LEZIONE 3

Machine Learning (supervised)
Parte II

Massimo Brescia
How to build invariance in ANNs

Classification systems must be invariant to certain transformations depending on the problem.
- For example, a system recognizing objects from images must be invariant to rotations and translations.
- At least three techniques exist for making classifier-type neural networks invariant to transformations.

1. Invariance by Structure
   - Synaptic connections between the neurons are created so that transformed versions of the same input are forced to produce the same output.
   - Drawback: the number of synaptic connections tends to grow very large.

2. Invariance by Training
   - The network is trained using different examples of the same object corresponding to different transformations (for example rotations).
   - Drawbacks: computational load, generalization mismatch for other objects.

3. Invariant feature space
   - Try to extract features of the data invariant to transformations.
   - Use these instead of the original input data.
   - Probably the most suitable technique to be used for neural classifiers.
   - Requires prior knowledge on the problem.
How to build invariance in ANNs

- Optimization of the structure of a neural network is difficult.
- Generally, a neural network acquires knowledge about the problem through training.
- The knowledge is represented by a distributed and compact form by the synaptic connection weights.
- Neural networks are not able to handle uncertainty, and do not gain from probabilistic evolution of data samples.
- A possible solution: integrate a neural network and artificial intelligence into a hybrid system.
MLP – learning – Back Propagation

Output error

\[ N(U, e, P(e)) = \frac{1}{2h} \sum_j (Y_j - P(e))^2 \]

Stopping threshold

\[ |Y - P(e)| < \varepsilon \]

Forward phase with the propagation of the input patterns through the layers

\[ net_{input_j} = \sum_k w_{jk}o_j \]

Backward phase with the back propagation of the error

\[ \delta = f'(o)(Y - P(e)) \]

Law for updating hidden weights

\[ net_{input_k} = \sum_i w_{ki}o_i \]

Momentum to jump over the error surface

\[ w_{ji}(new) = w_{ji}(old) + \eta \delta o_j + \alpha \Delta w_{ji}(old) \]

Activation function

\[ f(o) = \frac{1}{1 + \exp(-o)} \]
Other typical problems of the back-propagation algorithm are the speed of convergence and the possibility of ending up in a local minimum of the error function.

Back Propagation requires that the activation function used by the artificial neurons (or "nodes") is differentiable. Main formulas are:

\[ f(h) = \frac{1}{1 + e^{-\sum_i w_{ih}f(i)}} \quad (3) \]
\[ f(j) = \frac{1}{1 + e^{-\sum_h w_{jh}f(h)}} \quad (4) \]

\[ \delta_h = f'(j)(Y - P(e)) \quad (5) \]
\[ \delta_i = f'(h) \sum w\delta_h \quad (6) \]

- (3) and (4) are the activation function for a neuron of the, respectively, hidden layer and output layer. This is the mechanism to process and flow the input pattern signal through the “forward” phase;
- At the end of the “forward” phase the network error is calculated (inner argument of the (5)), to be used during the “backward” or top-down phase to modify (adjust) neuron weights;
- (5) and (6) are the descent gradient calculations of the “backward” phase, respectively, for a generic neuron of the output and hidden layer;
- (7) and (8) are the most important laws of the backward phase. They represent the weight modification laws, respectively, between output and hidden layers (7) and between hidden-input (or hidden-hidden if more than one hidden layer is present in the network topology) layers. The new weights are adjusted by adding to the old ones two terms:
  - \( \eta \delta_f(j) \): this is the descent gradient multiplied by a parameter, defined as “learning rate”, generally chosen sufficiently small in \([0, 1]\), in order to induce a smooth learning variation at each backward stage during training;
  - \( \alpha \Delta w_{old} \): this is the weight variation multiplied by a parameter, defined as “momentum”, generally chosen quite high in \([0, 1]\), in order to give an high change to the weights to prevent the “local minima”

\[ w_{new_{jh}} = w_{old_{jh}} + \eta \delta_{h} f(j) + \alpha \Delta w_{old_{jh}} \quad (7) \]
\[ w_{new_{hi}} = w_{old_{hi}} + \eta \delta_{i} f(h) + \alpha \Delta w_{old_{hi}} \quad (8) \]
MLP with BP can be used for regression and classification problems. They are always treated as optimization problems (i.e. minimization of the error function to achieve the best training in a supervised fashion).

