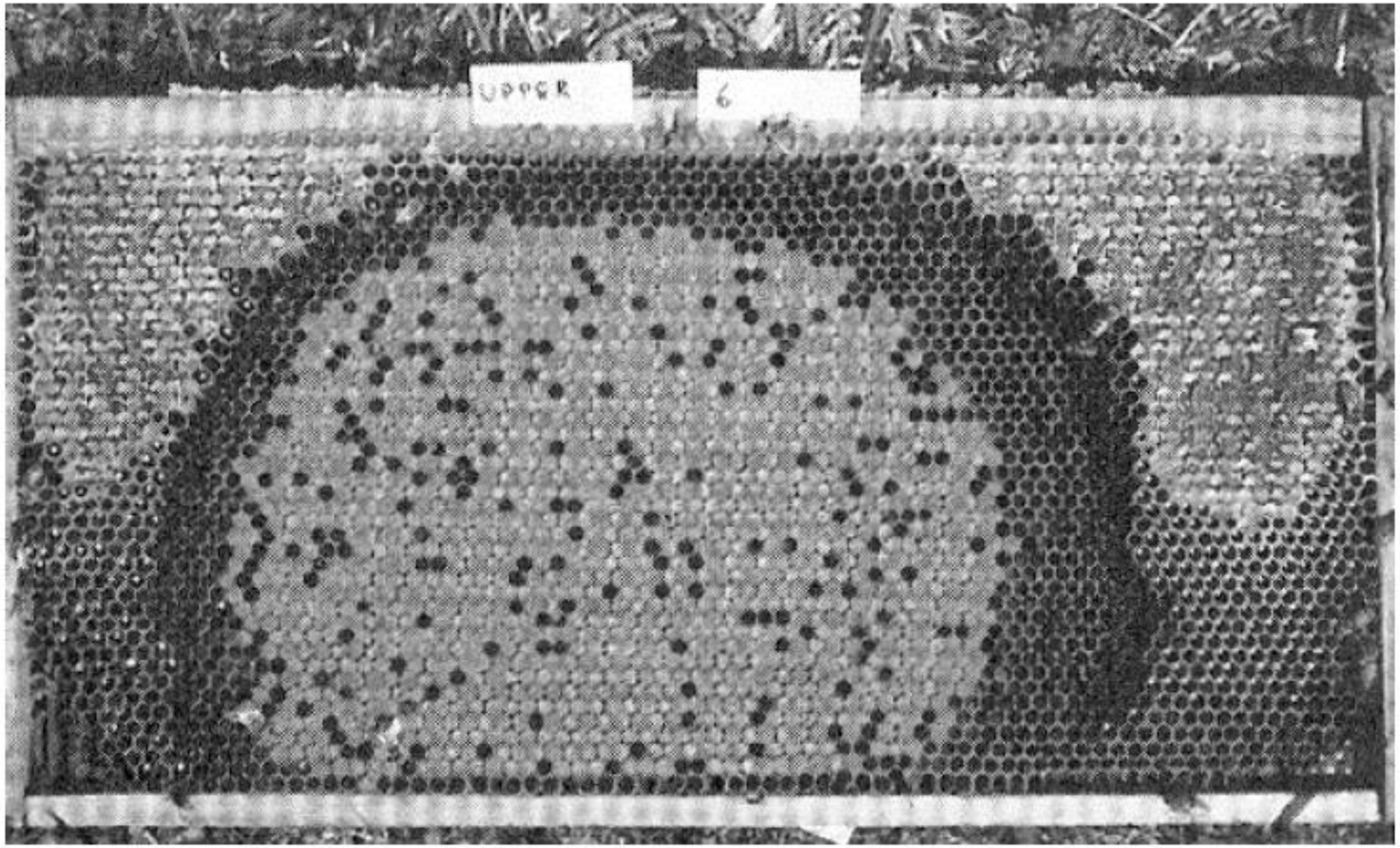
(Mitchel Resnick, 1994)

Termites Simulation



(Mitchel Resnick, 1994)

