Neural Network: Algorithms and Methods

February 22, 2010

Chapter 1

Alternative Learning Algorithms

An important issue in training neural networks is that the networks should not become paralyzed or stuck into poor local minima, which are far from the global minimum. We introduced several learning algorithms in the previous chapter to train neural works. The majority of those algorithms, specifically gradient based algorithms, have a tendency to trap into local minima. To alleviate this problem, we may apply a number of tricks such as restarting training with a new random set of weights, training with noisy exemplars, and perturbing the weights. All these tricks can be applied when we find that a network appears to prematurely converge. While these tricks may lead to improved solutions, there is no guarantee that such solutions will not also be only locally optimal. Further, the same suboptimal solution may be rediscovered, leading to fruitless oscillatory training behavior.

All learning algorithms introduced in the previous chapter use some sort of gradient information during training to update the weights of a neural network. This requirement enforces that an activation function of the neural network must be differentiable, although some activation functions may not be differentiable but suitable for practical purposes. For instance, a threshold logic activation function has only two states: on and off, which make make them easily implementable in hardware. However, it would not be possible to learn appropriate weights for neural networks with the threshold function because most classical learning algorithms requires a differentiable

activation function. We thus require learning algorithms that have abilities in escaping from local optima and can work with both differentiable or non-differentiable activation functions. In this chapter, we shall introduce three learning algorithms that neither suffer from entrapment in local minima nor depend on the differentiability of an activation function. These learning algorithms are developed based on metaheuristics and can be used independently or in conjunction with classical learning algorithms to train neural networks. Porto *et al.* (1995) termed the metaheuristic based learning algorithms as *alternative* learning algorithms.

A metaheuristic is a heuristic that combines user-specified black-box procedures to solve a very general class of computational problems for which there is no satisfactory problem-specific algorithm or heuristic; or when it is not practical to implement such a method. The name metaheuristics combines the Greek prefix "meta" ("beyond", here in the sense of "higher level") and "heuristic" ("to find"). The recent success of metaheuristics is clearly recognizable in an increasing number of books [3, 207], collections of articles [204, 194], special issues of journals [97, 202, 149] and a new conference series, Metaheuristics International Conference, that concentrates solely on this subject. Metaheuristics based training can avoid local minima either by accepting worse solutions or by generating good starting solutions to guide a training process towards better solutions. In the latter case, often the experience accumulated during training is used to guide the search in subsequent iterations. Furthermore metaheuristics based learning algorithms do not place restrictions on the network topology and activation function. A number of metaheuristic have been proposed in the literature such as simulated annealing, genetic algorithm, evolutionary programming, evolution strategies, ant colony optimization and tabu search. In this chapter, we shall briefly describe the first three metaheuristic methods and their application to training neural networks. For more detail of these methods, interested readers can look the books [][][].

1.1 Simulated Annealing

Simulated annealing is a robust and general search technique that has attracted significant attention due to its suitablity for optimization problems, especially for some problems where a desired global solution is hidden in

many poorer local solutions. This technique has been applied to the famous traveling salesman problem, integrated circuit design, image restoration, graph partitioning, routing and many other problems. While this list is far from exhaustive, it indicates that simulated annealing has a great scientific and industrial importance in solving various problems. Simulated annealing plays a special role within search for two reasons. Firstly, it appears to be quite successful when applied to a broad range of practical problems. Secondly, it uses a stochastic component that facilitates a theoretical analysis of its asymptotic convergence. The later feature makes simulated annealing very popular to mathematicians. At the heart of simulated annealing is an analogy to the statistical mechanics of annealing. We shall first describe the physics analogy of simulated annealing, then the method of simulated annealing and finally the use of simulated annealing to train neural networks.

1.1.1 The Physics Analogy

In metallurgy, annealing is a process of heating up a solid in a heat bath and then cooling slowly until it crystallizes. Simulated annealing uses an analogous set of "controlled cooling" operations for nonphysical optimization problems, in effect transforming a poor and unordered solution into a highly optimized and desirable solution. The annealing process consists of the following two steps.

- Increase temperature of the heat bath to a maximum value at which the solid melts.
- Decrease temperature carefully until the atoms of the melted solid rearrange themselves in a ground state.

The atoms have high energies at very high temperatures. This causes the atoms to become unstuck from their initial positions and gives them a great deal of freedom to move randomly through high energy states. As temperature reduces, energy decreases and the atoms lose their mobility. However, the slow reduction of temperature gives the atoms more chances to rearrange and form crystal, a highly regular atomic structure. This structure is the state of a minimum energy for the system. The amazing fact is that, for a slowly cooled system, nature is able to find such a minimum energy state. In contrast, if we reduce temperature too quickly, the crystal is not

formed; rather it ends up in a polycrystalline or amorphous state having somewhat higher energy. The process of quick reduction of temperature is called *quenching*.

We can model the physical annealing process by computer simulation based on Monte Carlo techniques that rely on repeated random samplings to simulate physical and mathematical systems. In a standard Monte Carlo technique, the state of a system is randomly changed and all such changes are accepted regardless the feasibility of the new states. This acceptance criterion may cause the system to fall into unreasonable states for much of the time. Thus computer simulation would have to run for a long time to sample properly all feasible states of the system. An introductory overview of the use of Monte Carlo techniques in statistical physics is given by Binder (1978). We here discuss one of the early techniques known as the *Metropolis algorithm*. In 1953, Nicholas Metropolis and coworkers proposed a new algorithm based on the Monte Carlo technique for simulating the evolution of solids in a heat bath to thermal equilibrium. The modification made by Metropolis and coworkers is the use of a probabilistic acceptance criterion to accept the new states produced by repeated random samplings. The probabilistic acceptance criterion is known as the *Metropolis criterion*.

Let the current state of the solid is i and its energy E_i. The Metropolis algorithm generates the next state j by perturbing the current state i, for instance, by displacing a single atom. This small change makes the new state near to the old one. Let E_j be the energy of the new state. If the energy difference, i.e., $E_i - E_j$, is less than or equal to 0, the Metropolis algorithm accepts j as a current state. Otherwise, the algorithm accepts j as the current state with some probability. The probability of accepting j as the new state is defined as

$$P\{\text{accept } j\} = \begin{cases} 1 & \text{if } E_j \leq E_i \\ \exp\left(\frac{E_i - E_j}{k_B T}\right) & \text{if } E_j > E_i \end{cases} \tag{1.1}$$

where k_B is a physical constant known as the Boltzmann constant and T denotes the temperature of the heat bath. A characteristic feature of the acceptance criterion is that it not only accepts improvement (energy reduction), but also accepts deterioration (energy increase) to some extent. This criterion is quite different from the one used in a greedy approach, e.g. gra-

dient descent learning algorithm, that accepts only improvement. Accepting deterioration in contrast to iterative improvement facilitates simulated annealing to escape from local optima.

Ref **K. Binder, Monte Carlo Methods in statistical physics, Springer, Berlin.**

1.1.2 The Method

S. Kirkpatrick and coworkers showed how the Metropolis algorithm provides a natural tool for bringing the techniques of statistical mechanics to bear on optimizations. By analogy the generalization of such an algorithm to an optimization problem is very straightforward. The current state of a thermodynamic system is analogous to the current solution of the optimization problem, the energy equation to an objective function, and the ground state to the global optimum (minimum or maximum). When simulated annealing was first introduced, it was used for designing complex integrated circuits and solving the traveling salesman problem.

Notice that the two applications cited above are both examples of combinatorial problems. The aim of these problems is to minimize an objective function. The space over which that function is defined is the N-dimensional space of N discrete parameters, like the set of possible orders of cities for the traveling salesman problem, or the set of possible allocations of element to construct integrated circuits. While combinatorial optimization problems have wide applicability, there is a large variety of optimization problems involving continuous parameters but not discrete. Assume we like to fit a function to a sum of exponentials. The choice of the decay constant is an optimization problem. Continuous parameter optimization problems also frequently arise in many other contexts such as engineering design, econometrics, and data analysis. In this chapter, the concept of simulated annealing will be employed for training neural networks, a kind of continuous parameter optimization problem. It is natural to describe simulated annealing in the context of continuous parameter optimization.

