(Concepts of) Machine Learning

Lecture 6: Advanced learning schemes & evolution

George Magoulas
gmagoulas@dcs.bbk.ac.uk

Contents

- Generalisation of machine learning models/the over-fitting thriller
- Ensemble learning
- Neuro-evolution
  - Hybrid GA
  - Differential evolution
  - Swarm intelligence - Particle swarm

Generalisation

- How does the classifier perform on a problem with respect to both seen and unseen data?
- How many training instances are required for good generalisation?
- What size network gives the best generalisation?
- What kind of neural architecture is best for modelling the underlying problem?
- What learning algorithm can achieve the best generalisation?

Generalisability

- **Generalisation theory** aims at providing general bounds that relate the error performance of a classifier with the number of training points, N, on one hand, and some classifier dependent parameters, on the other. So far, the classifier dependent parameters that we considered were the number of free parameters of the classifier and the dimensionality of the subspace, in which the classifier operates.

- **Generalisability** is the ability of a model to generalise on unseen data points (not used during training). A model is not effective if it is too specific to the training data, i.e., "too specific" means it is over-fitting and its performance on unseen data cannot be guaranteed.

- Let the classifier be a binary one, i.e.,

$$f : \Re^{\lambda} \rightarrow \{0,1\}$$

- Let F be the set of all functions f that can be realised by the adopted classifier (e.g., changing the weights of a given neural network different functions are implemented).
- The Vapnik – Chernovenkis (VC) dimension of a class $\boldsymbol{F}$ is the largest integer k for which $S(F,k) = 2^k$. If $S(F,N)=2^N$,

  we say that the VC dimension is <u>infinite</u>. However, NOT ALL dichotomies can be realised by the set of functions in $\boldsymbol{F}$.
- That is, VC is the largest integer for which the class of functions F can achieve all possible dichotomies, 2^k.

The VC dimension can be considered as an **intrinsic capacity** of the classifier, and only if the number of training vectors **exceeds** this number sufficiently, we can expect good generalisation performance.

- Let ... alised by ... of a ... d).

  we s... However, NOT ALL d... set of functions in $\boldsymbol{F}$.

- That is, V... largest intege... hich the class of functions F can achieve all possible dichotomies, 2^k.

- The VC dimension may or may **not** be related to the Input dimension λ and the number of free parameters.
  - Perceptron: $VC = \lambda + 1$
  - Multilayer network with hard limiting activation function

Floor operation: returns the largest integer less than its argument

$$2\left\lfloor \frac{k_n^h}{2} \right\rfloor \lambda \leq VC \leq 2k_w \log_2(ek_n)$$

where k_n^h is the total number of hidden layer nodes, k_n the total number of nodes, and k_w the total number of weights; e is the base of the natural logarithm; λ is the number of input nodes.

- A useful heuristic in practice is that the number of training patterns is about 10 times the VC or 10 times the number of weights, k_w, in the case of a multilayer network.

Generalisation Performance

- Let $P_e^N(f)$ be the error rate of classifier f, based on the N training points, also known as empirical error.
- Let $P_e(f)$ be the true error probability of f (also known as generalisation error), when f is confronted with data outside the finite training set.
- Let P_e be the minimum error probability that can be attained over ALL functions in the set $\boldsymbol{F}$.
  - for a large N :
    - $P_e^N(f)$ is close to $P_e(f)$, with high probability.
    - $P_e(f^*)$ is close to P_e , with high probability.
      - Let f^* be the function resulting by minimising the empirical (over the finite training set) error function.

Evaluating generalisability

- One systematic method to check for this is **cross-validation**. Cross-validation is a strategy where the model-building process is applied on a subset of the data such that the model has not "seen" all the data available. This learning process is then followed by an evaluation step on the "unseen" portion of the data. This is an approach to ensure that the model is not over-fitted to the data and an acceptable degree of generalisability is ensured.

- One approach is **the holdout method** where the data set is separated into two random sets, called the training set and the testing set. And the model is trained on the former and evaluated on the latter.
- Another version is the **k-fold cross-validation** approach, where the data set is divided into k subsets, and the holdout method is repeated k times. Each time, one of the k subsets is used as the test set and the other k-1 subsets are put together to form a training set. Then the average error across all k trials is computed.

Validation of a learning system

The goal is to estimate the error probability of the designed classification system

- Error Counting Technique
  - Let M classes
  - Let N_i data points in class ω_i for testing.

    $$\sum_{i=1}^{M} N_i = N$$ the number of test points.
  - Let P_i the probability error for class ω_i
  - The classifier is assumed to have been designed using another **independent** data set
  - Assuming that the feature vectors in the test data set are independent, the probability of k_i vectors from ω_i being in error follows the binomial distribution

http://en.wikipedia.org/wiki/Binomial_distribution

$$\text{prob}\{k_i \text{ in } \omega_i \text{ wrongly classified}\} = \binom{N_i}{k_i} P_i^{k_i} (1 - P_i)^{N_i - k_i}$$

- Since the P_i's are not known, estimate P_i by maximising the above binomial distribution which gives

$$\boxed{\hat{P}_i = \frac{k_i}{N_i}}$$

- Thus, count the errors and divide by the total number of test points in class

- A theoretically derived estimate of a sufficient number N of the test data set is

$$N \approx \frac{100}{P}$$

Thus, for $P \approx 0.01$, $N \approx 10000$. For $P \approx 0.03$, $N \approx 3000$

- Exploiting the finite size of the data set.

  - Resubstitution method:

    Use the same data for training and testing. It underestimates the error. The estimate improves for large N (data set size) and large $\frac{N}{l}$ ratios.

    $\leftarrow$ (dimension of feature space)

  - Holdout Method: Given N divide it into:

    N_1: training points

    N_2: test points

    $N=N_1+N_2$

    - Problem: Less data both for training and test

- Leave-one-out Method

The steps:

- Choose one sample out of the N. Train the classifier using the remaining N-1 samples. Test the classifier using the selected sample. Count an error if it is misclassified.
- Repeat the above by excluding a different sample each time.
- Compute the error probability by averaging the counted errors

- Advantages:
  - uses all data for testing and training
  - ensures independence between test and training samples
- Disadvantages:
  - Complexity in computations high
- Variants of it exclude $k>1$ points each time, to reduce complexity

Improving generalisation in Neural Networks

(i) **Network pruning:** remove weights without hurting network's performance. This leads to better generalisation and faster learning.

- ***Eliminate small weights***: determine what is "small"; what happens if the solution is sensitive to some small weights?
- ***Delete weights that have the least effect on the solution***
  - First train with backpropagation
  - Then delete the weights with the smallest **saliency**
  - The reduced network is retrained to obtain final solution.
- ***Remove nodes with lower relevance:*** define relevance as the difference in the value of the cost function without the node and with the node in the network.

Optimal Brain Damage (OBD)

$$saliency = \delta E_{ji} w_{ji}^2$$

Saliency for weight w_{ji}, where δE measures sensitivity of the cost function to small perturbations in w_{ji}

$$\delta E_{ji} \approx \frac{\partial^2 E}{\partial w_{ji}^2}$$

Sensitivity measure is approximated by second derivative.

Improving generalisation

(ii) Weight decay and elimination: Improve generalisation by favouring simple structures. Add a term to the cost function to penalise complex structures.

- In the **weight decay** approach the number of weights is reduced by encouraging near zero weights in the learning process. This is achieved by adding a term to penalise nonzero weights

Small positive constant

$$E = \frac{1}{2}\sum_j (t_j - y_j)^2 + \frac{\zeta}{2}\sum_i w_i^2$$

  - The weight is decayed in an amount proportional to its magnitude. Not all weights are decayed to small values. Only those weights which are not reinforced do so.

Improving generalisation

- In the **weight elimination** the cost function is formulated as

$$E = \frac{1}{2}\sum_j (t_j - y_j)^2 + \overset{\text{Small positive constant}}{\zeta} \sum_i \frac{w_i^2 / \overset{\text{Fixed weight normalisation factor}}{w_0^2}}{1 + (w_i^2 / w_0^2)}$$

  - When $w_i >> w_0$, the penalty terms is close to unity and this criterion essentially counts the number of weights
  - When $w_i << w_0$, the penalty term is proportional to w_i^2 and the criterion behaves like the weight decay criterion.