Adjust the weights in such a way that the output of ANN is consistent with class labels of training examples

\[ E = \sum [Y_i - f(w_i, X_i)]^2 \]

For regression problems the basic goal is to model the conditional distribution of the output variables, conditioned on the input variables.

This motivates the use of a sum-of-squares error function. But for classification problems the sum-of-squares error function is not the most appropriate choice.
By assigning a sigmoidal activation function on the output layer of the neural network, the output can be interpreted as posterior probabilities.

In fact, the output of the network trained by minimizing a sum-of-squares error function approximate the posterior probabilities of class membership, conditioned on the input vector, using the maximum likelihood principle by assuming that the target data was generated from a smooth deterministic function with added Gaussian noise. For classification problems, however, the targets are binary variables and hence far from having a Gaussian distribution, so their description cannot be given by using Gaussian noise model. Therefore a more appropriate choice of error function is needed.

Let us now consider problems involving two classes. One approach to such problems would be to use a network with two output units, one for each class. But let’s discuss an alternative approach in which we consider a network with a single output $y$. We would like the value of $y$ to represent the posterior probability $P(C_1 \mid x)$ for class $C_1$. The posterior probability of class $C_2$ will then be given by $P(C_2 \mid x) = 1 - y$

This can be achieved if we consider a target coding scheme for which $t = 1$ if the input vector belongs to class $C_1$ and $t = 0$ if it belongs to class $C_2$. We can combine these into a single expression, so that the probability of observing either target value is the Bernoulli distribution equation

$$P(t \mid x) = y^t (1 - y)^{1-t} \quad \longrightarrow \quad \prod_{n} (y^n)^{t^n} (1 - y^n)^{1-t^n}$$

By minimizing the negative logarithm, we get to the **cross-entropy** error function in the form

$$E = -\sum \left[ t^n \ln y^n + (1-t^n) \ln(1 - y^n) \right]$$
MLP heuristic rules – activation function

If there are good reasons to select a particular activation function, then do it:
- Mixture of Gaussian $\Rightarrow$ Gaussian activation function;
- Hyperbolic tangent;
- Arctangent;
- Linear threshold;

General “good” properties of activation function:
- Non-linear;
- Saturate – some max and min value;
- Continuity and smooth;
- Monotonicity: convenient but nonessential;
- Linearity for a small value of net;

Sigmoid function has all the good properties:
- Centered at zero;
- Anti-symmetric;
- $f(-\text{net}) = - f(\text{net})$;
- Faster learning;
- Overall range and slope are not important;
We can also use bipolar logistic function as the activation function in hidden and output layer. Choosing an appropriate activation function can also contribute to a much faster learning. Theoretically, sigmoid function with less saturation speed will give a better result.

It can be manipulated its slope and see how it affects the learning speed. A larger slope will make weight values move faster to saturation region (faster convergence), while smaller slope will make weight values move slower but it allows a refined weight adjustment.
MLP heuristic rules – scaling of data

Scaling input and target values

Standardize
- Large scale difference
  - error depends mostly on large scale feature;
- Shifted to Zero mean, unit variance
  - Need to be done once, before training;
  - Need full data set;

Target value
- Output is saturated
  - In the training, the output never reach saturated value;
    - Full training never terminated;
- Range [-1, +1] is suggested;
MLP heuristic rules – hidden nodes

Number of hidden units governs the expressive power of net and the complexity of decision boundary;

Well-separated $\rightarrow$ fewer hidden nodes;

complicated distribution or large spread over parameter space $\rightarrow$ many hidden nodes;

Heuristics rule of thumb:

More training data yields better result;

Number of weights $<<$ number of training data;

Number of weights $\approx$ (number of training data)/10 (impossible for massive data);

Adjust number of weights in response to the training data:
   Start with a “large” number of hidden nodes, then decay, prune weights...;
In the mathematical theory of ANNs, the **universal approximation theorem** states:

*A standard MLP feed-forward network with a single hidden layer, which contains finite number of hidden neurons, is a universal approximator among continuous functions on compact subsets of $\mathbb{R}^n$, under mild assumptions on the activation function.*

The theorem was proved by George Cybenko in 1989 for a sigmoid activation function, thus it is also called the **Cybenko theorem**. Kurt Hornik proved in 1991 that not the specific activation function, but rather the feed-forward architecture itself allows ANNs to be universal approximators.

Then, in 1998, Simon Haykin added the conclusion that a 2-hidden layer feed-forward ANN has more chances to converge in local minima than a single-hidden layer network.

