View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by IntechOpen

We are IntechOpen, the world's leading publisher of Open Access books

Built by scientists, for scientists

4,800
Open access books available

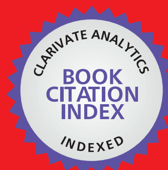
122,000
International authors and editors

135M
Downloads

154
Countries delivered to

Our authors are among the
TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities



WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

Chapter 10

Simulated Annealing Evolution

Sergio Ledesma, Jose Ruiz and Guadalupe Garcia

Additional information is available at the end of the chapter

http://dx.doi.org/10.5772/50176

1. Introduction

Artificial intelligence (AI) is a branch of computer science that seeks to create intelligence. While humans have been using computers to simplify several tasks, AI provides new options to use computers. For instance, voice recognition software uses AI to transform the sounds to the equivalent text words. There are several techniques that AI includes. An artificial neural network (ANN) is one of these techniques.

Humans use their intelligence to solve complex problems and perform daily tasks. Human intelligence is provided by the brain. Small processing units called neurons are the main components of the human brain. ANNs try to imitate partially some of the human brain behavior. Thus, artificial neurons are designed to mimic the activities of biological neurons.

Humans learn by experience: they are exposed to events that encourage their brains to acquire knowledge. Similarly, ANNs extract information from a data set; this set is typically called the training set and is organized in the same way that schools design their courses' content. ANNs provide an excellent way to understand better biological neurons. In practice, some problems may be described by a data set. For instance, an ANN is typically trained using a data set. For some problems, building a data set may be very difficult or sometimes impossible as the data set has to capture all possible cases of the experiment.

Simulated annealing (SA) is a method that can be used to solve an ample set of optimization problems. SA is a very robust technique as it is not deceived with local minima. Additionally, a mathematical model is not required to apply SA to solve most optimization problems.

This chapter explores the use of SA to train an ANN without the requirement of a data set. The chapter ends with a computer simulation where an ANN is used to drive a car. Figure 1 shows the system architecture. SA is used to provide a new set of weights to the ANN. The ANN controls the acceleration and rotation speed of the car. The car provides feedback by sending vision information to the ANN. The distance traveled along the road from the *Start* is used by the method of SA. At the beginning of the simulation the ANN does not know how to drive the car. As the experiment continues, SA is used to train the ANN. Each time the

INTECH
open science | open minds

©2012 Ledesma et al., licensee InTech. This is an open access chapter distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

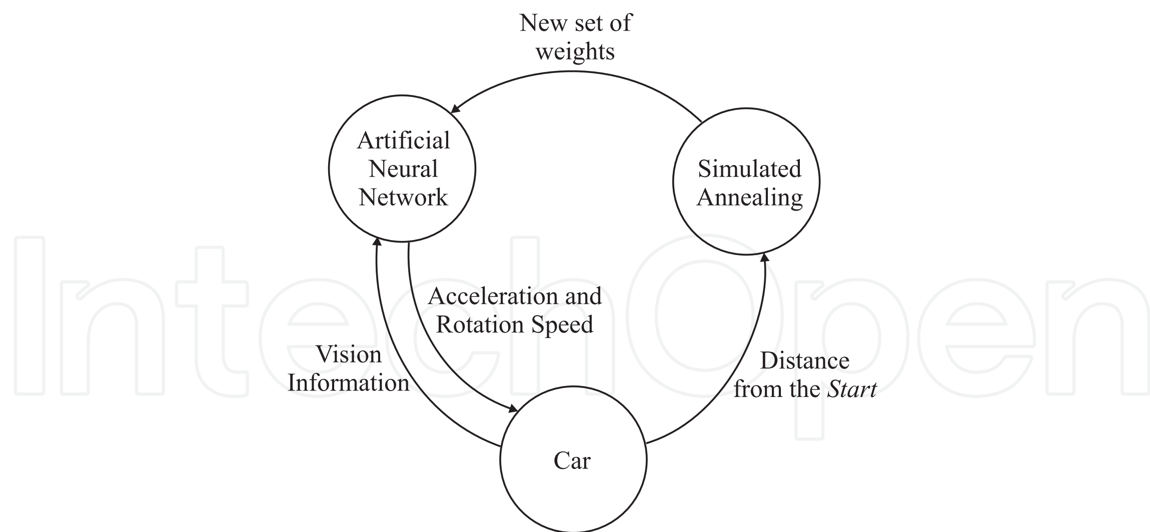


Figure 1. System architecture.

temperature decreases, the ANN improves its driving skills. By the end of the experiment, when the temperature has reached its final value, the ANN and the car have evolved to the point that they can easily navigate a maze.

2. Artificial neural networks

An ANN is a computational method inspired in biological processes to solve problems that are very difficult for computers or humans. One of the key features of ANNs is that they can adapt to a broad range of situations. They are typically used where a mathematical equation or model is missing, see [4]. The purpose of an ANN is to extract, map, classify or identify some sort of information that is allegedly hidden in the input, [13].

2.1. Neuron

The human brain is composed of processing units called neurons. Each neuron is connected to other neurons to form a neural network. Similarly, the basic components of an ANN are the neurons. Neurons are arranged in layers inside the ANN. Each layer has a fixed number of neurons, see [5]. For instance, the ANN shown in the Figure 2 has three layers: the input layer, the hidden layer, and the output layer.

An ANN accepts any kind of input that can be expressed as a set of numeric values; typical inputs may include: an image, a sound, a temperature value, etc. The output of an ANN is always dependent upon its input. That is, a specific input will produce a specific output. When a set of numeric values is applied to the input of an ANN, the information flows from one neuron to another until the output layer generates a set of values.