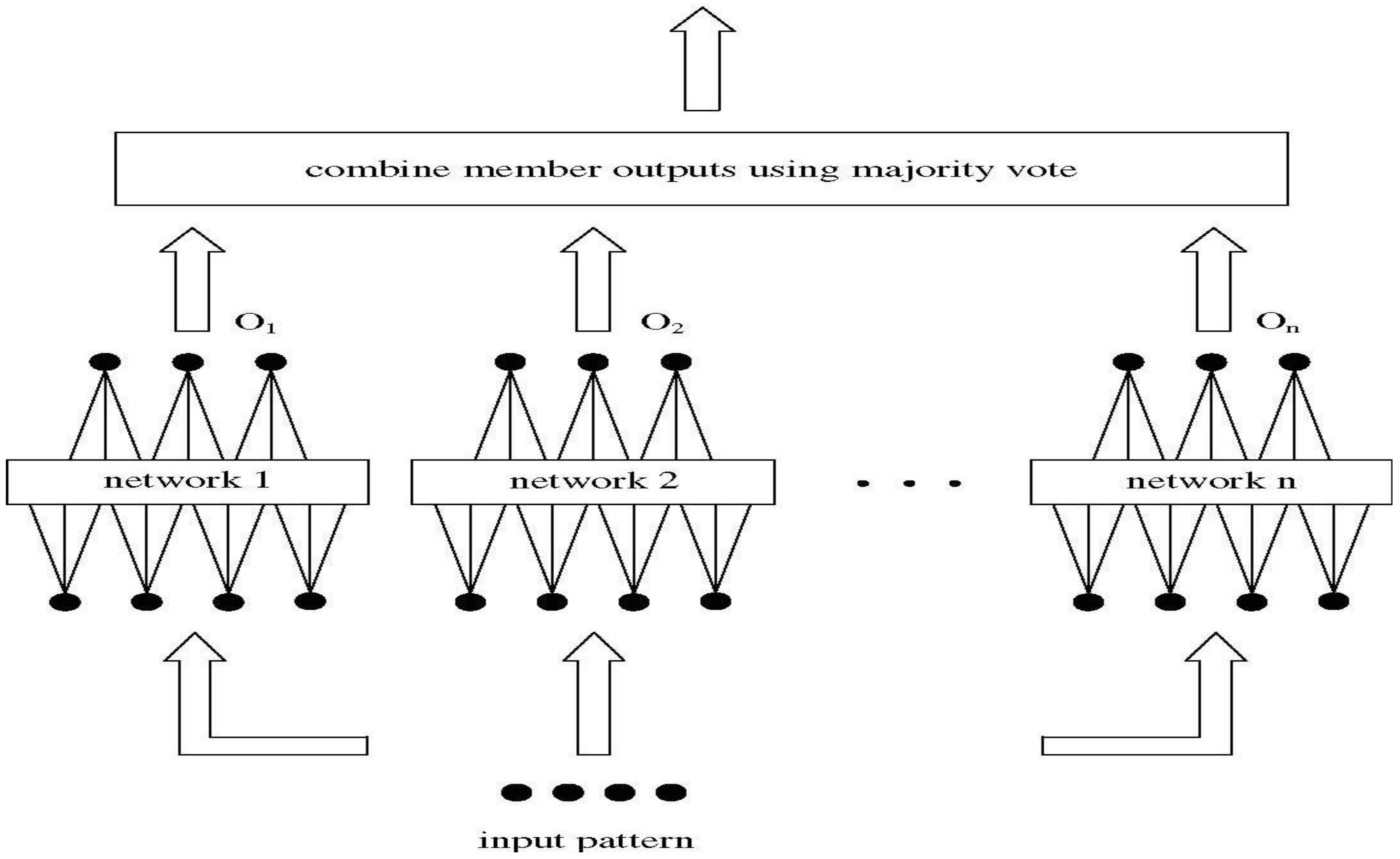
Ensemble learning

- Ensemble based methods enable an increase in generalisation performance by combining several individual models trained on the same task.
- Approach has been justified both theoretically and empirically.
- The creation of an ensemble is often divided into two steps : (i) generate individual ensemble members and (ii) appropriately combine individual members outputs to produce the output of the ensemble.

Weak learnability

- Schapire in [The strength of weak learnability. Mach. Learn. 5, 197–227 (1990)] showed that a model of learnability in which the learner is only required to perform slightly better than guessing is as strong as a model in which the learner's error can be made arbitrarily small.
- The proof of this result was based on the filtering of the distribution in a manner causing the weak learning algorithm to eventually learn nearly the entire distribution.
- Strong theory for boosting in binary classification and multiclass settings in batch and online learning and group models.

Ensembles of Multilayer Networks



Approaches to create neural ensembles

- **Simple Network Ensemble (SNE):** consists of *n* networks where each network uses the full training set and differs only in its random initial weight settings.
- **More advanced approach:** train networks on different subsets of the training set, e.g. using *bagging*, where each training set is created by resampling and replacement of the original one with uniform probability, or *boosting*, which also uses resampling of the training set, but the data points previously poorly classified, receive a higher probability
- **Diverse Network Ensemble (DNE):** use networks that disagree partially on their decisions.

Boosting

Technique first introduced in [Schapire (1990), JML; Schapire, Freund (1996), Experiments with a new boosting algorithm, 13th ICML]

- *Learners (models) are trained sequentially, using a sample from the original dataset, with the prediction error from the previous round affecting the sampling weight for the next round.*
- *After each round of boosting, the decision can be made to terminate and use a set of calculated weights to apply as a linear combination of the newly created set of learners.*

Generic boosting method

```
$D_0 \leftarrow uniform()$
$t \leftarrow 0$
while $notstoppingConditionReached()$ do
    $currentSubset \leftarrow pickFromSet(wholeSet, D_t)$
    $h_t \leftarrow newClassifier(currentSubset)$
    $\epsilon_t \leftarrow getError(h_t, wholeSet)$
    $\alpha_t \leftarrow learnerCoefficient(\epsilon_t)$
    $D_{t+1} \leftarrow nextDistribution(D_t, h_t, \alpha_t, wholeSet)$
    $t \leftarrow t + 1$
end while
$H \leftarrow aggregate(h_{0..t}, \alpha_{0..t})$
```

Each boosting variant builds a distribution of weights D_t, which is used to sample from the training set...

Re-sampled dataset is used to train a new classifier h_t, which is then incorporated in the group, with a weight α_t based on its classification error ϵ_t.

is updated at each iteration to increase the importance of the examples that are harder to classify correctly.

The main differences between each boosting variant are in how the `getError`, `learnerCoefficient`, `nextDistribution` and `aggregate` functions are implemented.

Diversity

Networks belonging to an ensemble are thought to be diverse with respect to a test set if they make different generalisation errors on that test set. Different patterns of generalisations can be produced when networks are trained either on different training sets, or from different initial conditions, or with different numbers or hidden nodes, or using different algorithms.

Diverse neural ensemble

A simple implementation of the DNE-based method:

Step1: Create *n* MNs where each one uses the same training set and differs only in its random initial weights.
Step2: Select the *k* Networks (*k<n)*, which when they do fail to classify the data, they fail on different inputs patterns so that failures on one network can be compensated by successes of others.
Step3: Combine the ensemble members' outputs using majority voting to get ensemble's output.

Proteins localisation

- Problem Description: In order to function properly, proteins must be transported to various localisation sites within a particular cell. Description of protein localisation provides information about each protein that is complementary to the protein sequence and structure data. (Ref. 7 for details)
- Characteristics: The datasets of the proteins are drastically imbalanced.

Yeast Dataset

Patterns	Class
463	cyt
5	erl
35	exc
44	me1
51	me2
163	me3
244	mit
429	nuc
20	pox
30	vac

Results in the Yeast Problem

Patterns	Class	HN	KNN	FNN	DNE
463	cyt	74.3%	70.7%	66.7%	80.5%
5	erl	60.0%	0.0%	99.6%	100%
35	exc	45.7%	62.9%	62.7%	74.3%
44	me1	63.6%	75.0%	82.9%	100%
51	me2	15.7%	21.6%	47.8%	70.6%
163	me3	85.3%	74.9%	85.6%	92.65%
244	mit	47.1%	57.8%	61.3%	71.4%
429	nuc	35.7%	50.7%	57.7%	70.1%
20	pox	0.0%	55.0%	54.6%	55.0%
30	vac	10.0%	0.0%	4.1%	20.0%
Mean		**43.7%**	**46.8%**	**62.3%**	**73.5%**

Neuro-evolution

- Although neural networks are used for solving a variety of problems, they still have some limitations.
- One of the most common is associated with neural network training. The back-propagation learning algorithm cannot guarantee an optimal solution. In real-world applications, the back-propagation algorithm might converge to a set of sub-optimal weights from which it cannot escape. As a result, the neural network is often unable to find a desirable solution to a problem at hand.

Hybrid Genetic Algorithms- 1

```
{
//start with an initial time
    t:=0;
//initialise a usually random population of
//individuals
STAGE 1: initpopulation P(t);
//evaluate fitness of all individuals of population
STAGE 2: evaluate P(t);
//test for termination criterion (time, fitness,
//etc.)
    while not done do
        //increase the time counter
        t:=t+1;
//select a sub-population for offspring production
STAGE 3: P':=selectparents P(t);
        //recombine the "genes" of selected parents
STAGE 4: recombine P'(t);
        //perturb the mated population stochastically
STAGE 5: mutate P'(t);
        //evaluate the new fitness
STAGE 6: evaluate P'(t);
        //select the survivors from actual fitness
STAGE 7: P:=survive P,P'(t);
    end
}
```

- Darwinian and Lamarckian principles are combined into hybrid genetic algorithms:

  The elements of the population are selected in "Darwinian" fashion from generation to generation, but can become better by modifying their parameters in "Lamarckian" way, that is, by performing some hill-climbing steps before recombining.

- Thus the use of GA is related to the concept of "evolution" of a population of individuals and that of hill-climbing methods to the concept of "life" of each individual.

Hybrid Genetic Algorithms - 2

Algorithmic description

a) Randomly generate an *initial population* of *chromosomes*
b) Compute the *fitness* of every member of the current population. For example, this can be used to initialise the hill-climbing method
c) Make an intermediate population by extracting members out of the current population by means of a *Lamarckian strategy*
d) Generate the new population by applying the *genetic operators* (crossover, mutation) to this intermediate population (for example the termination points of the hill-climbing method)
e) If there is a member of the current population that satisfies the problem requirements then stop, otherwise go to step (b)

Hill-climbing method -1

- Technique applied on a single point (current point) in the search space aiming at iterative improvement.
- During each iteration a new point is selected from the neighbourhood of the current point. If the new point provides better value in light of evaluation function, the new point becomes the current point.
- Otherwise, some other neighbour is selected and tested against the current point.
- Terminates if no further improvement is possible.