---


MLP heuristic rules – hidden layers

One or two hidden layers are OK, so long as differentiable activation function;
   But one layer is generally sufficient;

More layers \(\rightarrow\) may be induce more chance of local minima;

Single hidden layer vs double (multiple) hidden layer:
   single is good for any approximation of continuous function;
   double may be good some times, when the parameter space is largely spread;

Problem-specific reason of more layers:
   Each layer learns different aspects (different level of non-linearity);
   Each layer is an hyperplane performing a separation of parameter space;

Recently, according the experimental results discussed in Bengio & LeCun 2007, problems
where the data are particularly complex and with a high variation in the parameter space
should be treated by “deep” networks, i.e. with more than a computation layer.

Hence, the universal ANN theorem has evident limits! The choice must be driven by the
experience!!!
MLP heuristic rules – weight initialization

Not to set zero – no learning take place;
   Selection of good Seed for Fast and uniform learning;
   Reach final equilibrium values at about the same time;

For standardized data:
   Choose randomly from single distribution;
   Give positive and negative values equally \(-\omega < w < +\omega\);
      If \(\omega\) is too small, net activation is small – linear model;
      If \(\omega\) is too large, hidden units will saturate before learning begins;

the particular initialization values give influences to the speed of convergence. There are several methods available for this purpose.

The most common is by initializing the weights at random with uniform distribution inside the interval of a certain small range of number. In the MLP-BP we call this method HARDRANDOM.

Another better method is by bounding the range as expressed in the equation below. We call this method with just RANDOM

\[
\sqrt{\frac{\eta}{N_{\text{input}}}} < \|w_i\|
\]
Widely known as a very good weight initialization method is the Nguyen-Widrow method. We call this method as **NGUYEN**. Nguyen-Widrow weight initialization algorithm can be expressed as the following steps:

1. Random assign initial weights for all hidden nodes
2. For each hidden node \( j \), normalize its weight by
   \[
   w_{j;i}^{(1,-1)} = \beta \cdot w_{j;i}^{(1,-1)} / \|w_j^{(1,-1)}\|_2 \quad \text{where } \beta = 0.7^{\sqrt{n/m}}
   \]
   \[
   m = \# \text{ of hidden nodes}, \quad n = \# \text{ of input nodes}
   \]
   \[
   \|w_j^{(1,-1)}\|_2 = \beta \text{ after normalization}
   \]

Let \( \rho \geq 1 \) be a real number.

Remember that:

\[
\|x\|_p := \left( \sum_{i=1}^{n} |x_i|^p \right)^{1/p}
\]
MLP heuristic rules – parameters

Benefit of preventing the learning process from terminating in a shallow local minimum;

$\alpha$ is the **momentum** constant;

converge if $0 \leq | \alpha | < 1$, typical value = 0.9;

$\alpha = 0$: standard Back Propagation

Smaller **learning-rate** parameter makes smoother path;
increased rate of learning yet avoiding danger of instability;
First choice: $\eta \approx 0.1$;
Suggestion: $\eta$ is inversely proportional to square root of number of synaptic connection $(m^{-1/2})$;
May change during training;

There is also available an **adaptive learning** rule. The idea is to change the learning rate automatically based on current error and previous error. The formula is:

$$\Delta w_i = \eta_i E$$

$$\eta_i = (AEE_i + 1) \eta_i$$

$$E_i = E$$

The idea is to observe the last two errors rate in the direction that would have reduced the second error. Both variable $E$ and $E_i$ are the current and previous error. Parameter $A$ is a parameter that will determine how rapidly the learning rate is adjusted. Parameter $A$ should be less than one and greater than zero.
MLP heuristic rules – training error

Standard rules to evaluate the learning error are MSE (Mean Square Error) and RMSE (Root MSE)

\[
MSE = \frac{\sum_{k=1}^{NP}(t_k - Out(pat_k))^2}{NP}
\]

\[
RMSE = \sqrt{\frac{\sum_{k=1}^{NP}(t_k - Out(pat_k))^2}{NP}}
\]

sometimes it may happen that a better solution for MSE is a worse solution for the net. To avoid this problem we can use the so-called convergence tube: for a given radius \( R \) the error within \( R \) is placed equal to 0, obtaining:

\[
\text{if } (t_k - Out(pat_k))^2 > R: \quad E_k = (t_k - Out(pat_k))^2
\]
\[
\text{if } (t_k - Out(pat_k))^2 \leq R: \quad E_k = 0
\]

Don’t believe it? Let’s see an example
simple classification problem with two patterns, one of class 0 and one of class 1.

