Master Thesis

Combined Neural Networks and Genetic Algorithms as a method for reducing redundancy in steel design

Joo, Min Sung (朱 敏 成)

Department of Ferrous Technology

(Computational Metallurgy)

Graduate Institute of Ferrous Technology

Pohang University of Science and Technology

2008

Combined Neural Networks and Genetic Algorithms
as a method for reducing redundancy in steel design

2008 Min Sung Joo

신경회로망과 유전알고리즘의 결합을 통한 철강 디자인의 중복성 감소

Combined Neural Networks and Genetic Algorithms as a method for reducing redundancy in steel design

Combined Neural Networks and Genetic Algorithms as a method for reducing redundancy in steel design

by

Joo, Min Sung
Department of Ferrous Technology
(Computational Metallurgy)
Graduate Institute of Ferrous Technology
Pohang University of Science and Technology

A thesis submitted to the faculty of Pohang University of Science and Technology in partial fulfillments of the requirements for the degree of Master of Science in the Graduate Institute of Ferrous Technology (Computational Metallurgy)

Pohang, Korea
June 23th, 2008

Approved by

Prof. Lee, Hae Geon

Major Advisor

Prof. Bhadeshia, H. K. D. H.

Co-Advisor

Combined Neural Networks and Genetic Algorithms as a method for reducing redundancy in steel design

Joo, Min Sung

This dissertation is submitted for the degree of Master of Science at the Graduate Institute of Ferrous Technology of Pohang University of Science and Technology. The research reported herein was approved by the committee of Thesis Appraisal

June 23th, 2008

Thesis Review Committee

Chairman: Prof. Lee, Hae Geon
(signature)

Member: Prof. De Cooman, B. C.
(signature)

Member: Prof. Kim, In Gee
(signature)

Preface

This dissertion is submitted for the degree of Master of Engineering in Computational Metallurgy at Pohang University of Science and Technology. The research described herein was conducted under the supervision of Professor H. K. D. H. Bhadeshia, Adjunct Professor of Computational Metallurgy in the Graduate Institute of Ferrous Technology, Pohang University of Science and Technology and Professor of Physical Metallurgy, University of Cambridge, between September 2006 and June 2008.

Except where acknowledgement and reference is made to previous work, this work is, to the best of my knowledge, original. Neither this, nor any substantially similar dissertation has been, or is being, submitted for any other degree, diploma or other qualification at any other university.

Part of this work has been submitted to appear in the following publication:

Minsung Joo, Joohyun Ryu and H. K. D. H. Bhadeshia: Domains of Steels with Identical Properties, submitted to *Materials and Manufacturing Processes*.

Min Sung Joo

June 2008

Acknowledgement

I am extremely greateful to my supervisor, Professor Bhadeshia, H. K. D. H. for his constant guidance, support, great friendship and for his insight. Also thanks to Professor Hae-Geon Lee, Professor B. C. De Cooman, Professor Weijie Liu, Professor In Gee Kim and Professor Rongshan Qin for their advice, support and friendship.

I would like to express my thanks to all the people in the Graduate Institute of Ferrous Technology (GIFT) in Pohang University of Science and Technology, especially those members in Computational Metallurgy Laboratory (CML), for all their help and for all the memories we have. The life with CML members was quite pleasant and enjoyable. Especially, I gratefully acknowledge Joo Hyun Ryu for his helps on the neural network analysis.

I grateful acknowledge Drs Jae Kon Lee, Young Roc Im and Jung Hyeung Lee from Sheet Products and Process Research group of the steel company, POSCO, for their helps with the neural network analysis

I would like to thank Yong Ki Kim, Jae Yong Chae and You Young Song for help and friendship during being my stay in Cambridge.

Finally, I would like to take this opportunity to express my gratitude to my family members for their love, unfailing encouragement and support.

MFT 20062870
Joo, Min Sung
Combined Neural Networks and Genetic Algorithms as a method for reducing redundancy in steel design, Department of Ferrous Technology (Computational Metallugy) 2008
Advisor: Prof. Lee, Hae Geon; Prof. Bhadeshia, H. K. D. H.
Text in English

Abstract

The mechanical properties of ferrite-pearlite steels can be affected by many factors. Material scientists have conducted many trials to find new steel compositions, in an attempt to optimize and achieve properties. The data are generally analyzed using linear regression methods. However, they do not reveal the full complexity of the mechanical properties of steels.

This work begins by reviewing neural networks, the genetic algorithm method and the essential factors and hardening mechanisms that affect the mechanical properties of steels. The objective of this work was to find identical domains of input parameters which lead to a particular strength or ductility for the hot-rolled steels, all with a final microstructure which is a mixture of ferrite and pearlite. This can in principle be done by combining the neural network with a genetic algorithm method. The neural network is one of the most general ways of developing quantitative relationships when dealing with complex problems. It can find the relationship between the large number of variables, which determine the properties of the steel and the properties themselves. However, the neural network requires the user to

reach the desired solution by a trial and error choice of inputs, like many other modeling techniques. This difficulty can be resolved by combining the model with a genetic algorithm which can search the input space simply and efficiently. The combination can allow the user to set the objectives and let the genetic algorithm discover the domain of inputs which can automatically satisfy the desired outputs.

During this work, two neural network models were selected from Ryu's work which can predict the ultimate tensile strength and the elongation to failure respectively [Ryu, 2008], and combined with a genetic algorithm. The combined models with single objective were tested to find the domains of inputs which satisfy particular strength or ductility targets and the results were interpreted in a metallurgical context. The combined models with a single objective were then gathered to make another combined model with two objectives, the ultimate tensile strength and the elongation to failure. This model was tested to find the domains of inputs which simultaneously satisfy the two targets and analyzed to find new grades of steels which have high tensile strength and high ductility.

Contents

PREFACE I

ACKNOWLEDGEMENT II

ABSTRACT III

CONTENTS V

NOMENCLATURE VII

I. INTRODUCTION – LITERATURE REVIEW 1

1.1. Mechanical properties 2
1.1.1. Fundamentals 4
1.1.2. Mechanisms for strain hardening 10

1.2. Neural network 15
1.2.1. Fundamentals 16
1.2.2. Error estimation 18
1.2.3. Overfitting 21
1.2.4. Committee 23
1.2.5. Significance 24

1.3. Genetic algorithm 24
1.3.1. Process and operators 25
1.3.2. Potential pitfalls 33
1.3.3. Genetic algorithms for Bayesian neural networks 35
1.3.4. Multi-objective optimization 37

II. MODELING AND COMPUTER EXPERIMENTS 41

2.1. Neural network modeling 41

2.2. Genetic algorithm 46
2.2.1. Single objective 46
2.2.2. Multiple objective 50

2.3. Simulations for each combined model 51

2.4. Summary 53

III. RESULT AND DISCUSSION 54

3.1. Single objective model 56
3.1.1. The ultimate tensile strength model 56
3.1.2. The elongation model 69

3.2. Multiple objective model 75
3.2.1. 400 MPa strength in combination with elongation 75
3.2.2. 600 MPa strength in combination with elongation 76
3.2.3. 800 MPa strength in combination with elongation 78

3.3. Summary 80

IV. SUMMARY AND FUTURE WORK 81

4.1. Summary 81

4.2. Future work 83

REFERENCES 85

APPENDIX A 92

APPENDIX B 111

APPENDIX C 130

CURRICULUM VITAE 151

Nomenclature

ANN	Artificial neural network
EL	The elongation
GA	Genetic algorithm
HS	The hardness from Brinell hardness test
IHPE	Inverse of Hall-Petch effect
SPO	Strong Pareto optimal
TS	The tensile strength
T_C	Coiling temperature
T_{FR}	Finish-rolling temperature
UTS	The ultimate tensile strength
WPO	Weak Pareto optimal
A_0	The original cross-sectional area
C_i	I-th criteria in multi-objective genetic algorithm
E	The modulus of elasticity, Young's modulus
E_D	The overall error in neural networks
F	The instantaneous force or load
G	The shear modulus
P_i	The probability of being selected for individual i in genetic algorithm
U_r	The modulus of resilience
W_{Mn}	The contents of manganese in weight percent

W_{Nf}	The contents of free nitrogen in weight percent
W_{Si}	The contents of silicon in weight percent
b	The burgers vector
d_α	The ferrite grain size in millimeters
f_i	The fitness of *i*-th chromosome in genetic algorithm
f_{max}	The maximum fitness in genetic algorithm
$\overline{f}$	The average fitness in genetic algorithm
h_i	Hyperbolic tangents in neural networks
l_0	The original length
l_i	The instantaneous length
p	The dislocation density
t_j	The measured value
x_j	Input variables in neural networks
w_i	Weights in neural networks
w_{ij}	Weights in neural networks
y	Output in neural networks
z_i	The arguments of hyperbolic tangents in neural networks
γ	The shear strain
ε	The tensile elongation to failure
$\theta^{(1)}$	Biases in neural networks
$\theta^{(2)}$	Final bias in neural networks
σ	The engineering stress

σ_{U}	Ultimate tensile strength
σ_{Y}	Yield strength
τ	The shear stress
τ_{0}	The intrinsic strength of the material

I. Introduction – Literature review

Steels used in industry are designed to particular mechanical properties, which in turn are influenced by the thermo-mechanical processing and the chemical composition. Hot-rolled steels with a microstructure consisting of a mixture of ferrite and pearlite form the backbone of the steel construction industry because of the unique combination of strength, toughness and cost that they offer. They are also easy to fabricate using processes such as welding. It is possible that a reduction in the variety of steels available could result in greater economies in the production process.

There have been many attempts in the past to find connections between the mechanical properties and its controlling variables. Normally, this involves a well-designed series of experiments, but this is in principle unnecessary because a vast quantity of data already exists in the form of research, production and quality-control experiments. Neural networks and associated tools are useful in exploring such relationships without introducing bias by using the data.

Once a phenomenon has been modeled, the user has to reach the desired solution by a trial and error choice of inputs. This involves making educated guesses on what inputs to use in order to reach the target value of the output parameter. The non-linear nature of the model means that a very large number of trials is needed. This problem is better handled by combining the model with a genetic algorithm which can efficiently search the input space [Chakraborti, 2004; Delorme, 2003]. The

power of doing this has recently been demonstrated in reaching novel solutions out of the field of established knowledge, leading to the discovery of the so-called δ-TRIP steel [Chatterjee, 2006; Chatterjee *et al*., 2007].

1.1. Mechanical properties

When in service, steels are variously loaded. Deformation may then occur and this can result in fracture. Thus it is necessary know the characteristics of the steels and fitness for purpose. An understanding of the mechanical properties is therefore the basis for the understanding of the steels. Important mechanical properties include strength, hardness, ductility and stiffness. Thus, in the experiments, factors which can affect the results have to be carefully controlled. The basic mechanical behavior can be derived with simple stress-strain tests, in tension, compression, torsion or shear.

In tension or compression tests, a specimen is tested to an increasing uniaxial force at uniform speed with the simultaneous observation of elongation or contraction of the specimen. In this case, engineering stress σ defined by the relationship [Dieter, 1988]:

$$\sigma = \frac{F}{A_0} \tag{1-1}$$

where F is the instantaneous load applied perpendicular to the specimen cross section and A_0 is the original cross-sectional area before any load is applied. Engineering strain ε is defined by the relationship:

$$\varepsilon = \frac{l_i - l_0}{l_0} = \frac{\Delta l}{l_0} \quad (1\text{-}2)$$

where l_0 is the original length before any load is applied and l_i is the instantaneous length.

In shear tests, the shear stress τ is defined by the relationship:

$$\tau = \frac{F}{A_0} \quad (1\text{-}3)$$

where F is the load or force imposed parallel to the upper and lower faces, each of which has an area of A_0. The shear strain γ is defined as the tangent of the strain angle θ.

If the deformation is immediately recovered after the force is removed, the strain is elastic. In this manner, the engineering stress and engineering strain are proportional to each other through the relationship:

$$\sigma = E\varepsilon \quad (1\text{-}4)$$

This is known as Hooke's law, and the constant of proportionality E is the modulus of elasticity or Young's modulus [Dieter, 1988]. In the case of shear deformation, shear stress and shear strain are also proportional to each other through the relationship:

$$\tau = G\gamma \quad (1\text{-}5)$$

where G is the shear modulus.

In contrast with elastic deformation, if the specimen does not recover to its original dimension when applied forces are terminated, the deformation is said to be plastic. Figure 1.1a shows schematically the tensile stress-strain behavior into the plastic

region. A straight line is constructed parallel to the elastic portion at some specified strain offset, usually 0.002 (0.2%).

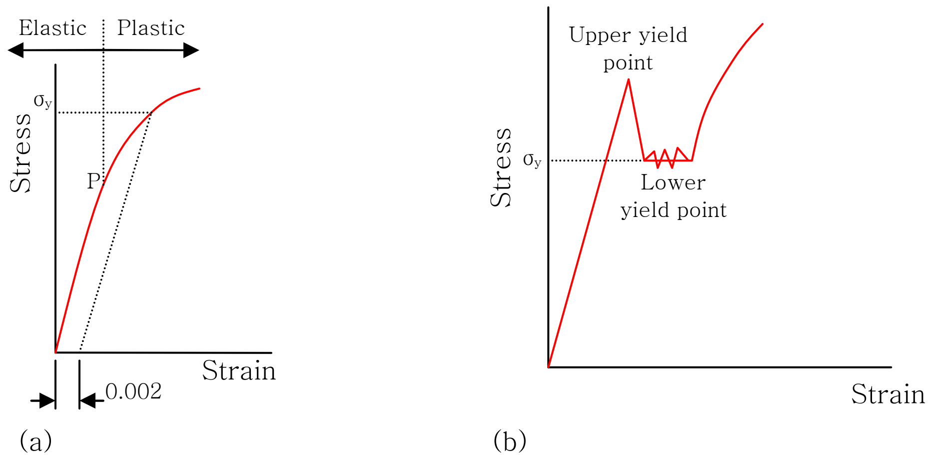


Figure 1.1 (a) Typical stress-strain behavior for a metal showing elastic and plastic deformations, the proportional limit *P*, and the yield strength σ_y, as determined using the 0.002 string offset method. (b) Yield point phenomenon.

1.1.1. Fundamentals

Yield strength

The onset of plastic deformation defines the yield strength. Before yield, the material undergoes elastic deformation within the limits of experimental error. Once the yield point is exceeded, some fraction of the deformation will be permanent and non-reversible. Some steels reach an upper yield point before the stress drops rapidly to a lower yield point. The material response is linear up until the upper yield point, but the lower yield point is used in structural engineering as a conservative value. The yield point phenomenon is illustrated in Figure 1.1b.

Yield strength depends on both the strain rate and the temperature at which the

deformation occurs. Early work [Alder *et al*., 1954] found that the relationship between yield stress and strain rate at constant temperature to be:

$$\sigma_y = C(\dot{\varepsilon})^m \tag{1-6}$$

where C is a constant and m is the strain rate sensitivity.

Later, better equations were proposed which could deal with both temperature and strain rate [McQueen *et al*., 1994]:

$$\sigma_y = \frac{1}{\alpha}\sinh^{-1}\left[\frac{Z}{A}\right]^{(1/n)} \tag{1-7}$$

where α, n and A are constants and Z is the temperature-compensated strain-rate, the Zener-Hollomon parameter [Zener *et al*., 1944]:

$$Z = (\dot{\varepsilon})\exp\left(\frac{Q_{HW}}{RT}\right) \tag{1-8}$$

where $\dot{\varepsilon}$ is the strain rate, R is the universal gas constant, T is the absolute temperature, Q_{HW} is the activation energy for the deformation.

Tensile strength

The tensile strength is the engineering stress at the maximum in the engineering stress-strain curve. All deformation up to this point is uniform. Figure 1.2 shows a typical engineering stress-strain curve. The tensile strength (TS) is indicated at point M.

Ductility

Ductility is a measure which describes how much plastic deformation a material can

sustain before fracture occurs. Ductility can be expressed quantitatively in two ways. The one is percent elongation, the other is percent reduction in area. %EL is the percentage of plastic strain at fracture (Figure 1.3):

$$\%EL = \left(\frac{l_f - l_0}{l_0}\right) \times 100 \tag{1-9}$$

where l_f is the fracture length and l_0 is the original gauge length.

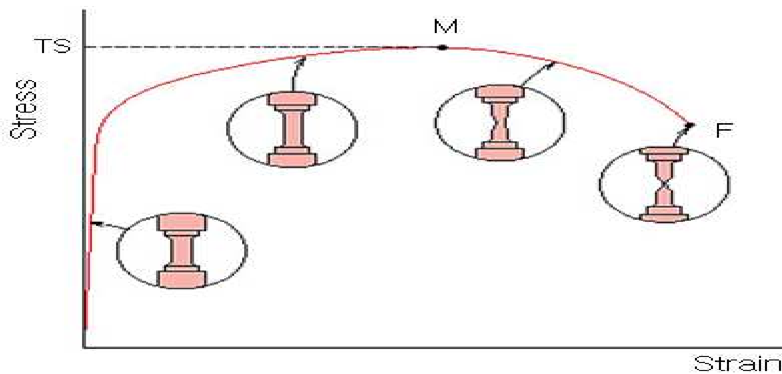


Figure 1.2 Typical engineering stress-strain behavior to fracture, point F [Callister, 2007].

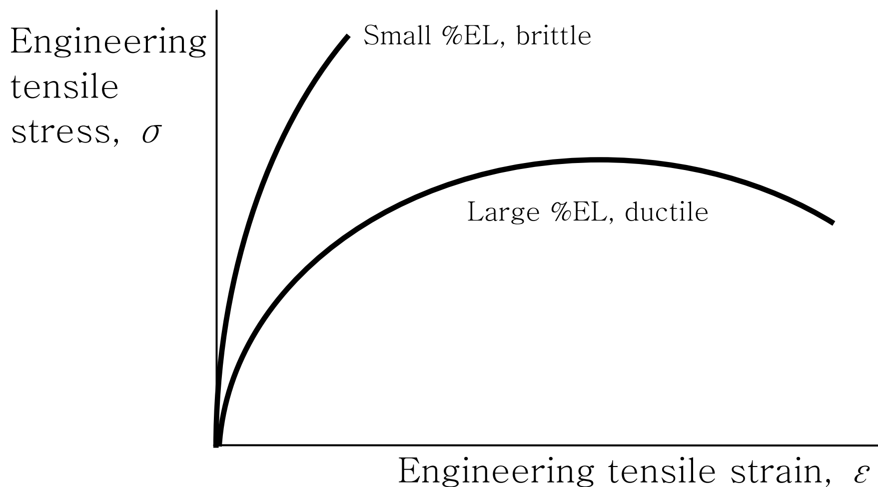


Figure 1.3 Strain-stress curve with %EL [Callister, 2007].

Percent reduction in area %RA is defined as:

$$\%RA = \left(\frac{A_0 - A_f}{A_0} \right) \times 100 \qquad (1\text{-}10)$$

where A_0 is the original cross-sectional area and A_f is the cross-sectional area at the point of fracture.

Resilience

Resilience is the property of a material to absorb energy when it is deformed elastically and then, upon unloading to have this energy recovered. In other words, it is the maximum energy per volume that can be elastically stored. It is represented by the area under the curve in the elastic region in the stress-strain curve. The related property is the modulus of resilience, U_r can be expressed:

$$U_r = \int_0^{\varepsilon} \sigma d\varepsilon \qquad (1\text{-}11)$$

Toughness

Toughness is a measure of the energy absorbed by a material during fracture. It can be measured approximately by the area under the stress-strain curve. The explicit mathematical description is:

$$\frac{energy}{volume} = \int_0^{\varepsilon_f} \sigma d\varepsilon \qquad (1\text{-}12)$$

where ε is strain, ε_f is the strain upon failure and σ is stress.