Consider a function $f(x_1, x_2, \cdots, x_n) = f(\mathbf{x})$ is defined over an n-dimensional space of continuous parameters, We like to minimize f with respect of $\mathbf{x}$. The function f may be the energy of a physical system, the error in a fitting problem or any other "objective function". It is clear from

the preceding section that the Metropolis algorithm has elements. In the context of our minimization problem, these elements are as follows. The function, f, is the objective function. The system state is $\mathbf{x}$. A generator of random variations to change the system states, that is, a procedure for taking a random step from $\mathbf{x}$ to $\mathbf{x} + \Delta\mathbf{x}$. Finally, a control parameter, something like a temperature, with an annealing schedule that describes how the temperature is to be lowered from high to low values. The pseudo code of the method of simulated annealing is presented in *Algorithm* 3

The method of simulated annealing proceeds by choosing an initial state $\mathbf{x}^{(0)}$ and making random steps $\Delta\mathbf{x}$. A natural way to chose $\Delta\mathbf{x}$ is to call a random number generator n times to produce $\Delta x_1, \cdots, \Delta x_n$. At each step, the method evaluates the change

$$\Delta f = f(\mathbf{x} + \Delta\mathbf{x}) - f(\mathbf{x}) \tag{1.2}$$

in the objective function value. If Δf is negative (improved) in terms of the objective function value, the method of simulated accepts the new state $\mathbf{x}^{(1)} = \mathbf{x}^{(0)} + \Delta\mathbf{x}$ as a current state. In contrast, if Δf is positive (deteriorated) in terms of the objective function value, the new state is accepted with a probability p, which is calculated based on the new fitness value, $f(\mathbf{x}^{(0)} + \Delta\mathbf{x})$, relative to the previous fitness value, $f(\mathbf{x}^{(0)})$. That is,

$$\begin{aligned} p &= \frac{\exp\left(-f(\mathbf{x}^{(0)} + \Delta\mathbf{x})/T\right)}{\exp\left(-f(\mathbf{x}^{(0)} + \Delta\mathbf{x})/T\right) + \exp\left(-f(\mathbf{x}^{(0)})/T\right)} \\ &= \frac{1}{1 + \exp\left(\Delta f/T\right)} \\ &\approx \exp\left(-\Delta f/T\right) \end{aligned} \tag{1.3}$$

where Δf is the increase in f and T is a control parameter, which by analogy with a thermodynamic system is "temperature" irrespective of the objective function being involved in the optimization process. The probability of accepting deterioration with respect to the value of f depends on the choice of T. The higher the value of T, the higher the probability of accepting deterioration. The method of simulated annealing decreases T during its execution. It uses a procedure called "cooling schedule" for decrementing T. Initially, at large values of T, large deteriorations will be accepted; as

T decreases, only smaller deteriorations will be accepted and finally, as the value of T approaches 0, no deteriorations will be accepted at all.

When T is reduced slowly enough, a system will avoid getting trapped into local minima because the probability of accepting new states expressed by (1.3) allows the deterioration f in contrast to iterative improvement to get over a barrier into a new local (or global) minimum. We can generate many new states and apply the aforementioned acceptance criterion many times successively for geting a series of accepted states in the parameter space. These states generate a random walk that explores the parameter space, and which at long times is governed by the probability distribution function

$$P(\mathbf{x}) = \frac{1}{Z}\exp\left(\frac{-\Delta f}{T}\right) \tag{1.4}$$

where $P(\mathbf{x})d^n x$ is the probability that the walk will be in the volume $d^n x$ on any given step at long times, and the normalization constant or "partition function" Z is given by

$$Z = \int d^n x \, \exp\left(\frac{-\Delta f}{T}\right). \tag{1.5}$$

We have now understood that the most important comonents in the method of simulated annealing are a generator of random changes and a colling schedule. The choice of these two component greatly influence the solution quality and convergence of simulated annealing. We describe these compoments at length in the following sections.

1.1.3 Generating Function

The aim of a generating function in simulated annealing is to facilitate the search traversal from one state to another, for instance, $\mathbf{x}$ to $\mathbf{x} + \Delta\mathbf{x}$, over an n-dimensional parameter space. We can generate $\Delta\mathbf{x}$ in a deterministic way or random fashion. The method of simulated annealing generates $\Delta\mathbf{x}$ randomly. The many applications of randomness have led to many different techniques for generating random numbers. These techniques may vary as to how unpredictable or statistically random they are, and how quickly they

can generate random numbers. The earliest methods for generating random numbers – dice, coin flipping, roulette wheels – are used mainly in games and gambling. These techniques can not be employed for generating $\Delta\mathbf{x}$ because they generate discrete numbers, thereby not suitable for continuous parameter optimization.

We like to generate random variables taking values in a continuum, such as $\mathbb{R}$ and the interval $[a, b]$. How we can calculate the probability of events associated with a random variable X that takes values on the continuum. In probability theory and statistics, the *cumulative distribution function*, also called *probability distribution function* or just *distribution function*, describes completely the probability distribution of the real-valued random variable X. The distribution of X for every real number x is given by

$$x \mapsto F_X(x) = P(X \leq x) \tag{1.6}$$

where the right-hand side represents the probability that the random variable X takes on a value less than or equal to x. The probability that X lies in the interval $(a, b]$ is therefore $F_X(b) - F_X(a)$ if $a < b$. A function $F : \mathbb{R} \to \mathbb{R}$ is the distribution function of some random variable if and only if it has the following properties:

i) F is non-decreasing (i.e., $F(x) \geq F(y)$ whenever $y > x$).

ii) F is right-continuous i.e., $F(x + \epsilon) \geq F(y)$

iii) F is normalized i.e., $\lim_{x\to\infty} F(x) = 0$, $\lim_{x\to\infty} F(x) = 1$

One such function that has the aforementioned properties is the *normal distribution*, also called the *Gaussian distribution*. This is an important family of continuous probability distributions applicable in many fields. Each member of the family can be defined by two parameters, location and scale: the mean ("average", μ) and variance ("standard deviation squared", σ^2), respectively. Carl Friedrich Gauss (1777-1855) became associated with this set of distributions when he analyzed astronomical data and defined the equation of its probability density function.

The continuous probability distribution is often described in terms of another function called the *probability density function*. The one-dimensional probability density function of the Gaussian distribution is

$$\varphi_{\mu,\sigma^2}(x) = \varphi(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(x-\mu)^2}{2\sigma^2}}, \quad x \in \mathbb{R} \tag{1.7}$$

If we choose the value of μ to be zero, the probability density function becomes

$$\varphi(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-x^2}{2\sigma^2}}, \quad x \in \mathbb{R} \tag{1.8}$$

The variance σ^2 controls the shape of the probability density function. To visualize the effect of σ^2, we plot the probability density function for different σs in the same scale (Fig. 2). This figure shows that, for a large σ, the probability density function distribution exhibits a central maximum with a long flat tail. This phenomena indicates that the probability density function involving a large σ^2 is likely to generate a large x. This feature is beneficial for exploring a search space. Similarly, a large central maximum with a small flat tail, which is obtained for a small σ^2, is suitable for exploitation (fine tuning). As mentioned before, simulated annealing uses a large T at the beginning and gradually decreases T as the annealing process progresses. This is equivalent to exploration at the beginning and exploitation (fine tuning) at the later stage. We have now understood that σ^2 can be thought equivalent to T. If we replace σ^2 and Δx of Eq.(1.8) by T and x, respectively, we obtain

$$\varphi(\Delta x) = \frac{1}{\sqrt{2\pi T}} e^{\frac{-\Delta x^2}{2T}}, \quad \Delta x \in \mathbb{R} \tag{1.9}$$

The form of (1.9) is called the Gaussian form of the *Boltzmann probability density function*. For n-parameters, this density function can be written as

$$\varphi(\Delta \mathbf{x}) = \frac{1}{\sqrt[n]{2\pi T}} e^{\frac{-\Delta \mathbf{x}^2}{2T}}, \quad \Delta x \in \mathbb{R} \tag{1.10}$$

Another commonly used continuous probability distribution is the *Cauchy distribution*, which is also known as the *Lorentz distribution* among physicists. The one-dimensional Cauchy density function centered at the origin is defined by

$$\varphi(x) = \frac{1}{\pi}\frac{c}{c^2 + x^2} \qquad x \in \mathbb{R} \tag{1.11}$$

where $c > 0$ is a scale parameter. To visualize the effect of c, we plot this density function for different cs in the same scale (Fig. 2). The shape of the Cauchy density function resembles that of the Gaussian density function, but the former one approaches the axis so slowly that an expectation does not exist. As a result, the variance of the Cauchy distribution is infinite. The Cauchy distribution is more likely to generate large variations due to its long flat tails than the Gaussian distribution. Any system with this feature have a higher probability of escaping from a local optimum or moving away from a plateau. If we replace c by T and x by Δx, the Cauchy density function for n-parameters can be written as

$$\varphi(\Delta \mathbf{x}) = \frac{1}{\pi}\frac{T}{[T^2 + \Delta x^2]^{(n+1)^2}} \qquad \Delta x \in \mathbb{R} \tag{1.12}$$

Amin will write here about a generating function based on the Gaussian and Cauchy distribution

1.1.4 Annealing Schedule

We have already seen that most features of simulated annealing, i.e., the state space, the new state generation and the objective function, are fixed by definition. The control parameter T is the only feature that simulated annealing modifies (decrements) during its execution using a annealing schedule through Eq.(1.3). The choice of this schedule is critical to the success of simulated annealing because it has affect on the acceptance criterion and the generating function as well. The principle underlying the choice of a suitable annealing schedule can easily be stated−the initial temperature should be high enough to "melt" a system completely and should be reduced as the search progresses towards the system's "freezing point"− but choosing an annealing schedule for practical purposes is not easy. A number of annealing scheduling have been proposed in the literature. We here describe some of them.

