DOI 10.4430/bgta0274

A tutorial on machine learning with geophysical applications

A.N. Qadrouh[1], J.M. Carcione[2], M. Alajmi[1] and M.M. Alyousif[1]

[1] *KACST, Riyadh, Saudi Arabia*
[2] *Istituto Nazionale di Oceanografia e di Geofisica Sperimentale - OGS, Sgonico (TS), Italy*

(Received: 24 December 2018; accepted: 14 March 2019)

ABSTRACT Machine learning (ML) is any predictive algorithm, or a combination of algorithms, that learns from data (that learns from "experience"), and makes predictions without being explicitly coded with a deterministic model. The most immediate example are neural networks, which are trained with data to minimise a cost function and perform predictions. In this work, we present some ML methods, with simple examples to grasp the basic concepts of each algorithm, avoiding formal mathematical complexities. The techniques involved in ML include gradient methods, genetic algorithms, simulated annealing, neural networks, and the novel field of quantum computing as an aid to speed the algorithms. Geophysical examples are given to illustrate practical applications.

Key words: machine learning, gradient descent, genetic algorithm, simulated annealing, neural networks, perceptrons, deep learning, artificial intelligence, data mining, seismic inversion, petrophysical prediction.

1. Introduction

Artificial intelligence started in the 1950s as a concept by which a computer has the same features of human intelligence (Samuel, 1959). Machine learning (ML) is a set of algorithms that learn from data and are able to make predictions. Specifically, deep learning is a subset of the previous techniques that uses multi-layered artificial neural networks to perform intelligent tasks, e.g. language translation, object detection, etc. (Schmidhuber, 2015). Writing a code to deal with any human task (e.g. driving a car) is very difficult. This is the goal of ML: feed the machine with the right data and it will generate the algorithm. A related field is data mining (Han *et al.*, 2011), which identifies patterns and establishes relations between elements in large data sets to predict new behaviours of the system under study.

The various ML methods involve the conventional artificial neural networks, gradient descent, genetic algorithms (GAs), simulated annealing, fuzzy decision tree, the imperialist competitive algorithm (ICA), particle swarm optimisation (PSO), hybrid methods, etc. Supervised learning uses a known data set (training data), and finds the model by minimising a cost function through back propagation of the error using, for instance, the gradient descent method to obtain the optimal weights. On the other hand, in unsupervised learning, there is no training data set and outcomes are unknown (Shalev-Shwartz and Ben-David, 2014). An example is image recognition: we have to recognise a set of images composed of squares and triangles. Supervised learning tells the algorithm that the first has 4 sides and the second has 3 sides. Unsupervised learning does not know what the figures are, and classifies the set into similar groups without aid.

Regarding hydrocarbon exploration, several works using ML have been published. Huang *et al.* (1996), Helle *et al.* (2001), and Hamada and Elshafel (2010) have predicted porosity and permeability from wireline logs using artificial neural networks. Helmy *et al.* (2010) used hybrid computational models for the characterisation of oil and gas reservoirs, based on support-vector machines (SVM) and functional networks (e.g. Shalev-Shwartz and Ben-David, 2014). Ali Ahmadi and Chen (2018) compared several ML techniques to obtain porosity and permeability. Aleardi (2015) estimated seismic velocity from well-log data with GAs and compared the results to those obtained with neural networks. Araya-Polo *et al.* (2017) trained a deep neural network to detect faults without seismic processing. Jia and Ma (2017) used the support-vector regression (SVR) method, a state-of-the-art ML regression tool, for learning interpolation of seismic data. Recently, geophysicists became aware that some problems, such as seismic modelling and full-waveform inversion (FWI) in 3D space, can be solved faster using quantum computers than classical computers (Moradi *et al.*, 2018). Quantum computing can make ML solutions exponentially faster than classical computing, although the coding is different (Feynman, 1982).

The goal of this tutorial is to provide a rigorous, yet easy to follow, introduction to the main concepts and to present some of the (albeit simplified) algorithms involved in ML, avoiding complex jargon and formal mathematical developments. It is intended for those who wish to start working in this wide field of research. For more complete methods and mathematical rigourosity, the reader may refer to the books of Smola and Vishwanathan (2008), Shalev-Shwartz and Ben-David (2014), and Alpaydin (2014), among others.

The paper is organised as follows: sections 2 to 6 illustrate the methods with simple examples, namely, gradient descent, GAs, simulated annealing, neural networks, and quantum computing. Finally, Section 7 presents geophysical applications, i.e. seismic inversion for reservoir/mining applications, and petrophysical prediction of well logs.

2. Gradient descent

Let us consider the n training data points $(x_k, \bar{y}_k)$ shown in Fig. 1 (dots) and obtain a best fit with the two-parameters predictor function:

$$y(x) = a_0 + a_1 x, \qquad (1)$$

where a_0 and a_1 are the parameters. Our goal is to minimise the cost function

$$C(a_0, a_1) = \frac{1}{2n} \sum_{k=1}^{n} [\bar{y}_k - y(x_k)]^2, \qquad (2)$$

i.e. find a_0 and a_1. To this purpose, we use gradient descent, which is an iterative method that modifies the a_i values till we arrive at a minimum.

The minimum represents the lowest cost our predictor can give us based on the training data. The goal is to "go downhill", and find the best a_i corresponding to the minimum. We initialise these parameters, say equal to 0, and start the iteration from here.

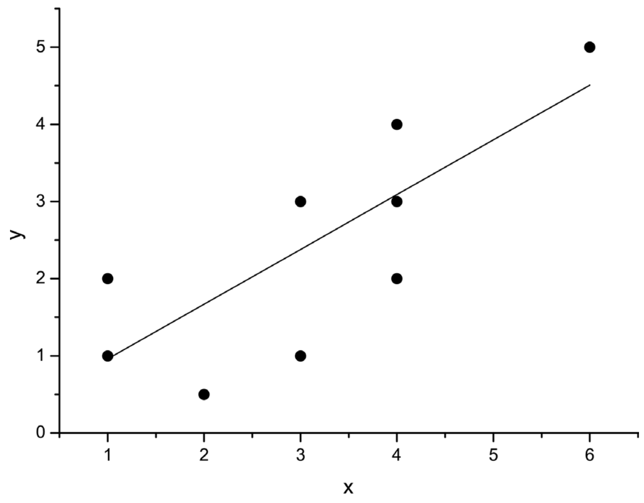

Fig. 1 - Data and fit with the gradient-descent method.

The algorithm giving us the descent is:

$$a_i \to a_i - \alpha \frac{\partial C(a_i)}{\partial a_i}, \tag{3}$$

where α is the learning rate, or how quick we move towards the minimum. The cost function C decreases fast if one departs from a_i in the direction of the negative gradient of C at a_i, $-\nabla C$. Then $C^{j+1}<C^j$, where j denotes the iterations. The learning rate α is generally small, e.g. 0.1, but depends on the problem, requiring preliminary tests.

On the basis of the cost function (Eq. 2), we obtain from Eq. 3:

$$\begin{aligned} a_0 &\to a_0 - \frac{\alpha}{n} \sum_{k=1}^{n} [y(x_k) - \bar{y}_k], \\ a_1 &\to a_1 - \frac{\alpha}{n} \sum_{k=1}^{n} [y(x_k) - \bar{y}_k] x_k. \end{aligned} \tag{4}$$

We repeat this procedure till convergence to the minimum, based on a given error ε, i.e., $|C^{j+1}-C^j| \le \varepsilon$. The solid line in Fig. 1 is the best fit, where $\alpha = 0.1$, and $\varepsilon = 10^{-6}$ requires 21 iterations. The optimal coefficients are: $a_0 = 0.249$ and $a_1 = 0.7099$, and the final cost function is $C = 0.3891$.

Another example, with 3 independent variables and 4 parameters, is:

$$y(x_1, x_2, x_3) = a_0 + a_1 x_1 + a_2 x_2^2 + a_3 x_1 x_3. \tag{5}$$

This is a 4D surface. Let us assume n points for each variable, i.e. we have n^3 points. The cost function is then:

$$C(a_0, a_1, a_2, a_3) = \frac{1}{2n^3} \sum_{k_1}^{n} \sum_{k_2}^{n} \sum_{k_3}^{n} [\bar{y}_{k_1 k_2 k_3} - y(x_1, x_2, x_3)]^2. \tag{6}$$

The parameter update is:

$$
\begin{aligned}
a_0 &\to a_0 - \frac{\alpha}{n^3}\sum_{k_1}\sum_{k_2}\sum_{k_3}[y(x_1,x_2,x_3) - \bar{y}_{k_1k_2k_3}],\\
a_1 &\to a_1 - \frac{\alpha}{n^3}\sum_{k_1}\sum_{k_2}\sum_{k_3}[y(x_1,x_2,x_3) - \bar{y}_{k_1,k_2,k_3}]x_{1k_1},\\
a_2 &\to a_2 - \frac{\alpha}{n^3}\sum_{k_1}\sum_{k_2}\sum_{k_3}[y(x_1,x_2,x_3) - \bar{y}_{k_1,k_2,k_3}]x^2_{2k_2},\\
a_3 &\to a_3 - \frac{\alpha}{n^3}\sum_{k_1}\sum_{k_2}\sum_{k_3}[y(x_1,x_2,x_3) - \bar{y}_{k_1,k_2,k_3}]x_{1k_1}x_{3k_3}.
\end{aligned}
\qquad (7)
$$

We consider the exact y points given by function in Eq. 5. Let us assume $\varepsilon = 10^{-20}$, $a_0 = -1$, $a_1 = 1$, $a_2 = 4$ and $a_3 = 8$, and $n = 10$ equispaced points for each variable, with $x_1 \in [0,1]$, $x_2 \in [-1,0]$ and $x_3 \in [1,2]$, and start the iteration from $a_0 = a_1 = a_2 = a_3 = 0$. With $\alpha = 0.01$ the algorithm requires 340 iterations to converge and we obtain the true a_i values given above. When $\alpha = 0.1$, the algorithm diverges.

Let us make the problem 3D, with $x_3 = x_2$ (now the normalisation in Eq. 7 is n^2 instead of n^3). Fig. 2 shows the surface y. The algorithm requires $\alpha = 0.1$ to converge, whereas $\alpha = 0.2$ diverges.

Let us perturbate the y surface ±40% its value with random numbers. Fig. 3 shows the perturbed surface. Using $\alpha = 0.1$, we converge to a minimum after 362 iterations, with $a_0 = -0.99$, $a_1 = 1.02$, $a_2 = 4.08$ and $a_3 = 8.27$, which give $C = 0.018$. A more advanced method is the stochastic gradient descent, where the update direction is not exactly based on the gradient. Instead, the direction is a random vector, such that its expected value at each iteration equals the gradient direction (e.g. Shalev-Shwartz and Ben-David, 2014).