Toughness tests can be done by using a pendulum and some basic physics to measure how much energy it will hold when released from a particular height. By

having a sample at the bottom of its swing a measure of toughness can be found, as the Charpy toughness tests.

The Charpy impact test is a standardized high strain-rate test which determines the amount of energy absorbed by a material during fracture. The apparatus consists of a pendulum hammer swinging at a notched sample of material. The energy transferred to the material can be inferred by comparing the difference in the height of the hammer before and after impact. The energy needed to fracture a material, can be used to measure the toughness of the material and the yield strength. Also, the strain rate may be studied and analyzed for its effect on fracture. The Charpy test is empirical, thus the data from the test cannot be used directly in enginerring design. It is, nevertheless, an essential quality control measure which is specified widely in international standards, and in the ranking of samples in research and development exercises [Bhadeshia, 2001].

The toughness as a material property can be measured by this equation:

$$K_{Ic} = Y\sigma_c\sqrt{\pi a} \tag{1-13}$$

where a is the length of the surface flaw or one half the length of an internal crack, Y is a dimensionless constant which depends on crack, specimen sizes, etc and σ_c is the critical stress for crack propagation as:

$$\sigma_c = \left(\frac{2E\gamma_s}{\pi a}\right)^{\frac{1}{2}} \tag{1-14}$$

where E is the moulus of elasticity and γ_s is specific surface energy.

K_{Ic} is called the fracture toughness which is a quantitative way of expressing a

material's resistance to brittle fracture when a crack is present. If a material has a large value of fracture toughness it will probably undergo ductile fracture. Brittle fracture is very characteristic of materials with a low fracture toughness value.

True stress and strain

In most engineering applications, the definitions of the engineering stress and strain are accurate enough because the cross-sectional area and length of the specimen do not change substantially when loads are applied. However if the cross-sectional area and the length of the specimen change substantially, the engineering stress and strain cannot be accurate any more. To overcome this problem, the true stress and strain are introduced.

True stress σ_t is defined as the load F divided by the instantaneous cross-sectional area A_i over which deformation is occurring:

$$\sigma_t = \frac{F}{A_i} \tag{1-15}$$

True strain ε_T is defined by:

$$\varepsilon_T = \ln\frac{l_i}{l_0} \tag{1-16}$$

If no volume change occurs during deformation:

$$A_i l_i = A_0 l_0 \tag{1-17}$$

The true and engineering stress and strain are related:

$$\sigma_T = \sigma(1+\varepsilon), \quad \varepsilon_T = \ln(1+\varepsilon) \tag{1-18}$$

Hardness

Hardness is a measure of material's resistance to permanent deformation. Large hardness means large amounts of the resistance to localized plastic deformation or cracking. Hardness tests are conducted more frequently than any other mechanical test because it is simple, inexpensive and nondestructive.

There are many kinds of tests to evaluate hardness for the materials. The most common of which are Brinell hardness test, Janka Wood Hardness Rating, Knoop hardness test, Meyer hardness test, Rockwell hardness test, Vickers hardness test and Barcol hardness test.

Some other mechanical properties can be obtained from hardness data such as tensile strength. Tensile strength in units of MPa is:

$$\text{TS (MPa)} = 3.45 \times \text{HB} \tag{1-19}$$

where HB is the Brinell hardness [Callister, 2007].

1.1.2. Mechanisms for strain hardening

Work hardening

For some metals and alloys, Hollomon suggested that the true stress-strain curve can be approximated as [Hollomon, 1945]:

$$\sigma_T = K\varepsilon_T^n \tag{1-20}$$

where n is work hardening exponent (See Equation 1-21) and K is a strength coefficient which is structure dependent and is influenced by processing. The value of the work hardening exponent lies between 0 and 1. A value of 0 means that a

material is a perfectly plastic, while a value of 1 indicates 100% elasticity. Most metals have an n value between 0.10 and 0.50.

$$n = \frac{d(\ln \sigma_T)}{d(\ln \varepsilon_T)} = \frac{\varepsilon_T}{\sigma_T}\frac{d\sigma_T}{d\varepsilon_T} \quad (1\text{-}21)$$

Work hardening is the strengthening of a material by increasing the material's dislocation density. It is a result of a plastic deformation. Plastic deformation moves existing dislocations and simultaneously produces a great number of new dislocations. The shear stress τ can be expressed as [Taylor, 1934]:

$$\tau = \tau_0 + G\alpha b\rho^{1/2} \quad (1\text{-}22)$$

where τ_0 is the intrinsic shear strength of the material with low a dislocation density, G is the shear modulus, α is a correction factor specific to the material, b is the lattice constant which called burgers vector also, and ρ is the dislocation density. The latter part, $G\alpha b\rho^{1/2}$, is associated with the increase in strength due to the interaction of dislocations.

Solid solution strengthening

Solid solution strengthening occurs because when solute atoms are introduced, local stress fields are formed that interact with those of the dislocations, impeding their motion and causing an increase in the yield stress of the material. The strengthening is caused by the interaction between dislocations and solute atoms, which may originate from relative size difference, electrical interaction, chemical interaction and configurational interaction.

There are two categories of solute atoms which are interstitial and substitutional. In

both cases, the overall crystal structure is essentially unchanged. In the substitutional case, if the sizes of the solute and solvent atoms differ by less than 15%, the strength is found to be proportional to $c^{1/2}$, where c is concentration of substituional solutes [Honeycombe, 1968]:

$$\sigma \propto c^{1/2} \tag{1-23}$$

If the size of the solute atom is less than half that of the solvent, interstitial solid solution occurs [Felbeck *et al.*, 1984]. The smaller solute atoms accumulate in the dilatation field of an edge dislocation, where they form a so-called 'Cloud'. By the interaction force between the solute atoms of this cloud and the dislocation, local stress fields occur and then the material is strengthened.

In order to achieve noticeable solid solution strengthening, alloying with solutes of higher shear modulus is needed. In addition, alloying with elements of different equilibrium lattice constants is needed. The greater the different in lattice parameter, the higher the local stress fields introduced by alloying. Alloying with elements of higher shear modulus or of different lattice parameters will increase the stiffness and introduce local stress fields respectively. In either case, dislocation propagation will be hindered at these sites, impeding plasticity and increasing yield strength proportionally with solute concentration [Cottrell, 1953].

Precipitation hardening

Precipitation hardening, also called age hardening or dispersion hardening, relies on changes in solid solubility with temperature to produce fine particles of a phase, which impede the movement of dislocations. The particles act as pinning points in a

manner similar to solutes.

Precipitation hardening is accomplished by two different heat treatments. First, solution heat treating involves the formation of a single-phase solid solution via quenching and leaves the material in a soft state. Secondly, an ageing heat treatment creates the dispersion of second phase particles which lead to an increase in the material's strength.

In precipitation hardening, moving dislocations eventually encounter particles on the slip plane and result in so called short-range interactions in two distinct ways. If the precipitate particles are small, the dislocations will cut through them. As a result, new surfaces are created leading to hardening [Kelly *et al*., 1963].

$$\tau = \frac{r\gamma\pi}{bL} \tag{1-24}$$

where τ is material strength, r is the second phase particle radius, γ is the surface energy per unit area, b is the magnitude of the Burgers vector, and L is the spacing between pinning points that inhibit the motion of dislocations, such as alloying elements.

For larger precipitate particles, looping or bowing of the dislocations occurs which results in dislocations getting longer [Orowan, 1947; Ashby, 1968].

$$\tau = \frac{Gb}{L - 2r} \tag{1-25}$$

where G is the shear modulus.

The increase in strength is much higher in the Orowan mechanism than in cutting because new dislocation density is introduced around the particles causing

additional hardening of the glide planes.

Grain boundary strengthening

Normally, dislocations pile up on grain boundaries which impedes further dislocation propagation. A change in the average grain size can influence the amount of grain boundary per unit volume, dislocation movement and yield strength. The Hall-Petch relationship deals with the connection between the grain size *d* and the yield strength of a material σ_y. The relation can be written as:

$$\sigma_y = \sigma_i + kd^{-1/2} \quad (1\text{-}26)$$

where σ_i is a friction stress contributed by other strengthening mechanisms, *k* is a constant [Hall, 1951; Petch, 1953]. The validity of the Hall-Petch relationship has been confirmed for grain sizes in the range 1.5 to 150 μm for ferritic steels [Morrison, 1966].

There have been many experiments on nanocrystalline materials and they demonstrated that if the grains reached a small enough size which is typically less than 100 nm, the yield strength would either remain constant or decrease with decreasing grains size [Conrad and Narayan, 2000]. This phenomenon is called the inverse Hall-Petch effect (IHPE). A number of different mechanisms have been proposed for explaining the IHPE. They are classified to four categories such as dislocation-based models, diffusion-based models, grain-boundary-shearing models and two-phase-based models [Carlton and Ferreira, 2007]. However, there is no perfect theory explaining the IHPE.

1.2. Neural network

Many researchers have tried to clarify the relationships between the mechanical properties of steels and its controlling variables by using linear regression methods [Jaiswal and Mclvor, 1989]. This is useful but such methods are inconsistent with the complexity of the problem. For example, the following linear equations are from Pickering's work [Pickering, 1978].

$$\sigma_Y = 53.9 + 32.3W_{Mn} + 83.2W_{Si} + 354.2W_{Nf}^{0.5} + 17.4d_{\alpha}^{-0.5} \quad (1\text{-}27)$$

$$\sigma_U = 294.1 + 27.7W_{Mn} + 83.2W_{Si} + 3.85(\%pearlite) + 7.7d_{\alpha}^{-0.5} \quad (1\text{-}28)$$

Where σ_Y is predicted yield strength in MPa and σ_U is predicted ultimate tensile strength in MPa, W_{Mn}, W_{Si} and W_{Nf} are the contents of manganese, silicon and free nitrogen in weight percent respectively, and d_a is the ferrite grain size in millimeters.

These empirical and linear equations have problems because there frequently is no consideration about the uncertainty of predictions, and the variables are independently considered even though it is known that they may interact. They require the prior assumption of a relationship as linear and are not sufficiently flexible to capture the complexity in the data.

A neural network is the most general way of developing quantitative relationships when dealing with the complexity [MacKay, 1992a, 1992b, 1992c, 1995a, 1995b, 2003; Bhadeshia 1999; Bishop, 1995]. It is a parameterized nonlinear model and its flexibility makes it possible to discover more complex relationships in data than

traditional modeling methods. Bayesian probability theory provides a unifying framework for data modeling which offers several benefits. First, the overfitting problem can be solved by using Bayesian method to control model complexity. Second, probabilistic modeling handles uncertainty in a natural manner.

1.2.1. Fundamentals

A neural network is composed of input nodes, hidden units and an output node. The inputs x_j such as the chemical composition, define the input nodes and the output (for example, the tensile strength) defines the output node. Each input is multiplied by a random weight w_{ij} and the products are aggregated together with the biases $\theta^{(1)}$ which are similar to the constants of linear function:

$$z_i = \sum_j w_{ij} x_j + \theta^{(1)} \tag{1-29}$$

And z then forms the arguments of hyperbolic tangents:

$$h_i = \tanh(z_i) \tag{1-30}$$

Each h_i is itself multiplied by a further weight w_i. The sum of these hyperbolic tangents with a second bias $\theta^{(2)}$ gives the output y as a non-linear function of x_j:

$$y = \sum_i w_i h_i + \theta^{(2)} \tag{1-31}$$

Figure 1.4 illustrates neural network model for n inputs, m hidden units and 1 output. Because the weights and the constants are chosen randomly, the value of the output is not adapted to experimental data at first. They are systematically changed until the output can describe the data well. Thus, the neural network describes the phenomenon finally.

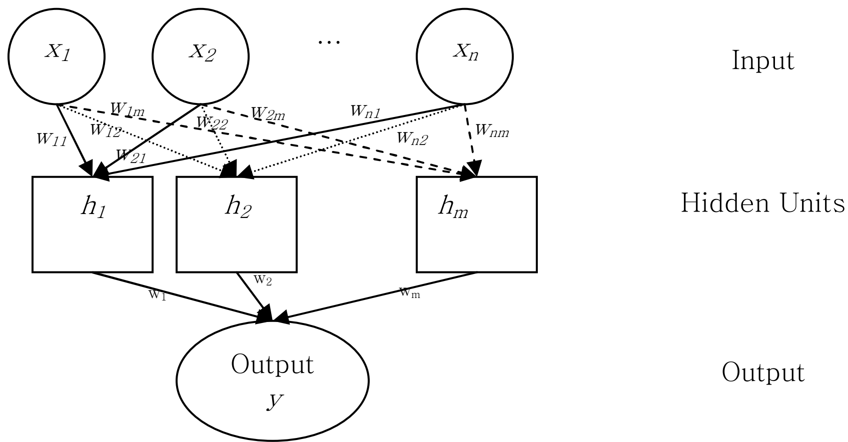


Figure 1.4 Schematic of neural network for n inputs (x_1, ⋯, x_n) and m hidden units (h_1, ⋯, h_m).

Consequently, a neural network has the ability to be flexible by varying the weights (Figure 1.5a), combining several hyperbolic tangents, i.e., changing the number of hidden units (Figure 1.5b) and controlling the number of inputs (Figure 1.5c) [Bhadeshia, 2006; MacKay, 1995]. These make it possible the neural network to capture arbitrary, non-periodic relationships.

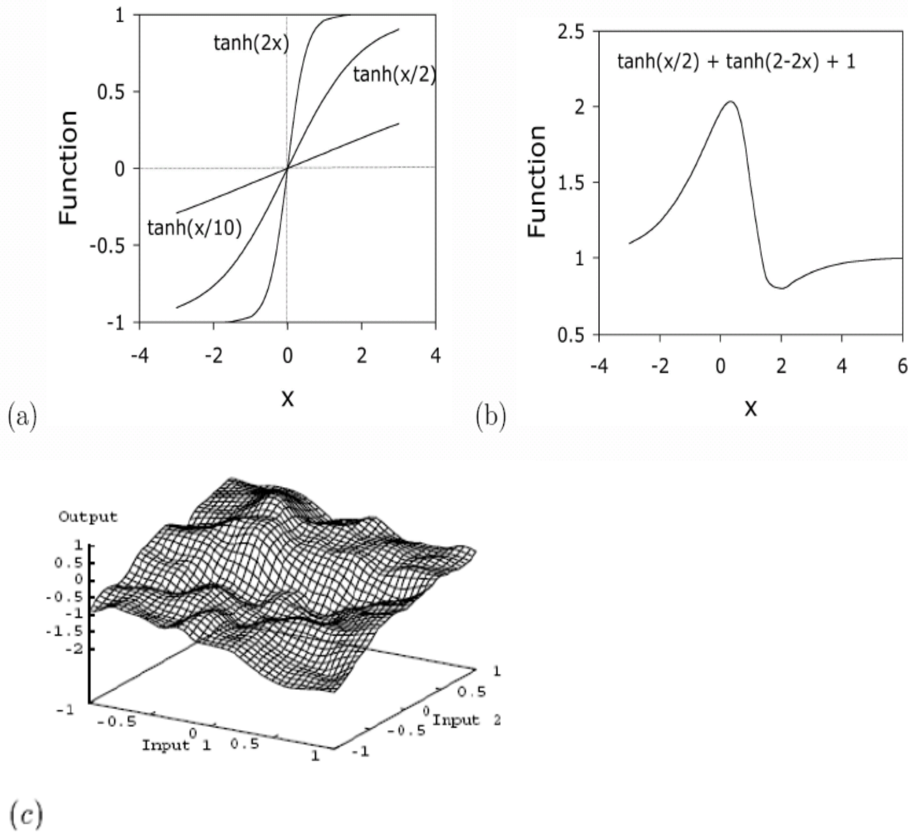


Figure 1.5 (a) Three different hyperbolic tangent functions obtained by varying the weights of one variable. (b) A combination of two hyperbolic tangents of one variable to produce a more complex model. (c) A typical function produced by two inputs network [Bhadeshia, 2006; MacKay, 1995].

1.2.2. Error estimation

The overall error in the neural network model, E_D is calculated by comparing the predicted values y_j of the output against those measured value t_j:

$$E_D \propto \sum_j (t_j - y_j)^2 \quad (1\text{-}32)$$

E_D is anticipated to increase if essential input variables have been excluded from the

analysis. E_D gives an overall perceived level of noise in the output [Bhadeshia, 1999]. 'Noise' is different from 'modeling uncertainty.' The former corresponds to the case where a different result is obtained when an experiment is repeated. This is because there are uncontrolled variables not included in the analysis. The noise in the output is a constant number and hence is not very useful in understanding the behavior of the model when extrapolating. A modeling uncertainty comes from an idea that many suitable mathematical relationships can adequately represent the same empirical data, but the individual relationships can behave differently during extrapolation. Unlike noise, the magnitude of the modeling uncertainty depends on the position in the input space where a calculation is done.

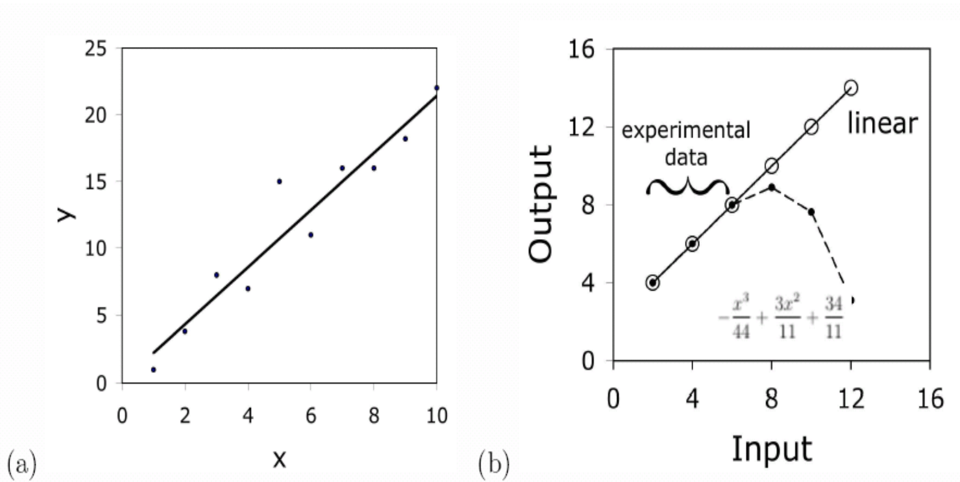


Figure 1.6 (a) Noise (b) Uncertainty [Bhadeshia, 2006]

Figure 1.6a shows the noise where data have been fitted to a straight line with a standard error of ±2 in the estimation of the output y. By contrast, Figure 1.6b shows the uncertainty of modeling where specific data (2, 4, 6) are fitted, exactly to

two different functions, one linear and the other non-linear. Both of the functions correctly reflect the known data but behave dramatically differently when extrapolated. The difference in the predictions of the two functions in domains where data are missing, is a measure of the uncertainty of modeling.

MacKay has developed a useful idea of neural networks in a Bayesian framework which allows the calculation of error bars representing the uncertainty in the fitting parameters [MacKay, 1992b]. Instead of calculating a unique set of weights, a probability distribution of sets of weights is used to define the fitting uncertainty. The error bars become large when data are sparse or noisy. However, large uncertainty always identifies a need for a research and leaves open possibilities for further investigations [Bhadeshia, 2006]. MacKay's method is useful that not only can it indicate noise, but also the modeling uncertainty. The latter gives an immediate indication of regions in the input space where data are sparse. Secondly, a large modeling uncertainty gives a clear indication of the input parameters for which new experiments are desirable. Thus, instead of using just a best-fit set of weights, a distribution of weights is calculated.