An annealing schedule has usually four components: an initial temperature T_0, a stopping criterion (something a final temperature), a rule for

temperature decrement, and **a finite number of state transitions at each temperature** k. The T_0 must be high enough so that search can traverse from the current state to its any neighbor. This may cause searching (at least in the early stages) into random walks. On the other hand, if T_0 is chosen not high enough, the final state would be very close to the starting state. It is, therefore, necessary to find an appropriate T_0. At present, there is no known method to find a suitable T_0 for all problems. According to Kirkpatrick *et al.* (1984), a suitable T_0 is one that results in an average probability χ_0 of a solution that increases f being accepted of about 0.8. The value of T_0 is clearly dependent on the scaling of f and, hence, be problem-specific. We can estimate such a by conducting initial search in which all increases in f are accepted and calculating $\overline{\delta f}$, an average increases of f. The initial temperature then can be defined as

$$T_0 = \frac{\overline{\delta f}}{\ln(\chi_0)} \tag{1.13}$$

Rayward and Smith (1996) suggested to start with a very high temperature and to cool rapidly until about 60% of worst solutions are being accepted. This forms a real T_0 that can now be cooled more slowly. A very similar idea was also suggested by Dowsland (1995). This kind of annealing schedule is analogous to physical annealing in which a metal is heated until it becomes liquid and then cooling begins. That is, once the metal is transformed into liquid it is pointless carrying on heating it.

An obvious stopping criterion is that we should stop executing simulated annealing when T_0 becomes zero. This choice may require a long time for obtaining a optimal or a near optimal solution of a given problem. However, in practice, it is not necessary to let T_0 becomes zero because as it approaches zero, the chances of accepting worse solutions are almost same as T_0 being equal to zero. That is, we can stop executing simulated annealing when T_0 becomes low. The magnitude of the low value can be difficult to determine and is problem-dependent. In practice, the stopping criterion is determined by fixing the number of temperature values to be used for solving a given problem, or the total number of solutions to be generated by simulated annealing. Alternatively, we can stop executing simulated annealing when no better solution is found in all trials of any temperature.

Once we have T_0 and the stopping criterion we can devise a temperature decrementing procedure.The most important consideration in such procedure is that temperature should to be decremented very s lowly. This consideration leads to the following temperature decrement procedure

$$T(t) = \left(\frac{T_1}{T_0}\right)^t T_0 = \alpha^t T_0 \tag{1.14}$$

where t, "time" is the step count and α is a constant close to, but smaller than, 1. This temperature decrement procedure was first proposed by Kirkpatrick *et al.* (1982) with $\alpha = 0.90$. A linear procedure can also be used in decrementing temperature very slowly. This can be expressed as

$$T(t) = T_0 - \beta t \tag{1.15}$$

where β is a constant and can be any value greater than zero. Of special theoretical importance is a logarithmic temperature introduced by Geman and Geman [7],

$$T(t) = \frac{T_0}{\log(d + t)} \tag{1.16}$$

where d is usually set equal to one. It has been proven that for T_0 being greater than or equal to the largest energy barrier of a given problem, the logarithmic temperature decrement procedure will lead a system to the global minimum state in the limit of infinite time. However, due to its asymptotically extremely slow temperature decrease, this procedure is impractical for solving many real-world problems. Szu and Hartley (1987) proposed a temperature decrement procedure

$$T(t) = \frac{T_0}{(1 + t)} \tag{1.17}$$

that is inversely linear in time.

Algorithm 1 Simulated Annealing($\mathbf{x_0}, T_0, n, m$)

$T \leftarrow T_0$
$\mathbf{x} \leftarrow \mathbf{x_0}$
$\mathbf{x}^* \leftarrow \mathbf{x}$
for $k = 1$ to n **do**
  for $j = 1$ to m **do**
    $\widetilde{\mathbf{x}} \leftarrow \text{Neighbor}(\mathbf{x})$
    $\Delta E \leftarrow E(\widetilde{\mathbf{x}}) - E(\mathbf{x})$
    $p \leftarrow \text{Min}\{\exp(-\Delta E/T), 1\}$
    $u \leftarrow \text{Uniform}(0, 1)$
    if $u < p$ **then**
      $\mathbf{x} \leftarrow \widetilde{\mathbf{x}}$
    end if
    if $E(\mathbf{x}) < E(\mathbf{x}^*)$ **then**
      $\mathbf{x}^* \leftarrow \mathbf{x}$
    end if
  end for
  $T \leftarrow T_0/(k+1)$
end for
return $\mathbf{x}^*$

1.1.5 Simulated annealing for training neural network

As already described in the previous chapter, the learning objective of the neural networks is to find a suitable connection weight values so as to the error function gets minimized. However, such objective poses a challenge due to the irregular surface characteristics, i. e., lots of local minima, of the error function. Classical algorithms like the gradient descent and the conjugate gradient do not guarantee of finding a global minimum. On the other hand, an alternative learning algorithm like the simulated annealing guarantees for a global minimum under some conditions. We are now going to discuss how a neural network can be trained with the simulated annealing algorithm.

Recall the analogy between function optimization and the physical crystallization process by heating at high temperature followed by slow cooling process. In the physical system, one is interested to drive the system to a microscopic configuration of the atoms for which the system has globally minimum energy. In fact, such condition makes the atoms very organized and orderly. A neural network resembles such a physical system with its connection weights as a microscopic configuration and the error function as an energy function. Our goal is to find an optimal connection weight so that the network gives minimum error.

We consider here a feed forward three layer neural network - one input layer, a hidden layer, and an output layer. At the beginning, the network will be at a random state (analogous to a random microscopic configuration). Generally it is realized by initializing the connection weights randomly within a small range. It will be easy to handle the connection weights if all the weights are arranged in a vector. If the network has a total of n weights including the bias connections simulated annealing treats it as a vector in the Rn space. Once we arrange the connection weights in a vector we have to keep track of which element of the vector denotes which connection. MATLAB provides a pair of commands, getx and setx, to switch between the two representations - a vector and a structure of connection weights. After the initialization we simply follow the simulated algorithm to train the neural network. A pseudocode for simulated algorithm is presented in Figure x.

The procedure starts with an initial temperature and an initial weight configuration. Since the temperature controls the amount of random per-

turbation of a configuration and also the acceptance probability of a bad configuration with higher energy, initial temperature should be high enough to let the network cross the barriers of local minima.

The algorithm then progresses by slowly decreasing the temperature and at each temperature stage by taking a random walk over the space Rn (i. e., the space of all configurations). A random walk consists of a sequence of steps, where each step results in a neighbor state. This random walk is also known as sampling. Simulated annealing controls the neighborhood with a distribution function having a scaling parameter proportional to the current temperature. A step is always accepted whenever it leads to a lower energy than that of the current state. An important feature of this algorithm is that it also accepts a bad step resulting higher energy with some probability. This acceptance probability is given by when a step causes higher energy, i. e., when E is positive. Acceptance probability decreases with the decrease of temperature. As the temperature T approaches to zero acceptance probability also approaches to zero. As a result, simulated annealing accepts more energy rising steps at its earlier stages than the later stages. Similar to the crystallization of solid, simulated annealing algorithm can find a global solution by the aforementioned cooling and sampling mechanism.

We can do both the sampling and cooling in different ways. Depending on the scheme, we have different simulated annealing algorithms. In the earlier section, we explained three major variants - classical, fast and very fast simulated annealing algorithms. To give an idea of how simulated annealing can be programmed for training a neural network, a complete MATLAB code of fast simulated annealing is given in Fig. X.

1.2 Genetic Algorithms

Mimicking biological evolution and harnessing its power for adaptation are problems that have intrigued computer scientists for several decades. Genetic algorithms are a family of computational models based on the underlying genetic process in biological organisms. These algorithms process a population of simple *chromosomes*-like data structures with three operators: *selection*, *crossover*, and *mutation*. The current framework of genetic algorithms was first proposed by John Holland in the late 1960s and early 1970s, although some of the ideas appeared as early as 1957 in the context

of simulating genetic systems (Fraser, 1957).

In his book Adaptation in Natural and Artificial Systems (Holland, 1992), Holland presented genetic algorithms in a general theoretical framework for adaptation in nature. Holland attempted to understand and link diverse natural phenomena, but he also proposed potential engineering applications of such phenomena. Since the publication of Holland's book, genetic algorithms have gained recognition as a general problem solving techniques in many applications including function optimization, image processing, system control and many other areas. In this section, we first describe genetic algorithms and then their application to training neural networks.

1.2.1 Basic Description

We have already seen that the training of neural networks can be considered as a function optimization problem. We here describe genetic algorithms in the context of function optimization where we like to optimize a set of variables either to maximize some profit or to minimize some cost. In more traditional terms, we wish to maximize (or minimize) some function

$$\text{maximize} \quad f(\mathbf{x}) \tag{1.18}$$

$$\text{subject to} \quad \mathbf{x} \in \Omega \tag{1.19}$$

Given a specific problem to be solved, the first step of genetic algorithms is to choose randomly a set of potential solutions to that problem, encoded them in some fashion on simple chromosomes, which represent search space solutions and evolve over time through a process of *selection* and controlled variations introduced by two evolutionary operators: *crossover* and *mutation*. The number of chromosomes n in the population is usually chosen at random and is kept fixed throughout the solution process. We call the initial set of chromosomes as the *initial population* and denote it by $P(0)$. We then evaluate each chromosome of $P(0)$ by a predefined evaluation function, sometimes called *fitness function*. Some chromosomes of $P(0)$ may get higher fitness scores and can be regarded as good chromosomes, while others can be regarded as bad chromosomes. Of course, we like to proceed towards the solution of the problem with the good chromosomes, but not with the bad chromosomes. Genetic algorithms, therefore, creates an *mating pool* or *intermediate population* $M(0)$ by probabilistically selecting the

chromosomes of $P(0)$ based on their fitness scores. Since $M(0)$ must have the same number of chromosomes as $P(0)$, the probabilistic selection may choose some good chromosomes of $P(0)$ multiple times and omit some bad chromosomes.