3. Genetic algorithms

GA is a search method that mimics the process of natural selection, based on concepts as crossovers and mutations to generate new genotypes. GA and genetic programming are the most

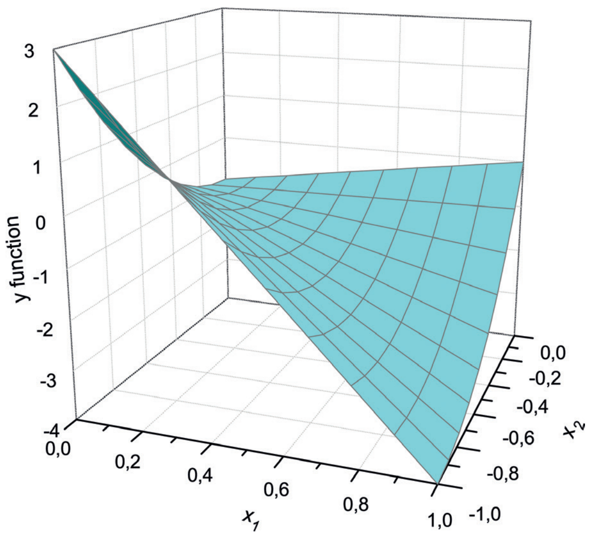

Fig. 2 - 3D surface. The coefficients are found with the gradient-descent method.

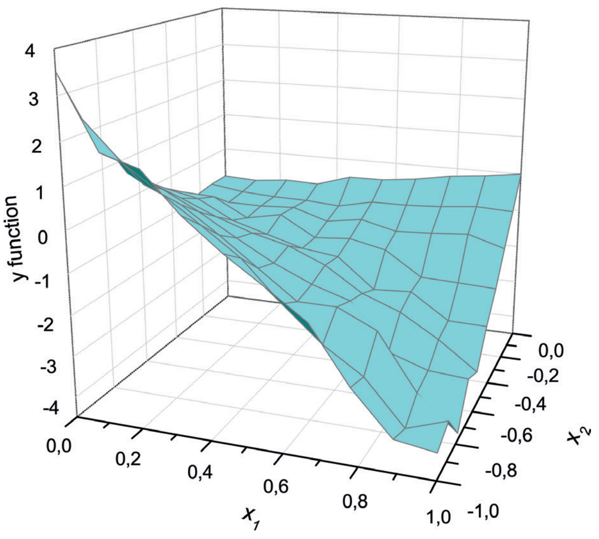

Fig. 3 - Perturbed 3D surface. The coefficients are found with the gradient-descent method.

prominent computational techniques, where Darwin's concept of evolution is adapted to solve a given problem.

The method defines a population of chromosomes (the unknowns to be found), which undergo a redistribution according to their fitness, and crossovers and mutations to find the fittest one, i.e. that chromosome (solution) that minimises an objective function. A chromosome is composed of genes whose values can be numerical, binary, symbols, etc. In the following example, each gene is a natural (positive) number. We present here a particular example (Hermawanto, 2013), but the algorithm can have many variations.

3.1. Examples

Let us consider the following equality:

$$A + 2B + 3C + 4D = 30, \tag{8}$$

where A, B, C and D are natural numbers. The GA will find these coefficients by minimising the objective function

$$o(A, B, C, D) = |A + 2B + 3C + 4D - 30| = 0. \tag{9}$$

Note that there are many solutions, for instance (A, B, C, D) = (7, 5, 3, 1), (19, 1, 3, 0), (1, 8, 3, 1) and (13, 2, 3, 1) are solutions.

Any set (A, B, C, D) is called a "chromosome" and each component is called a "gene". The "population" is the number of chromosomes. The GA process works as follows:

1. Initialization: consider 6 chromosomes and generate its initial values randomly, between 1 and 30 (solutions with one of the coefficients equal to zero are excluded):
   Chr 1: (26,16,3,20)
   Chr 2: (13,22,28,23)
   Chr 3: (8,2,23,10)

Chr 4: (19,23,30,11)
Chr 5: (8,30,22,23)
Chr 6: (20,3,19,27) .
The objective functions in Eq. 9 are:
$o_1 = 117$
$o_2 = 203$
$o_3 = 91$
$o_4 = 169$
$o_5 = 196$
$o_6 = 161$.

2. Obtain the cumulative probability:

$$C_k = \frac{1}{\sum_{j=1}^{6}(1+o_j)^{-1}} \sum_{n=1}^{k}(1+o_n)^{-1} = \frac{1}{\sum_{j=1}^{6} f_j} \sum_{n=1}^{k} f_n, , \quad k = 1, ..., 6, \tag{10}$$

where $o_j = o(A_j, B_j, C_j, D_j)$ and $f_j = (1 + o_j)^{-1}$ is the fitness function of the jth chromosome (1 is added to avoid dividing by zero). We get the following fitnesses and cumulative probabilities:

$f_1 = 0.00847$,	$C_1 = 0.204$
$f_2 = 0.00490$,	$C_2 = 0.323$
$f_3 = 0.01086$,	$C_3 = 0.585$
$f_4 = 0.00588$,	$C_4 = 0.728$
$f_5 = 0.00507$,	$C_5 = 0.850$
$f_6 = 0.00617$,	$C_6 = 1$.

3. The next process is to give more weight to those chromosomes with the lower objective function, or higher fitness function. Then, we redistribute the chromosomes by using the roulette-wheel technique. Generate random numbers between 0 and 1. We obtain $r_i = 0.27$, 0.43, 0.76, 0.48, 0.24 and 0.27. If $C_1 < r_1 < C_2$, then select Chr 2 as Chr 1, if $C_5 < r_2 < C_6$, then select Chr 6 as Chr 2, if $C_2 < r_3 < C_3$, then keep Chr 3 as Chr 3, etc. The redistribution is:
new Chr 1 = old Chr 2
new Chr 2 = old Chr 3
new Chr 3 = old Chr 5
new Chr 4 = old Chr 3
new Chr 5 = old Chr 2
new Chr 6 = old Chr 2.
Then, the new chromosomes are:
Chr 1: (13, 22, 28, 23)
Chr 2: (8, 2, 23, 10)
Chr 3: (8, 30, 22, 23)
Chr 4: (8, 2, 23, 10)
Chr 5: (13, 22, 28, 23)
Chr 6: (13, 22, 28, 23).
This process leaves the fittest chromosomes, not deterministically, but based on random numbers. In fact, the original chromosome 3 remained (the one with lowest objective function), but chromosome 1 was excluded even if it has a low objective function. To better

see this, let us assume that all the chromosomes have a fitness of 1 unless chromosome number 3 with a fitness of 10. The sum of the fitnesses is 15. Then C_1 = 1/15, C_2 = 2/15, C_3 = 12/15, C_4 = 13/15, C_5 = 14/15 and C_6 = 15/15. The interval between C_2 and C_3 is the largest (between 0.13 and 0.8) and any random number between 0 and 1 will have more probability to fall in this interval, so that chromosome 3 has more probability to be between the distributed population. This is the roulette-wheel selection, also known as fitness proportionate selection. A fitter individual has a greater pie on the wheel, i.e. a greater chance to fall in one of the slots, the higher the fitness the higher the chance the chromosome has to be selected. The concept is shown pictorially in Fig. 4.

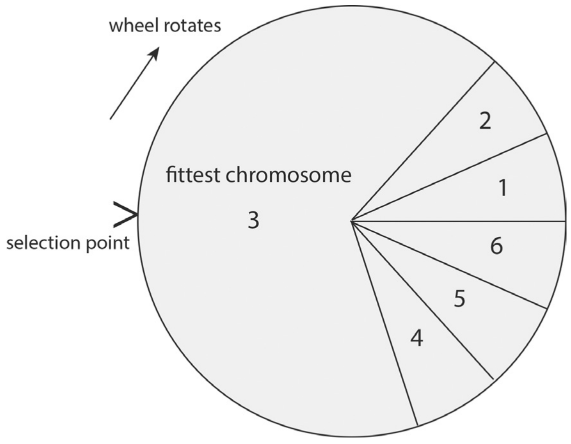

Fig. 4 - The roulette-wheel concept for chromosomes selection-redistribution.

4. Select parents that will crossover. Generate random numbers between 0 and 1. We obtain r_i = 0.35, 0.16, 0.48, 0.89, 0.90 and 0.06. Select a crossover rate c_R =0.4. If $r_i < c_R$, select the ith chromosome as a parent. Then, parents are Chr 1, 2 and 6 (3 parents).
5. Crossover: chromosomes 1, 2 and 6 will combine with each other as (1, 2), (2, 6) and (6, 1) [(2, 1), (6, 2) and (1, 6) are not required]. Here, the crossover means interchanging genes at a given single location (one-cut point). To determine the location for each crossover, we generate natural random numbers between 1 and 3 (being 3 the number of genes minus 1). We obtain: r_i = 3, 2 and 2. Then, for the 1st, 2nd and 3rd crossovers, the parent genes will be cut at genes 3, 2 and 2, respectively. This means:
   Chr 1 = Chr 1 × Chr 2 or (13, 22, 28, 23) × (8, 2, 23, 10) = (13, 22, 28, 10)
   Chr 2 = Chr 2 × Chr 6 or (8, 2, 23, 10) × (13, 22, 28, 23) = (8, 2, 28, 23)
   Chr 6 = Chr 6 × Chr 1 or (13, 22, 28, 23) × (13, 22, 28, 23) = (13, 22, 28, 23)
   (see Fig. 5), Actually, the last two in the third line are the same.
   Then, after the crossover, the population becomes:
   Chr 1: (13, 22, 28, 10)
   Chr 2: (8, 2, 28, 23)
   Chr 3: (8, 30, 22, 23)
   Chr 4: (8, 2, 23, 10)
   Chr 5: (13, 22, 28, 23)
   Chr 6: (13, 22, 28, 23).

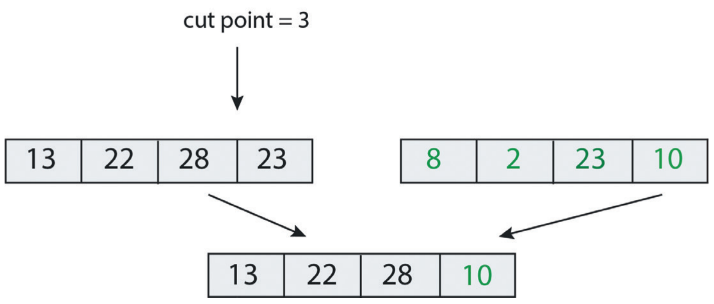

Fig. 5 - Crossover between chromosomes 1 and 2 at cut-point 3.

6. Mutation. The mutation rate, m_R, dictates the number of chromosomes to mutate. Let us take m_R = 0.1. Then, the number of mutations is equal to the number of genes (24 here) times the rate = 24×0.1=2. We now generate 2 natural random numbers between 1 and 24. We obtain 8 and 24. We also generate two natural random numbers between 1 and 30. We obtain 15 and 8. Then, we replace the 8th gene with 15 and the 24th gene with 8.
   Then, after the mutation, the population is:
   Chr 1: (13, 22, 28, 10)
   Chr 2: (8, 2, 28, 15)
   Chr 3: (8, 30, 22, 23)
   Chr 4: (8, 2, 23, 10)
   Chr 5: (13, 22, 28, 23)
   Chr 6: (13, 22, 28 ,8).
   The objective functions are:
   $o_1 = 151$
   $o_2 = 126$
   $o_3 = 196$
   $o_4 = 91$
   $o_5 = 203$
   $o_6 = 143$.
   These values have to decrease after each iteration. The process of mutation is illustrated in Fig. 6. Mutations preserve the population diversity to avoid convergence to local solutions. We stop the iterations when the objective function of one of the chromosomes is less than an ε (10^{-6} in this example).