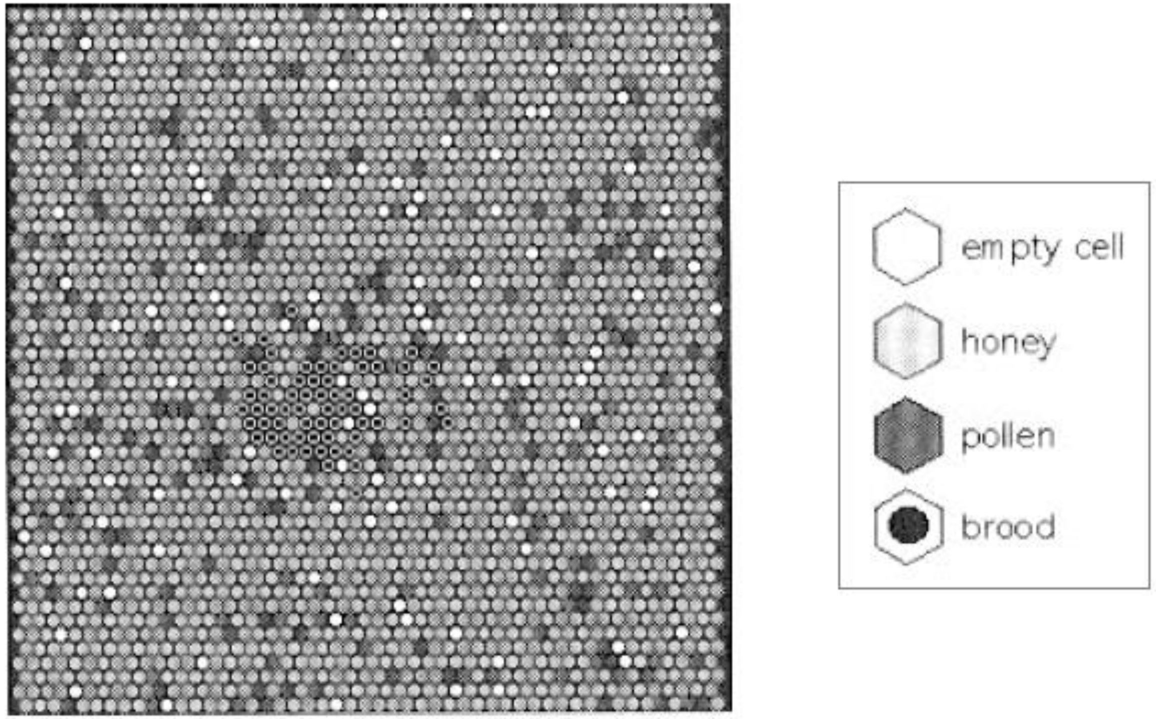
Honey Bees Nest Building



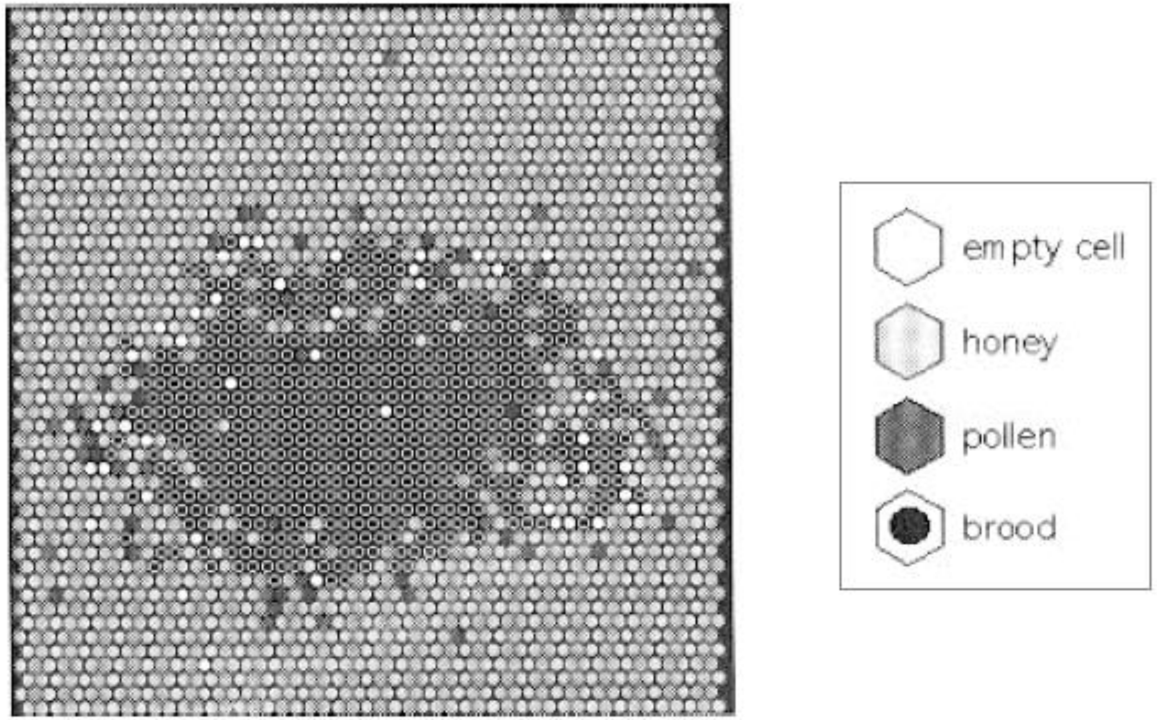
Honey Bees Nest Building

- the queen moves randomly over the combs
- eggs are more likely to be layed in the neighbourhood of brood
- honey and pollen are deposited randomly in empty cells
- four times more honey is brought to the hive than pollen
- removal ratios for honey: 0.95; pollen: 0.6
- removal of honey and pollen is proportional to the number of surrounding cells containing brood