Hill-climbing method – 2: Stochastic version

- Instead of checking all points in the neighbourhood of a current point and selecting the best one, select only one point from this neighbourhood.
- Accept this point with some probability that depends on the relative merit of these points.

$$P(\mathbf{x}_k \rightarrow \mathbf{x}_{k+1}) = \frac{1}{1 + e^{\frac{f(\mathbf{x}_k) - f(\mathbf{x}_{k+1})}{T}}}$$

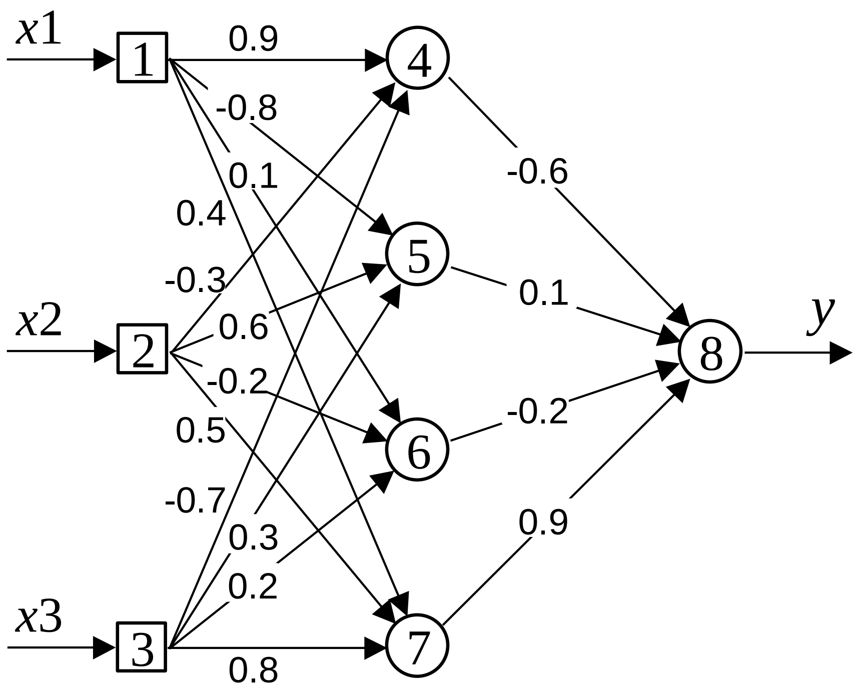
Evolving populations of neural networks with GA

Algorithmic description

1. Choose an appropriate error measure
2. Choose strings corresponding to all the weights and biases
3. Generate a pool of such strings, chosen randomly
4. Evaluate the error for each string
5. Convert the error to fitness
6. Apply the GA operators to obtain next generation
7. Repeat until find a string with sufficiently low error (high fitness)

Encoding a set of weights in a chromosome

From neuron:	1	2	3	4	5	6	7	8
To neuron: 1	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
4	0.9	-0.3	-0.7	0	0	0	0	0
5	-0.8	0.6	0.3	0	0	0	0	0
6	0.1	-0.2	0.2	0	0	0	0	0
7	0.4	0.5	0.8	0	0	0	0	0
8	0	0	0	-0.6	0.1	-0.2	0.9	0



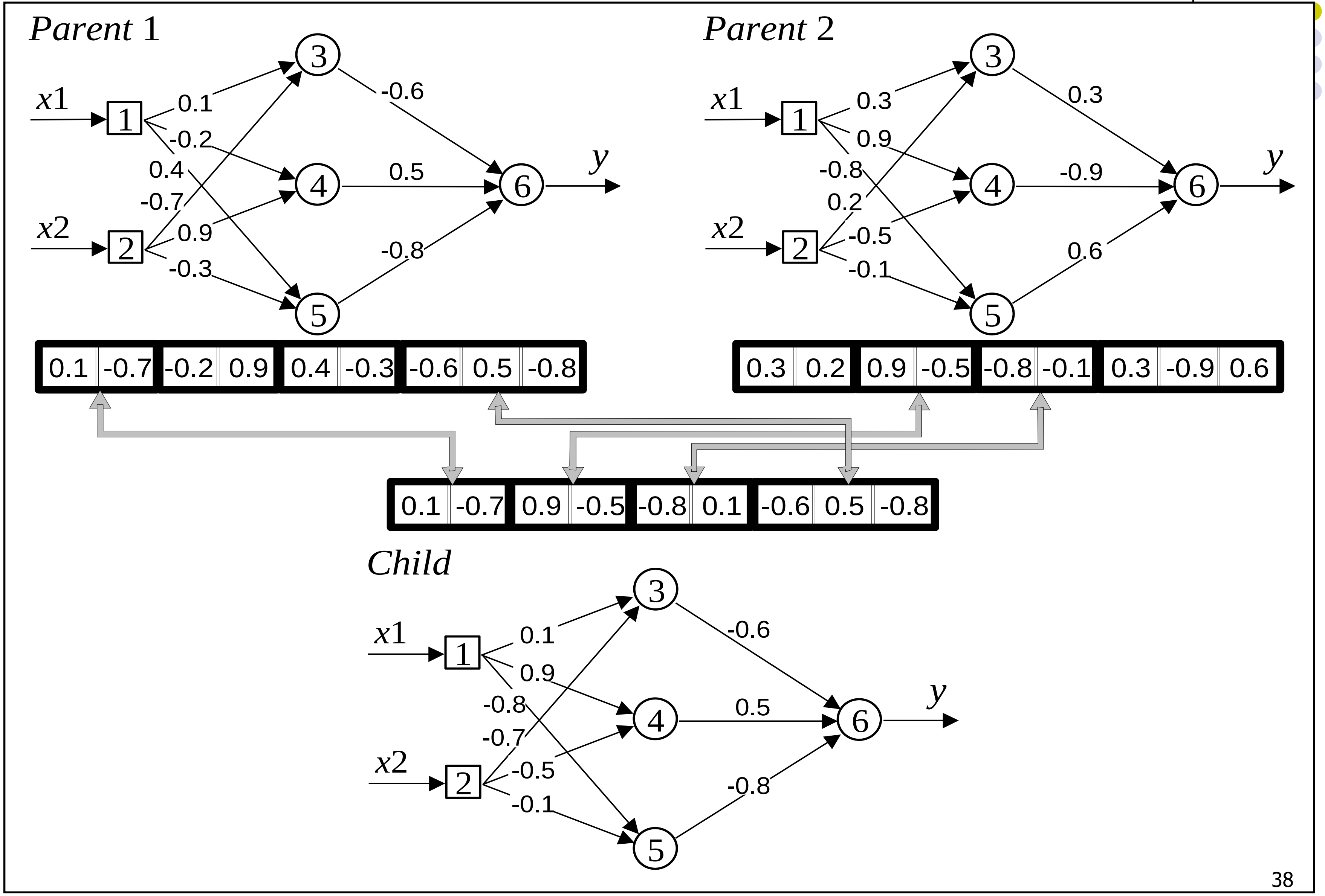
Chromosome:

0.9	-0.3	-0.7	-0.8	0.6	0.3	0.1	-0.2	0.2	0.4	0.5	0.8	-0.6	0.1	-0.2	0.9

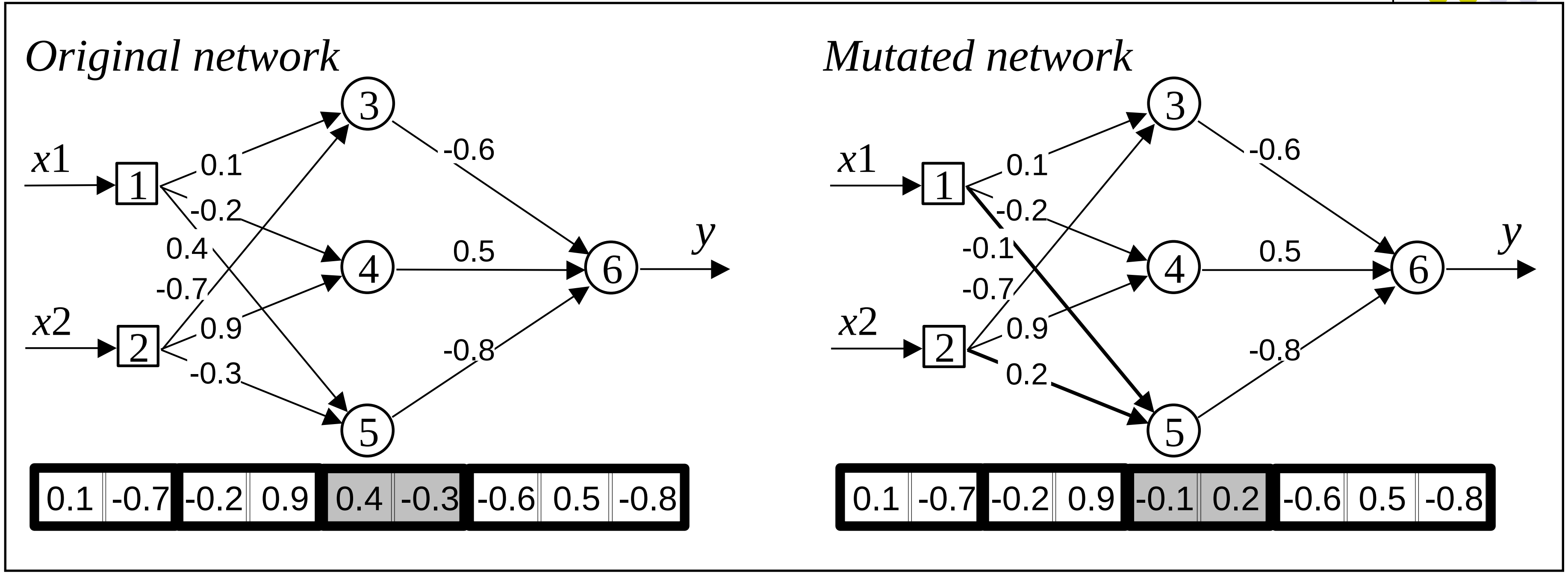
- The second step is to define a fitness function for evaluating the chromosome's performance. This function must estimate the performance of a given neural network. We can apply here a simple function defined by the sum of squared errors.
- The training set of examples is presented to the network, and the sum of squared errors is calculated. The smaller the sum, the fitter the chromosome. *The genetic algorithm attempts to find a set of weights that minimises the sum of squared errors.*

- The third step is to choose the genetic operators – crossover and mutation. A crossover operator takes two parent chromosomes and creates a single child with genetic material from both parents. Each gene in the child's chromosome is represented by the corresponding gene of the randomly selected parent.
- A mutation operator selects a gene in a chromosome and adds a small random value between -1 and 1 to each weight in this gene.