2.2. Activation function

The internal structure of an artificial neuron is shown in Figure 3(a). The output value z is given by:

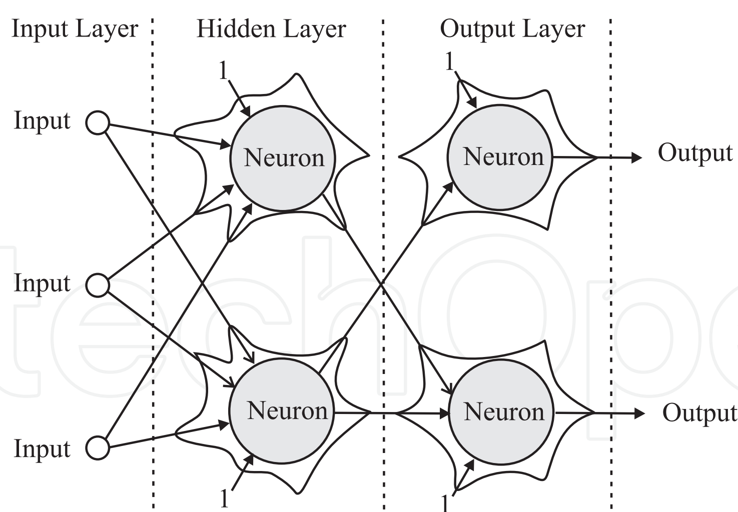


Figure 2. An artificial neural network.

$$z = f(y) = f(w_b + \sum_i x_i w_i), \tag{1}$$

where each w_i is called weight. A fixed input, known as *Bias*, is applied to the neuron, its value is always 1 and w_b is the respective weight for this input. The neuron includes also an activation function denoted by $f(y)$ in Figure 3(a). Without the *Bias*, the output of the network would be $f(0)$ when all inputs are zero. One common activation function used in multilayer ANNs is:

$$f(y) = logsig(y) = \frac{1}{1 + e^{-y}}. \tag{2}$$

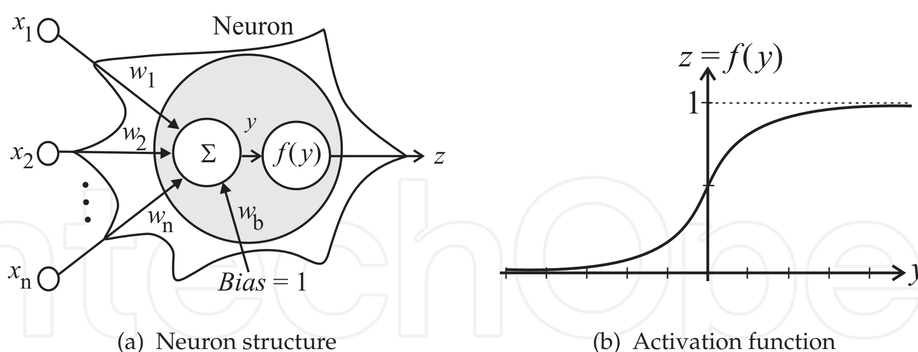
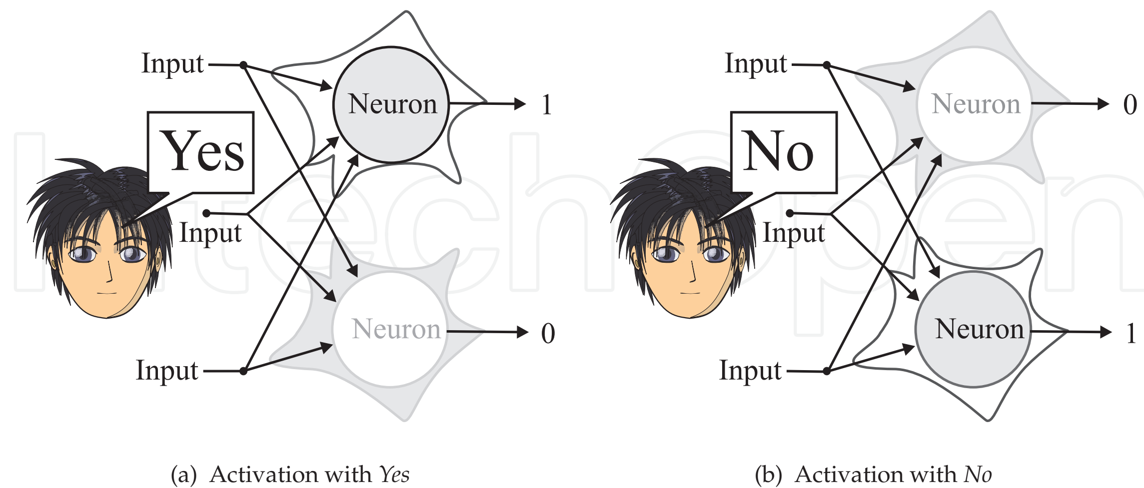


Figure 3. Artificial neuron.

The activation function of Equation 2 is plotted in Figure 3(b). The activation function (in this figure) is real, continuous, limited and has a positive derivative.

A neuron can be active or inactive, when the neuron is active its output is 1, when it is inactive its output value is 0. Some input values activate some neurons, while other values may activate other neurons. For instance, in Figure 4, the sound of the word *Yes* will activate the first output neuron, while the sound of the word *No* will activate the second neuron at the

output of the network. The structure of the ANN shown in this figure is very simple with only two neurons.



(a) Activation with *Yes* (b) Activation with *No*

Figure 4. ANN activation.

3. Learning

Before an ANN can be used for any practical purpose, it must be trained. An ANN learns during its training. For the duration of the learning process the ANN weights are recurrently adjusted.

3.1. Structured learning

In some instances, ANNs may learn from a data set. This set is typically known as the training set, and it is used on a new ANN (as its name indicates) for training. The training set has two parts: the input and the target. The input contains the set of inputs that must be applied to the network. The target includes the set of desired values at the output of the ANN. In other words, each sample (case) in the training set completely specifies all inputs, as well as the outputs that are desired when those inputs are presented, see [7]. During the training, each case in the training set is presented to the network, and the output of the network is compared with the desired output. After all cases in the training set have been processed, an epoch or iteration has completed by updating the weights of the network. There are several methods for updating the weights at each epoch or iteration. All these methods update the weights in such a way that the error (measured between the actual output of the network and the desired output) is reduced at each epoch, see [7].

Some training methods are based on the gradient of the error (they are called gradient based methods). These methods quickly converge to the closest local minima. The most typical gradient based methods to train an ANN are: the variable metric method, the conjugate gradient method, and method of Levenberg-Marquardt. To make the training of an ANN robust, it is always recommended to combine gradient based methods with other optimization methods such as SA.

When an ANN is trained using a data set, typically the set includes many training cases, and the training is done at once. This kind of training is called structured learning because the knowledge is organized in the data set for the network to learn. One of the main disadvantages of structured learning is that the training set has to be prepared to describe the problem at hand. Another disadvantage of structure learning is that if more cases are added to the training set, the ANN has again to be trained starting from scratch. As ANN training is time consuming, structured learning may be inadequate for problems that require continuous adaptation.