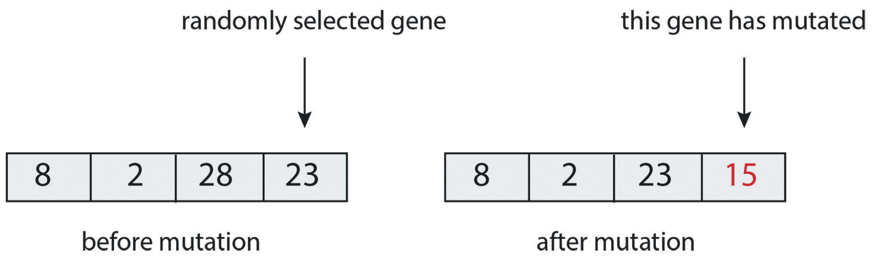

Fig. 6 - Mutation of gene number 8 in chromosome 2.

After 169 iterations of the previous steps, we find a chromosome whose objective function is zero, with the above values $c_R = 0.4$ and $m_R = 0.1$. We obtain the solution: $A = 1$, $B = 8$, $C = 3$ and $D = 1$, given by the 4th chromosome. If $c_R = 0.25$ and $m_R = 0.1$, we obtain the solution: $A = 3$, $B = 13$, $C = 2$ and $D = 3$, given by the 3rd chromosome after 118 iterations. If $c_R = 0.25$ and $m_R = 0.2$, we obtain the solution: $A = 2$, $B = 2$, $C = 9$ and $D = 1$, given by the 2nd chromosome after 41 iterations.

Let us apply now this algorithm to the first problem (see Fig. 1). We use 3 chromosomes with two genes each, corresponding to the unknown coefficients a_0 and a_1. We consider a crossover rate $c_R = 0.8$ and a mutation rate $m_R = 0.5$. Since there are 3 chromosomes, only one crossover occurs, according to the previous algorithm. In this case, we obtain $a_0 = 0.1472$ and $a_1 = 0.7208$, instead of $a_0 = 0.249$ and $a_1 = 0.7099$, computed with the gradient-descent method. The fit is shown in Fig. 7. The solution has been obtained with a minimum objective function in Eq. 2 equal to 0.3865 after 51567 iterations (gradient descent has a final cost value of 0.3891).

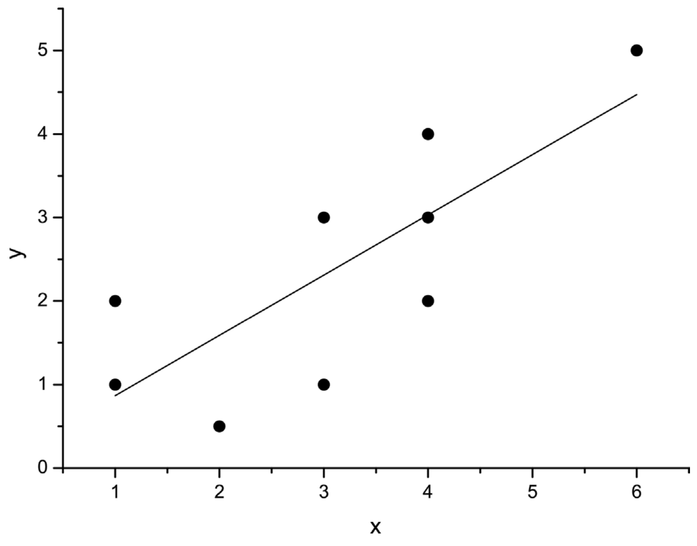

Fig. 7 - Fit of the data of Fig. 1 with the GA.

Public-domain algorithms are available. PGAPack (Levine, 1996) is a parallel GA library with the ability to be called from Fortran or C. (https://www. researchgate.net/publication/2813201_Users_Guide_to_the_PGAPack_Parallel_Genetic_Algorithm_Library). PIKAIA (http://www.hao.ucar.edu/modeling/pikaia/pikaia.php) is a general purpose function optimisation Fortran-77 subroutine based on a GA. PIKAIA is a public domain software available electronically from the anonymous ftp archive of the High Altitude Observatory. The subroutine is particularly useful (and robust) in treating multimodal optimisation problems. DISCIPULUS, from Register Machine Learning Technologies, Inc. (Foster, 2001), is the world's first commercially available, industrial strength genetic programming system. DISCIPULUS writes computer programs from examples you give it. These examples are contained in "training data," "validation data" and "testing data" that you provide to the algorithm.

4. Simulated annealing

Simulated annealing is an optimisation method that distinguishes between local optima (minimum or maximum). When minimising a function, any downhill step is accepted, but an uphill step may be accepted, in order to escape from a local minimum. It is based on the temperature T and the size of the downhill move (a vector $\mathbf{s}$) in a probabilistic manner. The smaller T and $\mathbf{s}$ are, the more likely that move will be accepted. The uphill decision is made by the Metropolis criterion (i.e. randomly). As the optimisation process proceeds, the length of the steps decreases and the algorithm converge to the global minimum. A physical analogy is the cooling of melted metal. After a slow cooling (annealing), the metal arrives at a low energy state. If the cooling is too quick, the algorithm might not escape local energy minima and when fully cooled it may contain more energy than annealed metal.

Let us find the maximum of function $f(\mathbf{x})$, where $\mathbf{x}$ is the initial vector of unknowns of length N. A new vector is defined as

$$x_i' = x_i + r_{-11} s_i, \tag{11}$$

where r_{-11} is a random number between -1 and 1, and s_i is a component of $\mathbf{s}$. Then, if $f' = \partial f/\partial x_i > f$, $\mathbf{x}'$ is accepted, $\mathbf{x}$ is set to $\mathbf{x}'$ and the algorithm moves uphill. If $f' \leq f$, the Metropolis criterion decides the acceptance: generate a random number between 0 and 1 = r_{01}; if

$$\begin{array}{ll} p = \exp[(f' - f)/T_0] > r_{01}, & \text{accept } \mathbf{x}', \text{ move downhill}, \\ p \geq r_{01}, & \text{reject } \mathbf{x}'. \end{array} \tag{12}$$

Downhill moves are decreased by a low T. After N_S steps (decided by the user), $\mathbf{s}$ is adjusted so that 50% of all moves are accepted. This samples the function widely. Then, $\mathbf{s}$ is increased and the number of acceptances decreases. After N_T steps (decided by the user), the new temperature is:

$$T' = R_T T, \tag{13}$$

where R_T is a number between 0 and 1. Lower temperatures decrease downhill moves. Finally, we compare the largest $\mathbf{x}$ at the end of each temperature reduction with the last one. If the difference is less than ε, the algorithms terminates. We use the algorithm developed by Goffe *et al*. (1994), based on Corana *et al*. (1987). The Fortran code can be found in: https://econwpa.ub.uni-muenchen.de/econ-wp/prog/papers/9406/9406001.txt.

4.1. Example 1. The Judge function

We consider the Judge function $J(x, y)$, which has two minima, two parameters (x and y) and 3 sets of 20 coefficients (see Fig. 8):

$$J(x,y) = \sum_{i=1}^{20} (x + a_{2i} y + a_{1i} y^2 - a_{0i})^2, \tag{14}$$

where the coefficients are given in Table 1. The local minimum is:
$(x_l, y_l) = (2.354, -0.319)$, with $J = 20.9895$,

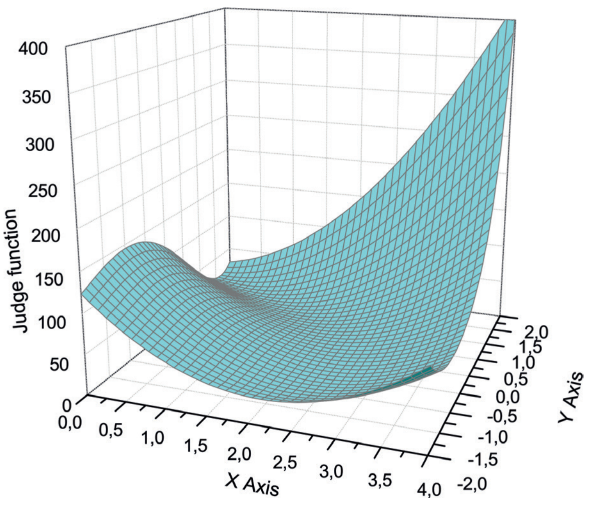

Fig. 8 - Judge function to test the simulated annealing method.

and the global minimum is:
$(x_g, y_g) = (0.86, 1.23)$, with $J = 16.0817$.

The input parameters are: initial temperature: $T_0 = 5$, number of parameters: $N = 2$; starting values of $\mathbf{x} = (0, 0)$; MAX = .FALSE. (it means that we look for the minimum, if MAX = .TRUE., we look for the maximum); $R_T = 0.5$; $\varepsilon = 10^{-6}$; $N_S = 20$; $N_T = 5$; number of final functions use to

Table 1 - Coefficients of the Judge function.

i	a_{0i}	a_{1i}	a_{2i}
1	4.284	0.645	0.286
2	4.149	0.585	0.973
3	3.877	0.310	0.384
4	0.533	0.058	0.276
5	2.211	0.455	0.973
6	2.389	0.779	0.543
7	2.145	0.259	0.957
8	3.231	0.202	0.948
9	1.998	0.028	0.543
10	1.379	0.099	0.797
11	2.106	0.142	0.936
12	1.428	0.296	0.889
13	1.011	0.175	0.006
14	2.179	0.180	0.828
15	2.858	0.842	0.399
16	1.388	0.039	0.617
17	1.651	0.103	0.939
18	1.593	0.620	0.784
19	1.046	0.158	0.072
20	2.152	0.704	0.889

decide upon termination: NEPS = 4; the maximum number of function evaluations: MAXEVL = 10^5; the lower bound for the allowable solution variables: LB (i) = -10^{25}, for all N elements; the upper bound for the allowable solution variables: UB (i) = 10^{25}, for all N elements; vector that controls the step length adjustment: $C(i) = 2$ for all N elements; the first seed for the random number generator: ISEED1 = 1; the second seed for the random number generator: ISEED2 = 2. To reach the global minimum, the algorithm uses 4801 function evaluations (of which 2218 accepted evaluations). The final temperature is $T = 0.596\times10^{-6}$. If we start at the local minimum $\mathbf{x}$ = (2.354, –0.319), the algorithm uses 5001 function evaluations.

4.2. Example 2. Linear regression

Let us now consider the first example, i.e. Eq. 1. We have to obtain a_0 and a_1 that minimise the cost function in Eq. 2:

$$C(a_0, a_1) = \frac{1}{2n}\sum_{k=1}^{n}(\bar{y}_k - a_0 - a_1 x_k)^2. \tag{15}$$

We use the same initial input data as before. To reach the global minimum, the algorithm uses 5401 function evaluations (of which 2555 accepted evaluations). The solution is $a_0 = 0.138$ and $a_1 = 0.723$. The final temperature is $T = 0.745\times10^{-7}$.