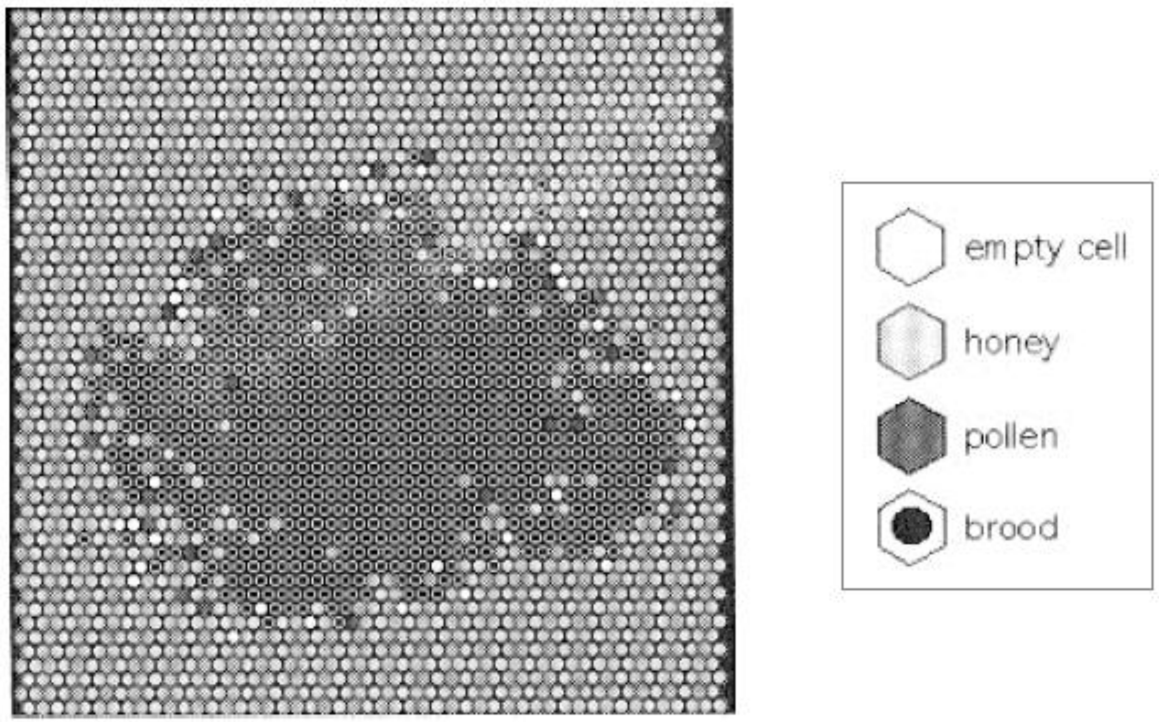
Honey Bees Nest Building



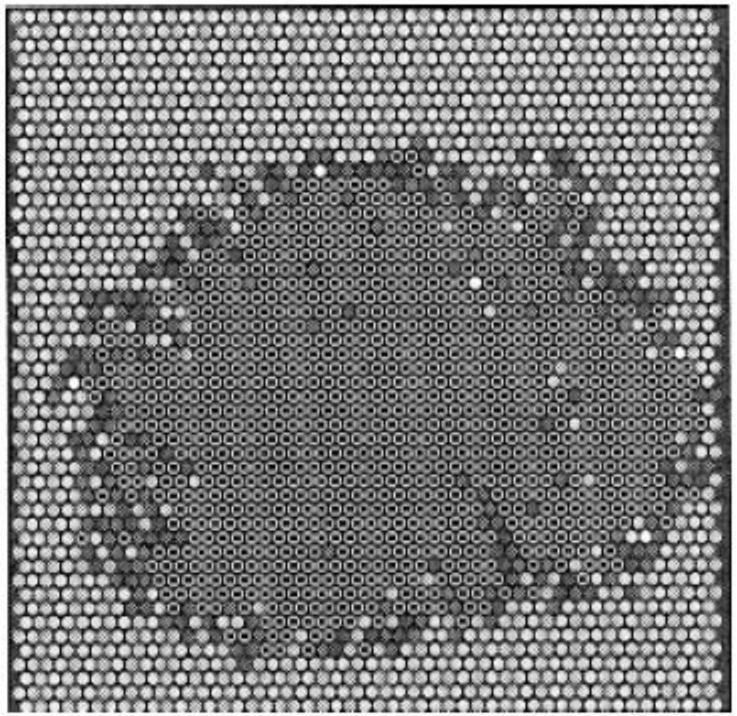
Honey Bees Nest Building

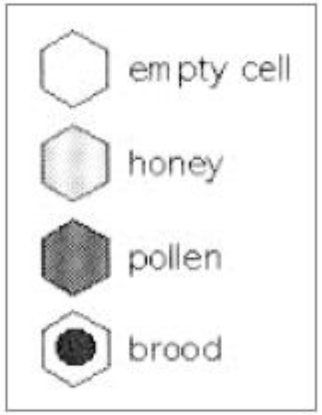