3.2. Continuous learning

In continuous learning, an ANN does not require a data set for training, the ANN learns by experience and is able to adapt progressively by incorporating knowledge gradually. Because some problems cannot be appropriately described by a data set, and because training using a data set can be time consuming, continuous learning is important for real-time computing (RTC) where there is a "real-time constraint".

3.3. Validation

After the ANN training has been completed, the network performance has to be validated. When using ANNs, the validation process is extremely important, as a matter of fact, the validation is as important as the training, see [7]. The purpose of the validation is to predict how well the ANN will behave in other conditions in the future. The validation process may be performed using a data set called the validation set. The validation set is similar to the training set but not equal. Under normal circumstances (when the ANN is properly used), the error obtained during training and during validation should be similar.

4. Simulated annealing

SA is an optimization method that can be used to solve a broad range of problems, [11]. SA is recommended for complex optimization problems. The algorithm begins at a specific temperature; as time passes the temperature gradually decreases following a cooling schedule as shown in Figure 5. The solution is typically described by a set of variables, but it can be described by other means. Once the algorithm has started, the solution approaches progressively the global minimum that presumably exists in a complex error surface, see [16] and [15]. Because of its great robustness, SA has been used in many fields including the training of ANNs with structured learning, [9].

One of the key features of SA is that it always provides a solution, even though the solution may not be optimal. For some optimization problems that cannot be easily modeled, SA may provide a practical option to solve them.

5. Simulated annealing evolution

Simulated annealing evolution includes the use of: ANNs, continuous learning and SA. In simulated annealing evolution, an ANN does not require a training set; instead the ANN

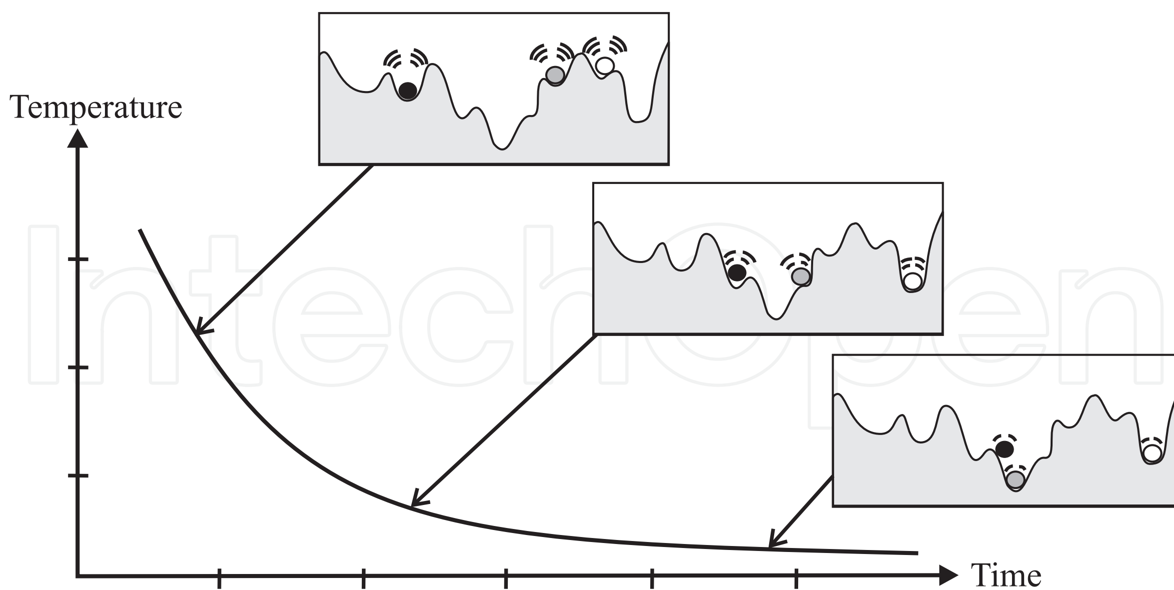


Figure 5. Cooling: three ANNs learning.

gradually learns new skills or improves existing ones by experience. Figure 5 shows how SA evolution works. In this figure, a typical cooling schedule used in SA is displayed. Suppose that there is a 2D landscape with valleys and hills as shown in this figure. Suppose also that it is desired to find the deepest valley on this landscape. Each of the balls, in this figure, represents an ANN. At the beginning of the simulation, the high initial temperature produces a state of high energy; the balls shake powerfully and are able to traverse easily through the high hills of the 2D terrain. In other words, each ANN is exploring, that is, the ANN is in the initial step of learning. As the temperature decreases, the energy of the balls decreases, and the movement of the balls is more restricted than at high temperatures, see [1]. Thus, as the temperature diminishes, the ANN has less freedom to explore new information as the network has to integrate new knowledge with the previous one. By the end of the cooling schedule, it is desired that one of the balls reached the deepest valley in the landscape, in other words, that one ANN learned a set of skills.

At each temperature, an ANN (in the set) has the chance to use its knowledge to perform a specific task. As the temperature decreases, each ANN has the chance to improve its skills. If the ANNs are required to incorporate new skills, temperature cycling can be used, see [6]. Specifically, an ANN may learn by a combination of SA and some sort of hands-on experience. Thus, simulated annealing evolution is the training of an ANN using SA without a training set.

Each ANN may be represented by a set of coefficients or weights. For illustrative purposes only, suppose that an ANN may be described by a set of two weights w_{11} and w_{12}. In Figure 6, there are three individuals, each is represented by a small circle with its two weights: w_{11} and w_{12}. At every temperature, each ANN is able to explore and use its abilities. At high temperatures, each ANN has limited skills and most of the ANNs will perform poorly at the required tasks. The gray big shadow circle in the figure indicates how the ANNs are considering different set of values w_{11} and w_{12} for testing and for learning. As temperature

decreases, the ANNs get closer to each other, illustrating the fact that most of them have learned a similar set of skills. By the time the temperature has reached its final value, the skills of each ANN will be helpful to the degree that the network is able to perform the task at hand. At this moment, the simulation may end or the temperature may increase, if there are new skills that need to be incorporated.