Class 0 → 0.49
Class 1 → 0.51

If the solution are 0.49 for the class 0 and 0.51 for the class 1, we have an efficiency of 100% (each pattern correctly classified) and a MSE of 0.24, a solution of 0 for the class 0 and 0.49 for the class 1 (efficiency of 50%) gives back a MSE equal to 0.13 so the algorithm will prefer this kind of solution

Class 0 → 0.49  MSE = 0.24 and efficiency = 100% → OK
Class 1 → 0.51  But the selected is the bad one! → GASP!

Class 0 → 0.0  MSE = 0.13 and efficiency = 50% → BAD
Class 1 → 0.49

So far, using convergence tube with \( R = 0.25 \) in the first case we have a MSE equal to 0 in the second MSE = 0.13 so the algorithm recognize the first solution as better than the second.

Class 0 → 0.49 (Ek = 0)  MSE = 0.0 and efficiency = 100% → OK
Class 1 → 0.51 (Ek = 0)  the selected is the good one! → OK!

Class 0 → 0.0  (Ek = 0)  MSE = 0.13 and efficiency = 50% → BAD
Class 1 → 0.49  (Ek = 0.26)
A linear model for a function $y(x)$ takes the form:

$$f(x) = \sum_{j=1}^{m} w_j h_j(x)$$

The model $f$ is expressed as a linear combination of a set of $m$ fixed functions often called basis functions.

Any set of functions can be used as the basis set although it helps if they are well behaved (differentiable). Combinations of sinusoidal waves (Fourier series):

$$h_j(x) = \sin\left(\frac{2\pi j (x - \theta_j)}{m}\right)$$

Logistic functions (common in ANNs):

$$h(x) = \frac{1}{1 + \exp(b^T x - b_0)}$$

Radial functions are a special class of functions. Their response decreases (or increases) monotonically with distance from a central point.

The center, the distance scale and the precise shape of the radial function are parameters of the neural model which uses radial functions as neuron activations.
Radial Functions

A typical radial function is the Gaussian which, in the case of a scalar input, is

\[ h(x) = \exp \left( -\frac{(x - c)^2}{r^2} \right) . \]

Its parameters are its centre \( c \) and its radius \( r \). Figure 2 illustrates a Gaussian RBF with centre \( c = 0 \) and radius \( r = 1 \).

A Gaussian RBF monotonically decreases with distance from the centre. In contrast, a multiquadric RBF which, in the case of scalar input, is

\[ h(x) = \frac{\sqrt{r^2 + (x - c)^2}}{r}, \]

monotonically increases with distance from the centre (see Figure 2). Gaussian-like RBFs are local (give a significant response only in a neighbourhood near the centre) and are more commonly used than multiquadric-type RBFs which have a global response. They are also more biologically plausible because their response is finite.
Radial Basis Function Networks

The radial basis function network are a special kind of MLP, where each of n components of the input vector $x$ feeds forward to m basis functions whose output is linearly combined with weights $w_j$ into the network model output $f(x)$. RBF are specialized as approximations of functions.

When applied to supervised learning, the least squares principle leads to a particularly easy optimization problem. If the model is

$$f(x) = \sum_{j=1}^{m} w_j h_j(x)$$

And the training set is $\{x, \hat{y}\}_{n}^{p=1}$

then the least squares recipe is to minimize the sum squared error

$$S = \sum_{i=1}^{p} (\hat{y}_i - f(x_i))^2$$

If a weight penalty term is added to the sum squared error, then the minimized cost function is

$$C = \sum_{i=1}^{p} (\hat{y}_i - f(x_i))^2 + \sum_{j=1}^{m} \lambda_j w_j^2$$
MLPQNA - I

L’MLPQNA è un tradizionale MLP che implementa come algoritmo di addestramento il modello della Quasi Newton Approximation (QNA).

Il metodo di Newton utilizza il calcolo delle derivate seconde, quindi dell’Hessiano, nella determinazione del minimo dell’errore. Tuttavia in molti casi tale calcolo risulta computationalmente troppo complesso.

La QNA è un’ottimizzazione dell’algoritmo di addestramento basata su di un’approssimazione statistica dell’Hessiano, attraverso un calcolo ciclico del gradiente, che è alla base del metodo di retropropagazione.