5. Neural networks

A neural network is a method inspired on how human brain works. It is composed of layers (columns) of "neurons" operating in parallel. A simple network has input, hidden and output layers, connected via variable weights, which represent parameters (to be found) to solve a specific problem. To ensure that each variable is treated equally in a model, data are usually rescaled to a certain interval such as [-1, 1], using an activation function (Kröse and van der Smagt, 1996; Haykin, 2009).

5.1. Example 1: the XOR problem

Let us first solve a simple problem related to the XOR function, i.e. predict the output, B, of XOR logic gates given two binary inputs A_1 and A_2. The XOR function returns a true value (= 1) if the two inputs are not equal and a false value (= 0) if they are equal (Table 2).

Table 2 - Input and output of the XOR function.

input A_1	input A_2	output B
0	0	0
0	1	1
1	1	0
1	0	1

Actually, solving the XOR problem is easy using Fortran. Why would you use a neural network to solve such a trivial task? The answer is that when we learn something new, we want to start with simple tasks and take away as much of the complexity as possible.

We consider first the "forward propagation". In Fig. 9 we have two values, A_n, $n = 1, 2$, at the input neurons and the true value B at the output neuron. This is called a multi-layer "perceptron" network. The task is to find the weights that predict the correct output. We generate 6 random numbers w_{ni} between 0 and 1 (in the iterations this constraint is relaxed), called weights, assigned to the synapses. Then, we compute:

$$\begin{aligned} a_1 &= w_{11}A_1 + w_{21}A_2, \\ a_2 &= w_{12}A_1 + w_{22}A_2, \\ a_3 &= w_{13}A_1 + w_{23}A_2 \end{aligned} \tag{16}$$

(or $a_i = w_{ni}A_n$).

Now we apply an "activation function" to a_i, generally a sigmoid:

$$f(x) = \frac{1}{1 + \exp(-x)}, \tag{17}$$

and obtain $f(a_i)$ [note that $\tanh(x) = 2f(2x) - 1$ could also be used as activation function]. The activation function maps the resulting values into the desired range between 0 to 1.

Then, we generate 3 weights, w_i (also random numbers between 0 and 1), and compute

$$b = \sum_{i=1}^{3} f(a_i)w_i, \tag{18}$$

and apply again the activation function:

$$b_1 = f(b) = f\left[\sum_{i=1}^{3} f(a_i)w_i\right]. \tag{19}$$

The cost function is:

$$C = \frac{1}{2}(B - b_1)^2, \tag{20}$$

where initially b_1 can be a real number (e.g. 0.54, and not 0 or 1). We need a set of weight values that ensure the network produces the expected output of Table 2. Next, we adjust the weights w_{ni}

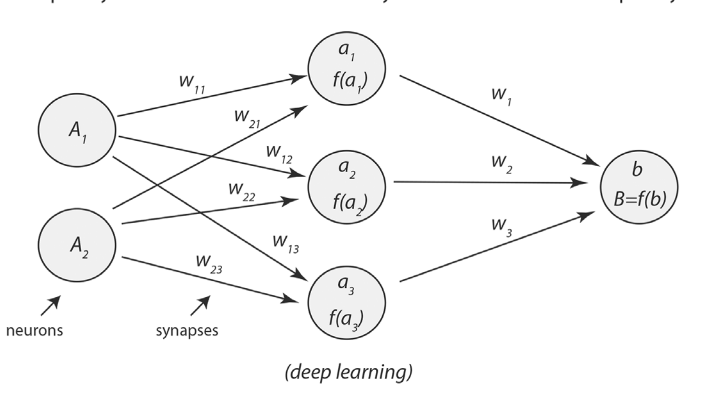

Fig. 9 - Multi-layer perceptron to solve the XOR problem.

and w_i so that the cost C decreases. We then perform the so-called "back propagation". Let us use the gradient-descent method. Simulated annealing, genetic programming or the Marquardt algorithm (e.g. Huang *et al.*, 1996) can be used as well. The update is:

$$w_i \rightarrow w_i - \alpha \frac{\partial C(a_i)}{\partial w_i}, \tag{21}$$

which after some calculation becomes:

$$w_i \rightarrow w_i + \alpha(B - b_1)\frac{\partial b_1}{\partial b} f(a_i) = w_i + \alpha(B - b_1) f'(b) f(a_i), \tag{22}$$

where:

$$f'(x) = \frac{\exp(-x)}{[1 + \exp(-x)]^2}. \tag{23}$$

Regarding the initial weights, we note that

$$b_1 = f(b) = f\left[\sum_i f(a_i) w_i\right] = f\left[\sum_i f\left(\sum_n A_n w_{ni}\right) w_i\right] \tag{24}$$

and we require

$$w_{ni} \rightarrow w_{ni} - \alpha \frac{\partial C(a_i)}{\partial w_{ni}}, \tag{25}$$

or

$$w_{ni} \rightarrow w_{ni} + \alpha(B - b_1) f'(b) f'(a_i) w_i A_n. \tag{26}$$

Now that we have the new weights, we start the forward propagation again. Both forward and back propagation are re-run many times on each input combination until the network can accurately predict the expected output of the possible inputs using forward propagation.

Let us assume $A_1 = 1$, $A_2 = 1$, $w_{11} = 0.3$, $w_{12} = 0.6$, $w_{13} = 0.3$, $w_{21} = 0.1$, $w_{22} = 0.7$, $w_{23} = 0.4$, $w_1 = 0.2$, $w_2 = 0.9$ and $w_3 = 0.5$. After 16 iterations and assuming $\alpha = 950$, the result is $b_1 = 6.72\times10^{-35}$ (the correct result is $B = 0$), with $w_{11} = 11.98$, $w_{12} = -19.30$, $w_{13} = -14.26$, $w_{21} = 11.78$, $w_{22} = -19.20$, $w_{23} = -14.16$, $w_1 = -78.68$, $w_2 = -102.34$ and $w_3 = -87.28$.

Let us assume $A_1 = 0$, $A_2 = 1$ and the same initial weights as above. After 2 iterations and assuming $\alpha = 950$, the result is $b_1 = 1$ (the correct result is $B = 1$), with $w_{11} = 0.3$, $w_{12} = 0.6$, $w_{13} = 0.3$, $w_{21} = 2.58$, $w_{22} = 10.65$, $w_{23} = 6.39$, $w_1 = 26.39$, $w_2 = 34.23$ and $w_3 = 30.36$.

5.2. Example 2: linear regression

The first problem in this tutorial is the linear regression related to Eq. 1. A neural network for this problem is shown in Fig. 10. The activation function is 1 here and the hidden layer has no effect (the hidden weight is 1). For the forward propagation, we start with random weights and compute y with Eq. 1. Then, we use an activation function $f(x) = x$ (see Eq. 17), such that $f(y) = y$, and $b = y$ (see Eq. 18); since $w = 1$ (see Eq. 19), it is $b_1 = y$. Next, we compute the cost function in Eq. 2 and back propagate to modify the weights, a process that we do with the gradient-descent method.

The above process implies actually no activation function, and the weights would simply do a

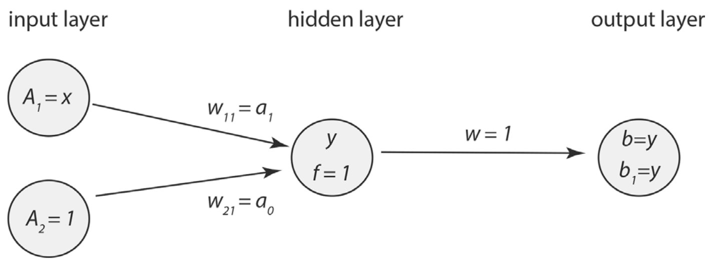

Fig. 10 - Neural network for the linear regression problem.

linear transformation, since the neural network is essentially a linear regression model.

Recent advances in neural networks can be found in Suzuki (2013), for various applications, and Poulton (2001) and Nikravesh *et al*. (2003), for hydrocarbon exploration. A public-domain software can be found on-line in: http://www.philbrierley.com/ code.html.

6. Quantum computing

Quantum computing uses quantum mechanical concepts to solve a given problem. Feynman (1982) developed the idea of a quantum computer, based on the superposition of quantum states and entanglement. The method uses quantum bits or qubits, which describe the superpositions.

A simple example illustrates the concept. Suppose that we have to find a password of $L = 3$ digits, where the digits are composed of 2 digits, 0 and 1 (binary numbers) (e.g. 101 = 5). A classical computer will analyse each of the 2^L numbers one by one, whereas a quantum computer will analyse the whole set simultaneously, based on the qubit $|\psi\rangle = \sum_{i=1}^{2^L} a_i |i\rangle$ that contains the superposition of all the states (numbers), where the ket $|i\rangle$ represents each number and a_i its probability ($|101\rangle = |5\rangle$ is the right ket). The following sections explain how this is done.

6.1. Quantum states and qubits

Let us see the mathematical meaning of the terms mentioned above. Instead of bits (0 and 1) used by ordinary computers, quantum computing uses "qubits". An example is:

$$|\phi\rangle = \begin{bmatrix} a \\ b \end{bmatrix} = a|0\rangle + b|1\rangle, \quad |0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \tag{27}$$

where, in the complex plane, ket $|0\rangle = (1, 0)$ and ket $|1\rangle = (0, 1) = \mathrm{i}$, where $\mathrm{i} = \sqrt{-1}$. Coefficients a and b can be complex and the sum means the "superposition" of two quantum states. States $|0\rangle$ and $|1\rangle$ have probabilities $|a^2|$ and $|b^2|$, such that $|a^2| + |b^2| = 1$. The bra of $|\phi\rangle$ is:

$$\langle\phi| = (a^*, b^*)^\top = a^*\langle 0| + b^*\langle 1|, \tag{28}$$

such that the braket or scalar product is:

$$\langle\phi|\phi\rangle = |a^2| + |b^2| = 1. \tag{29}$$

Now,

$$|00\rangle = |0\rangle \otimes |0\rangle, \quad |01\rangle = |0\rangle \otimes |1\rangle, \text{etc.}, \tag{30}$$

where $\otimes$ denotes the vector product. Define

$$|\phi_1\rangle = a|0\rangle + b|1\rangle, \quad \text{and} \quad |\phi_2\rangle = c|0\rangle + d|1\rangle, \tag{31}$$

and consider

$$\begin{aligned}|\psi\rangle &= \tfrac{1}{\sqrt{2}}(|00\rangle + |11\rangle) = |\phi_1\rangle \otimes |\phi_2\rangle = (a|0\rangle + b|1\rangle) \otimes (c|0\rangle + d|1\rangle) \\ &= ac|00\rangle + ad|01\rangle + bc|10\rangle + bd|11\rangle.\end{aligned} \tag{32}$$

There are no values a, b, c and d such that $ac = bd = \frac{1}{\sqrt{2}}$ and $ad = bc = 0$. Then $|\psi\rangle$ cannot be the superposition of two separate states ($|\phi_1\rangle$ and $|\phi_2\rangle$), so that state $|\psi\rangle$ is "entangled". Instead

$$\begin{aligned}|\psi\rangle &= \tfrac{1}{\sqrt{2}}(|00\rangle + |10\rangle) = |\phi_1\rangle \otimes |\phi_2\rangle = (a|0\rangle + b|1\rangle) \otimes (c|0\rangle + d|1\rangle) \\ &= ac|00\rangle + ad|01\rangle + bc|10\rangle + bd|11\rangle\end{aligned} \tag{33}$$

is not entangled since the choice $a = b = \frac{1}{\sqrt{2}}$, $c = 1$ and $d = 0$ satisfies Eq. 33.