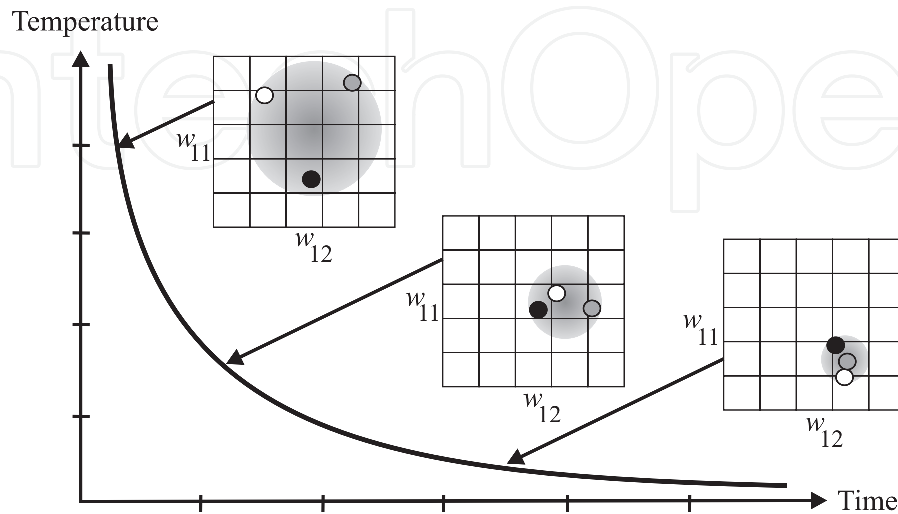


Figure 6. Each ANN tries to learn the same set of skills.

6. Problem description

To illustrate how to use simulated annealing evolution, this section presents a simple learning problem. The problem consists of using an ANN to drive a car in a simple road. Clearly, the problem has two objects: the road and the car. The road includes a *Start* point and a *Finish* point as shown in Figure 7. The road includes several straight lines and turning points. The car is initially placed at the *Start*. The driving is performed by an ANN that was integrated with the car. Specifically, the ANN manipulates indirectly several parameters in the car such as position, speed and direction. The purpose of the simulation is to train the ANN without a training set. At the beginning of the simulation, the ANN does not know how to drive; as the evolution continues the ANN learns and improves its skills being its goal to drive the car quickly from the *Start* to the *Finish* without hitting the bounds of the road (that is, the car must always stay inside the road). To solve this problem simulated annealing evolution was used to train the ANN.

The simulation was performed using object oriented programming (OOP); the respective UML diagrams for the simulation are shown in Figures 8 and 9. The two basis classes are shown in the diagram of Figure 8. This diagram includes two classes: Point and Bounds. The Point class in the diagram represents a point in a 2-Dimensional space, the diagram indicates that this structure includes only two floating point variables: x and y. The Bounds class, in the same UML diagram, is used to describe the bounds of an object when the object is at different positions and rotations angles. The main purpose of the Bounds class is to detect collisions when one object moves around other objects.

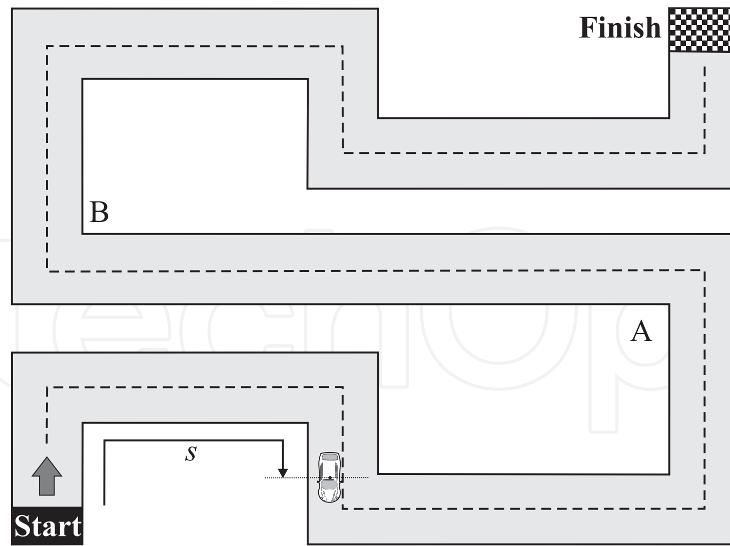


Figure 7. The car and the road.

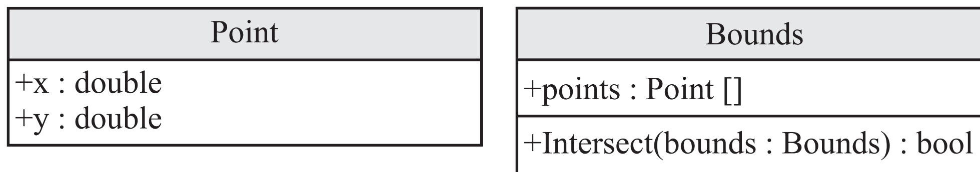


Figure 8. UML diagram showing the basic classes.

6.1. The object class

Figure 9 shows the respective UML diagram for the Object class. This class represents a static object in a 2-Dimensional space. The class name is displayed in italics indicating that this class is abstract. As it can be seen the *Render()* method is displayed in italics, and hence, it is a virtual method and must be implemented by the non-abstract derived classes. If the experiment includes some sort of visualization, the *Render()* method may be used to perform custom drawing. There are many computer technologies that can be used to perform drawing, some of them are: Microsoft DirectX, Microsoft GDI, Microsoft GDI+, Java Swing and OpenGL. From the UML diagram of Figure 9, it can be seen that each object has a position, a rotation angle (theta) and a set of bounds. The method **IsCollision()** takes another object of type Object and returns true when the bounds of one object collide with the bounds of another object. This method was implemented using a simple version of the algorithm presented by [14].

6.2. The mobile class

This class represents a moving object in a 2-Dimensional space. Each *Mobile* object has a speed, an acceleration and a rotation speed as shown in the UML diagram of Figure 9. This class is derived directly from the *Object* class, and therefore, is also abstract as the *Render()* method is not implemented. The **UpdateBounds()** method for this class takes the number of seconds at which the object bounds will be computed. This method is extremely useful when moving an object around other objects, for instance, if the bounds of one object intersect with