We finally create a new population $P(1)$ from $M(0)$ by applying crossover and mutation operators. We first apply crossover with some probability p_c on the chromosomes of $M(0)$, and then mutation with some probability p_m. The purpose of the crossover and mutation is to introduce some sort variations in the chromosomes of $M(0)$. We repeat the above procedure iteratively, generating populations $P(2), P(3), \cdots$, until a desired solution is found or an an appropriate stopping criterion is reached. The process of generating $P(k)$ from $P(0)$ is called an *evolutionary process*. We call one iteration of the evolutionary process as one *generation* and denote it by g. The pseudo code of genetic algorithms is given in *Algorithm*. In short, we can think genetic algorithms as a population-based model that uses selection, crossover and mutation operators to generate new sample points in a search space. The five basic components of genetic algorithms are chromosome and encoding, fitness function, selection and mating pool, crossover, and mutation. We discuss these components at some length in the following subsections.

1.2.2 Chromosome and Encoding

In genetic algorithms, the term chromosome typically refers to a candidate solution of a given problem, often encoded as a string of symbols. This is equivalent to mapping points in the space of Ω on to a string of symbols. The string length L for all chromosomes in $P(g)$ is same and is kept throughout the whole evolutionary process. A chromosome consists of elements from a chosen set of symbols, called the *alphabet*. As for example, a common alphabet is the set $\{0, 1\}$ in which case the chromosome is simply a binary string. The choice of chromosome length, alphabet, and encoding is called the *representation scheme* for the problem. Some terminologies and their synonyms are used widely in genetic algorithms and evolutionary community in general. On the side of the original problem context, candidate solution, phenotype, and individual are used to denote points in the space of Ω. This space is commonly called the *phenotype space*. On the

side of genetic algorithms, genotype, chromosome, and again individual are used for points in the encoded space where evolutionary search will actually take place through crossover and mutation operations. The encoded space is commonly called the *genotype space* (**Give a phenotype and genotype conversion figure here)**.

Assume our $f(\mathbf{x})$ is consisted of two variables X_1 and X_2, and the value of X_i lies between 0 and 31. If we use a binary string to represent a chromosome, five bits are necessary to encode each of X_i into the chromosome. The length of the chromosome, L, would be 10 bits. It is obvious that the representation of the chromosome will be crucial to the success and efficiency of genetic algorithms. A good representation will make a problem easier to solve, while a poor one will do the opposite. For instance, let we use six bits to represent X_i though five bits are sufficient. The chromosome length will increase in such a representation, eventually the search space size. Let M is the number of alphabets, L is the length of a chromosome, and N is the number of chromosomes in $P(g)$. The search space size involving $P(g)$ would be NM^L. Let X_i lies in the continuum $\mathbb{R}$ but we encode it as an integer variable. This representation is also problematic in the sense that genetic algorithms would not able to find an appropriate value for X_i. In general, an important issue faced by genetic algorithms is the same that faced by many artificial intelligence approaches, that is, representation.

Genetic algorithms traditionally use the binary representation, but they can also use non-binary representations that are gaining popularity in recent years. When X_i lies in the continuum $\mathbb{R}$, it would seem natural to represent the variable into a chromosome directly as a real number. A vector of real numbers then constitute the chromosome. Since $f(\mathbf{x})$ is consisted of two variables, a real-value vector $(30.5, 22.2)$ could be an example of the chromosome. It is clear that the goal of a representation scheme is to map the phenotype space into the genotype space. Identification of an appropriate representation scheme is the first step for solving problems using genetic algorithms. Once a suitable representation scheme has been found, the next phase is to create the initial population $P(0)$. The chromosomes of $P(0)$ are usually initialized in an entirely random fashion because we may not have any idea as to what constitutes a good initial set of potential solutions. If users have some idea about the potential solutions of a given problem, they can use it in creating $P(0)$.

1.2.3 Fitness Function

A fitness function is one of important components of genetic algorithms as it determines whether a given chromosome will contribute to future generations through reproduction. This function assigns a quality measure for each genotype of a population $P(g)$. An ideal fitness function correlates closely with an algorithm's goal, and yet may be computed quickly. The *Speed of execution* is very important, because genetic algorithms must iterate many, many times to produce good solutions for non-trivial problems.

Typically, a fitness function is composed from a quality measure in the phenotype space although the function measures the quality of a point (chromosome) in the genotype space. Consider we like to maximize $f(\mathbf{x}) = {X_1}^3 + {X_2}^3$, where X_1 and X_2 are integer variables whose value lies between 0 and 31. We can use five bits to represent each of $X_i, i \in 1, 2$. Let one chromosome of $P(k)$ is 00101 11000. The fitness score of this chromosome can be defined as the cube of its corresponding phenotype, that is, $5^3 + 24^3 = 13,949$.

1.2.4 Selection and Mating Pool

Selection is a operator in some sense a procedure that genetic algorithms use to construct a mating pool $M(g)$ by selecting copies of chromosomes from $P(g)$. To emulate natural selection, that is, survival of the fittest, it is obvious to select chromosomes based on their fitness scores. The chromosome with a higher fitness score usually have a greater chance of contributing more copies in $M(k)$, while the one with a lower fitness score has a less chance to contribute. The motivation is that highly fit chromosomes of a population possess good properties that, recombined in the right way, could lead to even better chromosomes.

Genetic algorithms traditionally use a fitness proportionate selection scheme in constructing $M(k)$. This scheme consists of two steps: the calculation of selection probability and the application of a simpling procedure. Assume our population $P(k)$ is consisted of N chromosomes, $C_1, \cdots, C_N$ and the fitness of a chromosome C_i is f_i. The average fitness, $\overline{f}$, over all chromosomes in $P(k)$ is

$$\overline{f} = \frac{1}{N} \sum_{i}^{N} f_i \tag{1.20}$$

In fitness proportionate selection, the probability of selecting a chromosome to be a member of $M(k)$ is proportional to the relative fitness score. That is, the probability of C_i, $p_s(C_i)$, to be in $M(k)$ is

$$p_s(C_i) = \frac{f_i}{\sum_i^N f_i} = \frac{f_i}{\overline{f}N} \tag{1.21}$$

It can be seen from Eq.(1.21) that chromosomes with a above-average fitness tend to have a better chance than those with a below-average fitness to be chosen for $M(g)$. Nevertheless, chromosomes with a below-average fitness are given a small, but a positive chance. Otherwise the search process could become too greedy resulting in stuck at local optima.

After computing the selection probability for all chromosomes in $P(g)$, we apply a sampling procedure to select chromosomes based on such probabilities. A widely used classical sampling procedure is stochastic sampling with replacement (Holland, 1975; Goldberg, 1989a). This sampling procedure maps chromosomes of $P(g)$ to contiguous segments of a line, one segment for each chromosome. The segment size of the chromosome is equal to its selection probability. We generate a random number and select a chromosome whose segment spans the random number. In the same fashion, we can select N chromosomes by generating N random numbers to form $M(g)$.
Example 1.1. Consider $P(k)$ is consisted of six chromosomes. We encode each chromosome by five bits, and the fitness of the chromosome is the decimal value of its binary representation. Table 1.2 shows chromosomes' representation, their fitness scores and selection probability. The mapping of these chromosomes to contiguous segments of a line is shown in **Fig.**. We allocate the segment sizes to the chromosomes according their selection probabilities. It can be seen from **Fig.** that the chromosomes 1 and 6 got the smallest and largest segments, respectively.

To construct $M(k)$, we generate six numbers uniformly at random between 0.0 and 1.0. We call the generation of a random number as a trail. Assume six random numbers generated by six trails are 0.81, 0.32, 0.96,

Table 1.1: A typical population with six chromosomes, their fitness scores and selection probabilities.

Chromosome number	1	2	3	4	5	6
Chromosome representation	00011	01101	01000	00110	01010	10000
Fitness score	3	13	8	6	10	16
Selection probability	0.05	0.23	0.14	0.11	0.18	0.29

Table 1.2: A typical mating pool constructed from Table 1.2 and the fitness scores of its chromosomes.

Chromosome number	1	2	3	4	5	6
Chromosome representation	00011	01101	01000	00110	01010	10000
Fitness score	3	13	8	6	10	16
Selection probability	0.05	0.23	0.14	0.11	0.18	0.29

0.01, 0.65, and 0.42. **Figure** shows the position of these numbers in the line segments by their corresponding trail numbers. We have now understood that the chromosome 2 will have two copies in $M(k)$, and chromosomes 2, 3, 4 and 5 will have one copy of each in $M(k)$. Table 1.3 shows chromosomes in $M(k)$ and their corresponding fitness scores.