A 2-qubit state is:

$$|\psi\rangle = a_0|00\rangle + a_1|01\rangle + a_2|10\rangle + a_3|11\rangle \equiv a_0|0\rangle + a_1|1\rangle + a_2|2\rangle + a_3|3\rangle. \tag{34}$$

A 3-qubit state is:

$$|\psi\rangle = a_0|000\rangle + a_1|001\rangle + a_2|010\rangle + a_3|011\rangle + a_4|100\rangle + a_5|101\rangle + a_6|110\rangle + a_7|111\rangle \equiv \sum_{i=0}^{2^3-1} a_i|i\rangle. \tag{35}$$

(e.g. binary number 110 is $1\times2^2+1\times2^1+0\times2^0 = 6$). An L-qubit state is:

$$|\psi\rangle = \sum_{i=0}^{2^L-1} a_i|i\rangle, \quad \sum_i |a_i|^2 = 1. \tag{36}$$

A measurement of $|\psi\rangle$ gives $|i\rangle$ with probability $|a_i|^2$.

For 2-qubits, the vector product is alternatively defined as:

$$|a\rangle \otimes |b\rangle = |ab\rangle = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} \otimes \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} a_1b_1 \\ a_1b_2 \\ a_2b_1 \\ a_2b_2 \end{bmatrix}. \tag{37}$$

A gate alters the state of a qubit. The NOT gate X acting on $|\psi\rangle = \alpha|a\rangle + \beta|b\rangle$ inverts the bits

$$X|\psi\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \left\{ \alpha \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} + \beta \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \right\} = \alpha \begin{bmatrix} a_2 \\ a_1 \end{bmatrix} + \beta \begin{bmatrix} b_2 \\ b_1 \end{bmatrix}. \tag{38}$$

Any qubit of amplitude A can be written as:

$$Q(\theta) = A(|0\rangle \cos\theta + |1\rangle \sin\theta) = A\exp(\mathrm{i}\theta), \tag{39}$$

where θ is the phase angle, and mod[A exp(i θ), exp($\pm$i2π)] = A exp(i θ), or $Q(\theta \pm 2\theta) = Q(\theta)$, while $Q(\theta \pm \theta) = -Q(\theta)$, where mod($m$, n) is the remainder when n divides each power of m, also denoted as m mod n.

The Hadamard gate

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{40}$$

creates a superposition of states with equal weights, i.e. acting on $|0\rangle$ rotates the qubit by $\theta=\pi/4$:

$$H|0\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}}. \tag{41}$$

Then, for an L-qubit

$$H|0\rangle \otimes H|0\rangle \ldots \otimes H|0\rangle = \frac{1}{\sqrt{2^L}} \sum_{i=0}^{2^L-1} |i\rangle. \tag{42}$$

The quantum discrete (linear) Fourier transform applied to a state $|k\rangle$ is:

$$\mathcal{F}(|k\rangle) = |\psi_k\rangle = \frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} \exp[\mathrm{i}2\pi ik/N]|i\rangle, \tag{43}$$

and this property holds:

$$\frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} \exp[\mathrm{i}2\pi ik/N] = \begin{cases} 1 & \text{if } k \text{ is a multiple of } N \\ 0 & \text{otherwise.} \end{cases} \tag{44}$$

The order to compute the quantum transform is $O(n)$, since there are n terms in the product, while the complexity in the classical transform is exponential, $O(n2^n)$ (Lavor *et al.*, 2003).

Classical computing works in two orthogonal spaces, 0 and 1, whereas quantum computing works in four orthogonal spaces, 1, –1, i and –i.

6.2. Shor's algorithm

Let us consider an example of quantum algorithm. Find the prime factors of a large integer number n, e.g. n = 694921 = 787×883. What is one possible practical application? RSA (Rivest–Shamir–Adleman) is a public-key cryptosystem used for secure data transmission. It is based on the practical difficulty of the factorisation of the product of two large prime numbers, the so-called "factoring problem". Factoring n can be done by choosing an integer m prime to n, and then finding the smallest positive integer P such that mod(m^P, n) = 1. Shor (1994) designed an algorithm to be run on a hypothetical quantum computer and solve this problem. On a classical (non-quantum) computer an algorithm requires an amount of time that is exponential with n, whereas a quantum computer would only require a polynomial up to power n. For example, to run the Shor algorithm to factorise a 2000-digit number, one million qubits are required. The evaluation time for this task with 1 MHz speed takes four days. The largest number that can be factorised on the most powerful supercomputers is a 768-bit number, and it takes two years (Fowler *et al.*, 2012).

Let n be a product of two prime numbers, $n = p \times q$, and let m be a natural number less than n which is not a factor of n and contains no common factors with n (e.g. m = 2, n = 15 = 3×5) [this means GCD(m, n) = 1, where GCD is the greatest common divisor]. Then, the sequence

$$\mathrm{mod}(m, n), \mathrm{mod}(m^2, n), \mathrm{mod}(m^3, n) \ldots \mathrm{mod}(m^r, n) \tag{45}$$

has as period P, the least natural number that satisfies

$$\mathrm{mod}(m^P, n) = 1. \tag{46}$$

If we find P, we can factor n, since P evenly divides $(p\text{-}1)\times(q\text{-}1)$, i.e. P is the largest even number that divides $(p\text{-}1)\times(q\text{-}1)$ (discovered by Leonhard Euler in 1760). Examples: $m = 2$, $n = 15$, the sequence is 2, 4, 8, 1, | 2, 4, 8, 1, . . ., and the period is $P = 4$; $m = 2$, $n = 21$, the sequence is 2, 4, 8, 16, 11, 1, | 2, 4, 8, 16, 11, 1,. . ., and $P = 6$; m =3, n = 91, the sequence is 3, 9, 27, 81, 61, 1 | 3, 9, 27, 81, 61, 1. . ., and $P = 6$.

In this case, quantum computing, to obtain the periodicity, has four steps. We follow Lomonaco (2002), Lavor *et al.* (2003), and Cooper (2006). We have to calculate mod(m^r, n) and extract the periodicity P. The quantum computer sets an initial state:

$$|\psi_0\rangle = |0\ldots0\rangle_{\text{input}}|0\ldots0\rangle_{\text{output}}, \tag{47}$$

where the input and output registers have both L qubits, respectively, which have been set to zero. Moreover, assume that L is chosen, such that P divides 2^L.

First, we apply the Hadamard gate in Eq. 40 to the input register:

$$|\psi_1\rangle = \frac{1}{2^{L/2}}(|0\rangle + |1\rangle)_0(|0\rangle + |1\rangle)_1\ldots(|0\rangle + |1\rangle)_L = \frac{1}{2^{L/2}}\sum_{i=0}^{2^L-1}|i\rangle|0\rangle. \tag{48}$$

Then, this register holds all the integers from 0 to $2^L - 1$.

If $|j\rangle$ and $|k\rangle$ are the states of the input and output registers, respectively, define the unitary linear operator $\mathcal{L}_m$ as:

$$\mathcal{L}_m\,(|j\rangle|k\rangle) = |j\rangle|k + m^j\rangle. \tag{49}$$

Now, we apply this operator to $|\psi_1\rangle$, to obtain:

$$|\psi_2\rangle = \frac{1}{2^{L/2}}\sum_{i=0}^{2^L-1}|i\rangle|m^i\rangle = \frac{1}{2^{L/2}}\sum_{i=0}^{2^L-1}|i\rangle|\text{mod}(m^i, n)\rangle, \tag{50}$$

since m^i is periodic with period P. Then, the input and quantum registers are entangled.

We then collect equal terms in $|\psi_2\rangle$. Because of the periodicity, we may substitute $aP + b$ for i, and obtain:

$$|\psi_2\rangle = \frac{1}{2^{L/2}}\sum_{b=0}^{P-1}\left(\sum_{a=0}^{\frac{2^L}{P}-1}|aP + b\rangle\right)|m^b\rangle. \tag{51}$$

Since mod(m^P, n) = 1, we have replaced m^b for m^{aP+b} in the second register. Let us measure the second register and assume that we obtain m^{b_0} . The result is:

$$|\psi_3\rangle = \sqrt{\frac{P}{2^L}}\left(\sum_{a=0}^{\frac{2^L}{P}-1}|aP + b_0\rangle\right)|m^{b_0}\rangle. \tag{52}$$

The new normalisation is due to the fact that there are $2^L/P$ terms in the sum. The states that contribute are $|b_0\rangle$, $|P + b_0\rangle$, . . ., $|2^L - P + b_0\rangle$.

Now, we find the period P using the quantum Fourier transform, since the transform of a periodic function with period P is a periodic function with period proportional to P^{-1}. The algorithm relies on the ability of a quantum computer to be in many states simultaneously (a superposition of states). Thus, to obtain the period, we evaluate the function at all points simultaneously. A measurement is applied to compute one possible value, destroying all the others. Because of this, we perform a Fourier transform to another state that returns the period with high probability.

We apply the quantum transform in Eq. 43 to the input register:

$$|\psi_4\rangle = \mathcal{F}(|\psi_3\rangle) = \sqrt{\frac{P}{2^L}} \sum_{a=0}^{\frac{2^L}{P}-1} \left(\frac{1}{\sqrt{2^L}} \sum_{i=0}^{2^L-1} \exp[\mathrm{i}2\pi i(aP+b_0)/2^L] \; |i\rangle \right) |m^{b_0}\rangle. \tag{53}$$

Alternatively,

$$|\psi_4\rangle = \frac{1}{\sqrt{P}} \left[\sum_{i=0}^{2^L-1} \left(\frac{1}{2^L/P} \sum_{a=0}^{\frac{2^L}{P}-1} \exp[\mathrm{i}2\pi ia/(2^L/P)] \right) \exp[\mathrm{i}2\pi ib_0/2^L] \; |i\rangle \right] |m^{b_0}\rangle. \tag{54}$$

Using Eq. 44, the expression in round parentheses is not zero if and only if $i = 2^L k/P$, with $k = 0, \ldots, P$ -1. Then,

$$|\psi_4\rangle = \frac{1}{\sqrt{P}} \left(\sum_{k=0}^{P-1} \exp[\mathrm{i}2\pi kb_0/P] \left| \frac{2^L k}{P} \right\rangle \right) |m^{b_0}\rangle. \tag{55}$$

Let $k_0 \in [0, P-1]$. A measurement gives $2^L p_0/P$. If k_0 is coprime to P, we just select the denominator. If k_0 and P have a common factor, P is not actually P but, say $D = Pc$, where c is the common factor. We run the algorithm iteratively till we find P . The process ends, because the iterations are less or equal to $\log_2 P$ (see Lavor *et al.*, 2003).