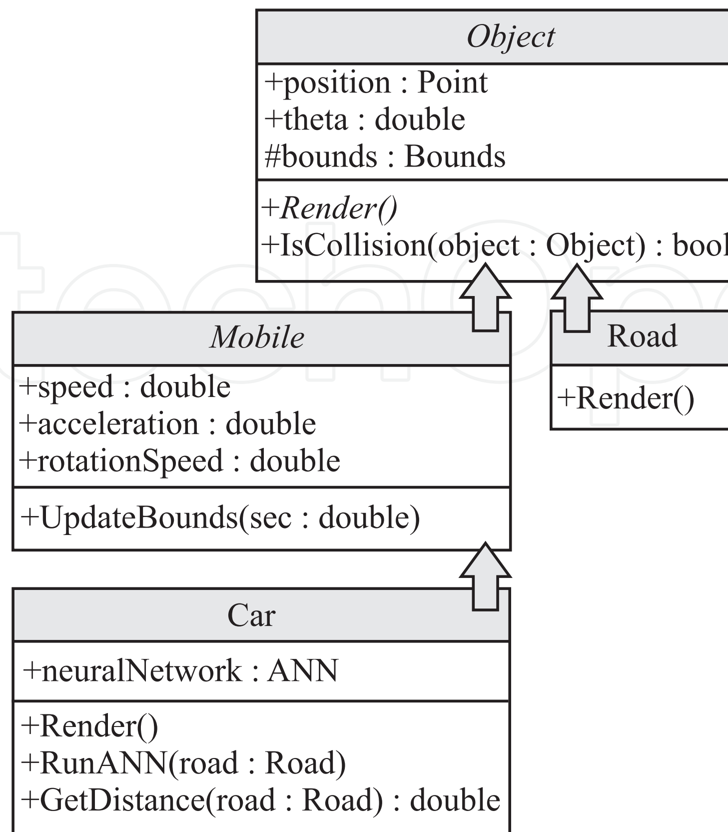


Figure 9. UML diagram showing the Car and the Road classes.

the bounds of another object, the object cannot move, this is implemented internally in the method **IsCollision()**. To update the bounds of a Mobile object, the speed of the object may be computed using Equation 3,

$$v_2 = v_1 + at, \tag{3}$$

where v_1 is the initial velocity of the object, v_2 is its final velocity, a is the acceleration, and t stands for the time for which the object moved. Similarly, the position of the object can be updated using another Kinematic equation. To compute the new position of the object Equation 4 can be used,

$$d = v_1 t + \frac{1}{2}at^2, \tag{4}$$

where the symbol d stands for the displacement of the object. In most cases, the object moves along a line described by its rotation angle and its position. Thus d has to be accordingly projected in the coordinate system as shown by Equations 5 and 6,

$$x_2 = x_1 + v_1 t \cos\theta + \frac{1}{2}at^2 \cos\theta, \tag{5}$$

$$y_2 = y_1 + v_1 t \sin\theta + \frac{1}{2}at^2 \sin\theta. \tag{6}$$

In some cases, the object may be rotating at a constant speed and, hence, the rotation angle has also to be updated at each period of time.

During the experiments, the methods **UpdateBounds()** and **IsCollision()** of the *Mobile* class are used together to prevent object collision. First, the simulator calls the **UpdateBounds()** method to compute the bounds of the object at some specific time, and then will call the **IsCollision()** method to check for potential collisions with other objects.

6.3. The road

The simulation experiments were performed using only two classes: the Road class and the Car class. The UML diagram of the Road class is shown in Figure 9. The Road class is derived directly from the *Object* class; the method **Render()** is implemented to draw the road displayed in Figure 7 using Microsoft GDI and Scalable Vector Graphics (SVG). When the car leaves the *Start*, the ANN has to make its first right turn at 90 $^\circ$, as the car is just accelerating this turn is easy. The following turn is also to the right at 90 $^\circ$ and the ANN should not have any trouble making this turn. Then, if the ANN wants to drive the car to point **A**, it has to make two turns to the left.

The straight segment from **A** to **B** should be easy to drive; unfortunately, the ANN may continually accelerate, and reach point **B** at a very high speed. The turn at point **B** is the most difficult of all the turns, because the car has to make a right turn at 90 $^\circ$ at high speed. Because the simulation is over when the car hits the bounds of the road, as soon as the ANN can see the turn of point **B**, it has to start reducing the speed of the car. Once the ANN has managed to drive to point **B**, reaching the *Finish* should not be difficult.

6.4. The car

The car used for the simulation is shown in Figure 10. The car has a position represented by x and y in Figure 10; the car rotation is represented by θ. The speed and acceleration vectors are represented by v and a respectively. The arrow next to the rotation speed in Figure 10 indicates that the car is capable of turning. The car has several variables to store its state (position, theta, speed, acceleration, rotationSpeed and neuralNetwork) as shown in the UML diagram of Figure 9. The Car class derives directly from the *Mobile* class and implements the method **Render()** to draw the car of Figure 10 using Microsoft GDI and SVG. When $v = 0$ and $a = 0$ the values of x and y do not change. When $v \neq 0$ and $a = 0$, the values of x and y will change while v remains constant. When $a \neq 0$, the values of x, y and v will change. The method **GetDistance()** computes the distance that the car has traveled along the road from the *Start*, this distance is represented by s in Figure 7.

Figure 11 illustrates how the car is able to receive information about its surroundings. The car had seven vision points illustrated by the arrows in the figure. To prevent the ANN from driving backwards, no vision lines were placed in the back of the car. Each value of d_1, d_2, ... d_7, represents the distance from the center of the car to the closest object in the direction of the vision line. These values were computed using the bounds of the road. To create a more interesting environment for the ANN, the values of d_1, d_2, ... d_7 were computed at low resolution and the car could not see objects located away from it.

In real life, a car driver is not able to modify directly the position or velocity of the car, the driver only controls the acceleration and the turning speed. As mention before, each car in the

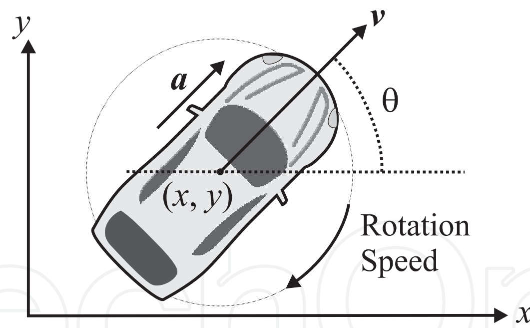


Figure 10. The car.