Honey Bees Nest Building

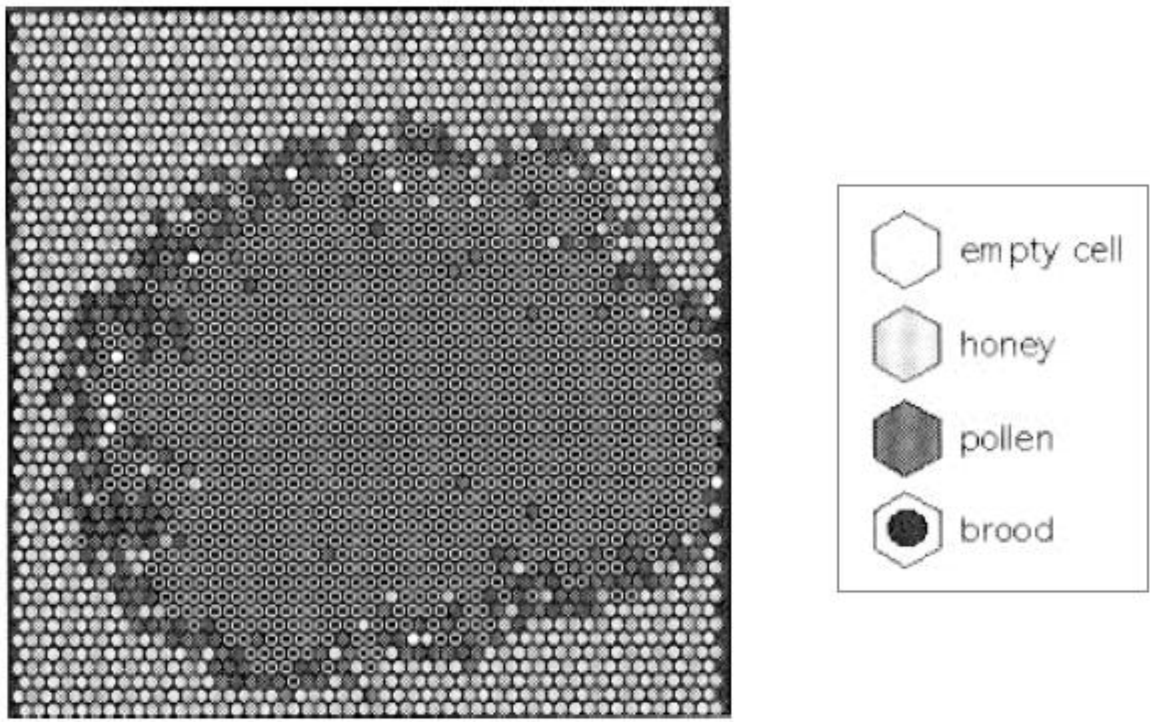


Honey Bees Nest Building

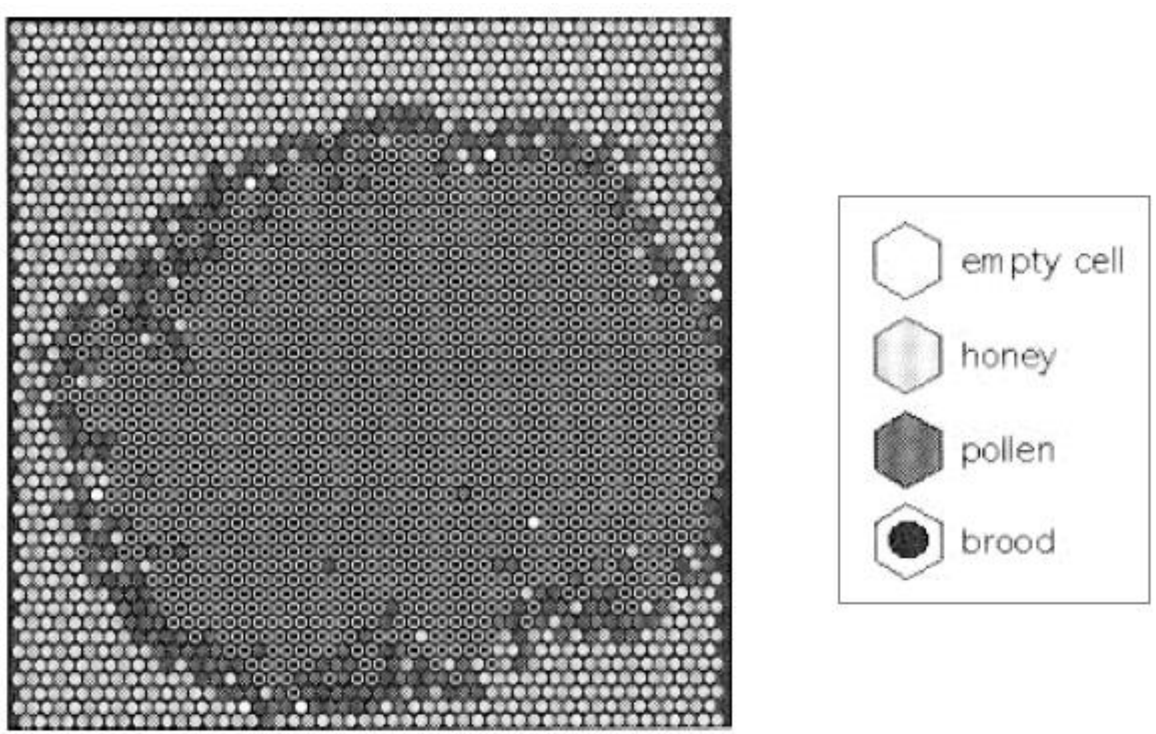