Table 1.3: Chromosomes of a population consists, their fitness and selection probability

Chromosome No.	1	2	3	4	5	6
Chromosome	0011	1101	1000	0110	0100	0001

The aforementioned sampling procedure is stochastic in nature and analogous to a game of chance popular in casinos and gambling, named after the French word *roulette* means "small wheel". The analogy to a roulette can be envisaged by imagining a small wheel in which each chromosome represents a pocket on the wheel; the size of the pocket is proportional to the chromosome's selection probability. Figure shows a roulette-wheel with pockets sized according to the probability of selection given in Table 1.2. By

spinning the roulette wheel N times, we can get N copies of chromosomes to construct $M(k)$. This is analogous to select N copies of chromosomes by generating N random numbers. This is why the stochastic sampling procedure with replacement is sometimes called the *roulette-wheel selection*. Although this sampling procedure provides a zero bias in selecting chromosomes, but does not guarantee a minimum spread.

One sampling procedure that provides zero bias and guarantees a minimum spread is the *stochastic universal sampling*. In stochastic universal sampling, the expected number of copies, $N(C_i)$ of a chromosome C_i to be in $M(k)$ is

$$N(C_i) = p_s(C_i).N = \frac{f_i}{\overline{f}} \tag{1.22}$$

The stochastic universal sampling procedure guarantees that the minimum and maximum number of copies of any chromosome are bounded by the floor and by ceiling of $f_i/\overline{f}$, respectively. As a result, the chromosomes with a above-average fitness ($f_i > \overline{f}$) tend to have more than one copies in $M(k)$, while they with a below-average fitness $f_i > \overline{f}$) tend to have no copies.

The stochastic universal sampling procedure maps the chromosomes of $P(g)$ to contiguous segments of a line like the stochastic sampling procedure. However, we here place N pointers that are equally spaced for selecting N chromosomes to construct $M(k)$. Since all pointers are equally spaced, the distance between any two pointers is $1/N$. This implies that we can place all pointers over the line if we know the position of the first pointer. In stochastic universal sampling, the position of the first pointer is a random number generated between 0 and $1/N$. The value of $N(C_i)$ is equal to the number of pointers that fall into the C_i's segment of the line. The graphical representation of the stochastic universal sampling is shown in **Fig.** We can think the stochastic universal sampling as a N-armed hand that is spun just once for selecting N chromosomes instead of single-armed hand of the stochastic sampling that is spun N times. This is the basic and only difference between these two sampling procedures.

1.2.5 Crossover

Crossover is a genetic operator that intends to simulate "sexual reproduction" of biological organisms in a simple way. This operator introduces variation in $M(k)$ by exchanging information between its chromosomes. The chromosomes that exchange information are called the *parents* and the new chromosomes that are produced due to exchange of information are called the *children* or *offspring*. The essence of crossover is the inheritance of information from parents to offspring, with the possibility that good parents may produce better offspring. Generally, crossover exchanges information between two parents, but multiple parents may be useful in some cases and produces two offspring, but, again, other numbers might be appropriate in some cases.

We first choose a pair of chromosomes from $M(K)$ randomly. These algorithms then generate a random number between 0 and 1. If this number is greater than or equal to the *crossover probability or crossover rate*, p_c, the chosen chromosomes exchange information in a specific way and produce two offspring chromosomes. We apply crossover to each pair of chromosomes in $M(g)$ with probability p_c. After the crossover operation, we replace the parent (chosen) chromosomes in $M(g)$ by their offspring offspring. The $M(g)$ has therefore been modified, but still maintained the same number of chromosomes. Since crossover plays a central role in genetic algorithms, P_c is set high, usually in the range of 0.6 to 0.95 (De Jong 1975; Schraudolph and Belew, 1992). In fact, crossover is considered to be one of genetic algorithms' defining characteristics, and it is one of the components to be borne in mind to improve the algorithms' behavior (Liepins et al., 1992).

There are many types of possible crossover operations that we can apply on two chosen chromosomes, say X and Y. The simplest one is the *one-point crossover* when we encode the chromosomes as a string of bits. Let X and Y can produce offspring according to p_c. What is the way to produce offspring by exchanging information between X and Y? In one-point crossover, we first generate a number, say r, between 1 and $L-1$ uniformly at random, where L is the length of X and Y. We refer to this number as the *crossing site*. We then produce two offspring by exchanging information between X and Y according to r and L. The first offspring consists of the first r bits of X and the last $L-r$ bits of the Y. The second offspring consists of the first

r bits of Y and the last $L-r$ bits of the X. We can generalize single-point crossover to k-point crossover, where $k > 0$. Given X and Y of length L. We generate k numbers, $r_1, r_2, \cdots, r_k$, between 1 and $L-1$ uniformly at random without repetition. We then produce offspring by exchanging information between X and Y through the crossing sties $r_1, r_2, \cdots, r_k$.
De. Jong, (1975). An analysis of the behavior of a class of genetic adaptive systems. PhD thesis, University of Michigan, Ann Arbor N. N. Schraudolph and R. K. Belew (1992). "Dynamic parameter encoding for Genetic Algorithms," Machine Learning, Vol. 9, pp. 9-21.
Example 1.1. Suppose the length $L = 8$. Consider the pair of parents X and Y is 00110000 and 11111111. Suppose the crossing site r is 4. Then, the crossover operation applied to the above parent chromosomes yields the two offspring 00111111 and 00111111. This operation is represented graphically in **Fig.**

Example 1.2. Suppose we have now two crossing sites at 3 and 7. Then, the crossover operation applied to the parents X and Y used in the previous example yields the two offspring 000111100 and 111000011. This operation is graphically represented in **Fig.**.

We usually apply k-point crossover on the binary coded chromosomes. However, we apply a different crossover when chromosomes are encoded as real-valued vectors and use the term "recombination" instead of crossover. A number of recombination operators have been proposed for genetic algorithms. Most recombination operators produce the genes of offspring via some form of combination of the parents' genes. Let us assume that $\mathbf{x} = (x_1, x_2, \cdots, x_n)$ and $\mathbf{y} = (y_1, y_2, \cdots, y_n)$ are two real-coded chromosomes selected for recombination, where $x_i, y_i \in (a, b) \subset \mathbb{R}$. We can produce an offspring $\mathbf{z} = (z_1, z_2, \cdots, z_n)$ as follows

$$z_i = x_i.a_i + y_i.(1 - a_i) \qquad i \in 1, 2, \cdots n \tag{1.23}$$

where a_i is a scaling factor chosen uniformly at random over an interval $[-d, 1 + d]$ for each z_i anew. The parameter d is specified by the user and defines the range of z_i. When the value of d is set to 0, z_i has the same range as the range of x_i and y_i. One problem with this value of d is that the range

for z_i may shrink over the generations because z_i may not be generated on the border of its range. This effect can be prevented by choosing a larger value for d. A value of $d = 0.25$ may ensure that the range of z_i is same as x_i and y_i. This kind of recombination is called sometimes *intermediate recombination*.

Example 1.3. Suppose we have chosen two real-coded chromosomes $\mathbf{x} = 123.5, 10.0, 22.5$ and $\mathbf{y} = 11.2, 27.0, 54.6$ for recombination. The value of d is set to 0.25. Since $\mathbf{x}$ and $\mathbf{y}$ are consisted of three variables, the offspring $\mathbf{z}$ will be consisted of three variables. Thus we need to generate three random numbers uniformly at random over an interval $[-0.25, 1.25]$ for the scaling factor a used in Eq.(1.23). Let these number are 0.2, -0.3, and 0.7. By Eq.(1.23), we get $z_1 =$, $z_2 =$ and $z_3 =$. The offspring Z is.

1.2.6 Mutation Operator

Mutation is also a variation operator applied to a single chromosome not to multiple chromosomes like crossover. Genetic algorithms use mutation after crossover on the chromosomes of $M(g)$. Two parameters involved in mutation: one is the *mutation probability* or *mutation rate*, p_m and one is the mutation size or the step size, δ. Mutation modifies randomly the alphabets (variables) of a chromosome based on the p_m.

Example 1.3. Let a chromosome $\mathbf{x} = (x_1, x_2, x_3, x_4, x_5)$, and $p_m = 0.02$. We like to produce an offspring $\mathbf{x}'$ from $\mathbf{x}$ by mutation. The most common way of implementing mutation involves generating one random number for each variable of $\mathbf{x}$. This number tells whether mutation will modify a particular variable. Since $\mathbf{x}$ is consisted of five variables, we generate five random numbers uniformly at random between 0 and 1. Let these numbers are 0.22, 0.01, 0.33, 0.51, and 0.40. Since the second number is within the range of p_m, mutation will modify x_2. Let mutation modifies x_2 to x_2', Thus $\mathbf{x}' = (x_1, x_2', x_3, x_4, x_5)$ is the offspring.