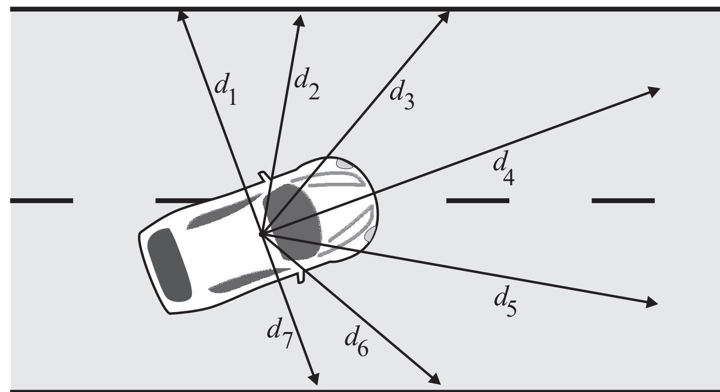


Figure 11. The car vision lines.

simulation has an ANN to do the driving. At each period of time, the ANN receives the vision information from the surroundings and computes the acceleration and the rotation speed of the car. Figure 12 shows the ANN of the car, the ANN has 8 inputs and two outputs. As it can be seen from this figure, the speed of the car is also applied to the input of the network; this is very important because the ANN needs to react differently depending on the current speed of the car. For instance, if the car moves a high speeds and faces a turn, it needs to appropriately reduce its speed before turning. As a matter of fact, the ANN needs to be ready for an unexpected turn and may regulate the speed of the car constantly.

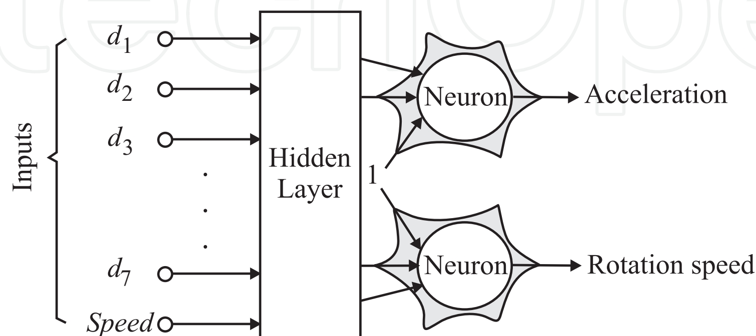


Figure 12. Artificial Neural Network for driving the car.

7. Experimental results

This section explains how SA was used to train the ANN. The implementation of SA was divided in three steps: initialization, perturbation and error computation.

The ANN training process using SA is illustrated in Figure 13. The simulation starts by randomly setting the network weights using a uniform probability distribution $U[-30, 30]$. In the second block, a copy of the weights is stored in a work variable. In the third block, the temperature is set to the initial temperature. For the simulation experiments, the initial temperature was set to 30. In the fourth block (iteration = 1), the optimization algorithm begins the first iteration. Then, the work variable (a set of weights) is perturbed. After the perturbation is completed, the ANN weights are set to these new weights. At this moment, the ANN is allowed to drive the car and the error is computed as shown in the figure. The temperature decreases exponentially and the number of iterations is updated as shown in the flow diagram. The simulation ends when the error reaches the desired goal or when the temperature reaches its final value (a value of 0.1 was used).

The cooling schedule used in the simulations is described by Equation 7,

$$T_{j+1} = cT_j, \tag{7}$$

where T_{j+1} is the next temperature value, T_j is the current temperature, and $0 < c < 1$. Clearly, the cooling schedule of Equation 7 is exponential and slower than a logarithmic one, therefore Simulated Quenching (SQ) is being used for the training of the ANN, see [3].

Observe, that each time the ANN weights are perturbed, the ANN is allowed to drive the car. Then, the error is computed and the oracle makes a decision about whether the new set of weights is accepted or rejected using the probability of acceptance of Equation 8, see [5] and [11]. Some implementations of SA accept a new solution only if the new solution is better than the old one, i.e. it accepts the solution only when the Error decreases; see [7] and [10]. The probability of acceptance is defined as

$$h(\Delta E) \approx exp(-\Delta E/T) \tag{8}$$

where

$$\Delta E = Error_{new\ solution} - Error_{currentsolution}.$$

A uniform probability distribution was used to generate states for subsequent consideration. At high temperatures, the algorithm may frequently accept an ANN (a set of weights) even if the ANN does not drive better than the previous one. During this phase, the algorithm explores in a very wide range looking for a global optimal ANN, and it is not concerned too much with the quality of the ANN. As the temperature decreases, the algorithm is more selective, and it accepts a new ANN only if its error is less than or very similar to the previous solution error following the decision made by the oracle.

7.1. SA initialization

Because of the properties of the activation function of Equation 2, the output of an ANN is limited. As mentioned before, an ANN is trained by adjusting the weights that connect the