1.2.3. Overfitting

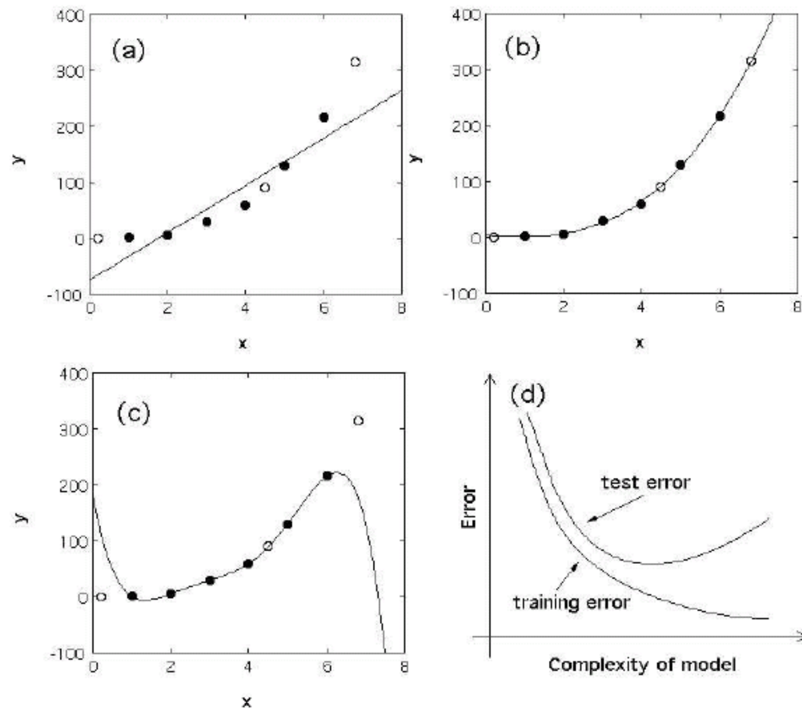


Figure 1.7 Variations in the test and training errors as a function of model complexity, for noisy data in a case where y should vary with x^3. The filled points were used to create the models (i.e., they represent training data), and the circles constitute the test data. (a) A linear function (b) A cubic polynomial function. (c) A fifth order polynomial function. (d) Schematic illustration of the variation in the test and training errors as a function of the model complexity [Bhadeshia, 2006].

A potential difficulty with the use of neural network modeling is the possibility of overfitting when the function is over complex because of too many hidden units. To prevent this, the available data are divided into two groups, a training and a test dataset. The model is produced using only the training data. The test data are then used to check that the model generalizes well.

Once a model has been trained, many models with varying complexity are produced. To calculate the error associated with each model, a regularization function (M_w) is used:

$$M_w = \beta E_D + \alpha E_w \qquad (1\text{-}33)$$

where α and β are parameters control model complexity, E_D is the test error; the difference between predicted and target values (Equation 1-32), and E_w is the parameter which is used to limit overfitting to penalize heavily weighted models:

$$E_w = \frac{1}{2}\sum_i w_i^2 \qquad (1\text{-}34)$$

where w_i are weights.

The α and β parameters define the assumed weight variances and Gaussian noise respectively:

$$\sigma_w^2 = \frac{1}{\alpha} \qquad (1\text{-}35)$$

$$\sigma_v^2 = \frac{1}{\beta} \qquad (1\text{-}36)$$

where σ_w is a weight variance and σ_v is a perceived level of noise.

The α encourages the weights to decay, so that impler models are preferred to explain output variation. A low α value results in a large σ_w value, and is therefore a good measure of the significance of each input which will be discussed later.

Figure 1.7 shows modeling noisy data for several cases where y should vary with x^3. A linear function is too simple to represent the data (Figure 1.7a), a cubic polynomial is optimum representation of both the training and test data (Figure

1.7b) and a fifth order polynomial generalizes poorly (Figure 1.7c).

Even though a model with high complexity in associated a small training error, it may generalize badly if the model is overfitted (Figure 1.7d) [Bhadeshia, 2006]. In practice, the number of hidden units can be one of the model control parameters [Bhadeshia, 1999].

1.2.4. Committee

Several reasonable models can be created given a set of data. Among them, the model which has the minimum test error is a best individual. However, several different models which behave well can be combined together to produce a committee of models. This approach can reduce the overall test error and allow more reliable extrapolation. Thus, a committee of models is used for a final prediction. The mean prediction of the committee model consisting of *N* equally weighted members:

$$\bar{y} = \frac{1}{N}\sum_{i=1}^{N} y^{(i)} \quad (1\text{-}37)$$

and the associated error in the mean prediction is

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N} {\sigma_y^{(i)}}^2 + \frac{1}{N}\sum_{i=1}^{N} (y^{(i)} - \bar{y})^2 \quad (1\text{-}38)$$

where $y^{(i)}$ and $\sigma_y^{(i)}$ are the prediction value and error of an individual model. Equation 1-37 and 1-38 gives the error bars for prediction of a committee [MacKay, 1995a].

An alternative measure of fitting error is the log predictive error (LPE) by MacKay

in a Bayesian framework which allows the calculation of error bars representing the uncertainty in the fitting parameters [MacKay, 1992b]. This error penalizes large test errors, but compensates if the prediction has large error bars.

$$LPE = \sum_i \left[\frac{1}{2} \frac{(t^{(i)} - y^{(i)})^2}{\sigma_y^{(i)^2}} + \log(\sqrt{2\pi}\sigma_y^{(i)}) \right] \tag{1-39}$$

where $t^{(i)}$ is the measured value.

1.2.5. Significance

A neural network based on a Bayesian framework can estimate the significance of individual input parameters [MacKay, 1995a]. The significance describes the level of contribution to the output, rather like a partial correlation coefficient in linear regression analysis. A high significance indicates that a particular input is able to explain a large amount of the variation obtained in the output, but it is not a just indication of the sensitivity of the output to that particular input [Fijii *et al*., 1996].

1.3. Genetic algorithm

A genetic algorithm is a search technique used in computing to find true or approximate solutions to optimization and search problems. It is based on the mechanics of natural selection and genetics as observed in the biological world [Goldberg, 1989]. It is a particular class of evolutionary algorithms that use techniques inspired by biology, such as inheritance, mutation, selection, and crossover [Chakraborti, 2004; Delorme, 2003; Goldberg, 1989; Michalewicz, 1996; Grefenstette, 1986]. It can readily be implemented as a computer simulation.

The genetic algorithm does not require any knowledge of how to get a solution for the problem to be solved. It only needs a way to evaluate possible solutions. This approach requires a lot of computing power, but has the immense practical advantage to provide near-optimal solutions to problems that do not have an algorithmical solution. The genetic algorithm is also easy to put to use. The implementation can be shared and there is little problem-specific code to write.

1.3.1. Process and operators

Table 1.1 shows description of the basic terminology for the genetic algorithm and Figure 1.8 shows the general process of the genetic algorithm.

Terminology	Description
Gene	Bit (0, 1), represent specific information
Genome, Chromosome	String of gene, i.e., 0110001…, represent of solution for the problem
Population	Group of chromosomes
Population size	The number of population in a generation
Reproduction	Process which makes new population by selection, crossover and mutation
Generation	Population at specific reproduction point of time

Table 1.1 Description of the basic terminology for the genetic algorithm

Modeling (Coding)

The first step is the representation of the solution in the form of binary bit strings. A range and a degree of precision of the solution determine the length of bit strings. A range indicates the difference between the largest and smallest value of solution, and a degree of precision describes the number of digits that are used to express a value of solution.

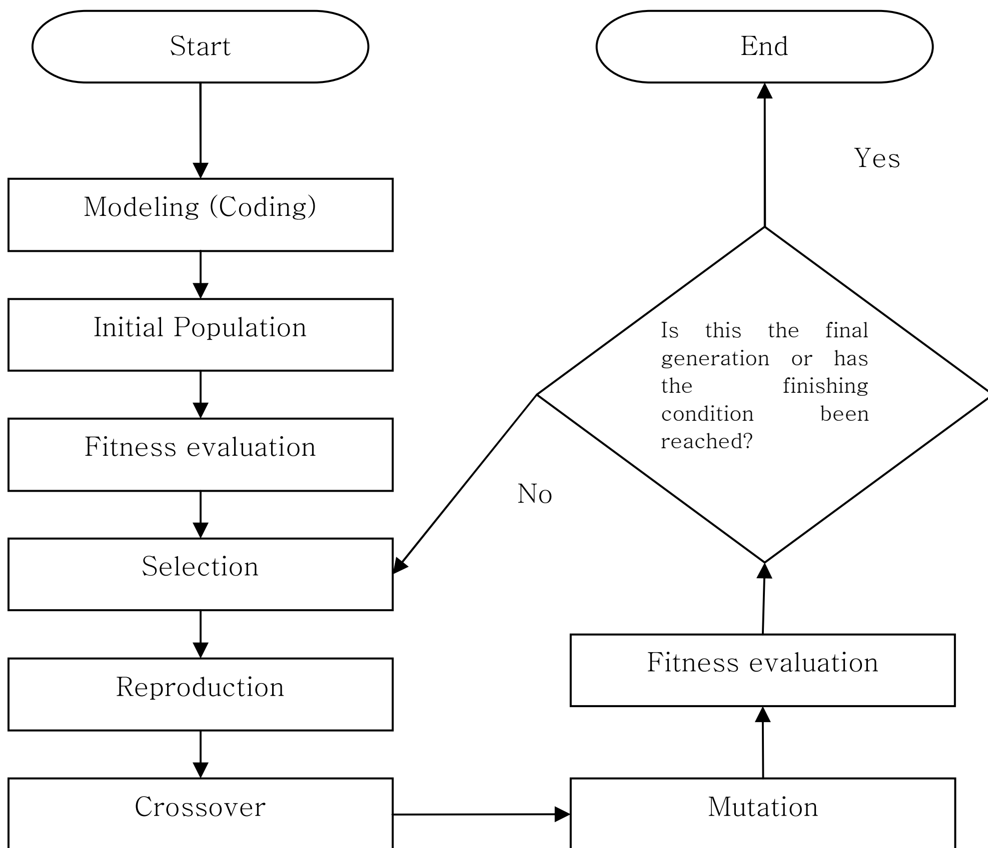


Figure 1.8 General process of the genetic algorithm

Initial Population

The initial population is the possible solution group for the given problem. Each possible solution is called an individual. The problem is encoded in a series of bit strings that are manipulated by the algorithm. These bit strings are coded representations of input variables such as the chemical composition. The choice of the initial sets is very important because it can have an influence on efficiency of the optimization process.

Fitness evaluation

A fitness function forms the basis for the evaluation of the suitability of each individual set relative to the derived value of the output. It determines how each individual is suited to the objective and hence its ability to survive to the next generation in the algorithm. Normally, the objective function governing given problem can be the fitness function. However, there are some disadvantages, like premature convergence and postmature convergence, as will be discussed later. Fitness scaling offers a way to alleviate these problems. There are three general scaling methods such as linear scaling, sigma truncation and power scaling.

Linear scaling determines the fitness score as following:

$$f_i' = af_i + b \qquad (1\text{-}40)$$

Where f_i' is scaled fitness value of f_i which is i-th chromosome's fitness, a and b are coefficients which can normally be fixed for the population life, not problem dependent. One problem with linear scaling is that the scaled fitness function may take on negative values if there are a few bad individuals. One solution is to

arbitrarily assign the value 0 to all negative fitness values. The other is to use sigma truncation. With sigma truncation, f_i is replaced as:

$$f_i' = f_i - (\bar{f} - c\sigma) \tag{1-41}$$

where $\bar{f}$ is the average fitness value of the population, i.e., if there are M chromosomes, $\bar{f} = \frac{1}{M}\sum_{i=1}^{M} f_i$, σ is the standard deviation of the population and c is a reasonable multiple of sigma (usually $1 \leq c \leq 3$). Negative results are arbitrarily set to 0. Sigma truncation removes the problem of scaling to negative values and truncated fitness values may also be scaled if desired.

Power law scaling is:

$$f_i' = f_i^k \tag{1-42}$$

where k is some suitable value. This method is not used very often. In general, k is problem dependent and may require dynamic change to stretch or shrink the range as needed.

Selection and reproduction

The selection is inspired by the role of natural selection in evolution. An evolutionary algorithm performs a selection process in which the most-fit individuals of the population survive, and the least fit individuals are eliminated. The reproduction is a process in which individual chromosomes are copied according to their fitness. Intuitively, the fitness function is some measure of profit that is to be maximized. Copying chromosomes according to their fitness means that the chromosomes with a higher value have a higher probability of contributing one

or more offspring in the next generation, which will be positioned at the place where the eliminated individuals were in the previous selection process.

There are many ways to select the survivors such as the fitness proportionate selection method which is also known as roulette-wheel selection, tournament selection and elitism selection. In fitness proportionate selection, a fitness is assigned to possible solutions or chromosomes. This fitness level is used to associate a probability of selection with each individual chromosome. If f_i is the fitness of individual i in the population, its probability of being selected is:

$$P_i = \frac{f_i}{\sum_{j=1}^{M} f_j} \tag{1-43}$$

where M is the number of chromosomes in the population. Thus, the best solutions or chromosomes will be selected more frequently. As this method prevents rapid lowering of diversity, it is helpful to avoid premature convergence of solutions.

In tournament selection, n individuals are selected at random and the fittest is selected. The most common type of tournament selection is binary tournament selection, where just two individuals are selected. It does not require global reordering and it is more naturally-inspired.

In elitism selection, the best chromosome (or a few best chromosomes) is copied to the population in the next generation. Elitism ensures that at least one copy of the best individual in the population is always passed onto the next generation. Elitism can very rapidly increase performance of GA, because it prevents losing the best found solution to date. Thus, the main advantage is that convergence is guaranteed, i.e., if the global maximum is discovered, the genetic algorithm converges to that

maximum. However, there is a risk of being trapped in a local maximum.

Crossover

The crossover is used as a further mechanism to vary the nature of individuals among generations. That is to say, it is a process of taking genes from two parents, mixing them and producing an offspring.

There are many ways to do crossover, such as single point crossover, two point crossover, uniform crossover and arithmetic crossover.

In one point crossover, a crossover point on the parent organism string is selected. All data beyond that point in the organism string is swapped between the two parent organisms. The resulting organisms are the children. Figure 1.9 shows the example where crossover point is 3.

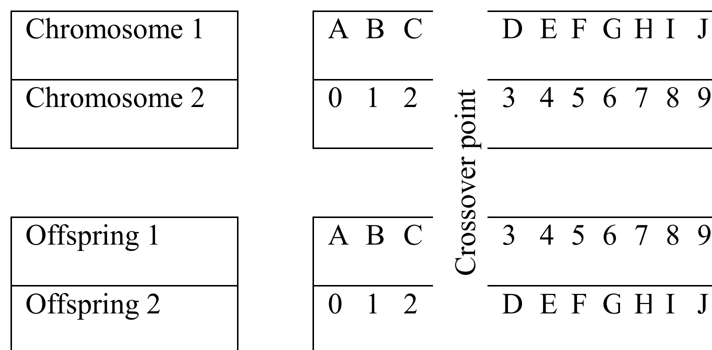


Figure 1.9 The single point crossover

In two point crossover, it calls for two points to be selected on the parent organism strings. Everything between the two points is swapped between the parent organisms, rendering two child organisms. Figure 1.10 shows the example where

crossover point are 2 and 9.

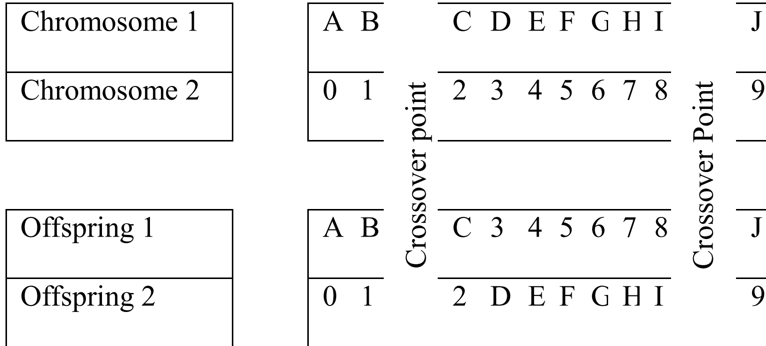


Figure 1.10 The two point crossover

In uniform crossover, genes are selected at random from either parent. Figure 1.11 shows the example.

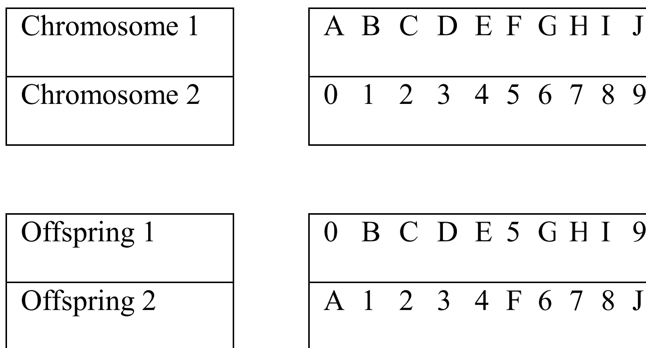


Figure 1.11 The uniform crossover

In arithmetic crossover, some arithmetic operation is performed on the two strings to create a new string. Figure 1.12 shows the example with the operation is AND.

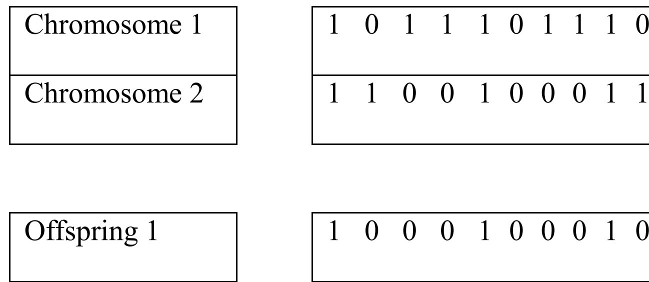


Figure 1.12 The arithmetic crossover

Mutation

Mutation is used to maintain genetic diversity from one generation to the next. The evolutionary algorithm periodically makes random changes or mutations in one or more members of the current population, yielding a new candidate solution. It is needed to avoid local minima and to offer the possibility of better solutions. For example, 1010101010 may become 1011101010 if the 4th bit is mutated.

End Condition

The following conditions can stop the algorithm:

- A solution is found that satisfies minimum criteria.
- A fixed number of generations is reached.
- The allocated budget (computation time/money) is exhausted.
- The highest ranking solution's fitness is reaching or has reached a plateau such that successive iterations no longer produce better results.
- Manual inspection.
- Combinations of the above.

1.3.2. Potential pitfalls

GA-deceptive functions

'GA-deceptive functions' are functions which select for one gene when a combination would be better. In some cases, this can eliminate better genes. This can be avoided by elitism which means the best chromosome must be preserved, the use of multiple populations or a high mutation rate which reintroduces genes.

Premature convergence

Premature convergence means that if a chromosome is far fitter than any other chromosome, it will dominate a population and lead to the loss of genes which have potential of better solutions. Assume that there is an individual i and fitness f_i is much larger than average fitness $\bar{f}$, but f_i is much smaller than the maximum fitness value f_{max} in an early generation. As generations are passed, the genes of individual i quickly spread all over the population. At that point, crossover cannot generate any new solutions (only mutation can) and $\bar{f}$ will be much smaller than f_{max} forever. This can be averted by a high mutation rate or fitness scaling which is a process that re-scales the too high fitness score with respect to the average score of the population.