Honey Bees Nest Building



Honey Bees Nest Building



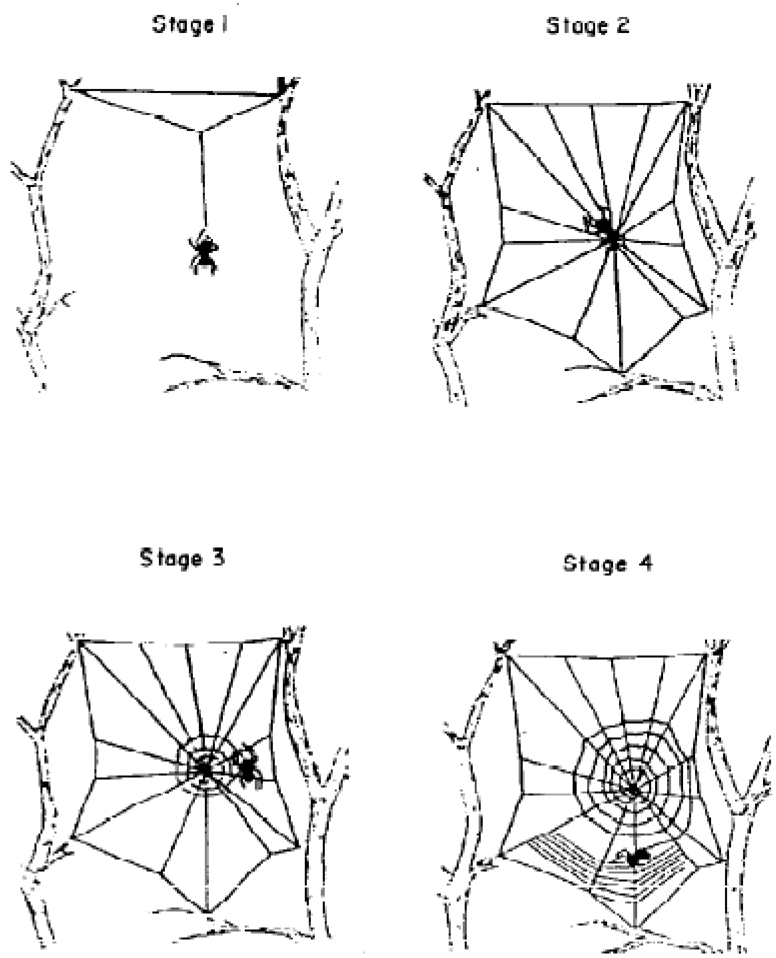
Stigmergy

Stigmergy: *stigma* (sting) + *ergon* (work)
= 'stimulation by work'