Crossover in weight optimisation



Mutation in weight optimisation



Evolving Neural Networks through Augmenting Topologies-NEAT

PyTorch version of NEAT (UberAI Lab)

Differential Evolution

- Stochastic, population based, real-valued algorithm
- Designed for challenging continuous problems where the objective/cost function could be non-differentiable, nonlinear and/or multimodal
- Few control parameters and good convergence behaviour.

Differential Evolution

- DE was introduced in 1996 by Storn (PhD student) and Price (supervisor).
  - The algorithm uses a population of size NP
  - Each vector $\boldsymbol{x}_{i,g} \in R^n$ is of size n, g denotes generations
  - variables are drawn from uniform random distribution
    - $x_{ij} \in [a, b], \forall i,j$ where a and b depend on the function
  - If an initial solution $\boldsymbol{x}_{nom,0}$ already exists, then population members are normally distributed random deviations of the initial solution

Differential Evolution

Basic steps of the algorithm:

- Generate **new vector** by adding weighted difference of two vectors to third (*mutation-type operation*); original vectors are called target vectors
- Mix **new vector** with **target** vector to yield **trial** vector (*crossover-type operation*)
- Replace **target** vector with **trial** vector if the latter is better in terms of fitness value (*selection-type operation*)

Let's see how these steps are implemented…

Differential Evolution

- For all **target** vectors $x_{i,g}$, a **mutant** vector $v_{i,(g+1)}$ is generated:

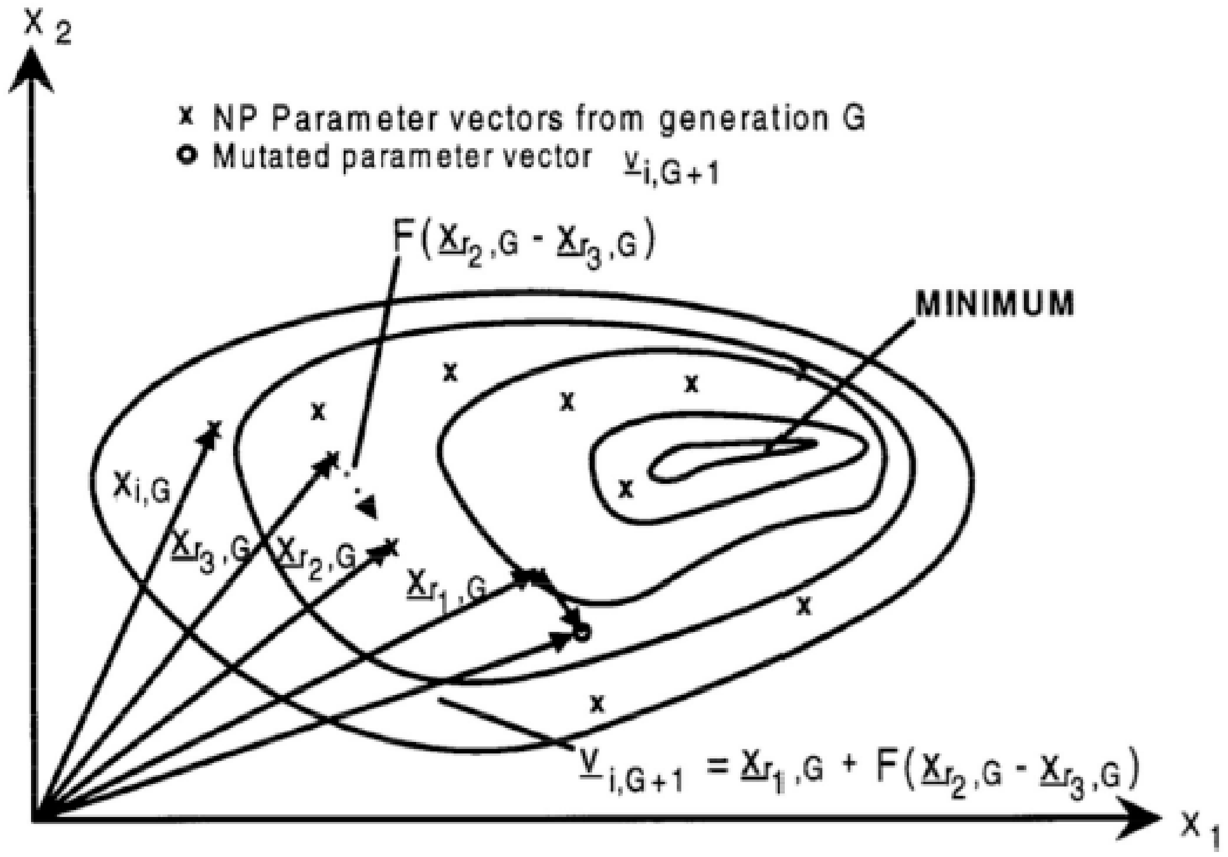
$$\mathbf{v}_{i,(g+1)} = \mathbf{x}_{r1} + F\,(\mathbf{x}_{r2,g} - \mathbf{x}_{r3,g})$$

Mutation-type operation

$r_1 \neq r_2 \neq r_3 \in \{1,2, \ldots\ NP\}$ are chosen randomly $i \notin \{r_1, r_2, r_3\}$

- $F \in (0, 2]$ is a real and constant factor
  - It controls the amplification of the differential variation $(\mathbf{x}_{r2} - \mathbf{x}_{r3})$
  - Other implementations impose different limits on *F*
- Unlike typical evolutionary algorithms, in DE mutation involves at least two other individuals from the population.

Differential Evolution


x_2
x NP Parameter vectors from generation G
o Mutated parameter vector $\underline{v}_{i,G+1}$
$F(\underline{x}_{r_2,G} - \underline{x}_{r_3,G})$
MINIMUM
$x_{i,G}$
$\underline{x}_{r_3,G}$
$\underline{x}_{r_2,G}$
$\underline{x}_{r_1,G}$
$\underline{v}_{i,G+1} = \underline{x}_{r_1,G} + F(\underline{x}_{r_2,G} - \underline{x}_{r_3,G})$
x_1

Differential Evolution

Crossover-type operation to produce diversity

Generate a **trial** vector u_i :

$$\boldsymbol{u}_{i,(g+1)} = [u_{i1,(g+1)}, u_{i2(g+1)}, \ldots, u_{in,(g+1)}]$$

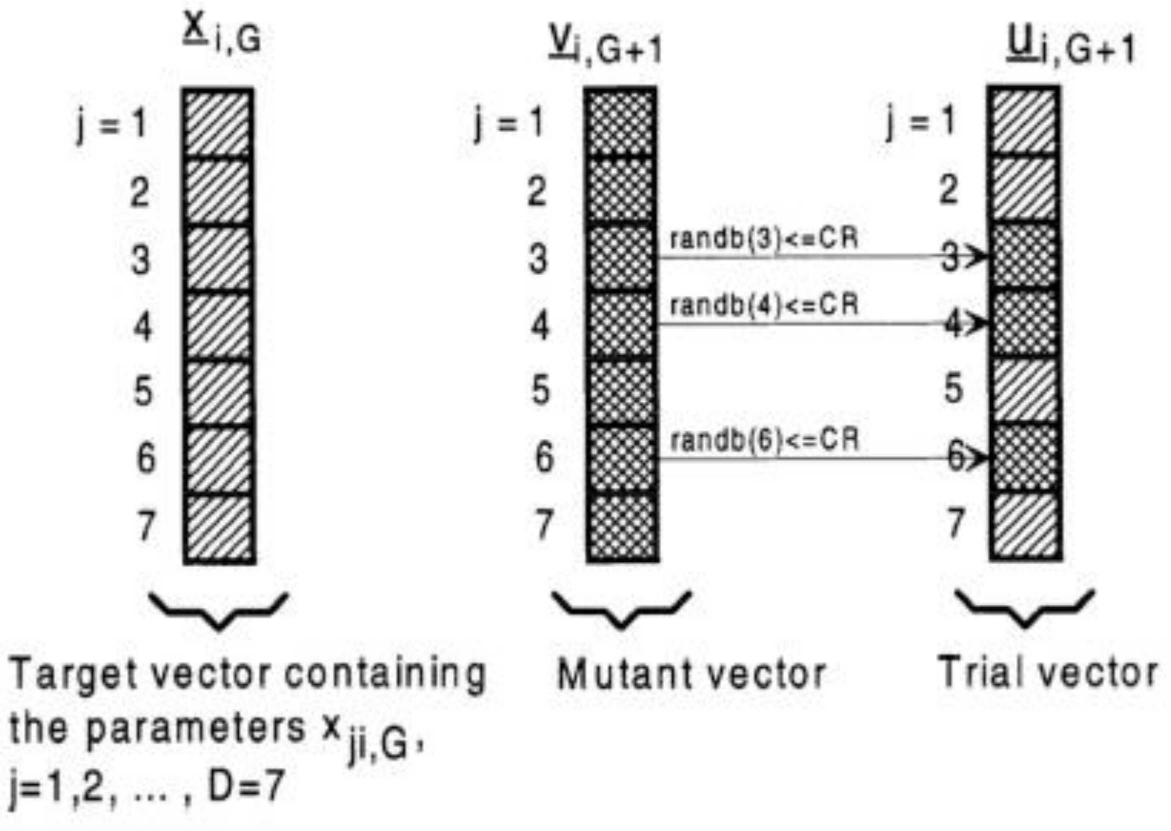
where for each element $j = 1, 2, \ldots, n$

$$u_{ij,(g+1)} = \begin{cases} v_{ij,(g+1)} & \text{if } (rnd([0,1]) \leq p_c) \text{ or } j = \delta \\ x_{ij,(g)} & \text{if } (rnd([0,1]) > p_c) \text{ and } j \neq \delta \end{cases}$$

$p_c \in [0, 1]$ is a crossover constant/rate

$\delta \in 1, 2, \ldots, n$ is a randomly chosen index (ensures $\boldsymbol{u}_{i,(g+1)}$ gets at least one parameter from $\boldsymbol{v}_{i,(g+1)}$