Postmature convergence

In the problem of postmature convergence, which means the lack of convergence towards the end of an optimization procedure, a population of similarly high-performing chromosomes will not compete strongly with one another. When all of a

population performs well, selection pressures wane. Assume that at the end of a run (i.e. in one of the consecutive generations), all individuals have a relatively high and similar fitness, i.e. f_i is almost f_{max} for all i. There is then virtually no selective pressure. This problem can be solved by fitness scaling. It helps the algorithm kept efficient and the competition between chromosomes is vigorous.

Excessive mutation

Too much mutation makes the process inefficient. It makes the algorithm begins to resemble a random walk rather than a directed process. However, too low a mutation rate may also make the algorithm lose the genes. The correct rate of mutation can be obtained by a research from previous works related the given problem.

Application to constrained problems

Genetic algorithms are by nature unconstrained, and can take any value even if they are unphysical. However, there are constrained problems to optimize. In this case, the penalty method replaces a constraint optimization problem by a series of unconstrained problems whose solutions must converge to the solution of the original constrained problem [Chen, 2002]. This procedure removes hard constraints on the parameter values. However, some constraints can be slightly violated, e.g., by a good solution close to the border of the space of valid solutions, which the method does not prevent. This might be allowable for some problems.

On the other hand, in many optimisation problems, the constraints actually enforce

the syntactic correctness of the solutions rather than simply restrict the space of valid solutions, i.e., solutions that violate the constraints are non-solutions. In this case, the penalty method is inadequate. One must resort to using special representations and genetic operators, e.g., use of a crossover method that prevents non-solution offsprings from being generated.

The meaning of fitness

Genetic algorithms maximise fitness, which therefore must be carefully defined. When applying genetic algorithm, defining an appropriate fitness function can make all the difference between success and an unexpectedly random result. For example, appropriate fitness function helps finding a set of optimal values that will give a specific output from a given problem.

1.3.3. Genetic algorithms for Bayesian neural networks

A genetic algorithm can help to find an optimised input set for a particular defined output (Figure 1.13). The black box is a function created by a neural network, ‘X’ is a specific output and ‘?’ is a set of optimal input values which will give ‘X’.

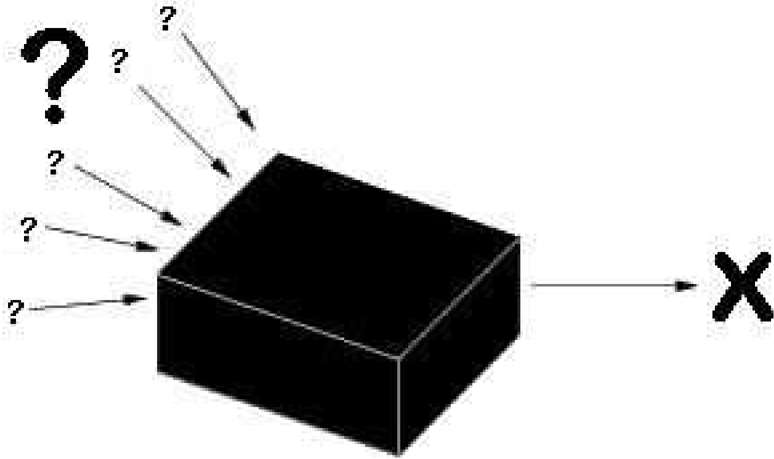


Figure 1.13 The purpose of genetic algorithm with neural networks.

For Bayesian artificial neural networks (ANNs), we have a set of input parameters and two output values - the prediction from the network and its associated uncertainty. Each model i created by the governing function of the model gives a result $y^{(i)}$ and the associated uncertainty $\sigma_y^{(i)}$. The average prediction of a committee of L models is:

$$p = \frac{1}{L}\sum_i y^{(i)} \tag{1-44}$$

The standard deviation error (σ) of p is as follows [Delorme, 2003]:

$$\sigma^2 = \frac{1}{L}\sum_i \sigma_y^{(i)2} + \frac{1}{L}\sum_i (t-p)^2 \tag{1-45}$$

where t is the target output. The score of an individual could be σ. In this case, we invert the error to get a better score and the fitness f is:

$$f = \frac{1}{\sigma} \tag{1-46}$$

The basic chromosome can consist of the set of inputs to the network but derived

inputs must be removed. For example, if there are both t and $ln(t)$, only one can be included to the set of inputs.

In genetic algorithms, there are a number of basic parameters such as computing parameters and genetic algorithm parameters. Computing parameters include number of populations and number of generations. Genetic algorithm parameters include the crossover rate, the mutation rate, the rate of population mixing and the population size.

The computing parameters are simple. More populations and generations can seek more areas of the network to be explored at once but it is then a greater requirement for computing power and time. The effects of these parameters have been explored in [Delorme, 2003]. Three populations of 20 chromosomes running for 3000 generations is a good start for many Bayesian neural network optimizations, with a crossover rate of 90% and a mutation rate of ±0.2%.

1.3.4. Multi-objective optimization

So far, it is assumed that solutions can be rated by a single function, which could be transformed into a fitness function and optimised via genetic algorithm. However, there are problems where several criteria must be considered simultaneously and it is not possible to combine these into a single objective. Such problems are known as multi-criteria or multi-objective optimization problems. Whereas optimality is well-defined in single-criterion optimisation as the highest/lowest value of the fitness function, it requires a different definition in the multi-criteria case since the integrity of the separate criteria must be respected.

Generally, definition of the standard multi-objective problem is:

$$\text{Minimize} \quad f = [f_1(x), f_2(x), ..., f_n(x)] \tag{1-47}$$

where each f_i is an objective function subject to $x \in \Omega$, Ω means the constraints on the space of design variables. There are some different techniques to achieve multi-objective optimization such as a weighting of objectives method and a slight twist method.

A weighting of objectives method, is also called Archimedean, is:

$$\text{Minimize} \quad f = w_1 f_1(x) + w_2 f_2(x) + ... + w_n f_n(x) \tag{1-48}$$

where $w_i > 0$ and $\Sigma w_i = 1$ subject to $x \in \Omega$.

A slight twist method is that picking one objective as primary, transforming remaining objectives into constraints (limitations):

$$\begin{gathered} \text{Minimize} \quad f_1(x) \\ \text{Subject to } f_2(x) < c_2, f_3(x) < c_3, ..., f_n(x) < c_m \end{gathered} \tag{1-49}$$

where c_i is a limit and $x \in \Omega$.

All of the methods mentioned above (weights or limits methods) are chosen by engineering judgment such as a trial and error, experience, etc. In order to resolve this problem, generally, the concept of Pareto optimality is used.

Pareto optimality

'Pareto efficiency', or 'Pareto optimality', is an important concept in economics with broad applications in game theory, engineering and the social sciences. Given a set of alternative allocations of goods or income for a set of individuals, a

movement from one allocation to another that can make at least one individual better off without making any other individual worse off is called a 'Pareto improvement'. An allocation is 'Pareto efficient' or 'Pareto optimal' when no further Pareto improvements can be made. This is called a 'strong Pareto optimal' (SPO) which is defined formally as follows.

A vector $x^* \in \Omega$ is Pareto optimal means that:

let < and ≤ be defined as 'strictly better' and 'at least as good' respectively,

$$f_i(x) \leq f_i(x^*) \quad \text{for all } i$$
$$\text{and} \quad f_i(x) < f_i(x^*) \quad \text{for at least one } i \text{ (one objective)} \qquad (1\text{-}50)$$

where i stands for i-th objective.

If only the 2nd condition above holds, x^* is weak Pareto optimal (WPO). If above condition (1-50) holds, then x^* is said to dominate x. If a vector is not dominated by any other, then it is said to be non-dominated.

The 'Pareto frontier' or 'Pareto set' or 'Pareto curve' is the set of choices that are Pareto efficient. That is, the Pareto frontier is the set of x* where there are no other solutions for which simultaneous improvement in all objectives can occur; non-dominated. The Pareto frontier is particularly useful in engineering by restricting attention to the set of choices which are Pareto-efficient, a designer can make tradeoffs within this set, rather than considering the full range of every parameter.

In multi-objective optimization, it is necessary to find compromises rather than a single solution. There are many solutions, with all solutions on the Pareto frontier being optimal. The particular solution that might be chosen depends on the nature of

the compromise. Consider the following example, where the cost of a component is plotted as a function of the number ordered.

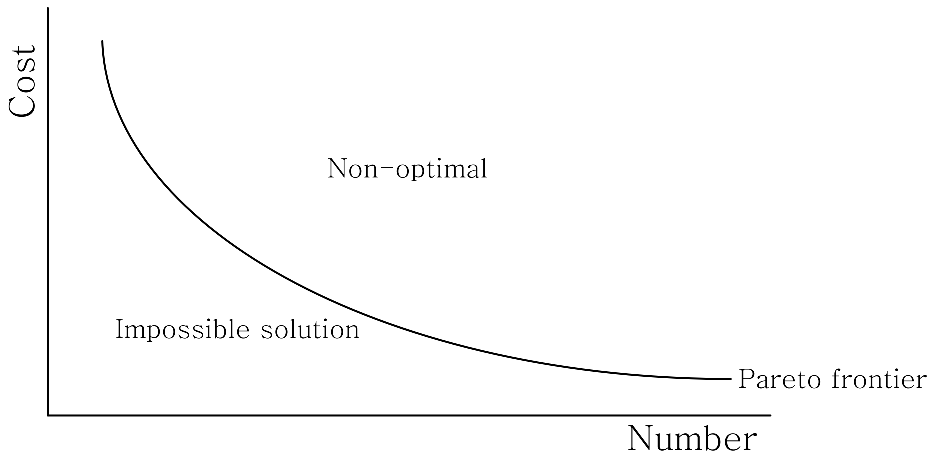


Figure 1.14 The example of Pareto frontier

There clearly are many solutions and other factors determine which compromise solution is acceptable to a particular purchaser.

In general, the Pareto frontier contains more than one element and there is no "automatic" way of selecting a single alternative. In practice, the decision maker must select one of the available answers from the Pareto optimal set as the solution. In a genetic algorithm, non-dominated (Pareto) points are needed to be identified and mated to find new ones [Osborne and Rubenstein, 1994].

II. Modeling and Computer experiments

The purpose of this work was to develop a combined model of a neural network and a genetic algorithm. At first, neural network models which on the mechanical properties of hot-rolled steels were selected, and then a genetic algorithm was combined to find an optimised input set for a particular defined output. Finally, computer experiments of the combined models were performed to get input domains for each target.

2.1. Neural network modeling

All of the data used in this work are from the Sheet Products and Process Research group, POSCO. The database has inputs consisting of the chemical composition and processing variables and the outputs which are mechanical properties of the steels. The database has been previously used to create neural network models within a Bayesian framework [Ryu, 2008]. Among Ryu's models, two were selected, one for strength and the other for elongation. They both have 17 input variables consisting of the chemical composition (carbon, manganese, silicon, phosphorus, sulphur, chronium, nickel, molybdenum, titanium, niobium, vanadium, aluminum, nitrogen, boron, copper) and the processing variables (the finish-rolling temperature (T_{FR}), the coiling temperature (T_C)); the output was either the ultimate tensile strength or the tensile elongation to failure.

Table 2.1 describes the characteristics of the database used to create the models.

The range from minimum to maximum, means and standard deviation of all the inputs are listed. The values do not define the range of applicability of the neural network model, as in linear regression analyses. Instead, the Bayesian framework is used to define the trained network applicability through the calculation of error bars. This is because each input may interact with others.

All the variables used within the model were normalized to allow an easy comparison of variables:

$$x_n = \left(\frac{x - x_{\min}}{x_{\max} - x_{\min}} \right) - 0.5 \qquad (2\text{-}1)$$

where x_n is the normalized value, x is the real value, and x_{min} and x_{max} are the minimum and maximum values of the dataset respectively. Through this operation, the input and output values are normalized between ±0.5. Nevertheless, any predictions made from the model can easily have values outside this region.

The models are strictly committees of models related to optimize the ability to make predictions. The ultimate tensile strength committee is combination of 11 different of best performing individual models whereas the elongation committee has an optimum membership of just 3 different models.

Figures 2.1 and 2.2 indicate the significances of the input variables, as perceived by the first five models in ultimate tensile strength committee and by the first three models in elongation committee respectively. C, Mn and Si are more significant than other solutes in influencing the ultimate tensile strength. In the case of the elongation model, C and Mn are the most significant determinants of the elongation. These are expected trends, which indicate that the models are reasonable. First, an

increase in the C concentration should lead to a greater fraction of pearlite, Fe_3C, which is harder than ferrite. Second, Mn not only has a strong effect on the stability of the austenite, but also provides solid solution strengthening. As Mn can depress the transformation temperature, it can also refine the microstructure. Finally, the addition of Si can lead to the solid solution hardening. Silicon addition over 1.0 wt% results in a significant increase in the volume fraction of retained austenite and the increment of elongation is attributed to the transformation of retained austenite into martensite during plastic straining and the resultant increase in work-hardening [Bhadeshia and Edmonds, 1980; Tsukatani *et al*, 1991]. As a result, σ_U increases and ε decreases. These trends are consistent with experimental observations (Equation 1-2) [Pickring, 1978].

Other inputs, except C, Mn and Si for the tensile strength and C and Mn for the elongation, were seen to offer at least a moderate contribution to the outputs. This therefore confirmed the good choice of inputs.

	Minimum	Maximum	Mean	Standard Deviation
C / wt%	0.0204	0.8684	0.1009	0.0833
Mn / wt%	0.167	1.41	0.471	0.2177
Si / wt%	0	0.217	0.0146	0.0262
P / wt%	0.004	0.022	0.0129	0.0027
S / wt%	0.002	0.015	0.007	0.0023
Cr / wt%	0	0.16	0.0189	0.0137
Ni / wt%	0	0.06	0.0132	0.006
Mo / wt%	0	0.02	0.0008	0.0028
Ti / wt%	0	0.004	0.0006	0.0009
Nb / wt%	0	0.004	0.0002	0.0004
V / wt%	0	0.003	0.0011	0.001
Al / wt%	0	0.064	0.0323	0.0105
N / ppm	0	87	33.8483	13.4774
B / ppm	0	2	0.2888	0.473
Cu / wt%	0	0.03	0.0075	0.0061
T_{FR} / ℃	808	925	868.88	14.089
T_C / ℃	478	714	618.80	29.5304
σ_U / MPa	317	1039	411.4461	69.6126
ε / %	14	50	38.4658	6.3109

Table 2.1 The dataset consisting of 3508 experiments and was used to make the models in the present work.

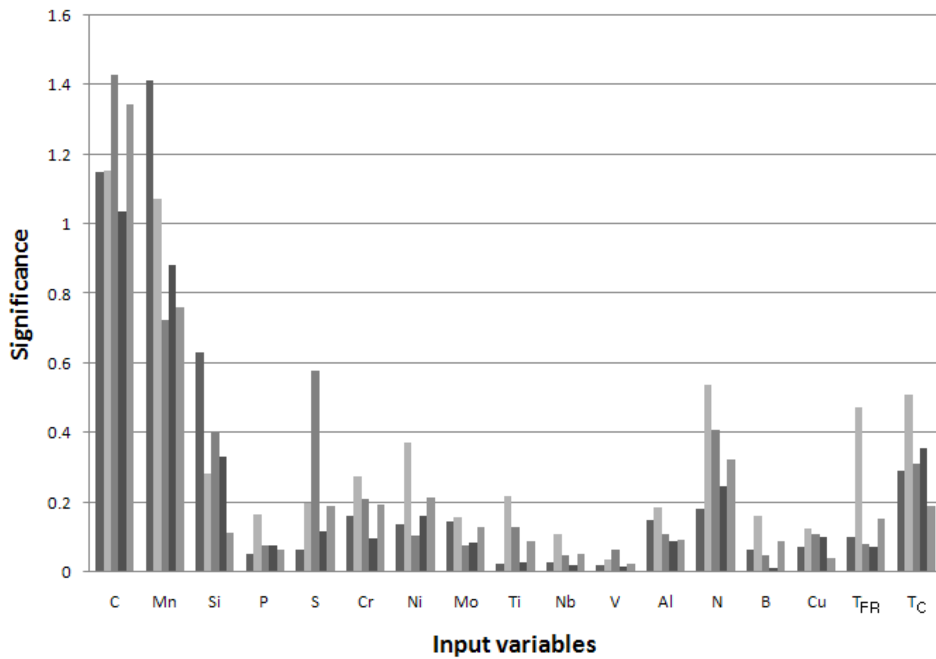


Figure 2.1 Significance of input variables in ultimate tensile strength model

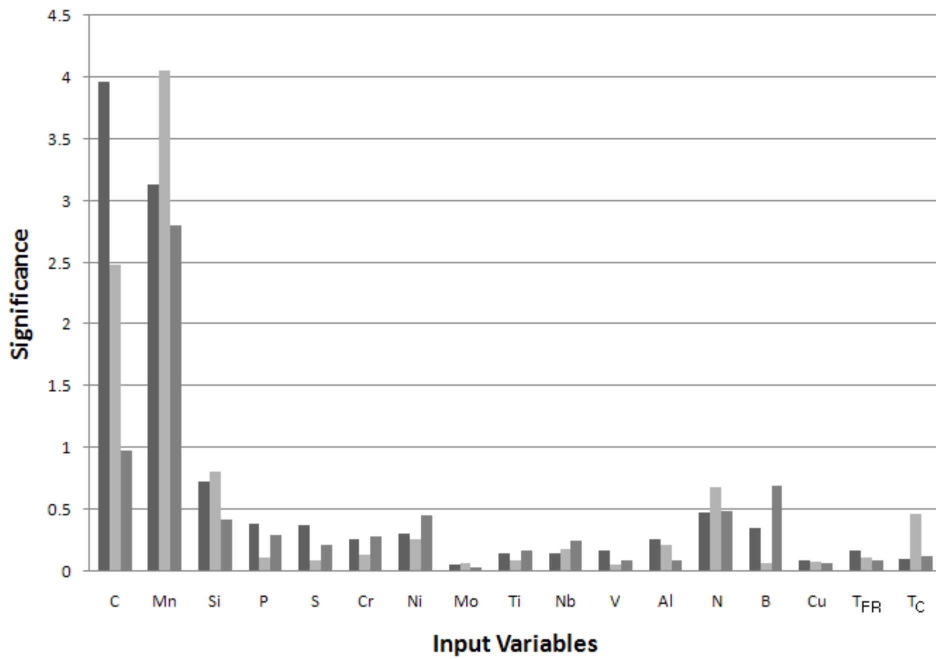


Figure 2.2 Significance of input variables in elongation model

2.2. Genetic algorithm

Using a genetic algorithm, two kinds of combinations with neural network models were made in this work. The first is a single objective combined model which had a target of either the ultimate tensile strength or the elongation. The other is multi-objective combined model with two targets to simultaneously aim for.

2.2.1. Single objective

Figure 2.3 shows the process of a genetic algorithm combined with a neural network. The initial populations are given random input values chosen by a random number generator. The range of each variable in the database which was used for creating the neural network was normalized between ± 0.5, as described previously. However, for the purpose of a genetic algorithm, the values are allowed to exist beyond this range and in order to search a wider range of input domains. The error bars associated with these predictions may well be large, but the uncertainty is the most critical issue. Indeed, the level of uncertainty can be used as a constraint in setting the environment of a genetic algorithm. However, because there is a distinct possibility that negative values which cannot be less than zero are studied as inputs, the search was confined to the following regions for each variable:

$$x_n \geq \left[\left(\frac{x_{\min}}{x_{\max} - x_{\min}} \right) + 0.5 \right] \qquad (2\text{-}2)$$

where x_n is input variable, x_{min} and x_{max} are minimum and maximum value for x_n

respectively on the database which was used to create the neural network committee model. This is helpful for input variables to be physically meaningful.