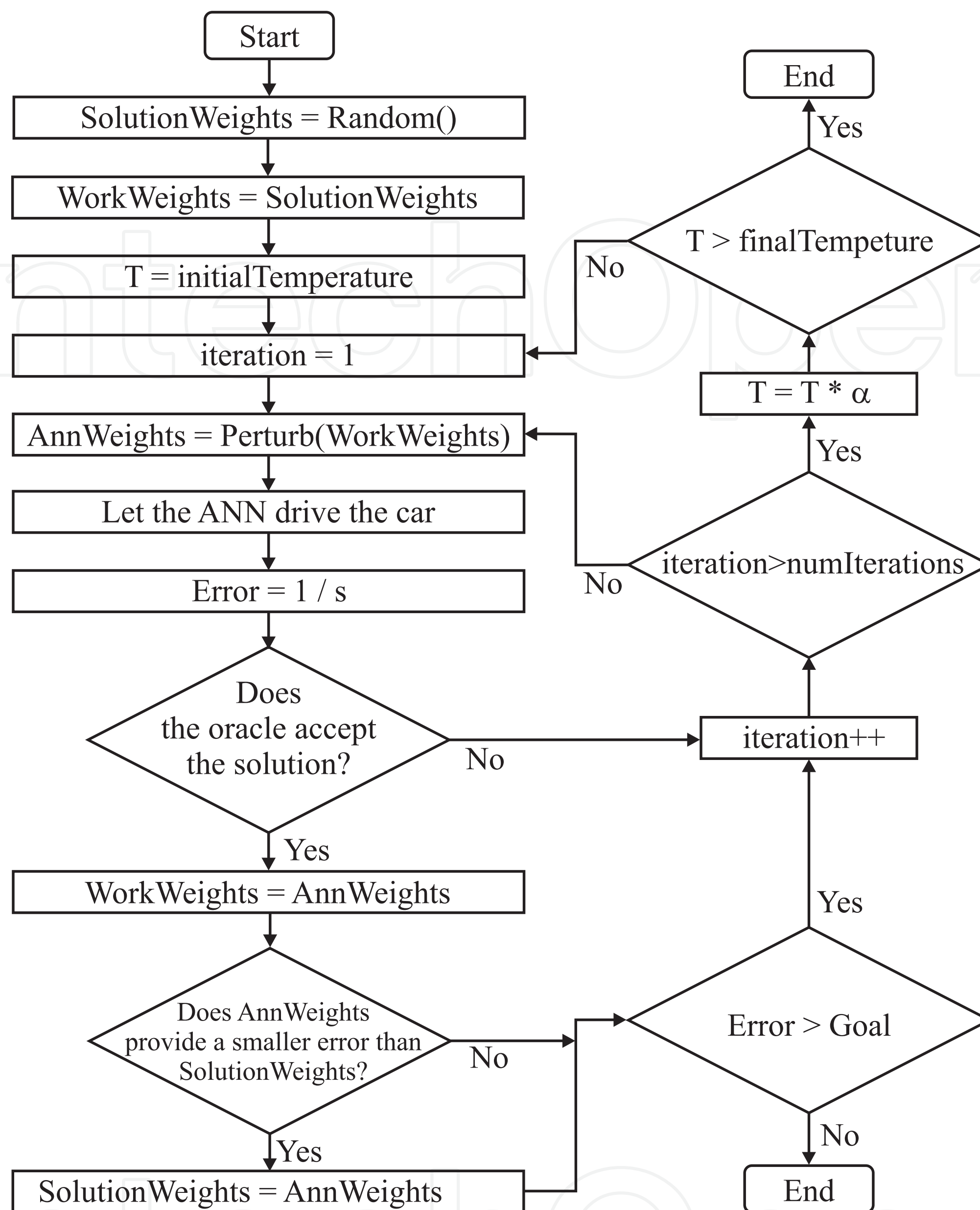


Figure 13. Flow diagram for simulated annealing evolution.

neurons. The training of an ANN can be simplified, if the input applied to the ANN is limited. Specifically, if the input values are limited from −1 to 1, then the ANN weights are limited to approximately from −30 to 30, [8] and [12]. To simplify the simulation, the input values of the ANN were scaled from −1 to 1. Therefore, the SA initialization consisted in simply assigning a random value from −30 to 30 using a uniform probability distribution to each of the ANN weights as shown in the C++ code shown in Figure 14. Observe that the random number generator uses the (*ISO/IEC TR 19769*) C++ Library Extensions **TR1**: *default_random_engine* and *uniform_real*. In this case, the ANN has two sets of weights: the hidden weights and the output weights. Each set of weights was stored in a matrix using the **vector** template from the Standard Template Library (STL); each matrix was built using a vector of vectors.

```
void Solution::SAInitialize()
{
   std::tr1::default_random_engine randGen;
   std::tr1::uniform_real<double> dist(-30.0, 30.0);
   //________________________ Hidden weights
   for(int i=0; i<hidRowCount; i++)
      for(int j=0; j<hidColCount; j++)
         hidWeight[i][j] = dist(randGen);
   //________________________ Output weights
   for(int i=0; i<outRowCount; i++)
      for(int j=0; j<outColCount; j++)
         outWeight[i][j] = dist(randGen);
}
```

Figure 14. Implementation of SA initialization using the C++ language.

7.2. SA perturbation

The code of Figure 15 shows the implementation of the SA perturbation using the C++ language. First, each ANN weight was perturbed by adding a random value from $-T$ to T using a uniform probability distribution (*tr1::uniform_real*), where T is the current temperature. Second, if the perturbed weight was outside the valid range from -30 to 30, the value was clipped to ensure that the weight remained inside the valid range.

7.3. SA error computation

In order to measure the driving performance of the ANN, an error function E was defined as shown in Equation 9,

$$E = \frac{1}{s}, \tag{9}$$

where the variable s represents the distance along the road measured from the *Start* to the current position of the car as shown in Figure 7. As it can be seen the value of the error decreases as the car drives along the road. The smallest error is accomplished when the car reaches the *Finish*.

The code of Figure 16 illustrates the implementation of the error function. The function starts by setting the ANN weights. The variable **deltaTimeSec** is used to refresh the simulation, a value of 16.7 milliseconds was used; it provides a refreshing frequency of 60 Hz (so that the simulation could be rendered on a computer display at 60 frames per second). Next, the function begins a **while** block, at each iteration the car bounds are updated and the simulation checks for a collision between the car and the road. If there is a collision the simulation stops and the error is computed. If there are not collisions, the ANN computes vision information and updates the acceleration and rotation speed of the car.

7.4. Results

Several experimental simulations were performed using different configurations to analyze the behavior of the ANN and the car.

```
void Solution::SimAnnealPerturb(Ann& preAnn, double temperature)
{
    std::tr1::default_random_engine randGen;
    std::tr1::uniform_real<double> dist(-temperature, temperature);
    //_________________________ Hidden weights
    for(int i=0; i<hidRowCount; i++)
    {
        for(int j=0; j<hidColCount; j++)
        {
            hidWeight[i][j]= preAnn.hidWeight[i][j]+dist(randGen);
            if (hidWeight[i][j]>30.0) hidWeight[i][j]=30.0;
            if (hidWeight[i][j]<-30.0) hidWeight[i][j]=-30.0;
        }
    }
    //_________________________ Out weights
    for(int i=0; i<outRowCount; i++)
    {
        for(int j=0; j<outColCount; j++)
        {
            outWeight[i][j]= preAnn.outWeight[i][j]+dist(randGen);
            if (outWeight[i][j]>30.0) outWeight[i][j]=30.0;
            if (outWeight[i][j]<-30.0) outWeight[i][j]=-30.0;
        }
    }
}
```

Figure 15. Implementation of SA perturbation using the C++ language.

```
double Solution::SimAnnealGetError()
{
    car.neuralNetwork.SetWeights(hidWeight, outWeight);
    const float deltaTimeSec = 0.0167f;
    float duration = 0.0f;

    while (duration<=300.0f) // 5 minutes max to travel the road
    {
        car.UpdateBounds(deltaTimeSec);
        if (car.IsCollision(road) == true) break;
        car.RunANN(road);
        duration+=deltaTimeSec;
    }
    return 1.0/car.GetDistance();
}
```

Figure 16. Implementation of SA error function the C++ language.

In the first simulation, the speed of the car was not applied at the input of the ANN, in all cases, the ANN was not able to turn at point **B**. At some unexpected point, the ANN was able to see the approaching turn of point **B** and did not have enough time to reduce the speed of the car.