Il QNA, anziché calcolare la matrice Hessiana, effettua una serie di passaggi intermedi, di minore costo computazionale, al fine di generare una sequenza di matrici che risulteranno un’approssimazione via via sempre più accurata di H.
MLPQNA – II

I parametri fondamentali dell’MLPQNA sono:

- Neuroni di input
- Hidden layers e numero di neuroni relativi
- Neuroni di output
- W-step (soglia di errore ad ogni approssimazione)
- Restarts (cicli di approssimazione)
- Decay (fattore di decadimento della legge di aggiornamento dei pesi)
- MaxIts (max numero di iterazioni)
- K-fold Cross Validation

Nel caso della classificazione si ottiene come risultato una matrice di confusione, insieme ai parametri caratterizzanti di efficienza, completezza, purezza e contaminazione per le rispettive classi.
TDA: MLP neural network + QNA

\[
\min_w E(w) = \frac{1}{2P} \sum_{p=1}^{P} E_p(w) = \frac{1}{2P} \sum_{p=1}^{P} (y(x^p;w) - d^p)^2
\]

\( E_p \) is a measure of the error related to the \( p \)-th pattern.

\[
\nabla^2 E(w^k)d^k \approx -\nabla E(w^k)
\]

Hessian approx. (QNA)

\( d^k \in \mathbb{R}^N \) DIRECTION OF SEARCH

\( \alpha^k \in \mathbb{R} \)

The implemented MLPQNA model uses Tikhonov regularization (AKA weight decay).

When the regularization factor is accurately chosen, then generalization error of the trained neural network can be improved, and training can be accelerated.

If it is unknown what Decay regularization value to choose (as usual), it could be possible to experiment the values within the range of 0.001 (weak regularization) up to 100 (very strong regularization).
TDA: MLP neural network + QNA

REGRESSION ERROR

Least Square error + Tikhonov regularization

\[
E = \sum_{i=1}^{N\text{patterns}} \frac{(y_i - t_i)^2}{2} + \frac{\|W\|^2 \lambda}{2}
\]

where, \( \lambda \) is the Decay, \( y \) and \( t \) are respectively, output and target for each pattern, while \( W \) is the weight matrix of MLP.

CLASSIFICATION ERROR

1. Cross entropy enabled \( \rightarrow \) cross-entropy per record estimated in bits (logarithm);

\[
E = \sum_{i=1}^{N\text{patterns}} \ln \left( \frac{1}{y_i} \right)
\]

1. Cross entropy disabled \( \rightarrow \) percentage of misclassified patterns at each cycle;
MLP trained by Quasi Newton rule

\[
\min_w E(w) = \frac{1}{2P} \sum_{p=1}^{P} E_p(w) = \frac{1}{2P} \sum_{p=1}^{P} (y(x^p; w) - d^p)^2
\]

\(E_p\) is a measure of the error related to the \(p\)-th pattern

\[
w^{k+1} = w^k + \alpha^k d^k
\]

\(d^k \in \mathbb{R}^N\) DIRECTION OF SEARCH

\[
d^k = -\nabla E(w^k)
\]

\[
d^k = \text{genetic operators}
\]

\[
\nabla^2 E(w^k)d^k \approx -\nabla E(w^k)
\]

Descent gradient (BP)

Genetic Algorithms (GA)

Hessian approx. (QNA)
MLP-BP Algorithm

By making the mathematical relations in practice, we can derive the complete standard algorithm for a generic MLP trained by BP rule as the following (ALG-1):

Let us consider a generic MLP with m output and n input nodes and with \( w_{ij}^{(t)} \) the weight between i-th and j-th neuron at time \( t \).

1) Initialize all weights \( w_{ij}^{(0)} \) with small random values, typically normalized in \([-1, 1]\);

2) Present to the network a new pattern \( p = (x_1, ..., x_n) \) together with the target \( c_p = (c_{p1}, ..., c_{pm}) \) as the value expected for network output;

3) Calculate the output of each neuron \( j \) (layer by layer) as: \( y_{pj} = f(\sum_i w_{ij} x_{pi}) \), except for the input neurons (in which their output is the input itself);

4) Adapt the weights between neurons at all layers, proceeding in the backward direction from output to input layer, with the following rule: \( w_{ij}^{(t+1)} = w_{ij}^{(t)} + \eta \delta_{pj} y_{pj} + \mu \Delta w^{(t)} \), where \( \eta \) is the gain term (also called learning rate) and \( \mu \) the momentum factor, both typically in \([0,1]\);

5) Goto 2 and repeat steps for all input patterns of the training set;
Let us consider a generic MLP with \( w(t) \) the weight vector at time (t).