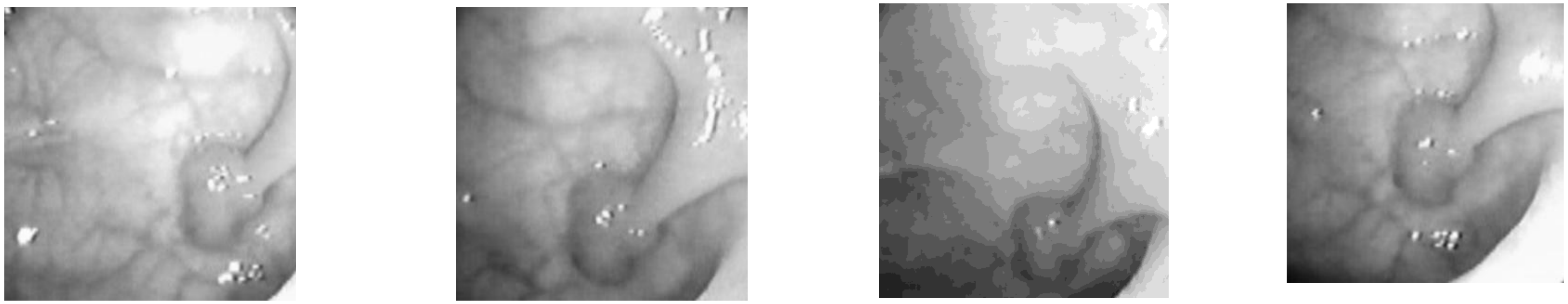
Differential Evolution



Differential Evolution

- **Selection:** Decide if trial vector enters the population at g+1
- Trial vector $\boldsymbol{u}_{i\,,(g)}$ is compared to target vector $\boldsymbol{x}_{i,(g)}$
  - Use **greedy criterion**:
    - if $\mathbf{u}_{i,(g)}$ *is better than* $\mathbf{x}_{i,(g)}$*, replace* $\mathbf{x}_{i,(g)}$ *with* $\mathbf{u}_{i,(g)}$
    - otherwise $\boldsymbol{x}_{i,(g)}$ “survives" and $\boldsymbol{u}_{i,(g)}$ is discarded

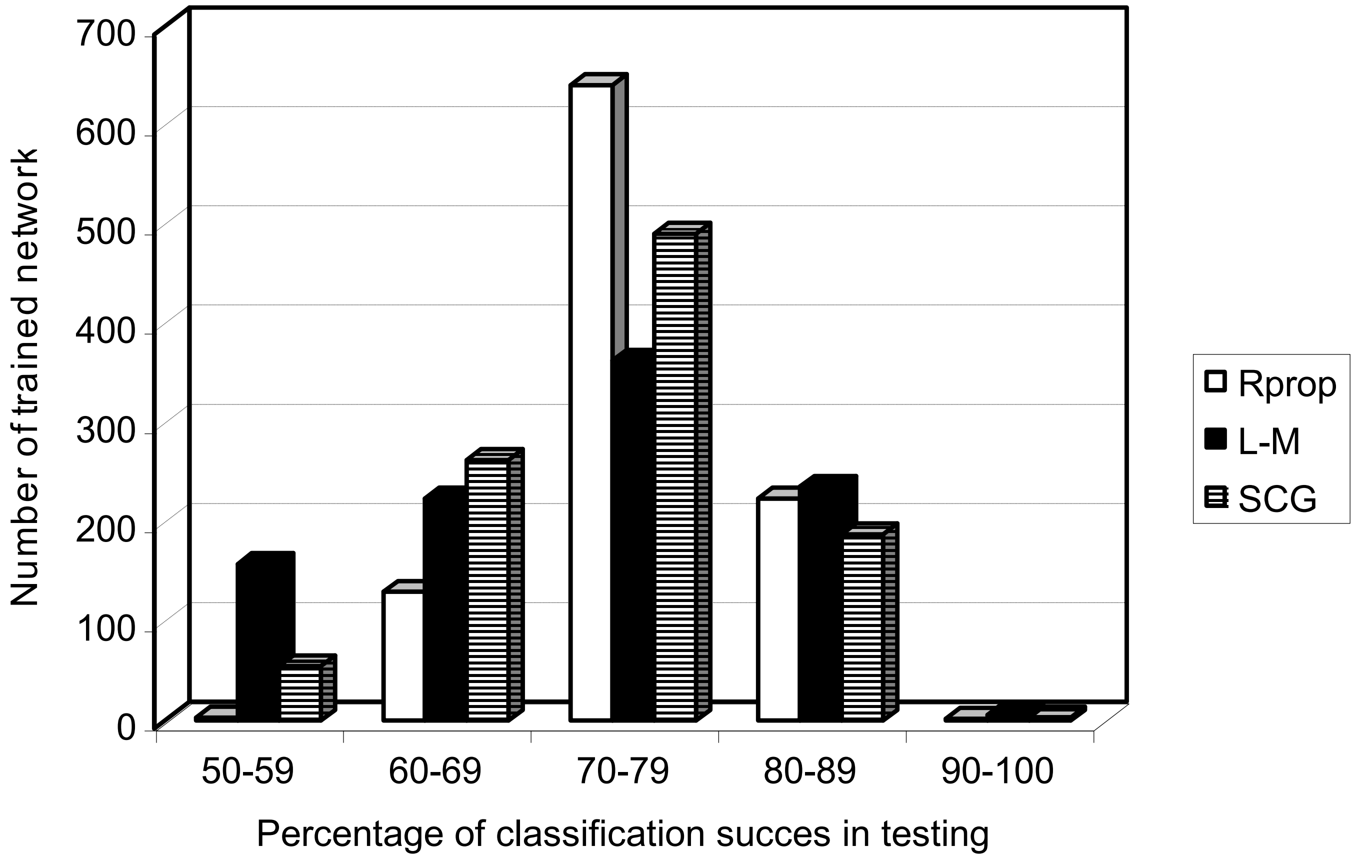
DE application in neural network training



Frames of a video sequence showing a polypoid tumor of the colon

The environment changes dynamically: shadows, reflections, various perceptual angles of physician, endoscope tip movements

Ref. 11 for details



Generalisation results for three batch-training algorithms

DE implementation

```
Step 0:  Initialize the population
Step 1:  Evaluate fitness of all individuals
Step 2:  Repeat
Step 3:     For $i$ = 1 to $NP$
Step 4:        MUTATION($w_i^k$) → Mutant_Vector.
Step 5:        CROSSOVER(Mutant_Vector) →
         Trial_Vector.
Step 6:        If E(Trial_Vector) ≤ E($w_i^p$), accept
         Trial_Vector for the next generation.
Step 7:     EndFor
Step 8:  Until the termination condition is met.
```

$i = 1, \ldots, NP$ (members of population);
$j = 1, \ldots, d$ (number of elements of the vector);
k = iteration (generation);

Technical details for key steps

ρ = crossover constant; r_j = random number; μ = mutation constant

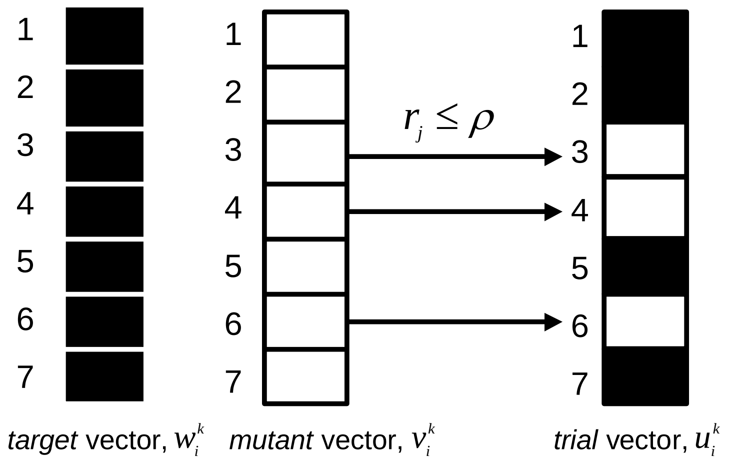
Step 4: mutant vector (mutation type operation)

$$v_i^{k+1} = w_i^k + \mu\left(w_{best}^k - w_i^k + w_{r1} - w_{r2}\right)$$

$$r_1, r_2 \in \{1,2,\ldots,i-1,i+1,\ldots NP\}$$

Step 5: trial vector ("crossover" type operation)

$$u_{i,j}^{k+1} = \begin{cases} v_{i,j}^{k+1} & \text{if } r_j \le \rho \text{ or } j = rand_i \\ w_{i,j}^{k} & \text{if } r_j > \rho \text{ and } j \ne rand_i \end{cases}$$



Step 6: selection

$$w_i^{k+1} = \begin{cases} u_i^{k+1} & \text{if } E_p(u_i^{k+1}) < E_p(w_i^k) \\ w_i^{k} & \text{if } E_p(u_i^{k+1}) \ge E_p(w_i^k) \end{cases}$$

Best approach is a Hybrid Scheme

Online learning produces an initial solution, then population members are normally distributed random deviations of the initial solution.