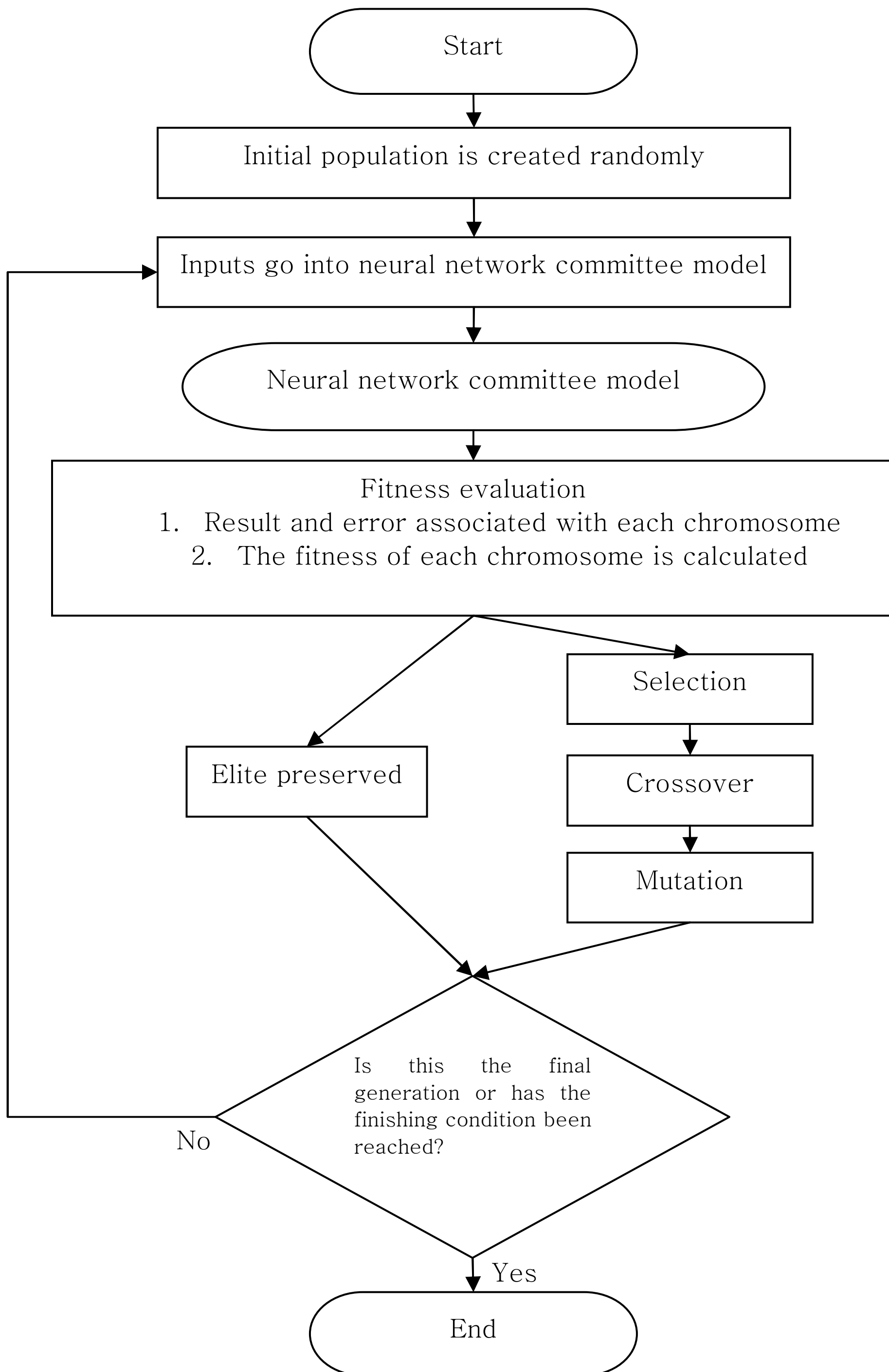


Figure 2.3 The process of a genetic algorithm combined with a neural network

The neural network committee model has two outputs; the result and the uncertainty. The fitness function should therefore consider both of these parameters. Therefore, if each model i created by the neural network gives a result $y^{(i)}$ and the associated uncertainty $\sigma_y^{(i)}$, the average prediction of a committee of L models is:

$$\bar{y} = \frac{1}{L}\sum_i y^{(i)} \tag{2-3}$$

The standard deviation error, σ of $\bar{y}$ is [Delorme, 2003]:

$$\sigma^2 = \frac{1}{L}\sum_i \sigma_y^{(i)2} + \frac{1}{L}\sum_i (t - \bar{y})^2 \tag{2-4}$$

where t is the target output. Then the fitness f is:

$$f = \frac{1}{\sigma} \tag{2-5}$$

To get better solutions, it is necessary to select the input sets which will survive a great number of generations in the selection operation (Chapter 1.2.1). In this work, the fitness proportionate selection is used (Equation 1-17). Thus, the best solutions are selected with the greatest probability. As this method can prevent the rapid lowering of diversity, it helps avoid a premature convergence and a postmature convergence (Chapter 1.2.2). In addition, in order to avoid 'GA-deceptive functions' (Chapter 1.2.2), it is necessary to keep the best chromosome. 'Elite preserved' in Figure 2.3 performs this role.

Further procedures are required to complete the genetic algorithm process. The crossover is used to vary the nature of individuals among generations. In this work,

the uniform-crossover method is used in which individual bits in the string are compared between two parents (Figure 1.8). It means that each gene of the offspring is selected randomly from the corresponding genes of the parents. The bits are swapped with a fixed probability, typically 0.5. And finally, mutation is used to maintain genetic diversity from one generation of a population to the next. This particularly helps in a wider exploration of the input space, thus avoiding local minima.

2.2.2. Multiple objective

It would be advantageous if several properties of the steels could be simultaneously modeled. It is possible to achieve an optimization among several objectives systematically using a genetic algorithm. In chapter 2.2.1, the simple genetic algorithm combined with only one neural network output is described. It is possible to make a multi-objective genetic algorithm combined with a pair of neural networks. There are two differences between simple genetic algorithm and multi-objective genetic algorithm. The first is in the definition of the fitness function. If each neural network model has a fitness $f_n(x)$, then the total fitness is:

$$f(x) = w_1 f_1(x) + w_2 f_2(x) \quad (2\text{-}5)$$

where $0 \leq w_1 \leq 1$ and $w_2 = 1 - w_1$.

In Equation 2-5, the random weights w_1 and w_2 are needed to widen the domains of the solutions. Figure 2.4 represents the search directions in a multi-objective genetic algorithm. If w_n are fixed at constant values, the total fitness has a constant search direction (Figure 2.4a). However, If w_n are not fixed and randomly generated, the

total fitness has a variety of search directions and it is possible to discover domains of solutions (Figure 2.4b).

Lastly, with the single objective genetic algorithm, the solution which has the best fitness score is preserved. However, in the multi-objective genetic algorithm, the elite preserve strategy needs to be altered. In multi-objective optimization problems, a solution with the highest fitness value of each objective can be regarded as an elite individual. Therefore, there are *n* elites for an *n*-objective problem. In this work, two objectives are handled. So, there are 2 elite individuals on each generation. During the operation of the optimization, for each population, 3 elite individually will survive at every generation; two elite individuals with respect to two objectives and the individual which has highest $f(x)$ in Equation 2-4. This scheme can be effective in preserving the variety within the population.

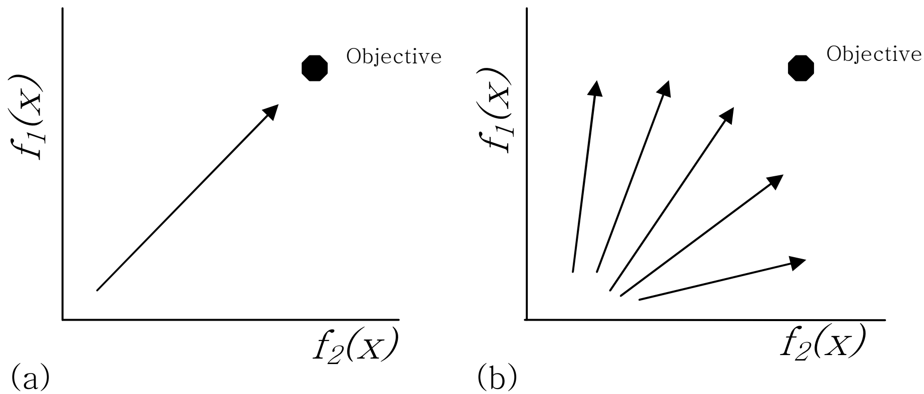


Figure 2.4 The meaning of random weights (a) constant direction (b) various directions [Murata *et al.*, 1996].

2.3. Simulations for each combined model

The genetic algorithm is developed using the computing language C. The original

genetic algorithm program source was obtained from [Delome, 2003]. It was then modified to adjust to the present work. There are two kinds of the program source; one for a single objective which means that it is combined with one neural network model, and the other for two objectives.

The performance of the genetic algorithm depends on several parameters. In this work, for the single objective combined models, it was found necessary to implement at most 10,000 generations to reach reasonable answers [Grefenstette, 1986] and 3 populations of 20 chromosomes were used [Delorme, 2003]. The crossover rate is typically 80-95 % [Obitko *et al*., 1999] but a lower value of 60 % was used in order to keep well-fitting solutions for as many generations as possible. The mutation rate was typically between 0 and 0.2 %. For each single objective combined model, three simulations were done. First, for the ultimate tensile strength model, the target values were 400, 600 and 800 MPa of tensile strength. For the elongation model, the objective values were 35, 45 and 55 % of elongation to failure.

For the multiple objective combined model, 5000 generations to reach answers were implemented to minimize computing time and 3 populations of 20 chromosomes were used. The crossover rate was 85 % and it was typical. The mutation rate was also typical value, 0 - 0.2 %. 9 simulations were done; 400 MPa tensile strength with 35, 45 and 55 % elongation, 600 MPa with 35, 45 and 55 %, and 800 MPa with 35, 45 55%.

2.4. Summary

Computer experiments were performed to predict optimized input domains for several particular defined outputs. These were done by combining a neural network and a genetic algorithm. The neural network models were selected from previous research [Ryu, 2008], and then a genetic algorithm method was combined to the selected models. There were two kinds of a combined model; one for single objective, the other for multiple objectives. In single objective combined model, the output was either the ultimate tensile strength or the percent elongation to failure whereas the targets were both of them in multi-objective experiments. In general, combined models cannot directly indicate theoretical compositions of given targets into reality. However, the aim of this is to narrow the focus of research into new and existing area which have not previously explored.

III. Result and Discussion

In ferrite-pearlite steels, as the percentage of pearlite increases, the strength increases and ductility decreases. It means that the strength is in this class of steels approximately in inverse proportion to the total elongation [Callister, 2007]. So, it is hard to achieve a high strength and a high elongation at once in ferrite-pearlite steels.

Computer simulations were performed in the content of this relationship between strength and ductility, to assess the ability of the neural network method. This can be covered by doing computer experiments using a single objective model which has a target of achieving 400 MPa of tensile strength or 35 % elongation. These targets are typical for ferrite-pearlite steels [Bhadeshia, 1998; Callister, 2007], so are easy to achieve in practice. A second computer experiment was to achieve 600 MPa of tensile strength or 45% elongation because this is possibly amongst the upper limits of common ferrite-pearlite steels (Figures 3.2 and 3.3). Finally, the simulations to achieve 800 MPa of tensile strength or 55 % elongation were performed to test for the highest values of mechanical properties which cannot easily be reached.

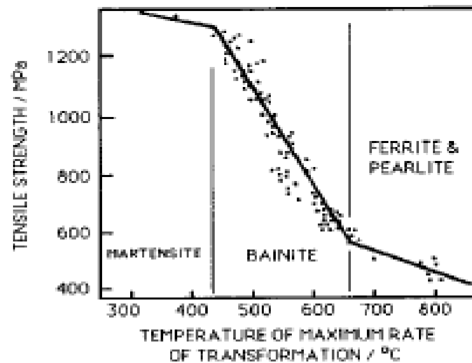


Figure 3.1 Variation in the tensile strength of structural steels as a function of the temperature at which the rate of transformation is greatest during continuous cooling heat treatment [Irvine *et al.*, 1957].

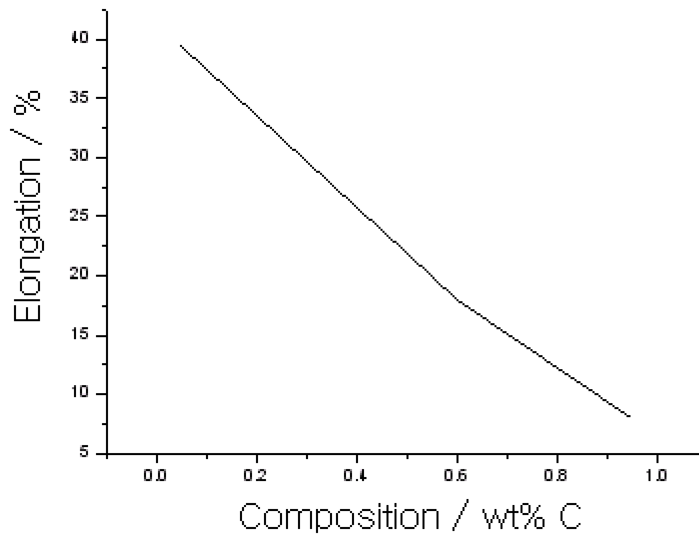


Figure 3.2 Elongation versus carbon concentration for plain carbon steels having microstructures consisting of fine pearlite and ferrite. [Data taken from *Metals Handbook: Heat Treating*, Vol. 4, 9th edition]

Different mechanical properties can often correlate [Callister, 2007], in which case it is necessary to consider several properties simultaneously during analysis. Multi-objective models can be helpful to recognize such inter-relationships between the

ultimate tensile strength and the percent elongation to failure. Of course, the desired targets may not be achievable in a ferrite-pearlite system. But that kind of information can be beneficial in assessing the potential of steels.

3.1. Single objective model

There were no constraints on the input parameters when the simulations were conducted in order to widen the domains of inputs as much as is possible. However, to make metallurgical sense, two constraints were set on the outputs; the target value and the modeling uncertainty. Sets of inputs which led to the output lying within the range ± 10 % of the target value were permitted as long as the computational uncertainty was within ± 15 % of the magnitude of the target.

3.1.1. The ultimate tensile strength model

Table 3.1 shows the characteristics of the simulations in which the objective was a tensile strength of 400 MPa. Some 94 combinations of input variables were discovered for which the target value had less than ± 15 % uncertainty.

These results were analyzed by considering C, Mn and Si, which are influential in determining strength, as described before (Chapter 2.1).

Figure 3.3 indicates the results from the first simulation in which the target was 400 MPa. The inputs have a few combinations which maintain the 400 MPa of the ultimate tensile strength criterion. Because remained inputs other than C, Mn and Si have relatively little influence, they are not plotted here.

	Minimum	Maximum	Mean	Standard Deviation
C / wt%	0.1091	0.2041	0.1727	0.0449
Mn / wt%	0.7050	0.8625	0.7433	0.0520
Si / wt%	0.0173	0.1085	0.0528	0.0444
P / wt%	0.0130	0.0310	0.0135	0.0026
S / wt%	0.0085	0.0215	0.0100	0.0036
Cr / wt%	0.0245	0.0800	0.0375	0.0204
Ni / wt%	0.0043	0.0900	0.0355	0.0108
Mo / wt%	0.0028	0.0100	0.0056	0.0033
Ti / wt%	0.0020	0.0060	0.0022	0.0009
Nb / wt%	0.0016	0.0060	0.0025	0.0013
V / wt%	0.0015	0.0045	0.0035	0.0014
Al / wt%	0.0320	0.0960	0.0397	0.0207
N / ppm	28.78	43.50	43.34	1.52
B / ppm	0.03	3.00	0.94	0.77
Cu / wt%	0.0094	0.0450	0.0361	0.0130
T_{FR} / ℃	828.40	983.50	876.04	40.97
T_C / ℃	596.00	709.17	653.35	46.37
σ_U / MPa	384.85	449.96	423.88	17.23
Uncertainty / MPa	25.63	59.98	44.81	9.74

Table 3.1 Characteristics of results of simulation for 400 MPa.

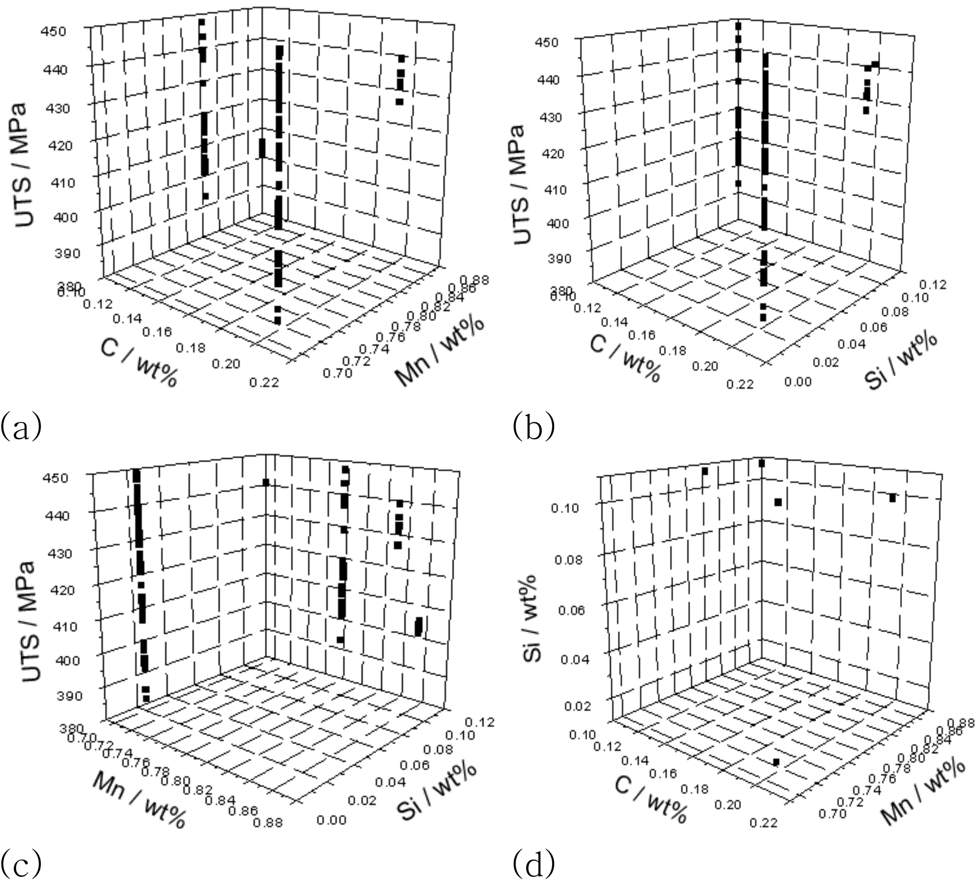
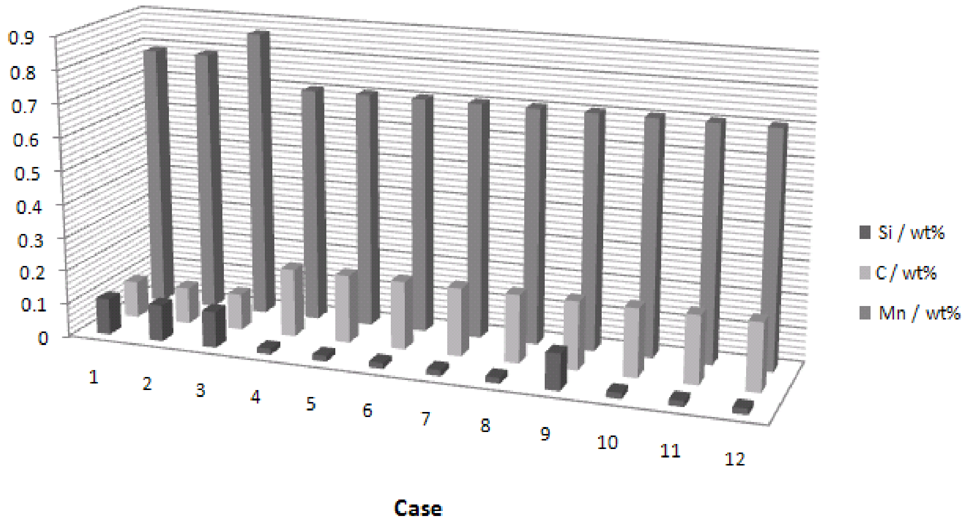
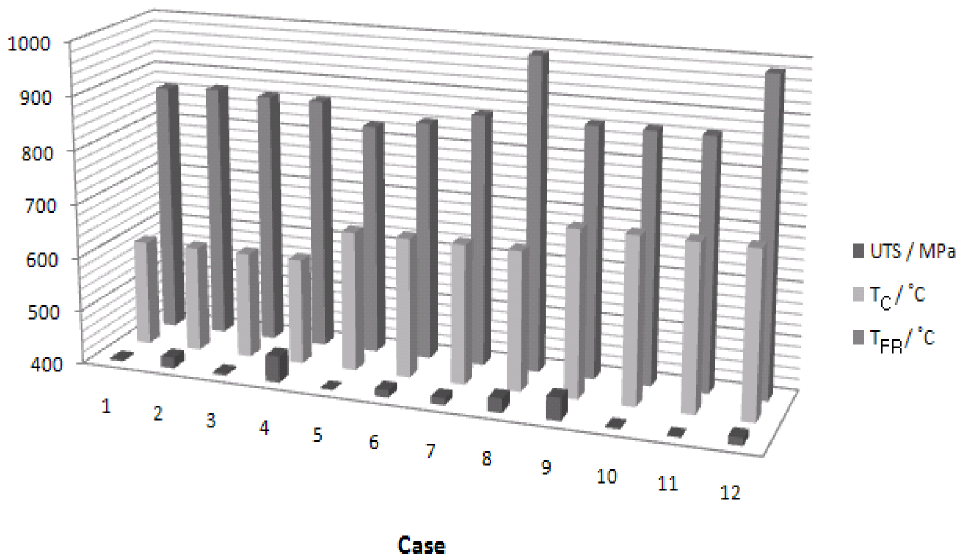