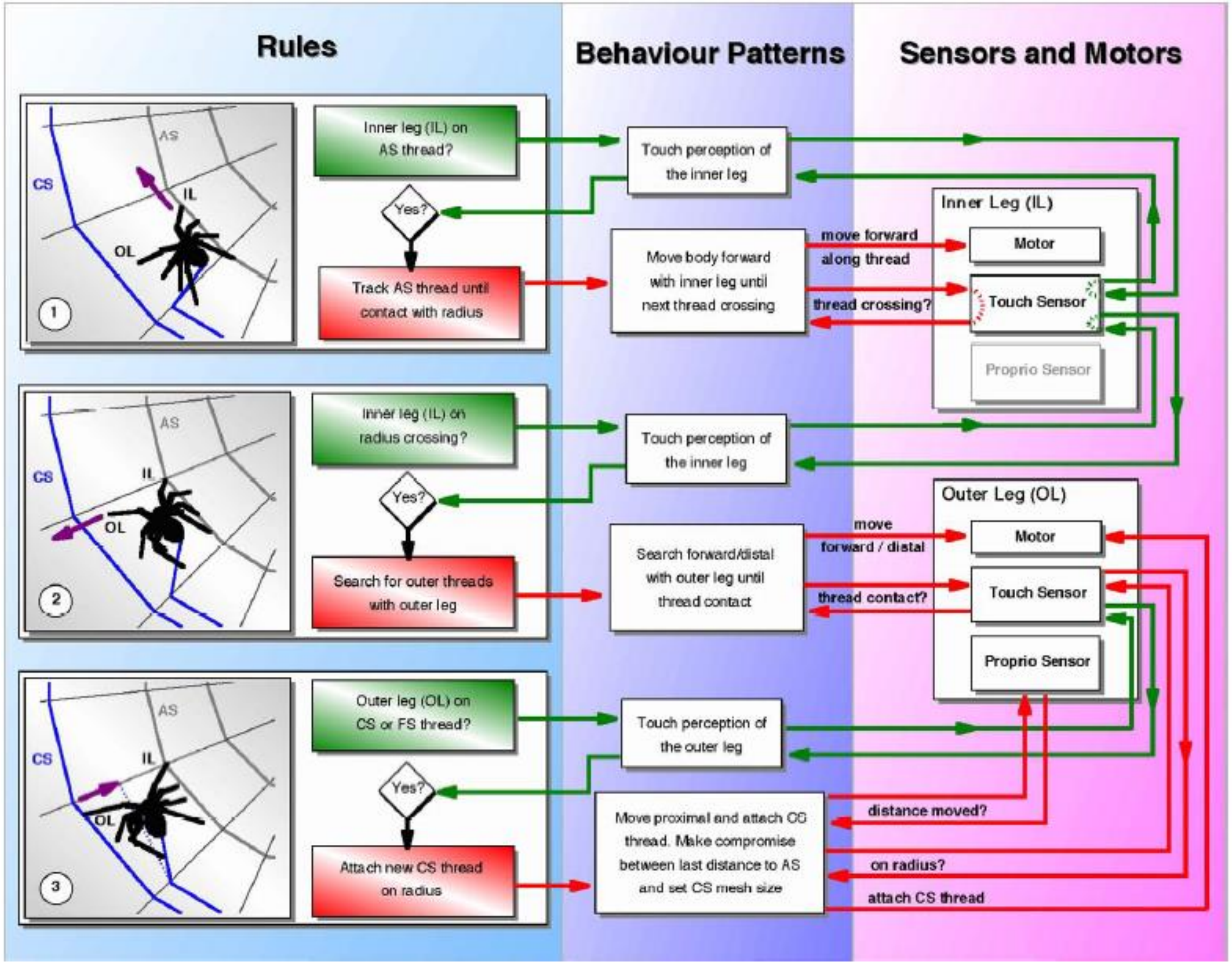
Characteristics of stigmergy

- indirect agent interaction modification of the environment
- environmental modification serves as external memory
- work can be continued by any individual
- the same, simple, behavioural rules can create different designs according to the environmental state

Stigmergy in Spiders



Stigmergy



Rules
Behaviour Patterns
Sensors and Motors
AS
CS
IL
OL
1
Inner leg (IL) on AS thread?
Yes?
Track AS thread until contact with radius
Touch perception of the inner leg
Move body forward with inner leg until next thread crossing
move forward along thread
thread crossing?
Inner Leg (IL)
Motor
Touch Sensor
Proprio Sensor
2
Inner leg (IL) on radius crossing?
Search for outer threads with outer leg
Touch perception of the inner leg
Search forward/distal with outer leg until thread contact
move forward / distal
thread contact?
Outer Leg (OL)
Motor
Touch Sensor
Proprio Sensor
3
Outer leg (OL) on CS or FS thread?
Attach new CS thread on radius
Touch perception of the outer leg
Move proximal and attach CS thread. Make compromise between last distance to AS and set CS mesh size
distance moved?
on radius?
attach CS thread

Motivation

Motivation and methods in biologically inspired IT

- there are analogies in distributed computing and social insects
- biology has found solution to hard computational problems
- biologically inspired computing requires:
    - identification of analogies
    - computer modelling of biological mechanisms
    - adaptation of biological mechanisms for IT applications

Principles

Two principles in swarm intelligence

- self-organization is based on:
  - activity amplification by positive feedback
  - activity balancing by negative feedback
  - amplification of random fluctuations
  - multiple interactions
- stigmergy - stimulation by work - is based on:
  - work as behavioural response to the environmental state
  - an environment that serves as a work state memory
  - work that does not depend on specific agents