On-line learning phase	*Step 0a:*	Initialize weights w^0, learning rate η^0, meta-learning rates γ_1, γ_2.
	Step 1a:	**Repeat**
	Step 2a:	Set iteration $k = k+1$
	Step 3a:	Pattern presentation
	Step 4a:	Calculate $E_p(w^k)$ and then $\nabla E_p(w^k)$.
	Step 5a:	Update the weights: $w^{k+1} = w^k - \eta^k \nabla E_p(w^k)$.
	Step 6a:	Adapt the learning rate $\eta^{k+1} = \eta^k + \gamma_1 \langle \nabla E_{p-1}(w^{k-1}), \nabla E_p(w^k) \rangle + \gamma_2 \langle \nabla E_{p-2}(w^{k-2}), \nabla E_{p-1}(w^{k-1}) \rangle$
	Step 7a:	**Until** `Termination_Condition_1` is met.
	Step 8a:	**Return** weights w^{k+1}.
Evolution phase	*Step 0b:*	Initialize population in the neighborhood of w^{k+1}, $\rho \in [0,1]$, $\mu_1 > 0$, $\mu_2 > 0$.
	Step 1b:	**Repeat**
	Step 2b:	Set generation counter $k = k+1$
	Step 3b:	Pattern presentation
	Step 4b:	**For** i = 1 to NP
	Step 5b:	`MUTATION(`w_i^k`)` $\rightarrow$ `Mutant_Vector` $v_i^{k+1} = w_i^k + \mu_1 (w_{best}^k - w_i^k) + \mu_2 (w_{r_1} - w_{r_2})$, $r_1, r_2 \in \{1,2,\ldots,i-1,i+1,\ldots,NP\}$
	Step 6b:	`CROSSOVER(`v_i^{k+1}`)` $\rightarrow$ `Trial_Vector` $u_{i,j}^{k+1} = \begin{cases} v_{i,j}^{k+1} & \text{if } r_j \le \rho \text{ or } j = rand_i \\ w_{i,j}^k & \text{if } r_j > \rho \text{ and } j \ne rand_i \end{cases}, j = 1,2,\ldots,n$
	Step 7b:	`SELECTION(`v_i^{k+1}`)` $\rightarrow$ `weights of next generation` $w_i^{k+1} = \begin{cases} u_i^{k+1} & \text{if } E_p(u_i^{k+1}) < E_p(w_i^k) \\ w_i^k & \text{if } E_p(u_i^{k+1}) \ge E_p(w_i^k) \end{cases}$
	Step 8b:	**EndFor**
	Step 9b:	**Until** `Termination_Condition_2` is met.

Improves stability and learning speed

Control contribution of the terms

Some results

Results when training a dedicated neural network for each frame

Method	Frame 1	Frame 2	Frame 3	Frame 4
Rprop	92%	91%	92%	93%
ABP	81%	85%	83%	81%

Results when employing a neural network for all frames

Method	Frame 1	Frame 2	Frame 3	Frame 4
On-line learning	83%	84%	77%	88%
Online learning+DE	93%	92%	84%	90%

Swarm Intelligence

- A **swarm** can be defined as a structured collection of interacting organisms or agents.
- Within the computational study of swarm intelligence, individual organisms have included ants, bees, wasps, termites, fish (in schools) and birds (in flocks).
- Individuals in these swarms are relatively simple in structure, but their collective behaviour can become quite complex.
- The global behaviour of a swarm of social organisms emerges in a nonlinear manner from the behaviour of individuals in that swarm.

Swarm Intelligence

- Interaction among individuals aids in refining *experiential knowledge* about the environment and enhances the progress of the swarm towards optimality.
- Interaction or cooperation among individuals is determined **genetically** or through **social interaction** (it depends on the particular method).

Swarm Intelligence

Example: **ACO- Ant Colony Optimisation**

- In the Ant Colony method, individuals specialise in one of a set of simple tasks.
- Collectively, the actions and behaviours of the ants ensure the building of optimal nest structures, protecting the queen, cleaning nests, finding food sources etc.

Swarm Intelligence

- Anatomical differences (genetics) may dictate the tasks performed by individuals. Minor ants (smaller and morphologically different from major ants) clean the nest, whereas major ants cut large prey and defend the nest.
- Social interaction can be direct or indirect. Direct interaction is through visual, audio or chemical contact. Indirect interaction occurs when one individual changes the environment and the other individuals respond to the new environment.

Particle Swarm Optimisation

- Originally designed by Kennedy (social psychologist) and Eberhart (electrical engineer) to simulate social behaviour (1995)
- *Basic idea*: **social interaction** is able to find optimal solutions to hard problems

Particle Swarm Optimisation

- Social behaviour: it increases the ability of an individual to adapt;

- Individual’s adaptability leads to intelligence: There is a relationship between adaptability and intelligence. Intelligence emerges from interactions among the individuals of the swarm.

Some notation

- A particle in the swarm: $x_i = (x_{i1}, x_{i2}, \cdots, x_{in})$
- The particle's best: $p_i = (p_{i1}, p_{i2}, \cdots, p_{in})$

$$if \;\; f(x_i) < f(p_i)\; then\; p_i = x_i$$

- "Simple nostalgia" (tendency of organisms to repeat past behaviours that have been reinforced or return to past successes): $v_i(t) = v_i(t-1) + c \cdot (p_i - x_i(t-1))$

- Emulate the success of others: $p_g = (p_{g1}, p_{g2}, \mathrm{K}, p_{gn})$
(the best particle of the swarm)

Particle Swarm Optimisation – The basic algorithm

1. Initialise population in hyperspace
2. Evaluate fitness of individual particles
3. Modify velocities based on previous best and global (or neighborhood) best
4. Terminate on some condition
5. Go to step 2

PSO Velocity Update Equations

$$x_i(t) = x_i(t-1) + v_i(t)$$

where

individual best (p_{best})

global best (g_{best})

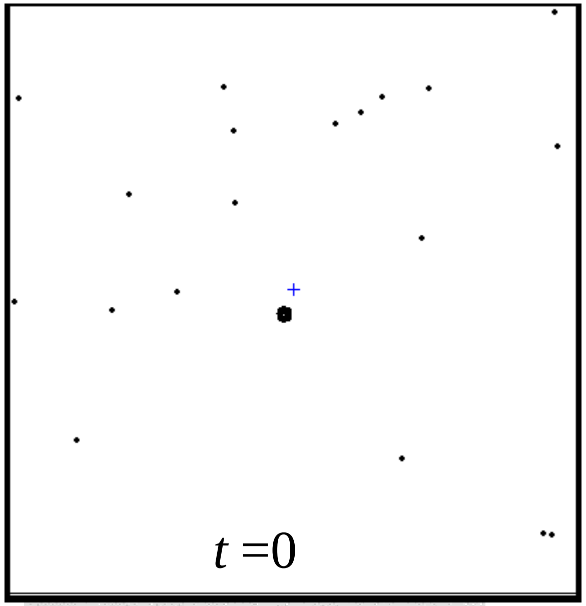
$$v_i(t) = w \cdot v_i(t-1) + c_1 \cdot r_1\left(p_i - x_i(t-1)\right) + c_2 \cdot r_2\left(p_g - x_i(t-1)\right)$$

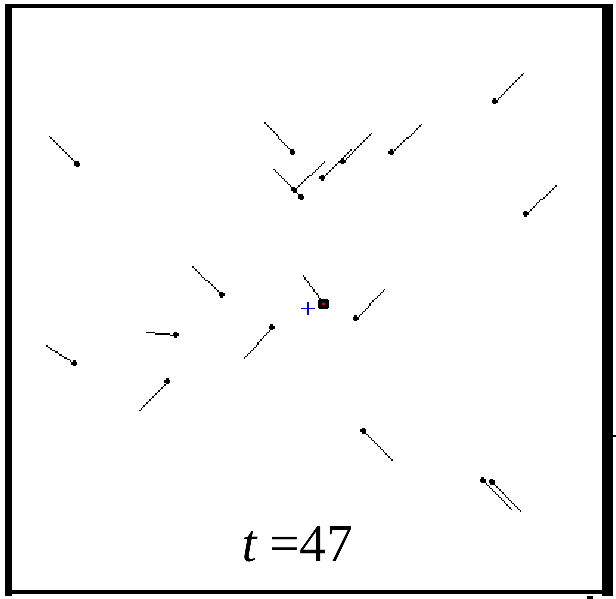
<u>THE HEURISTICS</u>

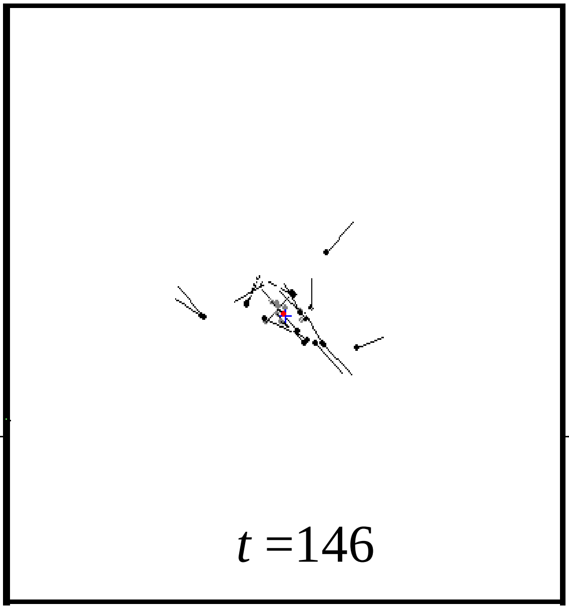
- r_1, r_2 random values in [0,1]
- Upper limit to $c_k, k \in \{1,2\}$
- Inertia weight: w
- Maximum velocity of change: $V_{\max}$

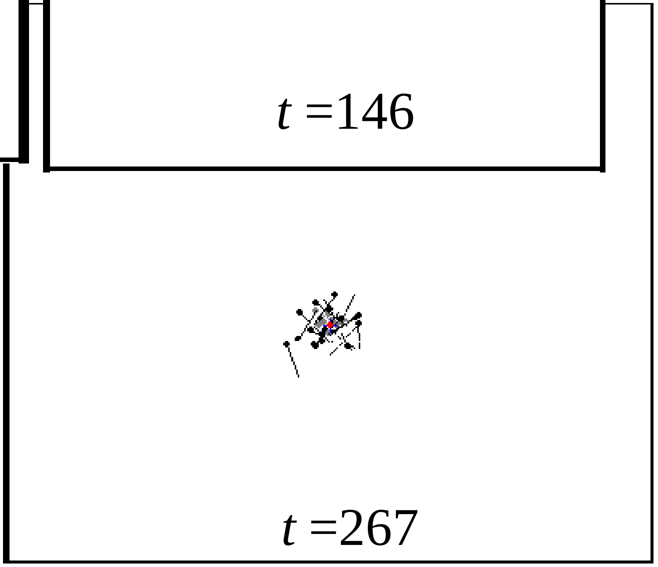
PSO parameters

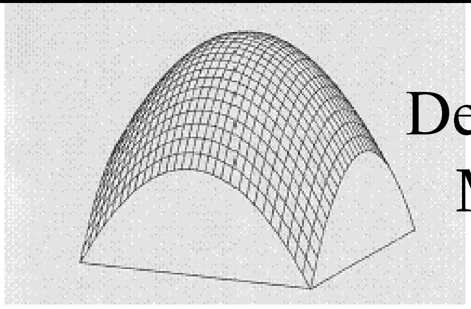
De Jong's test functions (F1 and F5) in two dimensions