1) Initialize all weights \( w(0) \) with small random values (typically normalized in \([-1, 1]\)), set constant \( \varepsilon \), set \( t=0 \) and \( g(0) = I \);

2) Present to the network all training set and calculate \( E(w(t)) \) as the error function for the current weight configuration;

3) If \( t=0 \)
   then \( d(t) = -\nabla E(t) \) (gradient of error function)
   else \( d(t) = -g(t-1)\nabla E(t-1) \)

4) Calculate
   \[
   w(t+1) = w(t) - \alpha d(t)
   \]
   where \( \alpha \) is obtained by line search expression \( \alpha(t) = -\frac{d(t)^T g(t)}{d(t)^T Hd(t)} \)

5) Calculate \( g(t+1) \) with equation
   \[
   g(t+1) = g(t) + \frac{pp^T}{p^Tv} - \frac{(g(t)v)v^T g(t)}{v^T g(t)v} + (v^T g(t)v)uu^T
   \]

6) If \( E(w(t+1)) > \varepsilon \) then \( t=t+1 \) and goto 2, else STOP

An hybrid solution combines control schemes with NN, (VSPI + NN = NVSPI), to obtain an optimized adaptive control system, able to correct motorized axis position in case of unpredictable and unexpected position errors.

VSPI = Variable Structure PI

\[ y = k_p e + k_i \int e dt \]  
\[ (\text{classic PI}) \]

\[ y = k_p(e) e + k_i(e) \int e dt \]  
\[ (\text{VSPI}) \]

\[ k_p = \frac{k_{op}}{1 + c_p |e|^n} \]
\[ (1.1) \]

\[ k_i = \frac{k_{oi}}{1 + c_i |e|^m} \]
\[ (1.2) \]

NVSPI = Neural VSPI

Submit to a MLP network a dataset (reference position trajectories) through the system in order to teach the NN to recognize fault condition of the VSPI response.

"A Neural Tool for Ground-Based Telescope Tracking control",
Machine Learning & Statistical Models

Neural Networks
- Perceptron
- Multi Layer Perceptron
- Radial Basis Functions

Feed Forward
- Competitive Networks
- Hopfield Networks
- Adaptive Reasoning Theory

Recurrent / Feedback

Hybrid
- Fuzzy Sets
- Genetic Algorithms
- K-Means
- Principal Component Analysis
- Support Vector Machine
- Soft Computing

Decision Analysis
- Decision Trees
- Random Decision Forests
- Evolving Trees
- Minimum Spanning Trees

Statistical Models
- Bayesian Networks
- Hidden Markov Models
- Mixture of Gaussians
- Principal Probabilistic Surface
- Maximum Likelihood
- $\chi^2$
- Negentropy
Genetic Algorithms

- A class of probabilistic optimization algorithms
- Inspired by the biological evolution process
- Uses concepts as “Natural Selection” and “Genetic Inheritance” (*Darwin 1859*)
- Originally developed by *John Holland* (1975)

A genetic algorithm maintains a population of candidate solutions for the problem at hand, and makes it evolve by iteratively applying a set of stochastic operators.
GA artificial vs natural

Genetic Algorithms

1. Start
2. Create initial population of chromosomes randomly
3. Create children using crossover and mutation/inversion
4. Find the best chromosome in the new population
5. Is this a new best?
   - Yes: Update the best solution
   - No: Created enough generations?
     - Yes: Return the best solution
     - No: repeat steps 3-5
6. End

Nature

1. Selection
2. Modification
3. Evaluation
4. Initiate & evaluate
5. Population
6. Modified offspring
7. Discard
8. Parents
9. Evaluated offspring
10. Deleted members

M. Brescia - Data Mining - lezione 3
# Genetic operators

## crossover

**Before:**
- \( s_1 = 1111010101 \)
- \( s_2 = 1110110101 \)

**After:**
- \( s_1' = 1110110101 \)
- \( s_2' = 1110101011 \)

## mutation

**Before:**
- \( s_1 = 1110110100 \)

**After:**
- \( s_1' = 1111010101 \)
- \( s_2' = 1111010101 \)

## elitism

Maintain best \( N \) solutions in the next population.

## Rank Tournament

Extracts \( k \) individuals from the population with uniform probability (without re-insertion) and makes them play a “tournament”, where the probability for an individual to win is generally proportional to its fitness. Selection pressure is directly proportional to the number \( k \) of participants.