Figure 3.3 Combinations of the input variables which lead to the tensile strength of 400 MPa (a) C and Mn (b) C and Si (c) Mn and Si (d) C, Mn and Si

Selected cases from these results are plotted in Figure 3.4, illustrating particularly the carbon, manganese, silicon, finish rolling temperature, T_{FR} and coiling temperature, T_C, together with the target strength. Notice that high carbon concentrations in cases 5 - 12 are automatically related to high T_C in order to maintain a value of strength close to 400 MPa. The coiling temperature can be controlled in industry by varying T_{FR} and the cooling rate at the runout table, which is a path between the finishing mill and the coiler in the hot rolling process. A low T_C lies in the range of 550-650 ℃ and is associated with a high cooling rate. A high coiling temperature is above A_{r1} which is the temperature that corresponds to the onset of cementite during the cooling. High T_C is associated with low cooling rates. So, this result is expected since an increase in strength due to carbon is compensated for by a reduction in the cooling rate due to the use of a high T_C. Similarly, large manganese concentrations are compensated for by low carbon concentrations in cases 1, 2 and 3 [Kirkaldy *et al.*, 1984; Reed *et al.*, 1992].



(a)



(b)

Figure 3.4 Combinations of variables, all of which lead to an ultimate tensile strength of approximately 400 MPa. (a) C, Mn and Si (b) T_{FR}, T_C and UTS for 12 cases

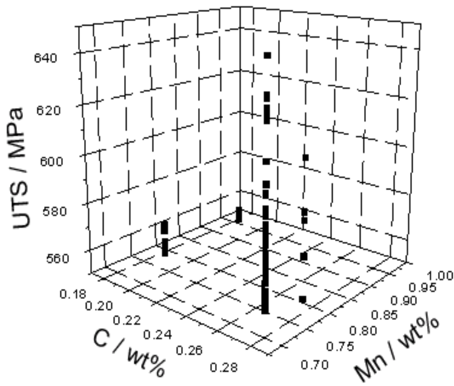
The range of the key inputs which lead to the required strength are as follows:

	C / wt%	Si / wt%	Mn / wt%	T_C / °C	T_{FR} / °C
Minimum	0.11	0.02	0.70	596	828
Maximum	0.20	0.11	0.79	709	983

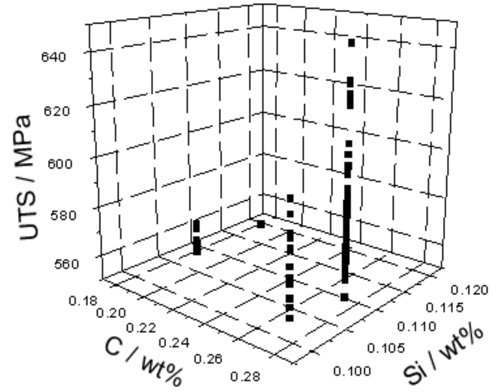
Table 3.1 The range of the inputs especially C, Si, Mn, T_C and T_{FR} for 400 MPa

As stated previously, there are of course many more variables involved in the analysis but they have a relatively minor effect and hence are not listed here. The important point is that a significant range of inputs can lead to the required strength of 400 MPa, with the analysis coming up with a total of 94 solutions.

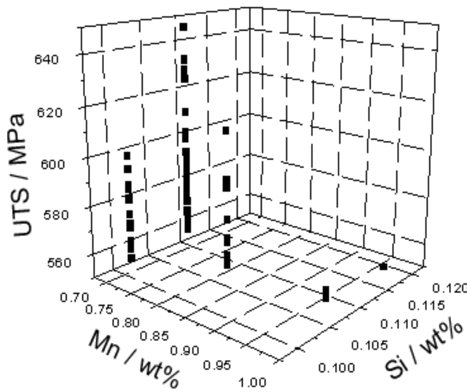
The number of solutions decreased to 75 and 18 when the required strength was increased to 600 and 800 MPa respectively. Tables 3.5 and 3.6 describe the characteristics of the simulations. These simulations are analogous to the previous results for the simulation for 400 MPa. Figures 3.5 and 3.6 show the tendencies of input variables of the results of simulations for 600 and 800 MPa, with C, Mn and Si having significant effects. The varieties are reduced as the target strength is increased, because it is difficult to reach the high values of the ultimate tensile strength in this alloy system.



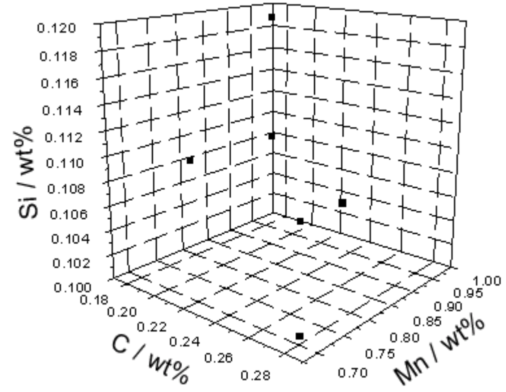
(a)



(b)



(c)



(d)

Figure 3.5 Combinations of the input variables which lead to the tensile strength of 600 MPa (a) C and Mn (b) C and Si (c) Mn and Si (d) C, Mn and Si

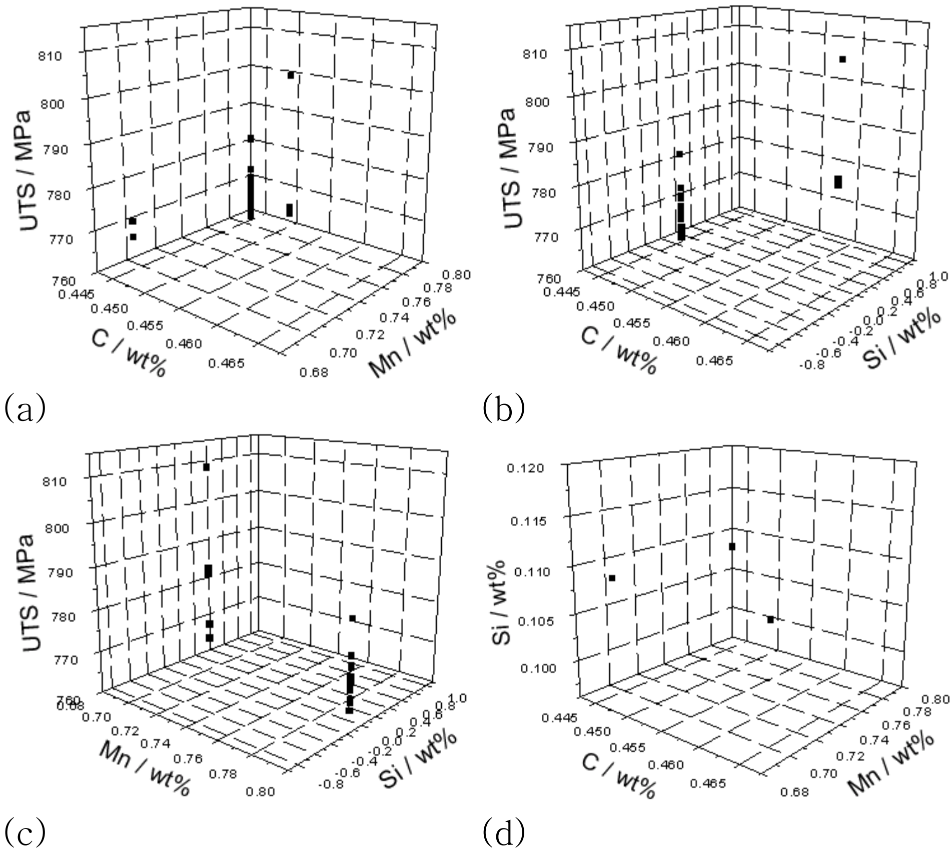


Figure 3.6 Combinations of the input variables which lead to the tensile strength of 800 MPa (a) C and Mn (b) C and Si (c) Mn and Si (d) C, Mn and Si

It is clear from the data listed below Tables 3.3 and 3.4 and from Figures 3.7 and 3.8, that not only has the number of available solutions decreased, but the permitted range of the inputs has decreased significantly.

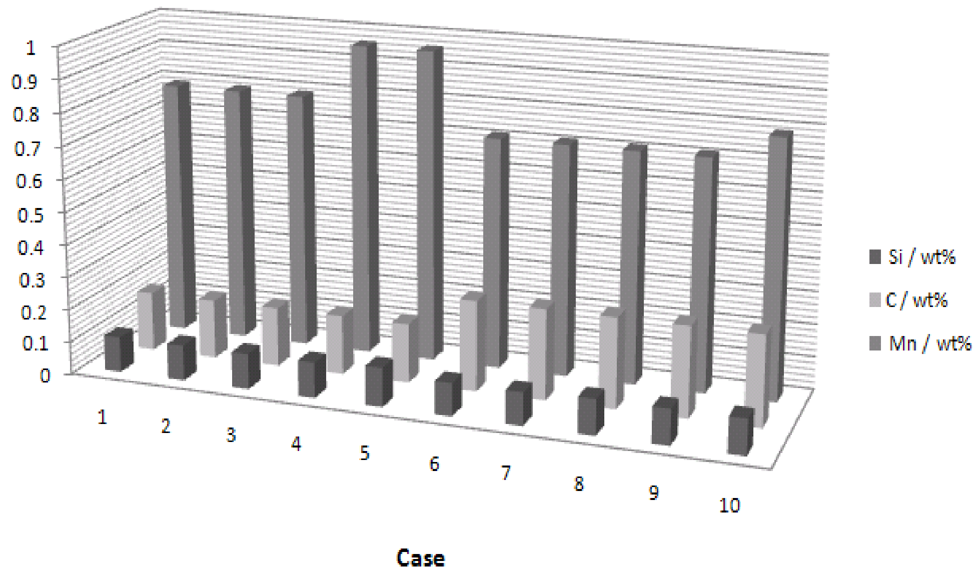
	C / wt%	Si / wt%	Mn / wt%	T_C / °C	T_{FR} / °C
Minimum	0.18	0.10	0.71	534	866
Maximum	0.28	0.12	0.95	600	984

Table 3.3 The range of the inputs especially C, Si, Mn, T_C and T_{FR} for 600 MPa

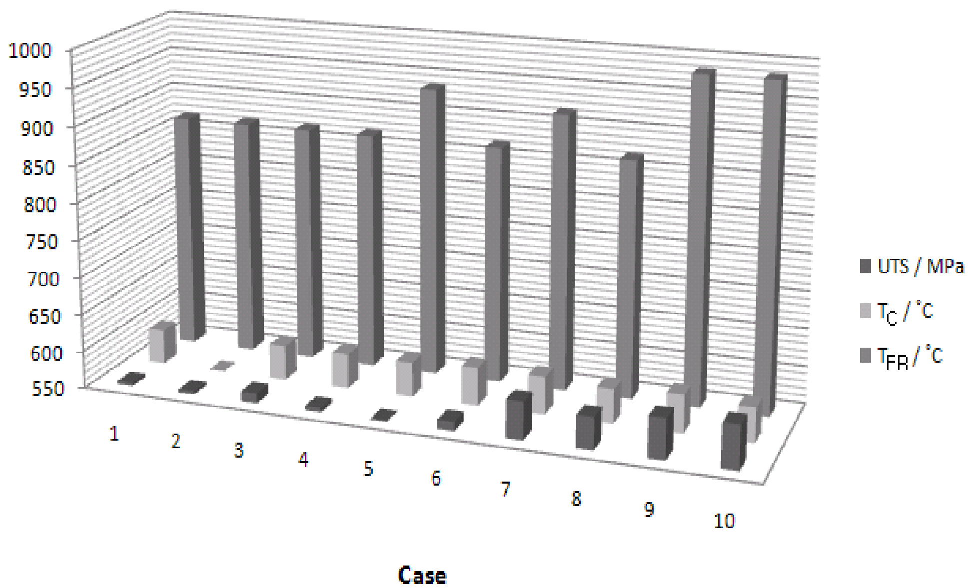
	C / wt%	Si / wt%	Mn / wt%	T_C / °C	T_{FR} / °C
Minimum	0.44	0.11	0.70	596	867
Maximum	0.47	0.11	0.79	622	867

Table 3.4 The range of the inputs especially C, Si, Mn, T_C and T_{FR} for 800 MPa

For the 800 MPa target, the permitted ranges of input variables is exceptionally small, indicating that it should be difficult in this alloy system to design steels as strength as this based on a mixture of ferrite and pearlite.
Another interpretation is that the alloy system considered in creating the model is too simple to allow flexibility in the choice of inputs when the target strength is high. In contrast, the 400 MPa is typical of ferrite-pearlite steels [Bhadeshia, 1998, Callister, 2007].

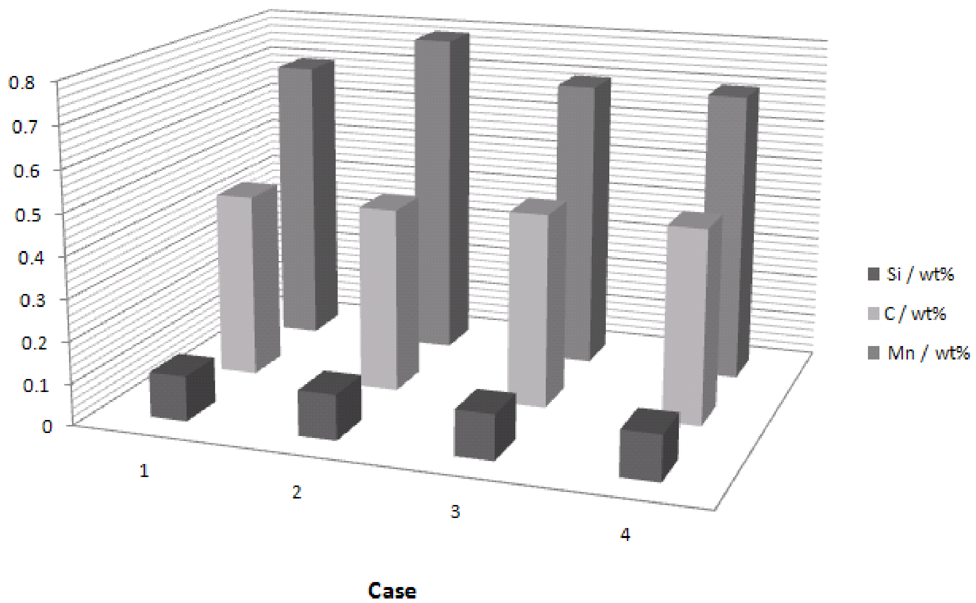


(a)

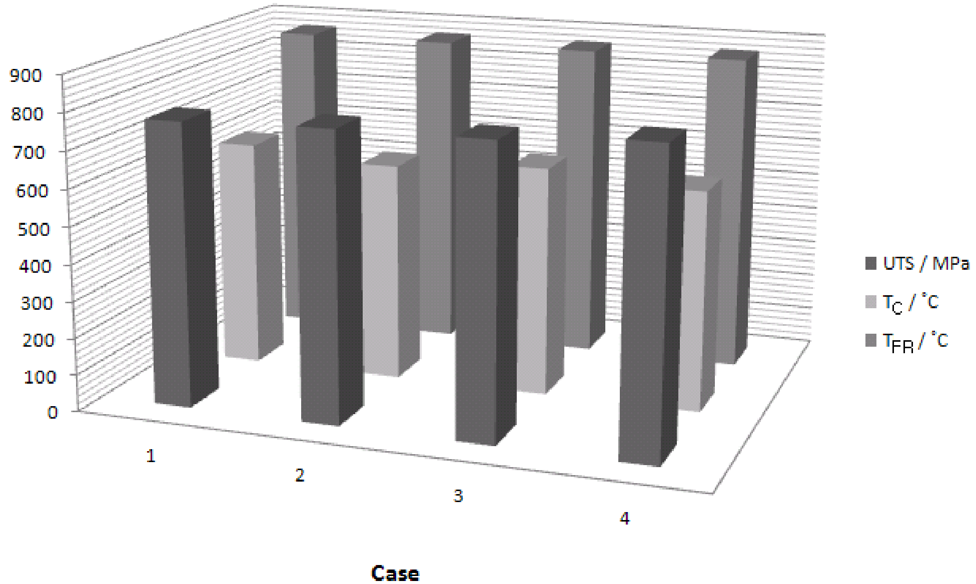


(b)

Figure 3.7 Combinations of variables, all of which lead to an ultimate tensile strength of approximately 600 MPa. (a) C, Mn and Si (b) T_{FR}, T_C and UTS for 10 cases



(a)



(b)

Figure 3.1 Combinations of variables, all of which lead to an ultimate tensile strength of approximately 800 MPa. (a) C, Mn and Si (b) T_{FR}, T_C and UTS for 4 cases.