6.2.1. Example: comparison between classical and quantum computing

Let us consider the factoring of $n = 91 = 7\times13$. The steps in a classical computer are as follows:

1. Choose $m<n$. If we take $m = 7$, we have GCD(7, 91) = 7 f= 1 and we have solved the problem, since m and n/m are factors of n. If GCD(m, 91) = 1, we continue.
2. Take a new m. If the period is odd, choose another m since P does not evenly divide (p-1) (q-1) (see Euler's argument above). Take m =3, where the period is even P = 6 (see above). Since P is even,

$$\mathbf{mod}[(m^{P/2}-1)(m^{P/2}+1), n] = \mathbf{mod}(m^P - 1, n) = \mathbf{mod}(3^6 - 1, 91) = 0. \tag{56}$$

3. If

$$\mathbf{mod}(m^{P/2}+1, n) = 0, \tag{57}$$

choose another m till Eq. 57 does not hold; m = 3 is suitable since the period is 6 and mod(3^3+1, 91) = 28 ≠ 0.

4. Compute GCD($m^{P/2}$-1, n) = GCD(3^3-1, 91) = GCD(26, 91) = 13, which is one of the factors. The other factor is GCD($m^{P/2}$+1, n) = 7.

The steps in a quantum computer follow the mathematical development above. The procedure is to find the period of the function $f(r)$ = mod(m^r, n). Lavor *et al.* (2003) give an example with m = 2 and n = 21, where the period is P = 6. They choose the number of qubits in the first register, t, such that $n^2<2^t<2n^2$. The choice is 9 and 5 qubits for the input and output registers. Then, the procedure is:

1. apply the Hadamard gate 9 times to create a superposition of states and obtain $|\psi_1\rangle$;
2. apply $\mathcal{L}_m$ to get $|\psi_2\rangle$ and re-order terms;
3. measure the second register to obtain $|\psi_3\rangle$;
4. look for the period by performing the quantum Fourier transform on the first register of $|\psi_3\rangle$ to obtain $|\psi_4\rangle$. We obtain several peaks related to the probability of each state of

the first register. The result is where P = 6 and L = 9 (see Eq. 54). Fig. 11 shows this probability.

We then choose a given j≠0 and proceed (use of partial fractions yields the inverse period 1/P). Then, implement steps 2 to 4 of the previous classical computing procedure to obtain the factors p and q. The period is P = 6, an even number, therefore GCD($2^{6/2}$ ± 1, 21) gives the two factors, p = 3 and q = 7.

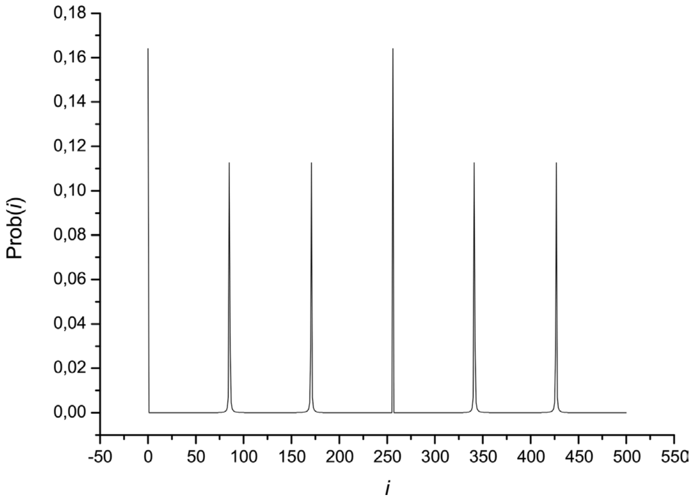

Fig. 11 - Probability of state |i⟩.

7. Geophysical applications

7.1. Seismic inversion

Applications of ML techniques in seismic technology are multiple. In hydrocarbon exploration it is often required to obtain the seismic velocity model. One example is the inversion of VSP (vertical seismic profile) data, where the source is located at the surface and the receiver at the borehole. A reverse VSP experiment (RVSP) requires a source at the borehole and receivers at the surface. These configurations can be used in ore mining with minor modifications. For instance, the drill bit in horizontal drilling (a RHSP configuration) is a source of seismic waves and receivers can be placed at the mine walls. The method is called seismic-while-drilling. RVSP and RHSP in mines can give useful information of the rock properties ahead of the bit. The techniques can be found, among others, in Poletto and Miranda (2004), Malehmir *et al.* (2012), and Zhou *et al.* (2015).

A 1D seismic velocity model is shown in Table 3. The low-velocity layer 4 may represent an overpressured formation. Let us assume that we know the location of the geological interfaces on the basis of nearby well-log information. The experiment has a source at the surface and receivers at a well. We have to find the velocity of the layers using simulated annealing. The cost function to minimise is:

$$C(c_1, c_2, c_3, c_4, c_5) = \sum_i^n \sum_j^m \left[\bar{s}(i,j) - s(i,j,c_l)\right]^2, \quad l = 1...., 5, \tag{58}$$

where c_l are the seismic velocities, $\bar{s}$ is the observed seismogram, s are the simulated seismograms,

n is the number of receivers and m is the number of samples of each seismic trace. The observed seismogram is shown in Fig. 12, where the direct and up-going waves can be seen.

The wave equation is recast in the velocity-stress formulation, with variable density and attenuation based on the Maxwell viscoelastic model. It has the form

$$\begin{aligned} \dot{v} &= (1/\rho)(\sigma_{,x} + e_\sigma), \\ \dot{\sigma} &= \rho c^2[v_{,x} - (1/\eta)\sigma - so + e_v], \\ \dot{e}_\sigma &= a\sigma_{,x} - be_\sigma, \\ \dot{e}_v &= av_{,x} - be_v, \end{aligned} \tag{59}$$

where v is the particle velocity, σ is the stress field, ρ is the mass density, c is the unrelaxed (∞-frequency) seismic velocity, η is the Maxwell viscosity, so is the source, and e_v and e_σ are auxiliary variables for the absorbing boundaries. A dot above a variable denotes time differentiation and the subscript "$,x$" indicates a spatial derivative. The density is given by $\rho = 1741(c/1000)^{1/4}$ (c in m/s, ρ in kg/m^3) and $\eta = Q\rho_0 c^2/(2\pi f_p)$ is uniform, where $Q = 150$ is the quality factor, ρ_0 and c_0 are the density and velocity of the upper layer, and f_p is the central frequency of the source, which has the time history: $h(t) = (V - 0.5)\exp(-a)$, $V = [\pi f_p(t - t_s)]^2$, $t_s = 1.4/f_p$, with $f_p = 5$ Hz.

The seismogram has been generated with a finite-difference o(2,4) algorithm, i.e. 2nd-order (staggered) in time and a 4th-order staggered operator to compute the spatial derivatives [in chapter 9 of Carcione (2014)]. The mesh has 500 grid points, the source is located at grid point

Table 3 - Velocity model.

Layer	Thickness (km)	Velocity (km/s)
1	0.5	2
2	1	2.5
3	1	2.8
4	0.5	2.1
5	∞	3.4

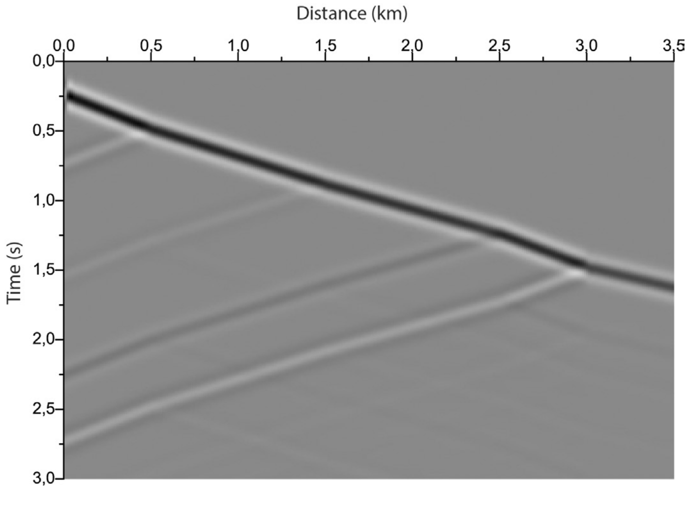

Fig. 12 - VSP seismogram.

100 (the surface), the grid spacing is dx = 10 m, and the time step is dt = 1 ms. The C-PML absorbing method is used (Martin and Komatitsch, 2009), where $a = 3c_0 \ln(0.001)(i-1)^2 dx^2/(2L^3)$, $i = 1, n_a$, $b = [\pi f_p(i-n_a)/(1-n_a)] - a$, where c_0 is the velocity at the strips of length $n_a = 30$, and $L = (n_a - 1)dx$. Outside the strips, $a = b = 0$ and $e_v = e_\sigma = 0$.

Simulated annealing is run with the following input parameters: initial temperature: $T_0 = 5$, number of parameters: $N = 5$ (the five velocities); starting values of $x = (2, 2, 2, 2, 2)$ km/s; MAX = .FALSE.; $R_T = 0.5$; $\varepsilon = 10^{-3}$; $N_S = 20$; $N_T = 5\ N = 25$; number of final functions use to decide upon termination: NEPS = 4; the maximum number of function evaluations: MAXEVL = 10^5; the lower bound for the allowable solution variables: LB (i) = 1.8 km/s, for all N elements; the upper bound for the allowable solution variables: UB (i) = 3.9 km/s, for all N elements; vector that controls the step length adjustment: $C(i) = 2$ for all N elements; the first seed for the random number generator: ISEED1 = 1; the second seed for the random number generator: ISEED2 = 2.

To reach a cost function $C = 0.48$, the algorithm uses 5001 function evaluations (of which 2570 accepted ones) and the final temperature is $T = 1.25$. Fig. 13 compares the final inversion results (blue dots) with the true velocity profile (solid line). The red dots correspond to an intermediate result with a cost function C=11467, 2501 function evaluations and a final temperature $T = 2.5$. Other examples can be found in Sen and Stoffa (1991), Ma (2002), and Pei *et al*. (2009).

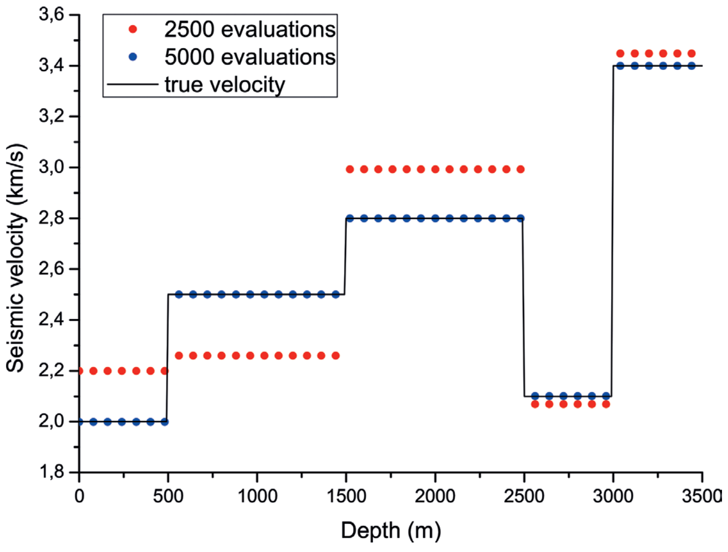

Fig. 13 - Seismic velocity inversion with simulated annealing. The blue dots represent the final result.