Individual \( i \) will have a probability to be chosen

\[
\frac{f(i)}{\sum_i f(i)}
\]

All above operators are quite invariant in respect of the particular problem. What drastically has to change is the **fitness function** (how to evaluate population individuals).
Chromosomes could be:

- Bit strings (0101 ... 1100)
- Real numbers (43.2 -33.1 ... 0.0 89.2)
- Permutations of element (E11 E3 E7 ... E1 E15)
- Lists of rules (R1 R2 R3 ... R22 R23)
- Program elements (genetic programming)

... any data structure ...
**Example of genetic evolution**

### Initial Population

<table>
<thead>
<tr>
<th>(5,3,4,6,2)</th>
<th>(2,4,6,3,5)</th>
<th>(4,3,6,5,2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2,3,4,6,5)</td>
<td>(4,3,6,2,5)</td>
<td>(3,4,5,2,6)</td>
</tr>
<tr>
<td>(3,5,4,6,2)</td>
<td>(4,5,3,6,2)</td>
<td>(5,4,2,3,6)</td>
</tr>
<tr>
<td>(4,6,3,2,5)</td>
<td>(3,4,2,6,5)</td>
<td>(3,6,5,1,4)</td>
</tr>
</tbody>
</table>
Select Parents

(5,3,4,6,2) (2,4,6,3,5) (4,3,6,5,2)
(2,3,4,6,5) (4,3,6,2,5) (3,4,5,2,6)
(3,5,4,6,2) (4,5,3,6,2) (5,4,2,3,6)
(4,6,3,2,5) (3,4,2,6,5) (3,6,5,1,4)

Try to pick the better ones.
Example of genetic evolution

Create Off-Spring – 1 point

(5,3,4,6,2)  (2,4,6,3,5)  (4,3,6,5,2)
(2,3,4,6,5)  (4,3,6,2,5)  (3,4,5,2,6)
(3,5,4,6,2)  (4,5,3,6,2)  (5,4,2,3,6)
(4,6,3,2,5)  (3,4,2,6,5)  (3,6,5,1,4)
(3,4,5,6,2)
Create More Offspring

Example of genetic evolution
Example of genetic evolution

Mutate

(5,3,4,6,2)  (2,4,6,3,5)  (4,3,6,5,2)

(2,3,4,6,5)  (4,3,6,2,5)  (3,4,5,2,6)

(3,5,4,6,2)  (4,5,3,6,2)  (5,4,2,3,6)

(4,6,3,2,5)  (3,4,2,6,5)  (3,6,5,1,4)

(3,4,5,6,2)  (5,4,2,6,3)
Example of genetic evolution

Mutate

(5,3,4,6,2)  (2,4,6,3,5)  (4,3,6,5,2)
(2,3,4,6,5)  (2,3,6,4,5)  (3,4,5,2,6)
(3,5,4,6,2)  (4,5,3,6,2)  (5,4,2,3,6)
(4,6,3,2,5)  (3,4,2,6,5)  (3,6,5,1,4)

(3,4,5,6,2)  (5,4,2,6,3)
Example of genetic evolution

Eliminate

(5,3,4,6,2)  (2,4,6,3,5)  (4,3,6,5,2)

(2,3,4,6,5)  (2,3,6,4,5)  (3,4,5,2,6)

(3,5,4,6,2)  (4,5,3,6,2)  (5,4,2,3,6)

(4,6,3,2,5)  (3,4,2,6,5)  (3,6,5,1,4)

(3,4,5,6,2)  (5,4,2,6,3)

Tend to kill off the worst ones.
Example of genetic evolution

Integrate

(5,3,4,6,2) (2,4,6,3,5) (5,4,2,6,3)
(3,4,5,6,2) (2,3,6,4,5) (3,4,5,2,6)
(3,5,4,6,2) (4,5,3,6,2) (5,4,2,3,6)
(4,6,3,2,5) (3,4,2,6,5) (3,6,5,1,4)
Example of genetic evolution

Restart

(5,3,4,6,2)  (2,4,6,3,5)  (5,4,2,6,3)
(3,4,5,6,2)  (2,3,6,4,5)  (3,4,5,2,6)
(3,5,4,6,2)  (4,5,3,6,2)  (5,4,2,3,6)
(4,6,3,2,5)  (3,4,2,6,5)  (3,6,5,1,4)
Alternate solutions are too slow or overly complicated
Need an exploratory tool to examine new approaches
Problem is similar to one that has already been successfully solved by using a GA
Want to hybridize with an existing solution
Benefits of the GA technology meet key problem requirements
Benefits of GAs

Concept is easy to understand

Modular, separate from application

Supports multi-objective optimization

Good for “noisy” environments

Always an answer; answer gets better with time

Inherently parallel; easily distributed

Many ways to improve a GA application as knowledge about problem domain is gained

Easy to exploit previous or alternate solutions

Flexible building blocks for hybrid applications
Se la matrice dei pesi di una MLP la identifichiamo come un cromosoma, avremo una popolazione di matrici di pesi (popolazione di MLP) evoluta tramite operatori genetici.