We use a different mutation for chromosomes with a different encoding. Mutation for the binary-valued chromosome is a lot simpler. We call such mutation as *bit mutation*. It is usually a bit-flipping operation; that is, we replace each bit of a chromosome from 0 to 1 or vice versa with probability p_m. We can also implement bit mutation without using the flipping operation. For instance, we can replace each bit of a binary-coded chromosome by

0 or 1 with equal probability, that is, with $p_m = 0.5$. The mutation size, δ, of the bit mutation is always one irrespective of the way of implementation. However, for a real-valued chromosome, there is no way to use the flipping or replacing operation in mutation and δ may not be one. What we can do is that we can add some real number based on δ, say Δx_i, with p_m to each variable x_i of a real-coded chromosome, $\mathbf{x}$. The most common way of getting Δx_i is the use of a distribution function, for instance, the Gaussian or Cauchy distribution function. That is,

$$\Delta x_i = \delta_i r \tag{1.24}$$

where r is a random number generated by the Gaussian or Cauchy distribution function.

Genetic algorithms use a very small p_m, usually in the range from 0.001 to 0.01 (De Jong 1975; Schraudolph and Belew, 1992) or $1/L$. Thus only a few chromosomes will undergo a change due to mutation, and of those that are mutated, only a few of the symbols are modified. The small p_m implies that mutation plays a minor role in the evolutionary process of genetic algorithms. The purpose of mutation in genetic algorithms is to allow the algorithms to avoid local minima by preventing the population of chromosomes from becoming too similar to each other that slows or even stops evolution. Consider we have a population of binary-coded chromosomes with $L = 10$. Assume the first bit of each chromosome in the population becomes one after several generations of an evolutionary process. However, we need a solution that requires a zero in the first bit of any chromosome. Can we get such a solution by applying only crossover again and again to the chromosomes of the population? The answer is not affirmative. Since mutation can change randomly any bit of a chromosome, we may get a zero in the first bit position by applying mutation again and again. Holland (1975) argues that mutation is what prevents the loss of variability at a given bit position.

1.2.7 Training network using genetic algorithms

So far we have concentrated on the basic ideas and components of genetic algorithms. It is time to look the employment of such algorithms into neural

networks. We here consider how genetic algorithms can be employed for training neural networks to learn suitable mappings from a given data set. As in previous training methods, learning will be based on an error function, which we like to minimize by iteratively adjusting the weights and biases of a neural network.

The first step of using genetic algorithms, as we have already seen, is the creation of a population of chromosomes. The chromosomes constitute a search space on which genetic algorithms work to find a optimal or near optimal solution of a given problem. Since the objective of a neural network training is to find a suitable set of weights, a chromosome must contain information (the value) of such weights. A straightforward way of constructing the chromosome is simply the concatenation of all the network's weights in a string. Do we need to concatenate the weights in a specific way or random fashion? As an illustration of the concatenation issue, consider a three layered feed-forward neural network with three input, two hidden and two output neurons. Let we construct two chromosomes, say X and Y, by randomly concatenating the weights of this network. Since X and Y encode all the weights of the same network, we use a different value to encode a same weight into X and Y. Let a_{ij}^l and b_{ij}^l denote the value of the same weight W_{ij}^l encoded into X and Y, respectively. **Figure** shows the graphical representation of the aforementioned network and two chromosomes.

What happen if we apply the single point crossover on X and Y. Let the crossing site is 7 and crossover produces two offspring, say X' and Y'. The offspring X' consists of the first seven weights of X and the last weights of Y, while Y' consists of the first seven weights of Y and the last weight of X. The graphical representations of such offspring are shown in **Fig. Y**. It is clear that the parent networks, that is, the networks with the weights encoded in X and Y, will have a very different functionality compared to the offspring networks, that is, the networks with the weights encoded in X' and Y', although crossover exchanges only one weight. As for example, a_{22}^2 is the value of the weight variable W_{22}^2 connecting the hidden neuron 2 to the output neuron 2. Since the network shown in **Fig.** can be used for two-class problems, a_{22}^2 will help to turn on and off the output neuron 2 for a training example belongs to class 2 and 1, respectively. However, the crossover exchanges u_{22}^2 with v_{11}^2, which have a opposite functionality. We have now understood that the crossover may change the functionality

of a network drastically when we form the chromosomes of a population by randomly concatenating the weights of a neural network. Besides the disruptive effect of crossover, there is another prominent problem is associated with crossover. This problem is called the *permutation* or *competing convention* problem, which we shall described in the next subsection.

We can avoid the disruptive effect of crossover by concatenating the network's weights in a specific way for all chromosomes of a population. One such way is concatenating the weights based on the processing units (hidden neurons) of the network. Let we form a chromosome in such a way that all incoming and outgoing weights of a hidden neuron are placed together closely in the coding representation of the chromosome. Specifically, for a particular hidden neuron, we first concatenate its all incoming weights then its all outgoing weights. If a network is consisted of n hidden neurons, we concatenate the weights of the first hidden neuron at beginning, then the second hidden neuron and so on. Let p and q are two chromosomes that we form as described above for our network shown in **Fig.**. The graphical representation of p and q is shown in **Fig.**. We again here consider the same crossing sit 7 for producing offspring. In this case, crossover produces two offspring, say p' and q', by exchanging a_{22}^2 of p with b_{22}^2 of q. We can expect that when crossover exchanges one weight, that is, a_{22}^2 to b_{22}^2 (or vice versa), the parent and offspring networks will have a similar functionality. This expectation is reasonable in the sense that both a_{22}^2 and b_{22}^2 represent the value of the same weight variable W_{22}^2 connecting hidden neuron 2 to output neuron 2. So we can say that since the single point crossover operator is more likely to disrupt genes that are far apart on a chromosome than to disrupt genes located close to each other, it is useful to place functional units of a neural network tightly together on a chromosome.

After deciding the way of forming chromosomes, we need to decide a coding scheme to encode the weights. The decision is basically the choice between the binary-valued and real-valued encodings. While standard genetic algorithms use the former encoding scheme, the later one seems to be beneficial here because the weights of a neural network are real numbers in general. The binary encoding results a large string (a large sized chromosome), specifically for a neural network with many weights. Let we use c and d bits to encode the integer part and decimal part of a weight, respectively. If the network has k weights, the size of a chromosome will be

$k\,(c+d)$ bits. The value of k is dependent on the number of neurons and their connectivities in the network. We shall describe various methods in chapters 3-5 to determine automatically a near optimal number of neurons, specifically hidden neurons, for a given problem. The value of c and d is dependent on the weights' size that the network requires to solve the problem. However, there is no way to find the weights' size in advance although we need to do so for training the network using standard genetic algorithms. A large value for c and d increases the chromosome's size, eventually the search space size. Thus genetic algorithms will take more time in finding a suitable set of weights. On the other hand, a small value for c and d reduces the search space size. Let we choose the value of c as 5 and d as 2. The weight's value in such a setting can lie in the range of **-15** to **+15**. Genetic algorithms do not able to find optimal weights if they do not lie in the aforementioned range. We have now understood that the binary encoding not only increases the search space size but also restrict it. This bottle neck make an evolutionary process inefficient when standard genetic algorithms are applied for training neural networks. So we can directly represent the weights in the chromosome as real numbers. The chromosome in this case will be a vector of real numbers.

Once we have chosen the way to form a chromosome and the way to represent the weights in the chromosome, we can create an initial population $P(0)$ consisting of n chromosomes. We then need a fitness function to measure the quality of each chromosome in the population. Since the chromosome contains all the network's weights, we can measure the chromosome's quality in terms of the network error for the training data. As mentioned before, the goal of training the network is to minimize its training error. The training error, therefore, does not directly translate into a fitness score which have to be the larger the better the chromosome. We can easily transform the error by using $1/e$ or $1/\,(1+e)$ as fitness score. We have already seen that crossover and mutation do not require any gradient information to modify the chromosomes, here the network's weights because the chromosomes encode such weights. We, therefore, can also use non-differentiable functions like the percentage of correct answers (classification accuracy) on the training set as a fitness function. However it is advantageous if the performance measure is continuous not just discrete because this allows genetic algorithms to better discriminate between the

performance of different chromosomes. Finally, we have to decide the type of selection, crossover and mutation to be used for evolution. Although we can use any selection scheme, the choice of crossover and mutation is dependent on the encoding scheme of chromosomes. As mentioned earlier, the real-coded chromosomes are suitable for training a neural network. So we can use intermediate crossover and Gaussian or Cauchy mutation.

1.2.8 Permutation Problem

The Permutation problem may arise when genetic algorithms use two representations: one for evolutionary search, specifically for crossover and one for fitness evaluation. The source of this problem is the mapping of many genotypes (chromosomes) to one phenotype (function). An encoding scheme that allows different genotypes for one phenotype is responsible for the permutation problem. One such encoding is the binary encoding. To understand this problem clearly, consider two neural networks that have the same number of input, and output neurons. We also consider the same number of hidden neurons for these neurons but the hidden neurons are shuffled. Let we use five bit to encode each weight of the networks. **Figure 2** shows the networks and their genetic representations.

Figure 2 essentially depicts a many-to-one mapping from genotype to phenotype as two networks have a different genotype representation but they are same in terms of functionality. In fact, all permutations over the set of hidden node indices, (H1, H2, H3), are equivalent in terms of the network functionality. This is because, in a purely feed-forward neural network, rearranging the order of the hidden neurons has no effect on the network functionality. That is, the same network have many different genotype representations. Given a fully connected feed-forward neural network with N hidden neurons, there will be $N!$ symmetries and up to $N!$ equivalent solutions. Genetic algorithms apply search operators, crossover and mutation, on the genotype space (representation). The redundancy in the genotype space will make searching inefficient.