In the second simulation, the number of neurons in the hidden layer was varied from 0 to 5. When the number of neurons in the hidden layer was zero, the ANN was able to drive the car to the **Finish** in 90% of the cases. When the number of neurons in the hidden layer was increased to one, the car was always able to get to the **Finish**. It was also observed that the ANN was driving faster when using more neurons in the hidden layer, thus, the car was getting to its destination quicker. When the number of neurons in the hidden layer was increased to 5, there were not any noticeable changes in the performance of the car than when the ANN had 4 neurons in this layer.

The third experiment consisted in varying the number of vision lines described in Figure 11. The number of vision lines was varied from 3 to 7. When using 3 vision lines, the ANN was able to reach 50% of the times to point **A**, 10% of the cases to point **B** and it was never able to get to the **Finish**. When the number of vision lines was set to 4, the ANN was able to drive the car to the **Finish** in 50% of the cases. When the number of vision lines was set to 5, 6 or 7, the ANN was always able to drive the car to the **Finish**. However, the ANN was always driving faster when using more vision lines.

The SA parameters were set to be compatible with the ANN weights. The initial temperature was 30, the final temperature was 0.1. Some experiments were performed by using lower final temperatures, but there were not any noticeable changes in the performance of the ANN. The number of temperatures was set to 10 using 20 iterations per temperature. Some tests were performed using more numbers of iterations, but there were not improvements. All the simulations were run using an exponential cooling schedule.

To validate the training of the ANN, another road similar to the shown in Figure 7 was built. In all cases, the ANN behaved similar in both roads: the road for training and the road for validation.

8. Conclusion

An ANN is a method inspired in biological processes. An ANN can learn from a training set. In this case, the problem has to be described by a training set. Unfortunately, some problems cannot be easily described by a data set. This chapter proposes the use of SA to train an ANN without a training set. We call this method simulated annealing evolution because, the ANN learns by experience during the simulation.

Simulated annealing evolution can be used to train an ANN in an ample set of cases. Because human beings learn by experience, simulated annealing evolution is similar to human learning.

An optimization problem was designed to illustrate how to use SA to train an ANN. The problem included a car and a road. An ANN was used to drive the car in a simple road. The road had several straight segments and turning points. The objective of the ANN was to

drive the car from the **Start** to the **Finish** of the road. At the beginning of the simulation, the car was placed at the **Start** and the ANN weights were set to random values. Obviously, the ANN could not drive too far the car without hitting the bounds of the road, and stopping the simulation. By the time the temperature reached its final value, the ANN was able to drive successfully to the **Finish** of the road as it will be briefly described.

During the simulation, the car had a set of vision lines to compute the distance to the closest objects. The distance from each vision line (measured from the car to the closest object) was applied to the input of an ANN. It was noticed that the ANN performed much better when the speed of the car was also applied to the input of the ANN.

The number of neurons in the hidden layer of the ANN was varied during the simulations. It was observed that when the number of neurons in the hidden layer was increased, the ANN was able to reach quicker the **Finish**. It was observed also that when using 5 or more neurons in the hidden layer, the performance of the ANN did not improve. It was noticed, however, that when using zero neurons in the hidden layer, the ANN could not always drive the car to the **Finish**.

The car vision consisted in a set of lines. Experimental simulations were performed varying the number of vision lines form 3 to 7. The experimental results indicated that when 3 vision lines are used, the ANN does not have enough information and cannot drive successfully to the **Finish**. It was observed also that when the number of vision lines was increased, the driving of the ANN was smoother. Finally, it was noticed that when the number of vision lines is increased to 8 or more, the ANN did not improve its performance (meaning that there were not observable changes in its driving).

Author details

Ledesma Sergio, Jose Ruiz and Guadalupe Garcia
University of Guanajuato, Department of Computer Engineering, Salamanca, Mexico

9. References

[1] Bandyopadhyay S., Saha S., Maulik U. & Deb K. A Simulated Annealing-Based Multi-objective Optimization Algorithm: AMOSA, IEEE Transactions on Evolutionary Computation 2008;12(3) 269-283.
[2] Buckland M. AI Techniques for Game Programming, Ohio USA: Premier Press; 2002.
[3] Ingber L. Simulated annealing: Practice versus theory, Mathematical and Computer Modelling 1993;18(11) 29-57.
[4] Jones M. T. AI Application Programming, Massachusetts: Charles River Media; 2005.
[5] Jones M. T. Artificial Intelligence: A Systems Approach, Massachusetts: Infinity Science Press LLC; 2008.
[6] Ledesma S., Torres M., Hernandez D., Avina G. & Garcia G. Temperature Cycling on Simulated Annealing for Neural Network Learning, In: A. Gelbukh and A.F. Kuri Morales (Eds.): MICAI 2007, LNAI 4827: proceedings of the Mexican International Conference on Artificial Intelligence MICAI 2007, 4-10 November 2007, Aguascalientes, Mexico, Springer-Verlag Berlin Heidelberg 2007.

[7] Masters T. Practical Neural Network Recipes in C++, San Diego, USA: Academic Press Inc.; 1993.
[8] Masters T. Signal and Image Processing with Neural Networks, New York, USA: John Wiley & Sons Inc.; 1994.
[9] Masters T. Advanced Algorithms for Neural Networks, New York, USA: John Wiley & Sons Inc.; 1995.
[10] Metropolis N., Rosenbluth M. N., Rosenbluth A. H. & Teller E. Equation of State Calculations by Fast Computing Machines. Journal of Chemical Physics 1953;21(6) 1087-1092.
[11] Press W. H., Teukolsky S. A., Vetterling W. T., & Flannery B. P. Numerical Recipes in C++: The Art of Scientific Computing (Third Edition), Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore and Sao Paulo: Cambridge University Press; 2007.
[12] Reed R. D. and Marks R. J. Neural Smithing, Cambridge, Massachusetts, USA: The MIT Press; 1999.
[13] Russel S. J. and Norvig P. Artificial Intelligence: A Modern Approach (3rd edition), Upper Saddle River, NJ USA: Prentice Hall; 2009.
[14] Shamos M. & Hoey D. Geometric intersection problems, 17th Annual Symposium on Foundations of Computer Science, October 1976, IEEE, Houston, Texas, USA; 1976.
[15] Smith K. I., Everson R. M., Fieldsend J. E., Murphy C., & Misra R. Dominance-Based Multi-objective Simulated Annealing, IEEE Transactions on Evolutionary Computation 2008;12(3) 323-342.
[16] Stefankovic D., Vempala S. & Vigoda E. Adaptive simulated annealing: A near-optimal connection between sampling and counting, Journal of the ACM 2009;56(3).

IntechOpen