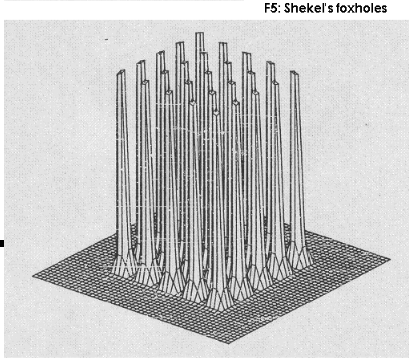
F1: Unimodal parabola

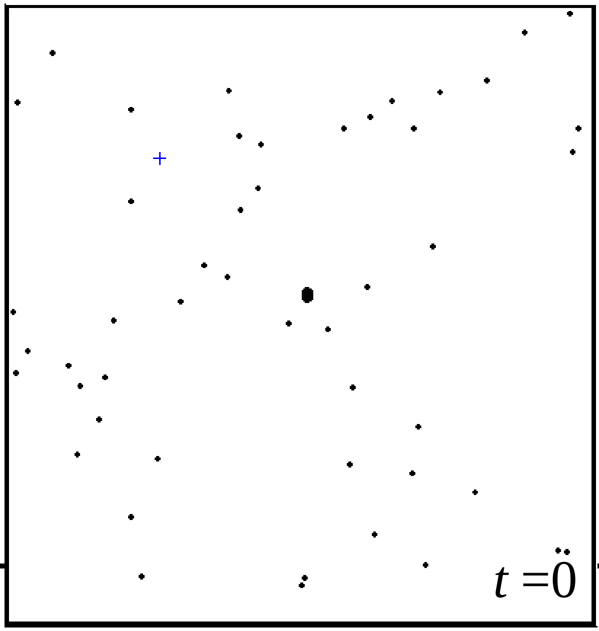
De Jong's F1: NP = 20,

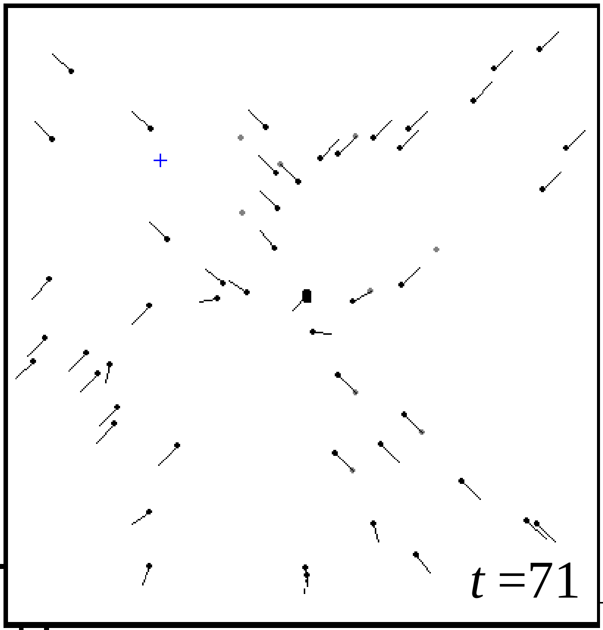
Max velocity = 0.5,

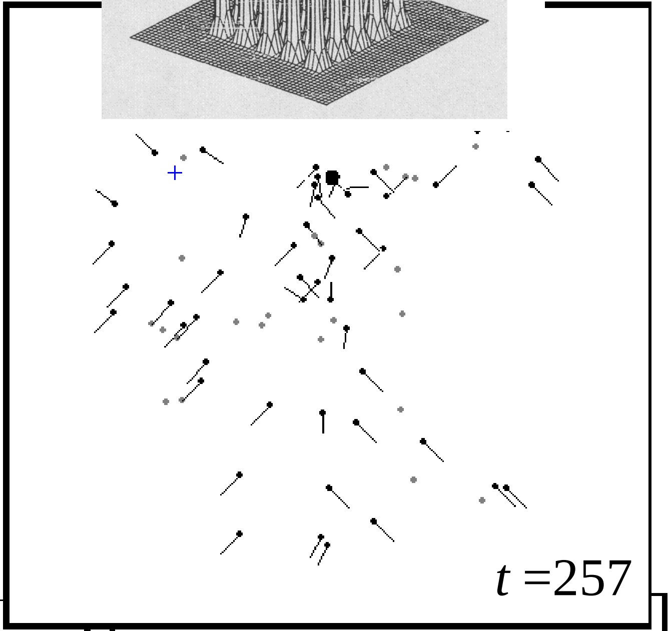
$c_1 = c_2 = 2$

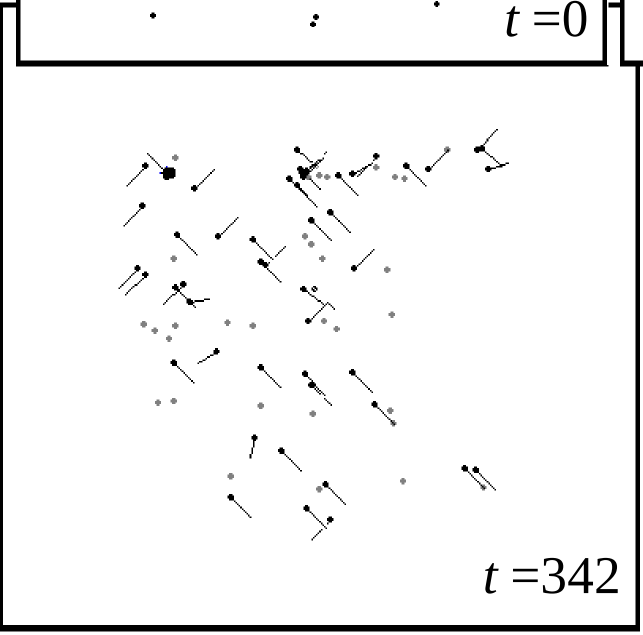
De Jong's F5: *NP* = 50, Max velocity = 4, $c_1 = c_2 = 2$