The permutation problem not only creates redundancy in the search space but also creates search basis towards *frequent phenotypes* and ineffectiveness in the crossover operation. Let us assume that G and S denote the genotype and phenotype space, respectively. **Igel ()** defines $r(s)$, the

redundancy of a phenotype $s \in S$, as the number of genotypes that map on to s. The average redundancy, $\overline{r}$, can be defined as

$$\overline{r} = \frac{1}{|S|} \sum_{s \in S} r(s) = \frac{|G|}{|S|} \tag{1.25}$$

A phenotype can be considered as the *frequent phenotype* or the *rare phenotype* if and only if $r(s) > \overline{r}$ or $r(s) < \overline{r}$. We can now define fraction of frequent phenotype as the ratio of the number of frequent phenotypes to the total number of phenotypes in the phenotype space. Fraction of frequent genotype can be defined as the ratio of the number of genotypes that map into a frequent phenotype to the total number of genotypes in the genotype space. Frequent phenotypes arise only when an encoding scheme suffers from permutation problem where different genotypes map into one phenotype introduces the redundancy of phenotypes to appear. A network with a increasing number of hidden neuron nodes the fraction of frequent genotypes increases and fraction of frequent phenotypes decreases. Therefore, an unbiased search operator in the genotype space is heavily biased in the phenotype space and the bias is specifically towards frequent phenotypes.

Effectiveness genetic algorithms greatly depend on the efficacy of crossover. We usually apply crossover between fitter parents to produce offspring by exchanging values of their lined up genes. This lining up is particularly of the foremost importance for genetic algorithms because without identifying and lining up the building blocks (genes) crossover will be meaningless and may result weaker offspring. Thus, the genetic process may eventually turn into a random walk. A proper lining up of the genes can be verified by applying crossover on two parents that have different genotypes but have a identical phenotype. Consider two such parents which are shown in (**Fig.**). What happen if we apply crossover in the dotted line position of the parents (**Fig.**). The offspring duplicate some elements of the parents and omit others.

two identical phenotypes. If the combination of encoding and recombination operator can identify which genes line up with which, the operation should result into the same individual. In case of different parent phenotypes, this effective lining up will result in values exchanges of comparable building blocks.

Due to permutation problem, this lining up of phenotypes becomes a strenuous task. In many cases, it is probably not possible. Schemes that cannot assist such lining up will result into futile recombination.
Example 1.4.

1.2.9 Evolutionary Programming

Evolutionary programming is also a population based computation model. It was first proposed by Lawrence J. Fogel in the mid 1960 as one way to achieve artificial intelligence through simulated evolution. In his work, Fogel applied evolutionary programming to the evolution of finite state machines. Mutation operators were used to create new solutions (finite state machines) that were being evolved for specific tasks. Evolutionary programming was dormant for many years. Later, in the late 80's, David B. Fogel reintroduced it to solve more general tasks. The pseudo code of evolutionary programming is given in *Algorithm.*

There are a couple of conceptual ideas associated with evolutionary programming. First, evolutionary programming works on the phenotype space unlike genetic algorithms that work on the genotype space. A philosophical tenant of evolutionary programming is that a search operator should act as directly as possible on the phenotype space to change the behavior of a system. Thus evolutionary programming uses a representation scheme typically tailored to the problem domain for encoding information in a chromosome. Second, evolutionary programming does not support exchanging information between chromosomes of a population. This computation model, therefore, does not apply crossover or recombination on the chromosomes of a population to produce offspring, but rather applies only mutation. This is the basic difference between evolutionary programming and genetic algorithms. Crossover is considered unnecessary in evolutionary programming and evolutionary computation in general!

Any kind of mutation can be used.

Let $P(k)$ is consisted of N chromosomes. Evolutionary programming applies mutation on each chromosome of $P(k)$ and produces N offspring. We can use bit **or** mutation for the binary valued chromosomes, while Gaussian or Cauchy mutation for the real valued chromosomes. As seen from **Fig. ,,** the choice of standard deviation is very important in Gaussian distribution.

Evolutionary programming, therefore, puts emphasis on the optimal value of standard deviation when it uses the Gaussian mutation. This emphasis is reasonable because mutation is the primary and only search operator in evolutionary programming. Do we use the same optimal standard deviation for all problems? Unfortunately, the optimal standard deviation is problem dependent as well as dimension dependent. Even for the same problem the optimal standard deviation is not same in the whole evolutionary process. A large standard deviation is suitable at the beginning of an evolutionary process because it allows a large jump that is beneficial for exploring a search space. On the other hand, a small standard deviation is suitable at the end of the process to facilitate exploitation (fine tuning). How can we get an optimal standard deviation at different stages of the evolutionary process? One way to achieve it is to evolve the standard deviation along with the chromosomes of $P(k)$. Evolutionary programming does the same thing exactly. It includes the standard deviation as part of an chromosome so that it can evolve automatically. Let we like to minimize $f(\mathbf{x})$, $\mathbf{x} \in \mathbb{R}^d$. The representation of a population of chromosomes to solve $f(\mathbf{x})$ is $(\mathbf{x}_i, \sigma_i)$, $\forall i \in \{1, \cdots, N\}$. In many implementations of evolutionary programming, σ_i is mutated first and then $\mathbf{x}_i$ using the new σ_i'. That is, for each parent $(\mathbf{x}_i, \sigma_i), i = 1, \cdots, N$, the Gaussian mutation produces a single offspring $(\mathbf{x}_i', \sigma_i')$ by: for $j = 1, \cdots, d$,

$$\sigma_i'(j) = \sigma_i(j) \exp\left(\tau' N(0,1) + \tau N_j(0,1)\right) \tag{1.26}$$

$$x_i'(j) = x_i(j) + \sigma_i'(j) N_j(0,1) \tag{1.27}$$

where $x_i(j)$, $x_i'(j)$, $\sigma_i(j)$ and $\sigma_i'(j)$ denote the j-th component of the vectors $\mathbf{x}_i$, $\mathbf{x}_i$, σ_i and σ_i, respectively. $N(0,1)$ denotes a normally distributed one-dimensional random number with mean 0 and standard deviation 1. $N_j(0,1)$ indicates that the random number is generated anew for each value of j. The factors τ and τ' have commonly set to $\left(\sqrt{2\sqrt{d}}\right)^{-1}$ and $\left(\sqrt{2d}\right)^{-1}$ (Bck and Schwefel, 1993; Fogel, 1994).

Third, evolutionary programming traditionally uses the tournament selection to construct $P(k+1)$ from all parents and offspring. Of course, from an algorithmic point of view, there is no reason why evolutionary programming can not use other selection schemes. However, the aim of the selection

scheme in evolutionary programming is different from that in genetic algorithms, which uses the selection scheme to form a mating pool for crossover. In the tournament selection scheme, as the name suggests, we arrange a pairwise competition over the union of parents and offspring. For each chromosome, we choose s opponents from the union uniformly at random. The chromosome receives a "win" if its fitness is better than an opponent. Since the competition is done in pairwise, the chromosome can receive at most s wins. We select N chromosomes out of N parent chromosomes and N offspring chromosomes that have the most wins to form $P(k+1)$. The most important aspect tournament selection is the parallelization as each tournament is independent. In the best-case scenario if we have the same number of processors as twice the population size all tournaments can be run simultaneously. However, the tournament size s introduces a sampling bias into the selection process. A larger s correspond to higher probability of the most fit chromosome being selected relative to the other chromosomes. Sokolov and Whitley (2005) review analytic results and present empirical evidence that shows this bias has a significant impact on search performance.

1.3 *Hint Evolutionary Programming*

The representations used in evolutionary programming are typically tailored to the problem domain [SJBF93]. One representation commonly used is a fixed-length real-valued vector. The primary difference between evolutionary programming and the other approaches (GA, GP, and ES) is that no exchange of material between individuals in the population is made. Thus, only mutation operators are used. For real-valued vector representations, evolutionary programming is very similar to evolutionary strategies without recombination.

they differ mainly in their representations of potential solutions and their operators used to modify the solutions.

There are a couple of conceptual ideas that are closely associated with evolutionary programming. First, evolutionary programming does not use recombination and there is a general philosophical stance that recombination is unnecessary in evolutionary programming and in evolutionary computation in general! The second idea is related to the first. Evolutionary programming is viewed as working in the phenotype space whereas genetic

algorithms are seen as working in the genotype space. A philosophical tenant of evolutionary programming is that operators should act as directly as possible in the phenotype space to change the behavior of a system. Genetic algorithms, on the other hand, make changes to some encoding of a problem must be decoded and operationalized in order for behaviors to be observed and evaluated. Sometimes this (partially philosophical) distinction is clear in practice and sometimes it is not.