	Minimum	Maximum	Mean	Standard Deviation
C / wt%	0.1814	0.2768	0.2603	0.0364
Mn / wt%	0.7108	0.9592	0.7431	0.0702
Si / wt%	0.1002	0.1192	0.1071	0.0035
P / wt%	0.0130	0.0310	0.0216	0.0068
S / wt%	0.0079	0.0215	0.0107	0.0048
Cr / wt%	0.0542	0.0800	0.0745	0.0106
Ni / wt%	0.0143	0.0900	0.0322	0.0123
Mo / wt%	0.0030	0.0300	0.0097	0.0030
Ti / wt%	0.0002	0.0060	0.0023	0.0015
Nb / wt%	0.0005	0.0060	0.0026	0.0013
V / wt%	0.0012	0.0045	0.0038	0.0013
Al / wt%	0.0059	0.0960	0.0310	0.0106
N / ppm	18.71	52.58	41.49	7.98
B / ppm	0.33	3.00	1.91	1.05
Cu / wt%	0.0150	0.0450	0.0360	0.0137
T_{FR} / ℃	866.50	983.50	895.49	42.56
T_C / ℃	534.28	600.44	596.24	7.50
σ_U / MPa	550.70	647.37	581.93	23.18
Uncertainty / MPa	42.19	89.89	66.28	14.16

Table 3.5 Characteristics of results of simulation for 600 MPa.

	Minimum	Maximum	Mean	Standard Deviation
C / wt%	0.4444	0.4656	0.4491	0.0091
Mn / wt%	0.6964	0.7885	0.7578	0.0447
Si / wt%	0.1085	0.1085	0.1085	0
P / wt%	0.0130	0.0310	0.0160	0.0069
S / wt%	0.0085	0.0085	0.0085	0
Cr / wt%	0.0800	0.0800	0.0800	0
Ni / wt%	0.0300	0.0326	0.0301	0.0006
Mo / wt%	0.0099	0.0100	0.0099	0
Ti / wt%	0.0015	0.0020	0.0016	0.0002
Nb / wt%	0.0013	0.0020	0.0015	0.0003
V / wt%	0.0015	0.0045	0.0042	0.0010
Al / wt%	0.0180	0.0320	0.0265	0.0070
N / ppm	43.50	43.50	43.50	0
B / ppm	0.47	3.00	1.02	0.77
Cu / wt%	0.0150	0.0150	0.0150	0
T_{FR} / ℃	866.50	866.5	866.5	0
T_C / ℃	596.00	622.37	603.32	12.15
σ_U / MPa	760.77	810.10	773.41	12.01
Uncertainty / MPa	77.28	118.92	92.52	12.46

Table 3.6 Characteristics of results of simulation for 800 MPa.

3.1.2. The elongation model

Table 3.7 shows the characteristics of simulations in which the objective was an elongation of 35 %. Some 129 combinations of input variables were discovered for which the target value had less than ± 15 percent uncertainty.

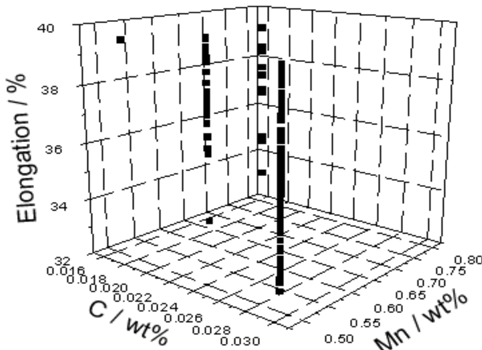
	Minimum	Maximum	Mean	Standard Deviation.
C / wt%	0.0168	0.0291	0.0269	0.0040
Mn / wt%	0.5026	0.7885	0.5354	0.0851
Si / wt%	0.0541	0.1085	0.0946	0.0238
P / wt%	0.0100	0.0310	0.0134	0.0047
S / wt%	0.0081	0.0215	0.0095	00029
Cr / wt%	0.0325	0.0800	0.0675	0.0210
Ni / wt%	0.0064	0.0300	0.0298	0.0021
Mo / wt%	0.0100	0.0300	0.0135	0.0076
Ti / wt%	0.0016	0.0060	0.0023	0.0011
Nb / wt%	0.0012	0.0020	0.0017	0.0004
V / wt%	0.0011	0.0045	0.0030	0.0015
Al / wt%	0.0320	0.0960	0.0496	0.0285
N / ppm	12.43	43.50	43.23	2.75
B / ppm	0.90	2.22	1.00	0.16
Cu / wt%	0.0031	0.0450	0.0325	0.0142
T_{FR} / ℃	817.95	983.5	900.11	62.22
T_C / ℃	440.72	629.51	521.48	56.39
ε / %	32.55	39.98	36.66	1.74
Uncertainty / %	2.00	5.24	3.95	0.87

Table 3.7 Characteristics of results of simulation for 35 %.

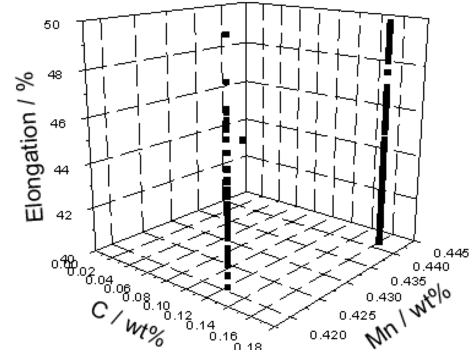
This simulation was used to check the behavior of the combining of neural network and genetic algorithm. This target is expected to be achieved easily since it is about average value of dataset used to create neural network model (Table 2.1).

Tables 3.8 and 3.9 show the characteristics of results from simulations for 45 % and 55 % respectively. Target of 45% was to check the effect of combining of neural network and genetic algorithm. This value is higher than the average elongation of typical ferrite-pearlite steels [Bhadeshia, 1998; Callister, 2007]. Target of 55% was used to test the high values of elongation which we can't reach easily (Figure 3.2).

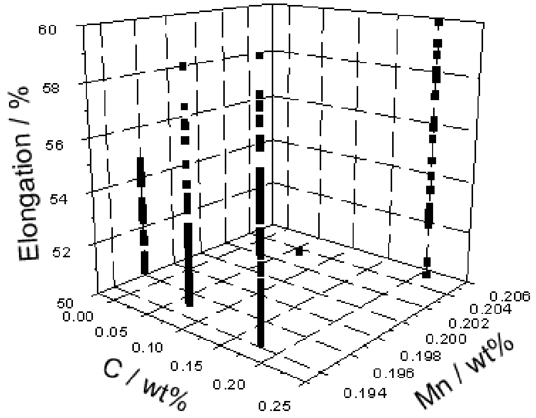
It is expected that domains of input parameters for higher elongation is smaller than for lower elongation simulation, because the higher the elongation, the harder the steels be obtained. However, In 35% case, 129 varieties were obtained. In 45% case, 171 varieties of inputs were discovered and in 55% case, 175 varieties of inputs were found. It means that it is possible to get even high elongation steels in this alloy system. Figure 3.9 shows the combinations of C and Mn in these simulations.



(a)



(b)



(c)

Figure 3.9 Combinations of the input variables which lead to the elongation to failure. (a) 35 % (b) 45 % (c) 55 %

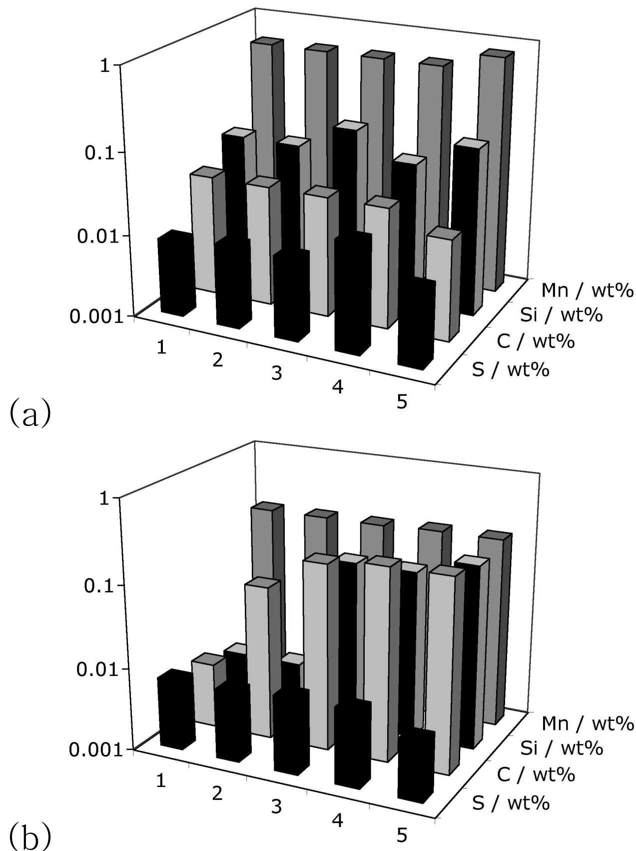


Figure 3.10 Combinations of variables which lead to an elongation of (a) 35%, (b) 55%

Figure 3.10 shows both cases of 35% and 55%. The range of compositions capable of producing the required elongation is very limited. This implies that the domain of compositions becomes much smaller if an attempt is made to simultaneously satisfy both strength and elongation targets. A high elongation corresponds to a high carbon and low manganese concentration and vice versa. A high concentration of carbon would lead to a greater fraction of pearlite and hence a larger degree of work hardening; the latter in turn would delay plastic instability and hence enhance

elongation.

	Minimum	Maximum	Mean	Standard Deviation
C / wt%	0.0150	0.1610	0.1487	0.0208
Mn / wt%	0.4205	0.4413	0.4354	0.0095
Si / wt%	0.0677	0.1255	0.0718	0.0077
P / wt%	0.0015	0.0310	0.0146	0.0050
S / wt%	0.0013	0.0215	0.0088	0.0023
Cr / wt%	0.0128	0.2400	0.0794	0.0175
Ni / wt%	0.0160	0.0300	0.0281	0.0039
Mo / wt%	0.0100	0.0300	0.0207	0.0096
Ti / wt%	0.0002	0.0060	0.0025	0.0013
Nb / wt%	0	0.0060	0.0021	0.0010
V / wt%	0.0003	0.0045	0.0036	0.0015
Al / wt%	0.0026	0.0960	0.0315	0.0229
N / ppm	31.37	73.27	41.02	7.58
B / ppm	0.36	1	0.81	0.28
Cu / wt%	0.0005	0.0450	0.0293	0.0151
T_{FR} / ℃	829.67	983.50	910.55	57.20
T_C / ℃	596.00	832.00	658.67	61.57
ε / %	40.06	49.98	43.92	2.47
Uncertainty / %	2.13	6.74	4.57	1.23

Table 3.8 Characteristics of results of simulation for 45 %.

	Minimum	Maximum	Mean	Standard Deviation
C / wt%	0.0060	0.2117	0.1212	0.0762
Mn / wt%	0.1941	0.2050	0.1963	0.0040
Si / wt%	0.0044	0.1665	0.0595	0.0526
P / wt%	0.0121	0.0310	0.0154	0.0032
S / wt%	0.0053	0.0215	0.0079	0.0012
Cr / wt%	0.0178	0.0800	0.0542	0.0258
Ni / wt%	0.0054	0.0300	0.0225	0.0077
Mo / wt%	0.0100	0.0300	0.0208	0.0095
Ti / wt%	0.0013	0.0060	0.0028	0.0017
Nb / wt%	0.0014	0.0023	0.0019	0.0002
V / wt%	0.0015	0.0045	0.0043	0.0008
Al / wt%	0.0159	0.0960	0.0489	0.0290
N / ppm	9.49	43.50	37.59	9.51
B / ppm	0.08	1	0.72	0.35
Cu / wt%	0.0005	0.0450	0.0240	0.0182
T_{FR} / ℃	857.13	983.50	923.53	58.21
T_C / ℃	578.92	832.00	773.79	87.19
ε / %	50.08	59.97	53.59	2.30
Uncertainty / %	1.84	8.21	5.59	2.05

Table 3.9 Characteristics of results of simulation for 55 %.

3.2. Multiple objective model

There is only one difference between single the objective and multi-objective models in interpreting the results, i.e., the permitted ranges of uncertainties. There are two uncertainties to be considered, one for the ultimate tensile strength, the other for elongation to failure. A larger uncertainty of ± 30 percent was allowed for each target in the multi-objective model, in order to give a larger probability of getting the desired values than was the case with the single objective models.

3.2.1. 400 MPa strength in combination with elongation

Table 3.10 shows the number of results for each simulation in which the target set at 400 MPa strength and the specified values of elongation. The number of solutions decrease as the target elongation is increased.

Simulation	400 MPa, 35 %	400 MPa, 45 %	400 MPa, 55 %
A number of solutions	156	71	1

Table 3.10 A number of solutions of 400 MPa strength in combination with elongation.

These results are reliable because as expected, it is hard to achieve 400 MPa with very high elongation. Figure 3.11 shows the ultimate tensile strength versus elongation from the results. Table 3.11 shows the single outcome of the 400 MPa with 55 % elongation simulation.

UTS (MPa)	Uncertainty of UTS (MPa)	Elongation (%)	Uncertainty of Elongation (%)
427.17	89.09	52.11	9.93

Table 3.11 The result of 400 MPa strength with 55 % elongation.

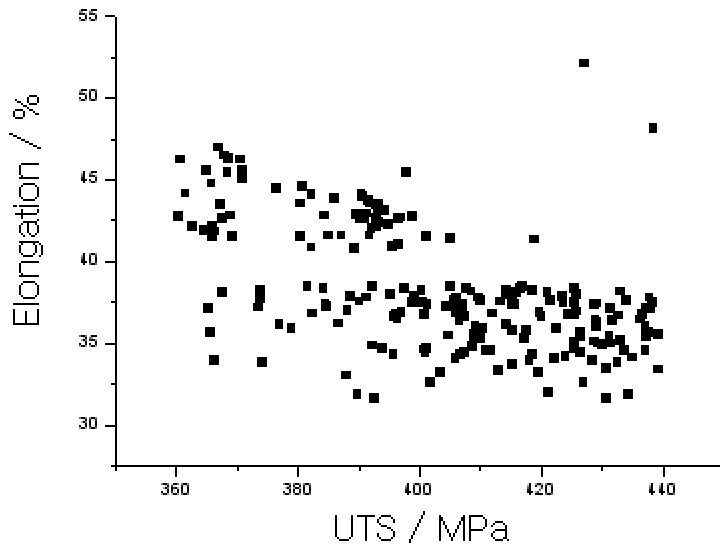


Figure 3.11 UTS vs. EL from 400 MPa strength in combination with elongation.

3.2.2. 600 MPa strength in combination with elongation

Table 3.12 shows the number of results for each simulation in which target set at 600 MPa strength and the specified values of elongations and Figure 3.12a shows the UTS and elongation diagram for these simulations. In contrast with the results of Chapter 3.2.1, the number of solutions does not decrease as target elongation is increased.

Simulation	600 MPa, 35 %	600 MPa, 45 %	600 MPa, 55 %
A number of solutions	26	33	1

Table 3.12 A number of solutions of 600 MPa strength in combination with elongation

This can be evidence of the possibility to make new steels which have about 600 MPa strength with 45 % elongation. Table 3.13 shows the characteristics of that simulation and Figure 3.13b shows the results of that with each uncertainty. There can be some risks to reach the targets because uncertainties are high. However, though it shows high uncertainties, the work suggests the design of new steels.

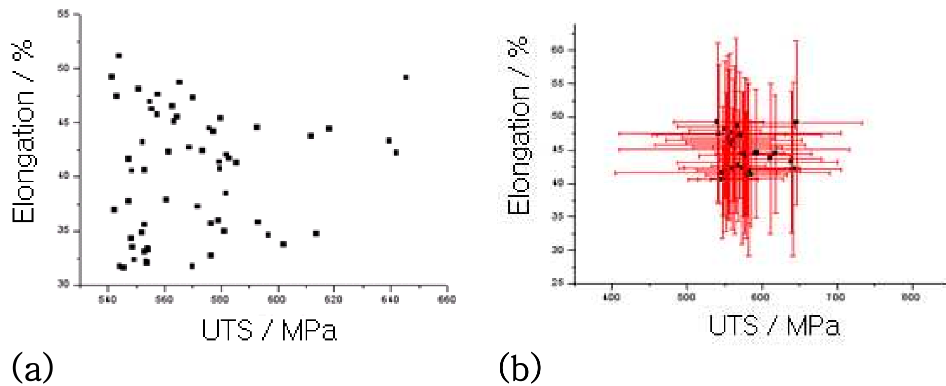


Figure 3.12 UTS vs. Elongation (a) 600 MPa strength in combination with elongation. (b) 600 MPa vs. 45 % with each uncertainty.

	Minimum	Maximum	Mean	Standard Deviation
C / wt%	0.2558	0.2768	0.2660	0.0107
Mn / wt%	0.4671	0.8931	0.7994	0.1069
Si / wt%	0.0086	0.1170	0.0904	0.0295
P / wt%	0.0202	0.0310	0.0283	0.0044
S / wt%	0.0007	0.0160	0.0069	0.0027
Cr / wt%	0.0279	0.1506	0.0450	0.0323
Ni / wt%	0.0111	0.0549	0.0326	0.0079
Mo / wt%	0.0068	0.0300	0.0127	0.0076
Ti / wt%	0	0.0018	0.0006	0.0005
Nb / wt%	0.00001	0.0041	0.0015	0.0008
V / wt%	0.0017	0.0046	0.0041	0.0010
Al / wt%	00.0285	0.0960	0.0535	0.0308
N / ppm	1.97	130.50	30.05	21.72
B / ppm	0.06	1.60	0.47	0.50
Cu / wt%	0.0081	0.0450	0.0364	0.0123
T_{FR} / ℃	764.23	983.50	896.26	64.16
T_C / ℃	541.45	656.60	594.59	15.08
σ_U / MPa	541.28	645.48	575.23	27.82
Uncertainty of σ_U / MPa	33.27	153.93	69.25	31.73
ε / %	40.59	49.23	44.43	2.66
Uncertainty ε / %	5.32	13.06	10.07	2.09

Table 3.13 Characteristics of results of simulation for 400 MPa with 45 %

3.2.3. 800 MPa strength in combination with elongation

Table 3.14 shows the number of results for each simulation in which target set at 800 MPa strength and the specified values of elongations. It seems that it is

impossible to obtain high UTS with high elongation. The combinations of 800 MPa with 45 % and 800 MPa with 55 % is indeed possible but the number of cases is very limited.

Simulation	800 MPa, 35 %	800 MPa, 45 %	800 MPa, 55 %
A number of solutions	0	1	1

Table 3.14 A number of solutions of 800 MPa strength in combination with elongation

3.3. Summary

Simulations based on neural networks combined with genetic algorithms have been conducted in two ways. Single objective simulations using a combination of neural network and genetic algorithms have been used to explore particular strength or ductility values. The results make metallurgical sense. However, it is not wise to focus an individual property. In the elongation simulations, there were some unphysical solutions. This can be improved by considering several properties at once. Thus, it is necessary to focus on multi-objective combined models. In multi-objective simulation, both the ultimate tensile strength and ductility are considered simultaneously. With this, not only is it possible to avoid the side effects of the single objective simulation, but new steels are indicated (Chapter 3.2.2). It is necessary to conduct real experiments to confirm the calculations.

IV. Summary and Future Work

4.1. Summary

The aim of the work presented in this thesis was to examine the possibility of defining the domains of steels which have essentially identical properties but different compositions and processing parameters, by making a combination of neural networks and genetic algorithms. The neural network models had been developed in previous work [Ryu, 2008], covering the ultimate tensile strength and the tensile elongation to failure of the steels with mixed microstructures of ferrite and pearlite. These models were combined with genetic algorithms to identify systematically the population of independent parameters which lead to a particular strength or ductility.