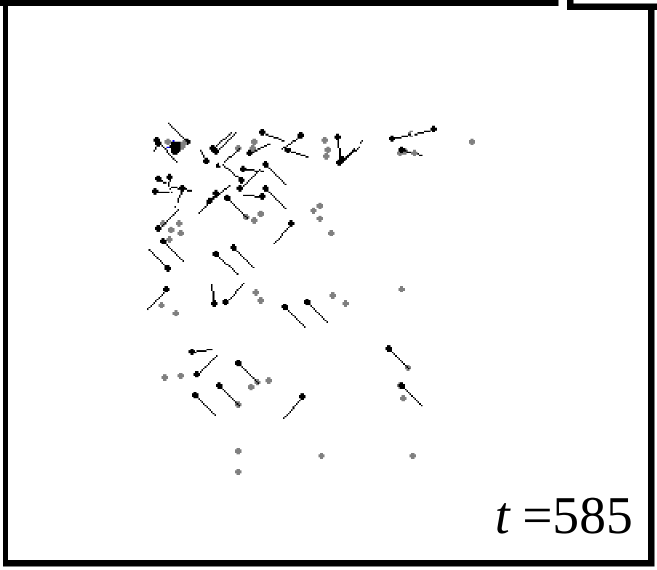


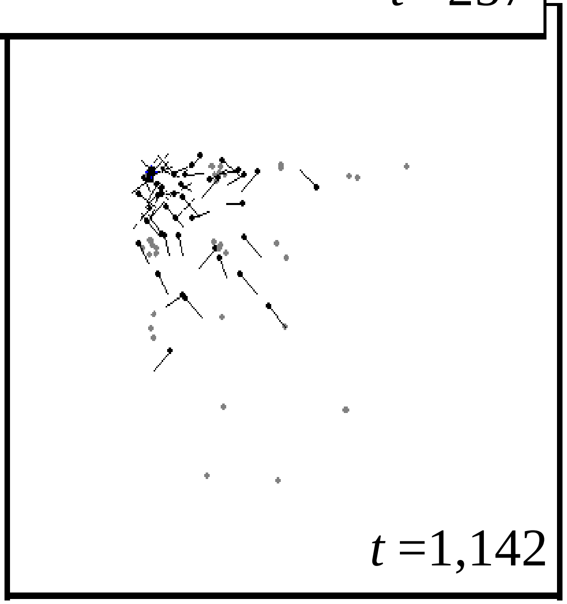






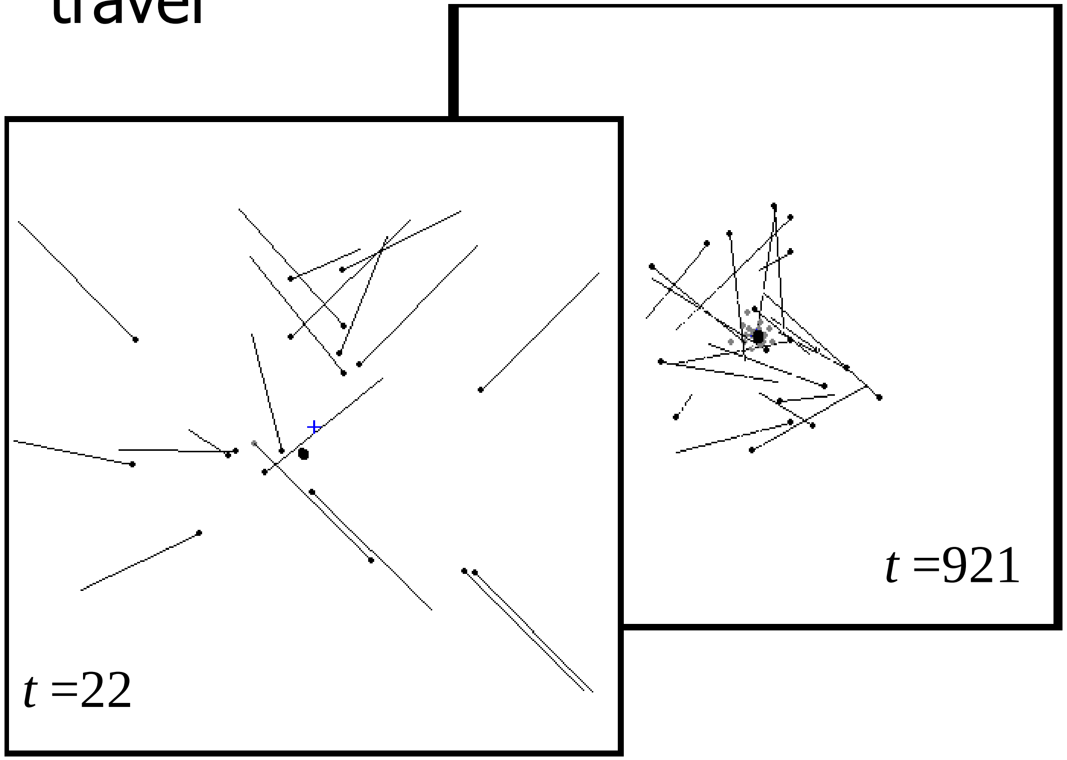


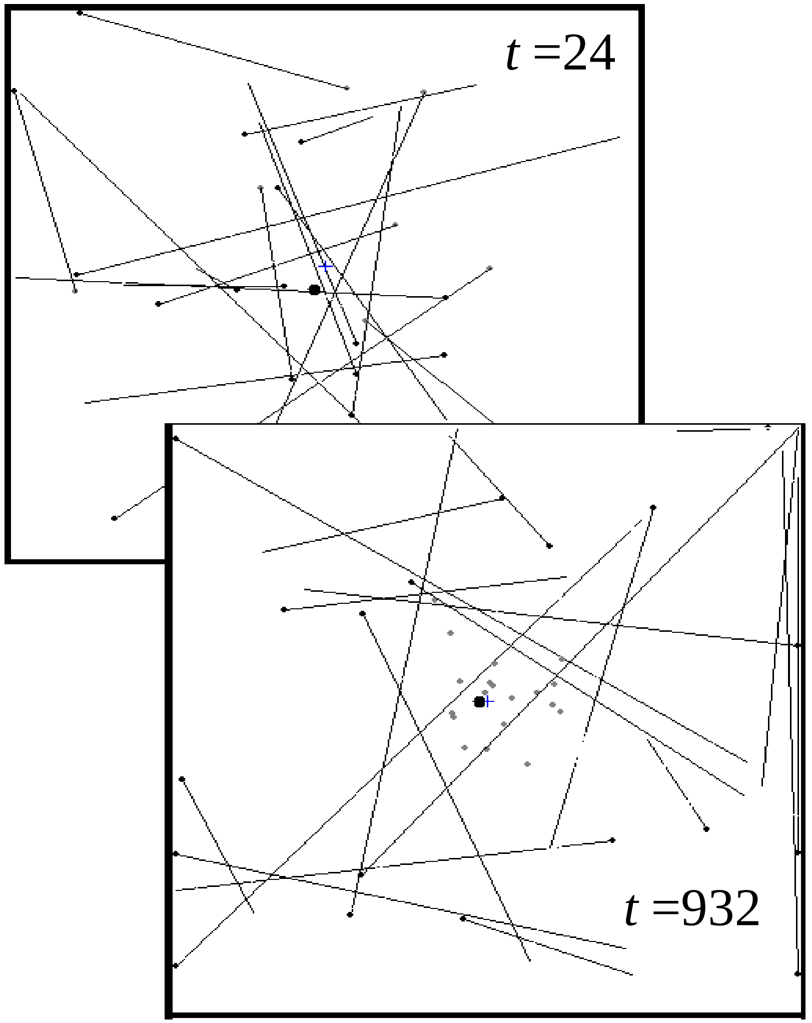




PSO parameters

max velocity=the maximum distance a particle can travel





De Jong's F1: $NP = 20$, $c_1 = c_2 = 2$
Max velocity = 2 (left), 10 (right)

Upper limit of *c*: "any time the sum of the two coefficients c_k exceeds the value 4.0, both the velocities and positions explode toward infinity" [Kennedy & Eberhart, 1999]

- The *golden* numbers: $c_1 = 2.8 \quad c_2 = 1.3$

Maximum velocity: "reducing $V_{\max}$ by too much impedes the ability of the Swarm to search" [Carlisle & Dozier, 2001]

- *Constriction factor*:
[Clerc, 1999;
Carlisle & Dozier, 2001]

$$K = \frac{k}{\left|2-(c_1+c_2)-2\sqrt{\left|(c_1+c_2)^2-4(c_1+c_2)\right|}\right|},$$

$$c_1+c_2>4 \quad \text{and} \quad k\in(0,1]$$

Inertia weight: "influences the trade-off between global and local exploration abilities of the particle" [Shi & Eberhart, 1998]

Evolving Neural Networks with Particle Swarm Optimisation-1

- Eberhart, Dobbins, and Simpson (1996) first used PSO to evolve network weights (replaced backpropagation learning algorithm)

- PSO can also be used to indirectly evolve the structure of a network. An added benefit is that the preprocessing of input data is made unnecessary.

Evolving Neural Networks with Particle Swarm Optimisation-2

- Evolve both the network weights ***and*** the slopes of sigmoid activation functions of hidden and output nodes.

- If activation function now is: $\text{output} = 1/(1 + e^{-k*\text{input}})$ then we are evolving k in addition to evolving the weights.

- The method is general, and can be applied to other topologies and other activation functions (does not require derivatives of activation functions to be calculated).

- Flexibility is gained by allowing slopes to be positive or negative. A change in sign for the slope is equivalent to a change in signs of all input weights.

Useful reading-1

Hybrid

1. Negnevitsky, *"Artificial Intelligence: a Guide to Intelligent Systems",* section 8.5.

Generalisation

2. Theodoridis S., Koutroumbas K. (2009), sections 10.1-10.3, Pattern Recognition, Academic Press. Available online at: https://drive.google.com/file/d/0By995HEqDrWQbnBfTGJEVXZrRkE/view?usp=sharing
3. Reed R. (1993), Pruning algorithms: a survey, IEEE Transactions Neural Networks. Available online at: http://axon.cs.byu.edu/~martinez/classes/678/Papers/Reed_PruningSurvey.pdf
4. Gupta A., Lam S.M., Weight decay backpropagation for noisy data (1998), Neural Networks, 11 (6), pp. 1127-1138. Available at the BBK Library: https://doi.org/10.1016/S0893-6080(98)00046-X

Ensemble learning

5. Z.-H. Zhou. Ensemble learning. In: S. Z. Li ed. Encyclopedia of Biometrics, Berlin: Springer, 2009, pp. 270-273. https://cs.nju.edu.cn/zhouzh/zhouzh.files/publication/springerEBR09.pdf
6. Sharkey, A.J.C. and Sharkey, N.E. (1997) Combining diverse neural nets. The Knowledge Engineering Review, 12 (3). pp. 231-247. http://eprints.whiterose.ac.uk/1630/1/sharkey.a.j.c1.pdf
7. Anastasiadis A. and Magoulas G.D., Analysing the Localisation Sites of Proteins through Neural Networks Ensembles, Neural Computing & Applications, vol. 15(3), 277 – 288, 2006. http://www.dcs.bbk.ac.uk/~gmagoulas/proteins.pdf

Useful reading-2

Hybrid Genetic Algorithms

8. El-Mihoub T. A., Hopgood A. A., Nolle L., Battersby A. (2006), Hybrid Genetic Algorithms: A Review, Engineering Letters, 13:2, EL_13_2_11 http://www.engineeringletters.com/issues_v13/issue_2/EL_13_2_11.pdf

Differential evolution

9. Storn R. M., Price K. V. (1997) Differential Evolution – A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces, Journal of Global Optimization 11: 341–359. Available online at: http://www1.icsi.berkeley.edu/~storn/TR-95-012.pdf
10. Magoulas G.D., Plagianakos V.P., and Vrahatis M.N., Neural Network-based Colonoscopic Diagnosis Using On-line Learning and Differential Evolution, Applied Soft Computing, Vol. 4(4), 369-379, 2004. Available online at: http://www.dcs.bbk.ac.uk/~gmagoulas/ApplSoftComp.pdf

Swarms

11. Poli R., Kennedy J., Blackwell T. (2007), Particle Swarm Optimisation: an overview, Swarm Intelligence Journal, vol. 1, no. 1, 33-57. Available online at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.206.2896&rep=rep1&type=pdf

Useful reading-3

Differential evolution

12. Storn R. M., Price K. V. (1997) Differential Evolution – A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces, Journal of Global Optimization 11: 341–359. Available online at: http://www1.icsi.berkeley.edu/~storn/TR-95-012.pdf
13. Magoulas G.D., Plagianakos V.P., and Vrahatis M.N., Neural Network-based Colonoscopic Diagnosis Using On-line Learning and Differential Evolution, Applied Soft Computing, Vol. 4(4), 369-379, 2004. Available online at: http://www.dcs.bbk.ac.uk/~gmagoulas/ApplSoftComp.pdf

Next week

- Lab: Malet St, MAL 109 (ITS)