The term evolutionary programming dates back to early work in the 1960's by L. Fogel [16]. In this work, evolutionary methods were applied to the evolution of finite state machines. Mutation operators were used to alternate finite state machines that were being evolved for specific tasks. Evolutionary programming was dormant for many years, but the term was resurrected in the early 1990s. The new evolutionary programming, as reintroduced by D. Fogel, [15, 14, 13], is for all practical purposes, nearly identical to an evolution strategy. Mutation is done in a fashion that is more or less identical to that used in evolution strategies. A slightly different selection process (a form of Tournament Selection) is used than that normally used with evolution strategies, but this difference is not critical. Given that evolution strategies go back to the 1970's and predate the modern evolutionary programming methods by approximately 20 years, there appears to be no reason to see evolutionary programming as anything other than a minor variation on the well-established evolution strategy paradigm. Historically, however, evolution strategies were not well known outside of Germany until the early 1990's and evolutionary programming has now been widely promoted as one branch of Evolutionary Computation.

There arc a couple of conceptual ideas that are closely associated with evolutionary programming. First, evolutionary programming does not use recombination and there is a general philosophical stance that recombination is unnecessary in evolutionary programming and in evolutionary computation in general! The second idea is related to the first. Evolutionary programming is viewed as working in the phenotype space whereas genetic algorithms are seen as working in the genotype space. A philosophical tenant of evolutionary programming is that operators should act as directly as possible in the phenotype space to change the behavior of a system. Genetic algorithms, on the other hand, make changes to some encoding of a problem must be decoded and operationalized in order for behaviors to be observed

and evaluated. Sometimes this (partially philosophical) distinction is clear in practice and sometimes it is not.

Evolutionary Programming (EP) is presented by L. J. Fogel [FOW66]. He initially studied this method to develop the artificial intelligence and succeeded in evolving a mathematical automaton that predicts a binary time series. Later, in the middle of 80s, his son David Fogel further developed it to solve more general tasks including prediction problems, optimization, and machine learning [Fog95]. Since this approach modeled organic evolution at the level of evolving species, the original EP does not rely on any kind of recombination.

The representations used in evolutionary programming are typically tailored to the problem domain [SJBF93]. One representation commonly used is a fixed-length real-valued vector. The primary difference between evolutionary programming and the other approaches (GA, GP, and ES) is that no exchange of material between individuals in the population is made. Thus, only mutation operators are used. For real-valued vector representations, evolutionary programming is very similar to evolutionary strategies without recombination.

A typical selection method is to select all the individuals in the population to be the N parents, to mutate each parent to form N offspring, and to probabilistically select, based upon fitness, N survivors from the total 2N individuals to form the next generation.

In general, each parent generates an offspring by the Gaussian mutation and better individuals among parents and offspring are selected as parents of the next generation. EP has been applied successfully for the optimization of the real-valued functions [FA90] [FS94] and other practical problems [SS]. For more details see [BRS93].

Evolutionary programming is one of the four major evolutionary algorithm paradigms.

It was first used by Lawrence J. Fogel in 1960 in order to use simulated evolution as a learning process aiming to generate artificial intelligence. Fogel used finite state machines as predictors and evolved them.

Currently evolutionary programming is a wide evolutionary computing dialect with no fixed structure or (representation), in contrast with some of the other dialects. It is becoming harder to distinguish from evolutionary strategies. Some of its original variants are quite similar to the later genetic

programming, except that the program structure is fixed and its numerical parameters are allowed to evolve.

Its main variation operator is mutation; members of the population are viewed as part of a specific species rather than members of the same species therefore each parent generates an offspring, using a (+) survivor selection

1.4 Hint for Genetic Algorithms

Mimicking biological evolution and harnessing its power for adaptation are problems that have intrigued computer scientists for at least four decades. Genetic algorithms (GAs), invented by John Holland in the 1960s, are the most widely used approaches to computational evolution. In his book Adaptation in Natural and Artificial Systems (Holland, 1992, also reviewed in this issue), Holland presented GAs in a general theoretical framework for adaptation in nature. Hollands motivation was largely scientific he was attempting to understand and link diverse types of natural phenomenabut he also proposed potential engineering applications of GAs. Since the publication of Hollands book, the field of GAs has grown into a significant sub-area of artificial intelligence and machine learning. Nowadays one can find several international conferences each year as well as a number of journals devoted to GAs and other evolutionary computation approaches. Research on GAs has spread from computer science to engineering and, more recently, to fields such as molecular biology, immunology, economics, and physics. One result of this growth in interest has been a division of the field of GAs into several subspecies. One major division is between research on GAs as engineering tools and research

1.5 Hint

In Chapter 5 backpropagation was introduced. Backpropagation is a very effective means of training a neural network. However, there are some inherent flaws in the back propagation training algorithm. One of the most fundamental flaws is the tendency for the backpropagation training algorithm to fall into a local minima. A local minimum is a false optimal weight matrix that prevents the backpropagation training algorithm from seeing the true solution.

The most widely used method for supervised training of multilayer perceptrons turns out to be the back-propagation algorithm, which retrieves the desired weight matrix by optimizing a proper cost function. Unfortunately, the steepest descent technique, normally employed in the minimum search, is often slowed down by the presence of flat regions and by the high number of local minima in the surface of the considered cost function. Then, it can be convenient to adopt alternative training methods that ensure the achievement of the global optimum, such as the simulated annealing, widely used in the solution of complex problems, or the genetic algorithms, which perform the optimization process by following an approach similar to that employed in the evolution of living beings.

The different characteristics of these two global optimization techniques have been combined to obtain a new training method that maintains the reliability of simulated annealing and the efficiency of genetic algorithms, reducing at the same time their major drawbacks. The Interval Genetic Algorithm (IGA) represents a natural extension of classic genetic algorithm to the continuous gene setting [14]. Its ability of reaching the global minimum of a cost function is considerably better than that showed by simulated annealing and, in addition, the convergence speed of IGA turns out to be on the average 10 times higher.

Annealing, as mentioned , is a method used to toughen materials. Its importance is that it prevents the creation of defects in the atomic scale.

Defects like vacancies, misplacement etc. can really hurt the strength of the material. For example, in this glass factory (taken from "bullseye glass factory" homepage) it is extremely important to anneal the glass before it is ready or else it will be very fragile and even removing it from its mold will be impossible. and I quote:

"Annealing, the controlled cooling of a glass, is critical to its longevity. Glasses which are not properly annealed will contain stress which may result in breakage before or at any time subsequent to their removal from the kiln. "

So how do we anneal?

Actually the process is very simple in concept. you just need to cool down over a relatively long period of time with frequent reheating. (as oppose to quenching which means rapid cooling.) the reheating is meant to prevent the defects (which are considered mathematically as a local minima)

and stabilized the material at its optimized position.

more on the math behind the process can be read at Dr Joan Adler and Amihay Silverman's simulated annealing page

And what can we do with it?

Well , besides its physical application, annealing is also used as an optimization tool. How is this made? our problem is finding the global optima of the function. the function is evaluated at each point and then and the annealing process begins. any downhill step is accepted and the process repeats from this new point. An uphill step may be accepted as it can escape from local optima. This uphill decision is made by the Metropolis criteria as described here(this is taken from the "MALLBA project" homepage) . Of course that by using this criteria we have some assumptions made on the function which must be obeyed.

usages:

glass making, ice cream making, mathematical optima problems, strength of materials etc.

a nice 1 dimensional annealing process can be viewed here . (this is taken from the "Taygeta scientific inc" page)

more about annealing can be read in the links section.

Simulated annealing is a generalization of a Monte Carlo method for examining the equations of state and frozen states of n-body systems [Metropolis et al. 1953]. The concept is based on the manner in which liquids freeze or metals recrystalize in the process of annealing. In an annealing process a melt, initially at high temperature and disordered, is slowly cooled so that the system at any time is approximately in thermodynamic equilibrium. As cooling proceeds, the system becomes more ordered and approaches a "frozen" ground state at T=0. Hence the process can be thought of as an adiabatic approach to the lowest energy state. If the initial temperature of the system is too low or cooling is done insufficiently slowly the system may become quenched forming defects or freezing out in metastable states (ie. trapped in a local minimum energy state). The original Metropolis scheme was that an initial state of a thermodynamic system was chosen at energy E and temperature T, holding T constant the initial configuration is perturbed and the change in energy dE is computed. If the change in energy is negative the new configuration is accepted. If the change in energy is positive it is accepted with a probability given by the Boltzmann factor

exp -(dE/T). This processes is then repeated sufficient times to give good sampling statistics for the current temperature, and then the temperature is decremented and the entire process repeated until a frozen state is achieved at T=0.

By analogy the generalization of this Monte Carlo approach to combinatorial problems is straight forward [Kirkpatrick et al. 1983, Cerny 1985]. The current state of the thermodynamic system is analogous to the current solution to the combinatorial problem, the energy equation for the thermodynamic system is analogous to at the objective function, and ground state is analogous to the global minimum. The major difficulty (art) in implementation of the algorithm is that there is no obvious analogy for the temperature T with respect to a free parameter in the combinatorial problem. Furthermore, avoidance of entrainment in local minima (quenching) is dependent on the "annealing schedule", the choice of initial temperature, how many iterations are performed at each temperature, and how much the temperature is decremented at each step as cooling proceeds.

1.6 Hint for Mathematical Notations

$N = \{v \in \mathbb{R} / v * \sqrt{2} \in \mathbb{N}\}$

$\lim_{x \to \infty} f(x) = 0$