**Errore output**

\[ N(U,e,P(e)) = \frac{1}{2k} \sum_i (Y_i - P(e))^2 \]

\[ |Y - P(e)| < \varepsilon \]

**Soglia di convergenza**

**Fase a ritroso con retro propagazione dell’errore**

**Diverse configurazioni di pesi (popolazioni di reti neurali) ottenute con evoluzione genetica**

**Fase in avanti con propagazione dell’input attraverso le funzioni di attivazione**

\[ net_{input_j} = \sum_k w_{jk}O_k \]

\[ net_{input_i} = \sum_i w_{hi}O_i \]

\[ w_{ji}(new) = w_{ji}(old) + \eta \delta_j + \alpha \Delta w_{ji}(old) \]

\[ f(\phi) = \frac{1}{1 + \exp(-\phi)} \]

**Funzione di attivazione**
Linked genes represent individual neuron weight values and thresholds which connect this neuron to the previous neural network layer. The genetic algorithm of the optimization over genes, structured like this, is standard.
Given a generic dataset with $N$ features and a target $t$, $pat$ a generic input pattern of the dataset, $pat = (f_1, \ldots, f_N, t)$ and $g(x)$ a generic real function, the representation of a generic feature $f_i$ of a generic pattern, with a polynomial sequence of degree $d$ is:

$$G(f_i) \equiv a_0 + a_1 g(f_i) + \cdots + a_d g^d(f_i)$$

Hence, the $k$-th pattern ($pat_k$) with $N$ features may be represented by:

$$Out(pat_k) \equiv \sum_{i=1}^{N} G(f_i) \equiv a_0 + \sum_{i=1}^{N} \sum_{j=1}^{d} a_j g^j(f_i) \tag{1}$$

The target $t_k$, concerning to pattern $pat_k$, can be used to evaluate the approximation error of the input pattern to the expected value:

$$E_k = (t_k - Out(pat_k))^2$$

With $NP$ patterns number ($k = 1, \ldots, NP$), at the end of the “forward” phase (batch) of the GA, we have $NP$ expressions (1) which represent the polynomial approximation of the dataset. In order to evaluate the fitness of the patterns, the Mean Square Error (MSE) or Root Mean Square Error (RMSE) may be used:

$$MSE = \frac{\sum_{k=1}^{NP} (t_k - Out(pat_k))^2}{NP} \quad RMSE = \sqrt{\frac{\sum_{k=1}^{NP} (t_k - Out(pat_k))^2}{NP}}$$

We use the trigonometric polynomial sequence, given by the following expression,

\[ g(x) = a_0 + \sum_{m=1}^{n} a_m \cos(m \cdot x) + \sum_{m=1}^{n} b_m \sin(m \cdot x) \]

\[ \text{NUM}_{\text{CHROMOSOMES}} = (B \cdot N) + 1 \]

where \( N \) is the number of features of the patterns and \( B \) is a multiplicative factor that depends from the \( g(x) \) function, in the simplest case is just 1, but can arise to 3 or 4

\[ \text{NUM}_{\text{GENES}} = (d \cdot B) + 1 \]

where \( d \) is the degree of the polynomial.

With 2100 patterns, 11 features each, the expression for the single \((k-th)\) pattern, using (1) with degree 6, will be:

\[ \text{Out}(\text{pat}_k) \approx \sum_{i=1}^{11} G(f_i) \approx a_0 + \sum_{i=1}^{11} \sum_{j=1}^{6} a_j \cos(j \cdot f_i) + \sum_{i=1}^{11} \sum_{j=1}^{6} b_j \sin(j \cdot f_i) \]

for \( k = 1,...,2100 \).

\[ \text{NUM}_{\text{CHROMOSOMES}} = (2 \cdot 11) + 1 = 23 \]

\[ \text{NUM}_{\text{GENES}} = (6 \cdot 2) + 1 = 13 \]
References