7.2. Petrophysical log prediction

We predict reservoir permeability from well logs data by using a neural network. Usually, the technique is applied to log data to match the available core permeability. The choice of the input variables is very important. For instance, the gamma ray log provides evidence of clay, which has a big influence on permeability. The bulk density, sonic, and neutron logs are inverse functions of porosity and shale content. Therefore, they contribute to the permeability.

We consider a synthetic example. The benefits of using synthetic data is that it allows us to vary the training set as we wish and determine the number of neurons in each layer and the number of hidden layers, before processing real data. We choose the density, sonic, gamma ray, and neutron-porosity logs as the 4 input neurons.

The model assumes a fully brine saturated formation from 2 to 3 km depth. We define porosity and clay content variations versus depth, $2 \leq z(\text{km}) \leq 3$, as:

$$\begin{aligned} \phi &= \frac{1}{400}[60 - (10 - y - 0.5y^2 + 0.1y^3)], \quad y = 10 + 20(z-3), \\ C &= \frac{1}{10}[5 + (y^5 - 8y^3 + 10y + 6)\sin(15y)], \quad y = z - 3, \end{aligned} \tag{60}$$

respectively.

The density is:

$$\rho = (1-\phi)[(1-C)\rho_s + C\rho_c] + \phi\rho_f, \tag{61}$$

where ρ_s= 2650 kg/m^3 and ρ_c = 2600 kg/m^3 are the densities of the sand and clay particles, and ρ_f = 1040 kg/m^3 is the brine density.

The sonic transit time is:

$$\Delta t = (1-\phi)[(1-C)\Delta t_s + C\Delta t_c] + \phi\Delta t_f, \tag{62}$$

where Δt_s = 58 μs/ft, Δt_f = 207 μs/ft and Δt_c = 82 μs/ft are the sand, fluid and clay transit times (actually slownesses), which correspond to the P-wave velocities 5212 m/s, 1471 m/s and 3721 m/s, respectively (typical values to process sonic logs). Eq. 62 is based on the time-average equation (Wyllie's equation). The velocities of the solids correspond to a Poisson medium (the Lamé constants are equal), i.e. the P-wave velocity is $3\sqrt{K/(5\rho)}$, where K is the bulk modulus.

The neutron porosity is given by

$$\phi_N = \phi H_b + (1-\phi)[CH_c + (1-C)H_s], \tag{63}$$

where H_b = 1.2, H_c = 0.05 and H_s = 0.001 are the hydrogen indices of brine, clay, and sand, respectively (Schlumberger, 1991).

The gamma-ray log γ is produced from the clay model by using the empirical equation of Stieber (1970):

$$C = \frac{\text{IGR}}{3 - 2\ \text{IGR}}, \quad \text{IGR} = \frac{\gamma - \gamma_{\text{min}}}{\gamma_{\text{max}} - \gamma_{\text{min}}}, \tag{64}$$

where γ_{min} = 20 API is the reading for pure sand and γ_{max} = 140 API is the reading for pure clay. We obtain:

$$\gamma = \gamma_{\text{min}} + \text{IGR}\ (\gamma_{\text{max}} - \gamma_{\text{min}}), \quad \text{IGR} = \frac{3C}{1+2C}. \tag{65}$$

Since clay has a big influence, we use the permeability model developed by Carcione *et al.* (2000). The model assumes that the rock is composed of sand and clay grains of total porosity ϕ and clay content C and has a permeability given by

$$\kappa = \left\{ \frac{(1-\phi)^2}{a\phi^3} \left[(1-C)^2 + C^2 b^2\right] \right\}^{-1}, \tag{66}$$

where $a = r_s^2/45$ and $b = r_s/r_c$, where r_s and r_c are the radii of the sand and clay particles, respectively. We consider r_s = 50 μm and r_c = 1 μm. The logs are shown in Fig. 14, where the permeability is $\varkappa = 10^{-p}$ m^2.

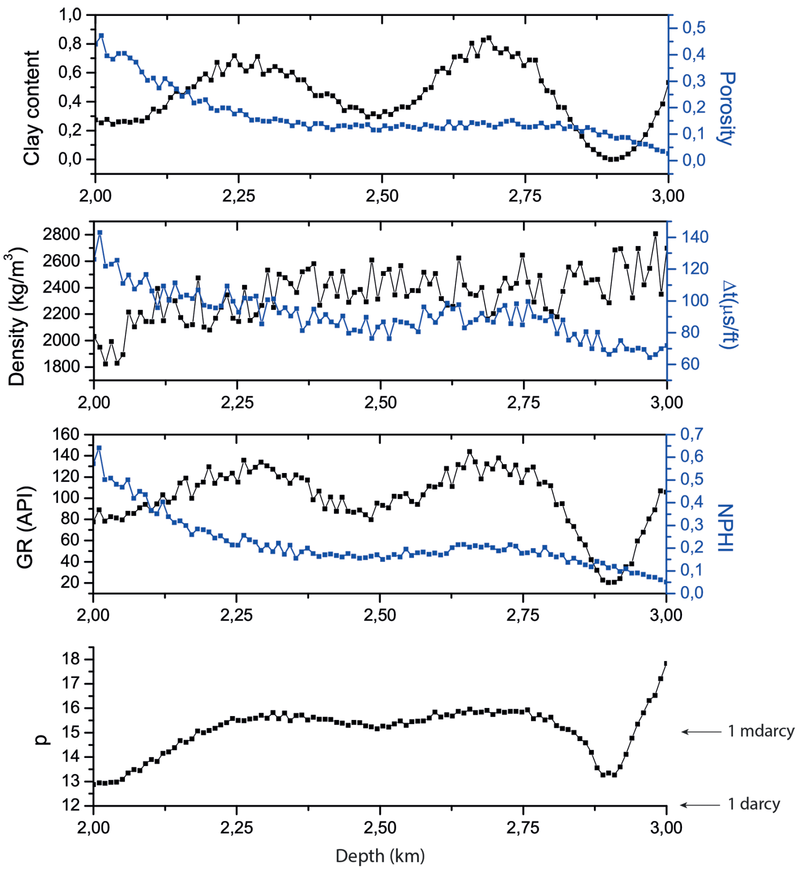

Fig. 14 - Synthetic well logs, with 10% noise added, used for permeability prediction. Permeability is $\varkappa = 10^{-p}$ m^2.

The input neurons correspond to the density, sonic, gamma ray, and neutron- porosity logs, and there is one hidden layer with 10 neurons. First, the network is trained on 50% of the data, i.e. between 2 and 2.5 km, and tested on the remaining 50%.

7.2.1. Training

The training phase involves the calculation of the weights, w_{in}, $i = 1, 10$; $n = 1, 4$ (40 values) and w_i (10 values). We consider the depth from 2 to 2.5 km ($M = 150$ samples of each log). The density is scaled as kg/m^3/10, the transit time Δt as μs/ft/100, and the gamma ray as API/100. The experimental permeability index p is re-scaled to [0,1] as

$$\bar{b}_1 = \frac{p - p_{\min}}{p_{\max} - p_{\min}}, \quad p = p_{\min} + (p_{\max} - p_{\min})\bar{b}_1, \tag{67}$$

where $p_{\min}$ and $p_{\max}$ are the minimum and maximum values.

The cost function involves the data at <u>all</u> depths:

$$C = \frac{1}{2M} \sum_{m=1}^{M} (b_{1m} - \bar{b}_{1m})^2, \tag{68}$$

where b_{1m} is the computed value and $\bar{b}_{1m}$ is the true value.

For each depth level m, the forward propagation computes

$$b_m = \sum_{i=1}^{10} f(a_{im}) w_i, \tag{69}$$

where a_{im} are the outputs of the hidden neurons, and applies again the activation function, to obtain

$$b_{1m} = f(b_m) = f\left[\sum_{i=1}^{10} f(a_{im}) w_i\right]. \tag{70}$$

The back propagation requires to update the weights as

$$w_i \to w_i - \alpha \frac{\partial C}{\partial w_i}, \tag{71}$$

which after some calculations becomes

$$w_i \to w_i + \frac{\alpha}{M} \sum_{m=1}^{M} (b_{1m} - \bar{b}_{1m}) f'(b_m) f(a_{im}), \tag{72}$$

where $f'(x) = \exp(-x)/[1 + \exp(-x)]^2$.

Regarding the initial weights we note that

$$b_{1m} = f(b_m) = f\left[\sum_{i=1}^{10} f(a_{im}) w_i\right] = f\left[\sum_{i=1}^{10} f\left(\sum_{n=1}^{4} A_{nm} w_{ni}\right) w_i\right], \tag{73}$$

where A_{nm} is the input data corresponding to well log n and depth m. Let us recall here that 4 is the number of logs, or input neurons at each depth level, and 10 is the number of hidden neurons. We require:

$$w_{ni} \to w_{ni} - \alpha \frac{\partial C}{\partial w_{ni}}, \tag{74}$$

or

$$w_{ni} \to w_{ni} + \frac{\alpha}{M} \sum_{m=1}^{M} (b_{1m} - \bar{b}_{1m}) f'(b_m) f'(a_{im}) w_i A_{nm}. \tag{75}$$

With the new weights (listed in Table 4), we start the forward propagation again until we

Table 4 - Optimal weights for the permeability prediction.

i	w_{1i}	w_{2i}	w_{3i}	w_{4i}	w_i
1	–0.63	–0.23	0.54	–4.67	0.63
2	–3.63	–0.013	–4.87	8.74	–4.61
3	1.97	3.06	6.12	–3.15	19.22
4	0.12	–0.57	1.78	–7.42	1.92
5	29.11	177	172	38.47	26.82
6	255	1248	1187	413	–50.09
7	–2.87	–19.03	–16.95	–9.47	24.95
8	106	708	635	175	88.05
9	0.25	–3.53	–5.08	–3.75	4.77
10	174	894	857	314	–76.03

obtain the optimal values. Fig. 15a shows the comparison between the experimental and computed permeability values (index p), from 2 to 2.5 km. We have used $\alpha = 12$ and 61000 iterations.

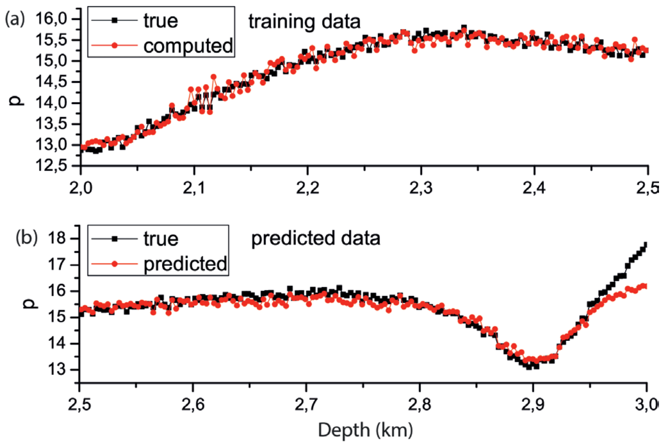

Fig. 15 - Training (a) and predicted (b) permeability data. Permeability is $\varkappa = 10^{-p}$ m^2. Training data from 2 km to 2.5 km.

7.2.2. Prediction

Having obtained the weights w_{ni} and w_i, we predict the permeability index at depth m as

$$b_{1m} = f(b_m) = f\left[\sum_{i=1}^{10} f(a_{im})w_i\right], \tag{76}$$

where

$$a_{im} = \sum_{n=1}^{4} A_{nm}w_{ni}. \tag{77}$$

Thus, the permeability is:

$$\kappa_m = 10^{-p_m}[\mathrm{m}^2], \quad p_m = p_{\min} + (p_{\max} - p_{\min})b_{1m}, \tag{78}$$

or

$$\kappa_m = 10^{12-p_m}[\mathrm{darcy}]. \tag{79}$$

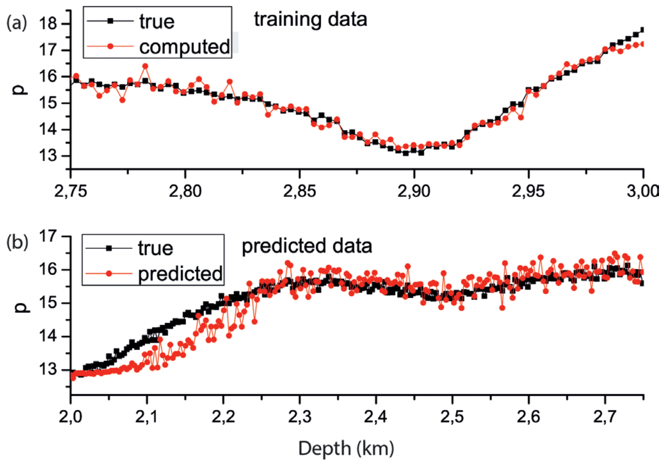

Fig. 16 - Training (a) and predicted (b) permeability data. Permeability is $K = 10^{-p}$ m^2. Training data from 2.75 km to 3 km.

Fig. 15b shows the comparison between the experimental and predicted permeability values (index p), from 2.5 to 3 km. As can be appreciated, the prediction is satisfactory. The relatively lack of prediction for $p \in [15, 18]$ is due to the fact that there is no training data in this range. If we choose the training data between 2.75 and 3 km (a shorter training interval), the results are shown in Fig. 16, where there is some misprediction between 2 and 2.2 km (higher permeabilities) (we have used $\alpha = 6$ and 9000 iterations).

8. Conclusions

ML is an emerging set of algorithms designed to learn from data and discover features and relationships hidden in large data sets. We present the basic fundamentals of some types of ML techniques, models and algorithms. The approaches, based on smart data analysis, can help us to solve a variety of problems. We provide a detailed theoretical account of the mathematical derivations necessary to solve practical problems, so that the reader can program their own codes. Each technique is illustrated with simple examples. Finally, we present two geophysical applications, namely seismic inversion and petrophysical prediction.

REFERENCES

Aleardi M.; 2015: *Seismic velocity estimation from well log data with genetic algorithms in comparison to neural networks and multilinear approaches*. J. Appl. Geophys., **117**, 13-22.

Ali Ahmadi M. and Chen Z.; 2018: *Comparison of machine learning methods for estimating permeability and porosity of oil reservoirs via petro-physical logs*. Petroleum, in press.

Alpaydin E.; 2014: *Introduction to machine learning, 3rd ed*. The MIT Press, Cambridge, MA, USA, 613 pp.

Araya-Polo M., Dahlke T., Frogner C., Zhang C., Poggio T. and Hohl D.; 2017: *Automated fault detection without seismic processing*. The Leading Edge, **36**, 208-214.

Carcione J.M.; 2014: *Wave fields in real media: wave propagation in anisotropic, anelastic, porous and electromagnetic media, 3rd ed*. Elsevier Sciences, Oxford, UK, vol. 38, 690 pp.

Carcione J.M., Gurevich B. and Cavallini F.; 2000: *A generalized Biot-Gassmann model for the acoustic properties of shaley sandstones*. Geophys. Prospect., **48**, 539-557.

Cooper J.W.; 2006: A re-evaluation of Shor's algorithm, <arxiv.org/abs/quant-ph/0612077>.

Corana A., Marchesi M., Martini C. and Ridella S.; 1987: *Minimizing multi-modal functions of continuous variables with the simulated annealing algorithm*. ACM Trans. Math. Soft., **13**, 262-280.

Feynman R.P.; 1982: *Simulating physics with computers*. Int. J. Theor. Phys., **21**, 467-488.

Foster F.; 2001: *Review: Discipulus: a commercial genetic programming system*. Genet. Program. Evol. Mach., **2**, 201-203.

Fowler A.G., Mariantoni M., Martinis J.M. and Cleland A.N.; 2012: *Surface codes: towards practical large-scale quantum computation*. Phys. Rev. A, **86**, 032324.

Goffe W.L., Ferrier G.D. and Rogers J.; 1994: *Global optimization of statistical functions with simulated annealing*. J. Econom., **60**, 65-99.

Hamada G.M. and Elshafel M.A.; 2010: *Neural network prediction of porosity and permeability of heterogeneous gas sand reservoirs using NMR and conventional logs*. NAFTA, **61**, 451-460.

Han J., Kamber M. and Pei J.; 2011: *Data mining: concepts and techniques, 3rd ed*. Morgan Kaufmann Publishers, Burlington, MA, USA, 744 pp.

Haykin S.; 2009: *Neural networks and learning machines, 3rd ed*. Prentice Hall, Upper Saddle River, NJ, USA, vol. 10, 906 pp.

Helle H.B., Bhatt A. and Ursin B.; 2001: *Porosity and permeability prediction from wireline logs using artificial neural networks*. Geophys. Prospect., **49**, 431-444.

Helmy T., Fatai A. and Faisal K.; 2010: *Hybrid computational models for the characterization of oil and gas reservoirs*. Expert Syst. Appl., **37**, 5353-5363.

Hermawanto D.; 2013: *Genetic algorithm for solving simple mathematical equality problem,* <arxiv.org/abs/1308.4675>.

Huang Z., Shimeld J., Williamson M. and Katsube J.; 1996: *Permeability prediction with artificial neural network modelling in the Venture gas field, offshore eastern Canada*. Geophys., **61**, 422-436.

Jia Y. and Ma J.; 2017: *What can machine learning do for seismic data processing? An interpolation application*. Geophys., **82**, V163-V177.

Kröse B. and van der Smagt P.; 1996: *An introduction to neural networks, 8th ed*. University of Amsterdam, Amsterdam, The Netherlands, 135 pp.

Lavor C., Manssur L.R.U. and Portugal R.; 2003: *Shor's algorithm for factoring large integers,* <arxiv.org/abs/quant-ph/0303175>.

Levine D.; 1996: *Users guide to the PGAPack parallel genetic algorithm library*. U.S. Department of Energy, USA, 79 pp., doi: 10.2172/366458.

Lomonaco S.J. Jr.; 2002: *Shor's quantum factoring algorithm*. In: Proc. Symposia in Applied Mathematics, American Mathematical Society (quant-ph/0010034), vol. 58, 19 pp.

Ma X.-Q.; 2002: *Simultaneous inversion of prestack seismic data for rock properties using simulated annealing*. Geophys., **67**, 1877-1885.

Malehmir A., Durrheim R., Bellefleur G., Urosevic M., Juhlin C., White D.J., Milkereit B. and Campbell G.; 2012: *Seismic methods in mineral exploration and mine planning: a general overview of past and present case histories and a look into the future*. Geophys., **77**, WC173-WC190.

Martin R. and Komatitsch D.; 2009: *An unsplit convolutional perfectly matched layer technique improved at grazing incidence for the viscoelastic wave equation*. Geophys. J. Int., **179**, 333-344.

Moradi S., Trad D. and Innanen K.A.; 2018: *Quantum computing in geophysics: algorithms, computational costs, and future applications*. In: Expanded Abstracts, SEG Technical Program, Anaheim CA, USA, 5 pp.

Nikravesh M., Zadeh L.A. and Aminzadeh F. (eds); 2003: *Soft computing and intelligent data analysis in oil exploration, 1st ed*. Elsevier Science, Oxford, UK, vol. 51, 754 pp.

Pei D., Quirein J.A., Cornish B.E., Quinn D. and Warpinski N.R.; 2009: *Velocity calibration for microseismic monitoring: a very fast simulated annealing (VFSA) approach for joint-objective optimization*. Geophys., **74**, WCB47-WCB55.

Poletto F. and Miranda F.; 2004: *Seismic while drilling: fundamentals of drill-bit seismic for exploration, 1st ed*. Elsevier Science, Oxford, UK, vol. 35, 546 pp.

Poulton M.M. (ed); 2001: *Computational neural networks for geophysical data processing*. Elsevier Science, Oxford, UK, vol. 30, 352 pp.

Samuel A.; 1959: *Some studies in machine learning using the game of checkers*. IBM J. Res. Dev., **3**, 210-229.

Schlumberger; 1991: *Log interpretation. Principles and applications*. Schlumberger Educational Services, Houston, TX, USA.

Schmidhuber J.; 2015: *Deep learning in neural networks: an overview*. Neural Networks, **61**, 85-117.

Sen M.K. and Stoffa P.L.; 1991: *Nonlinear one-dimensional seismic waveform inversion using simulated annealing*. Geophys., **56**, 1624-1638.

Shalev-Shwartz S. and Ben-David S.; 2014: *Understanding machine learning: from theory to algorithms*. Cambridge University Press, Cambridge, UK, 410 pp., doi: 10.1017/CBO9781107298019.

Shor P.W.; 1994: *Algorithms for quantum computation: discrete logarithms and factoring*. In: Proc. 35th Annual Symposium on the Foundations of Computer Science, Goldwasser S. (ed), IEEE Computer Society, Los Alamitos, CA, USA, vol. 35, 124-134.

Smola A. and Vishwanathan S.V.N.; 2008: *Introduction to machine learning*. Cambridge University Press, Cambridge, UK, 234 pp.

Stieber S.J.; 1970: *Pulsed neutron capture log evaluation - Louisiana Gulf Coast*. In: Proc. Annual Fall Meeting of the Society of Petroleum Engineers of AIME, Houston, TX, USA, SPE-2961-MS, 7 pp.

Suzuki K. (ed); 2013: *Artificial neural networks - Architectures and applications*. InTech, Rijeka, Croatia, 264 pp., doi: 10.5772/3409.

Zhou B., Mason I., Greenhalgh S. and Subramaniyan S.; 2015: *Seeing coal-seam top ahead of the drill bit through seismic-while-drilling*. Geophys. Prospect., **63**, 166-182.

Corresponding author: José M. Carcione
Istituto Nazionale di Oceanografia e di Geofisica Sperimentale (OGS)
Borgo Grotta Gigante 42c, 34010 Sgonico (TS), Italy
Phone: +39 040 2140-345; e-mail: jcarcione@inogs.it