The initial work using the combined models focused on individual targets, i.e., either the ultimate tensile strength or the ductility. The outcomes of the search have been found to be physically meaningful in terms of the metallurgy expected of hot-rolled steels with a ferrite-pearlite microstructure. Naturally, solutions either did not arise or were few and far between when the target values were too ambitious. It is important to realize that constraints were placed during the computer search first by designing the permitted deviation from the set target, and second by rejecting solutions whose uncertainty is unacceptable. This latter constraint is particularly important in dealing with non-linear models became ordinary neural networks

without the facility of the modeling uncertainty suffer from the danger of unreasonable extrapolation. The Bayesian method adopted here greatly limits this danger by demonstrating with calculated uncertainty, the cases where calculations are being done in regions of input space where knowledge is sparse or non-existant.

Engineering design rarely involves the definition of just one target. For example, it is bad practice to maximize strength without considering the consequences on ductility. This poor practice in typical of 'new materials' such as those working on ultrafine grained structure or amorphous metals [Fukuda et. al., 2002; Schuh *et al.*, 2007]. Therefore, an effort was made to construct a multi-objective algorithm. Two objectives, the ultimate tensile strength and the ductility, were therefore considered simultaneously by combining the two neural network models with a genetic algorithm. This revealed more interesting predictions than the single objective approach. Not only normally unreachable domains discovered, but also it was possible to find the new domains of input parameters for the target steels, although it remains to be proven whether this is a consequence of the larger level of permitted uncertainty. The results might be resolved by performing real experiments.

To summarize, the combination of the neural network and genetic algorithm is a good methodology for finding domains of input parameters which lead to particular mechanical properties of steels. The computational cost of doing so is large but not impossibly large, and a trial and error search of the input space probably should incur a greater expense and may not reach optimum solutions. The work has also highlighted the need for researching new grades of steels. It would be helpful to produce such steels discovered by the method and to test the mechanical properties.

4.2. Future work

In this work, two models for the ultimate tensile strength model and the elongation to failure were considered. However, the approach used can be applied to any other models produced by neural network, or even any physical model include ab initio methods. Targets can be anything such as Charpy toughness or hardness. It is possible that the domains of input parameters leading to any particular choice of mechanical properties can be obtained.

Two objectives were considered at once in this work. Actually, several models can be considered simultaneously by genetic algorithm, but the more targets are combined, the more complexity should be incurred. It means that in order to use the genetic algorithm on multiple aspect optimization problems, multiple objective functions should be combined into a fitness function.

However, there is normally no way to satisfy the multiple objectives at the same time because there may exist some contradictions among them like the inverse relationship between the ultimate tensile strength and the elongation to failure. Thus the multiple paths to the targets should be selected by considering carefully the priority of the problem. In this case, it is necessary that optimal paths for multiple objectives should be searched by considering all possible tradeoffs among the multiple and conflicting objective functions.

So, the concept of Pareto optimality, which is known as best method for multi-objective genetic algorithm, is needed. The fitness function of the two-objective model in this work is simply coordinated; randomly weighted for fitness from each

objective and linearly summed. It can be improved by considering Pareto optimal solution to reflect the complexity of multiple objectives.

References

Alder, J. F. and Phillips, V. A., *J. Inst. Met.* 83: 80-86, 1954.

Ashby, M. F., Oxide Dispersion Strengthening, Ansell, G. S., Cooper, T. D. and Lenel, F. V., Eds., Gorden and Breach, New York, 143, 1968.

Bhadeshia, H. K. D. H. and Edmonds, D. V., Analysis of the Mechanical Properties and Microstructure of a High-Silicon Dual-Phase Steels, *Metal Science*, 14: 41-49, 1980.

Bhadeshia, H. K. D. H., Alternatives to the ferrite-pearlite microstructures, *Materials Science Forum* 29: 284-286, 1998.

Bhadeshia, H. K. D. H., Neural networks in materials science, *ISIJ International*, 39: 966-979, 1999.

Bhadeshia, H. K. D. H., Bainite in steels, IOM Communications Ltd, London, 2nd edition, 2001.

Bhadeshia, H. K. D. H., The Importance of Uncertainty, *Neural Networks and Genetic Algorithms in Materials Science and Engineering*, Tata McGraw-Hill

Publishing Company, Bengal Engineering and Science University, India, 2006.

Bishop, C. M., Neural Networks for Pattern Recognition, Oxford University Press, London, 1995.

Callister, W. D., Material Science and Engineering: An Introduction 7th edition, Wiley, 2007.

Carlton, C. E., Ferreira, P. J., What is Behind the Inverse Hall-Petch Behavior in Nanocrystalline Materials?, *Materials Research Society Symposium Proceedings*, 976:19-24, 2007.

Chakraborti, N., Genetic algorithms in materials design and processing, *International Materials Reviews*, 49: 246-260, 2004.

Chatterjee, S., Transformations in TRIP-assisted steels: microstructure and properties, Ph.D. Thesis, University of Cambridge, 2006.

Chatterjee, S. and Bhadeshia, H. K. D. H., δ-TRIP steel, *Materials Science and Technology*, 23: 819-827, 2007.

Chen, B. L., Optimization Theories and Algorithms, Tsinghua University Press, Beijing, 2002.

Conrad, H. and Narayan, J., On the grain size softening in nanocrystalline materials, *Scripta Materrialia*, 42:1025–30, 2000.

Cottrell, A. H., Dislocations and Plastic Flow in Crystals, Oxford University Press, London, 1953.

Delorme, A., Genetic algorithm for optimization of mechanical properties, Technical report, University of Cambridge, 2003.

Dieter, G. E., Mechanical Metallurgy SI Metric Edition, McGraw Hill, London, 307−308, 1988.

Felbeck, D. K. and Atkins, A. G., Strength and Fracture of Engineering Solids, Prentice-Hall Inc., 1984.

Fujii, Hidetoshi, MacKay, D. J. C. and Bhadeshia, H. K. D. H., Bayesian neural network analysis of fatigue crack growth rate in nickel base superalloys, *ISIJ International*, 36: 1373-1382, 1996.

Fukuda, Y., Oh-ishi, K., Horita, Z. and Langdon, T. G., Processing of a low-carbon steel by equal-channel angular pressing, *Acta Materialia*, 50: 1359-1368, 2002

Goldberg, D. E., Genetic Algorithms in Search, Optimization and Machine Learning, Addison-Wesley, 1989.

Grefenstette, J. J., Optimization of Control Parameters for Genetic Algorithms, *IEEE Transactions on Systems, Man and Cybernetics*, 16: 122-128, 1986.

Hall, E. O., Proc. Phys. Soc. London, 643: 747, 1951.

Honeycombe, R. W. K., The Plastic Deformation of Metals, Edward Arnold, 1968.

Hollomon, J. H., *Trans AMIE,* 162:268-290, 1945.

Irvine, K. J., Pickering, F. B., Heselwood, W. C. and Atkins, M., *Journal of the Iron and Steel Institute*, 195: 54-67, 1957

Jaiswal, S. and Mclvor, I. D., *Ironmaking and Steelmaking*, 16: 49, 1989.

Kelly, A. and Nicholson, R. B., Progress in Materials Science, 10 - 4, Pergamon Press, New York, 1963.

Kirkaldy, J. S. and Venugopalan, D., Prediction of microstructure and hardenability in low alloy steel, *Phase Transformations in Ferrous Alloys*, eds A R Marder & J I Goldstein, TMS-AIME, Warrendale, Ohio, 125-148, 1984.

MacKay, D. J. C., Bayesian interpolation, *Neural Computation*, 4: 415-447, 1992a.

MacKay, D. J. C., A practical Bayesian framework for backpropagation networks, *Neural Computation*, 4: 448-472, 1992b.

MacKay, D. J. C., Evidence framework applied to classification networks, *Neural Computation*, 4: 720-736, 1992c.

MacKay, D. J. C., Bayesian non-linear modelling with neural networks, University of Cambridge programme for industry: Modelling Phase Transformations in Steels, 1995a.

MacKay, D. J. C., Probable networks and plausible predictions – a review of practical bayesian methods for supervised neural networks, *Computation in Neural Systems*, 6: 469-505, 1995b.

MacKay, D. J. C., Information theory, inference and learning algorithms, Cambridge University Press, U. K., 2003.

McQueen, H. J., Blum, W., Zhu, Q. and Demuth, V., Advances in Hot Deformation Textures and Microstructures, *TMS-AIME, Warrendale, PA*, 235–250, 1994.

Michalewicz, Z., Genetic Algorithms + Data Structures = Evolution Programs, Springer, 1996.

Morrison, W. B., The effect of grain size on the stress-strain relationship in low carbon steel, *Trans. ASM*, 59:824-846, 1966.

Murata, T., Ishibuchi, H. and Tanaka, H., Multi-Objective Genetic Algorithm and its Applications to Flowshop Scheduling. Computers and Industrial Engineering, 30(4): 957-968, 1996.

Obitko, M. and Slavik. P., Visualization of Genetic Algorithms in a Learning Envionment, Spring Conference on Computer Graphics, SCCG'99, *Bratislava: Comenius University*, 101-106, 1999.

Orowan, E., Discussion in "Symposium on Internal Stress", *Inst. Metals*, London, 451, 1947.

Osborne, M. J. and Rubenstein, A., A Course in Game Theory, *MIT Press*, 1994.

Petch, N. J., J. Iron Steel Inst. London, 173: 25, 1953.

Pickering, F. B., Physical Metallurgy and the Design of Steels, *Applied Science Publishers*, London, 50, 1978.

Reed, R. and Bhadeshia, H. K. D. H., Reconstructive austenite-ferrite transformation in low alloy steel, *Materials Science and Technology*, 8: 421-435, 1992.

Ryu, J. H., Model for mechanical properties of Hot-Rolled Steels, Master Thesis, POSTECH, 2008.

Schuh, C. A., Hufnagel, T. C. and Ramamurty, U., Mechanical behavior of amorphous alloys, *Acta Materialia*, 55: 4067-4109, 2007.

Taylor, G. I., Mechanism of plastic deformation of crystals, *Proc. Roy. Soc.*, A145: 362, 1934.

Tsukatani, I., Hasimoto, S. and Inoue, T., Effect of Silicon and Mananese Addition on Mechanical Properties of High-Strength Hot-Rolled Sheet Steel Containing Retained Austenite, *ISIJ international*, 31: 992-1000, 1991

Zener, C. and Hollomon, H., *J. Appl. Phys.*, 15: 22, 1944.

Appendix A

This is the documentation for the single objective genetic algorithm program, as described in chapter 2.2.1. The target property is the UTS. This is associated documentation following the MAP format,

http://www.msm.cam.ac.uk/map/mapmain.html.

Program MAP_HOTROLLEDSTEEL_UTS

1. Provenance of code.

2. Purpose of code.

3. Specification.

4. Description.

5. References.

6. Parameter descriptions.

7. Error indicators.

8. Accuracy estimate.

9. Any additional information.

10. Example.

11. Auxiliary subroutines required.

12. Keywords.

13. Sources.

1. Provenance of source code.

Min Sung Joo and H. K. D. H. Bhadeshia

Graduate Institute of Ferrous Technology (GIFT)

Pohang University of Science and Technology

Pohang, Kyoungbuk, Republic of Korea

athpimo@postech.ac.kr

2. Purpose of code.

An application of the genetic algorithm (GA) for reaching a solution given a fitting function. This can in theory be applied to any problem, where a database of inputs and outputs has trained a neural network.

3. Specification.

Language: C

Product form: Source code and executable files for UNIX/Linux machines.

Complete program.

4. Description.

The single_UTS.tar.gz file, which can be downloaded from here, contains the

following files. A working version of a GA is also included to optimize the ultimate tensile strength of the hot-rolled steels as a function of chemical composition and processing variables.

genetic.c	- A C program used to run the genetic algorithm.
gareadme.doc	- A manual giving further details about running and using the GA program for altering different settings to optimize the process.
gareadme.txt	- A txt version of *gareadme.doc*
single_UTS	- An executable program of *genetic.c*.
generate44	- It reads the normalized input data file, *input/norm_test.in*, and uses the weight files in subdicrectory *input*. The results are written to the temporary output file *_ot, _out, _res* and *_sen*.
score_output	- A file that contains the scroes of each chromosome after the program has concluded.
unnormalise/treatout.c	- A C program for un-normalising the output data files.
unnormalise/treatout	- An executable program of *unnormalise/treatout.c*.
unnormalise/nn-output_b	- A file which contains the normalized inputs and outputs of the GA after the program has concluded.
unnormalise/result	- A file which contains the unnormalized values for inputs, outputs and error of each chromosome after unnormalising data in the normalized *nn-output_b* file.

input/labels.txt - A list of input variables.

input /nn-input - Normalised inputs file (target and accuracy values for the GA).

input /norm_test.in - A text file which contains the normalized input variables initialized by the GA to be fed into the neural network.

The following are concerned with the neural network files that work with the GA, but can be changed according to user requirements:

*input /_w*f* - The weights files for the different models.

input /.lu* - Files containing information for calculating the size of the error bars for the different models.

input /spec1.t1 - An altered version of *spec.t1* which is a dynamic file, created by neural network. It is read by the program *generate44*.

input /outran.x - A normalized output file that was created when developing the model. It is accessed by *generate44* via *input/spec1.t1*.

input /MINMAX - Minimum and maximum values for the input variables used in the original database.

The file gareadme.txt contains a manual, which provides a detailed set of instructions for downloading and running the GA program. A summary of the

information in this manual is given here.

5. References.

Ryu, J. H., Model for mechanical properties of Hot-Rolled Steels, Master Thesis, POSTECH, 2008.

Shah, I., Tensile Properties of Austenite Stainless Steel, M. Phil. Thesis, University of Cambridge, 2002.

Delorme, A., Genetic algorithm for optimization of mechanical properties, Technical report, University of Cambridge, 2003.

6. Parameter descriptions.

Input parameters

To specify the target value and accuracy desired, *input/nn-input* must be amended. The normalized target value relates to a real value, according to *the input/MINMAX* file used.

Row1	Normalised target value
Row2	Accuracy (in decimal e.g. 0.1 for 10%)

To specify the permitted range of the output, *genetic.c* must be changed.

#define TARGET_UNNORMAL	Unnormalised target value
#define MAX_UNNORMAL	Maximum of target value in

	intput/MINMAX
#define MIN_UNNORMAL	Minimum of target value in *input/ MINMAX*
#define TARGET_PERCENT_PERMIT	Permitted range for the target (in decimal e.g. 0.1 for 10%)
#define UNCERTAINTY_PERCENT_PERMIT	Permitted range for the uncertainty (in decimal e.g. 0.1 for 10%)

To initiate the GA search, the inputs are randomly generated and placed in *input/norm_test.in*. It should be noted that each chromosome generally relates to a different steel composition, but this could change over the course of optimization. For the current ultimate tensile strength model, the composition and processing variables, totally 17 data items, are specified:

Gene number	Variable
1	Normalised C, wt%
2	Normalised Mn, wt%
3	Normalised Si, wt%
4	Normalised P, wt%
5	Normalised S, wt%
6	Normalised Cr, wt%
7	Normalised Ni, wt%

8	Normalised Mo, wt%
9	Normalised Ti, wt%
10	Normalised Nb, wt%
11	Normalised V, wt%
12	Normalised Al, wt%
13	Normalised N, ppm
14	Normalised B, ppm
15	Normalised Cu, wt%
16	Normalised Finishing rolling temperature, ˚C
17	Normalised Coiling temperature, ˚C

Each input is normalized using the equation:

Normalized value = (value – min)/(max-min) – 0.5

Where the values for min and max are defined as follows:

Gene number	Variable	Min	Max
1	C, wt%	0.0204	0.8684
2	Mn, wt%	0.1670	1.4100
3	Si, wt%	0.0000	0.2170
4	P, wt%	0.0040	0.0220
5	S, wt%	0.0020	0.0150

6	Cr, wt%	0.0000	0.1600
7	Ni, wt%	0.0000	0.0600
8	Mo, wt%	0.0000	0.0200
9	Ti, wt%	0.0000	0.0040
10	Nb, wt%	0.0000	0.0040
11	V, wt%	0.0000	0.0030
12	Al, wt%	0.0000	0.0640
13	N, ppm	0.00	87.00
14	B, ppm	0.00	2.00
15	Cu, wt%	0.0000	0.0300
16	Finishing rolling temperature, ˚C	808.00	925.00
17	Coiling temperature, ˚C	478.00	714.00

Output parameters

Two output files are produced by the GA program: *unnormalise/nn-output_b* and *score-output.*

score-output simply prints out the scores for each chromosome.

unnormalise/nn-output_b contains the inputs, prediction and (prediction + error). This is done for the best chromosomes within all populations:

Column 1-17	The normalized predicted inputs

Column 18	The normalized predicted output
Column 19	A value for the error, which includes the experimental noise of the database, an estimation of the uncertainty in the prediction and the test error. However, this is added to the prediction value so that the *input/MINMAX* file can easily handle the value.

The normalized values in all columns must be un-normalised using the equation:

actual value = (normalized value + 0.5) * (max-min) + min

The C program, *unnormalise/treatout.c*, is used to translate the output files to produce the actual values of inputs and outputs which are written to *unnormalise/result*.

7. Error indicators.

None.

8. Accuracy estimate.

See:

Input parameters, output parameters.

9. Any additional information.

See:

gareadme.txt

10. Example.

1. Program text

Complete program

2. Program data

The input file (*input/nn-input*) is:

-0.385042

0.1

The input variables (*genetic.c*) are:

#define TARGET_UNNORMAL	400.0
#define MAX_UNNORMAL	1039.0
#define MIN_UNNORMAL	317.0
#define TARGET_PERCENT_PERMIT	0.1
#define UNCERTAINTY_PERCENT_PERMIT	0.15

3. Program results

The output file unnormalise/nn-output_b, which contains the normalised values for

the ultimate tensile strength model including compositions and processing variables (following result is only single selected case):

C	**Mn**	**Si**	**P**	**S**
-0.283412	-0.067178	-0.420334	0.000000	0.000000
Cr	**Ni**	**Mo**	**Ti**	**Nb**
-0.346624	0.104346	-0.338664	0.000000	0.000000
V	**Al**	**N**	**B**	**Cu**
1.000000	0.000000	0.000000	-0.486950	0.378970
FRT	**CT**	**Pred**	**Pred + Err**	
0.000000	0.479550	-0.363828	-0.292822	

These are then run with *unnormalise/treatout.c*, where the normalised values are converted into the actual values to *unnormalise/result*:

C	**Mn**	**Si**	**P**	**S**
0.204067	0.704998	0.017288	0.013000	0.008500
Cr	**Ni**	**Mo**	**Ti**	**Nb**
0.024540	0.036261	0.003227	0.002000	0.002000
V	**Al**	**N**	**B**	**Cu**

0.004500	0.032000	43.50000	0.026100	0.026369

FRT	**CT**	**Pred**	**Pred + Err**
866.5000	709.1738	415.3162	51.2663

11. Auxiliary subroutines required.

None.

12. Keywords.

hot-rolled steel, genetic algorithm, ultimate tensile strength, neural network.

13. Sources.

genetic.c

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#include <sys/time.h>

/* NN related */
#define NUM          17     /* Total number of genes
*/
#define LIMIT        150    /* Maximum number of
inputs the system can handle */
#define SESSIONS     1000   /*      Number      of
generations that we'll put the system through */
/* GA related */
#define POPS         2      /*      Number      of
populations */
#define SIZE         17     /* Size of vector in the
genetic algorithms */
#define MAXPOP                  20         /*  Size
of population */
#define BESTPOP                 8          /*
Number of individuals taken from the best */
#define SELPOP                  12         /*
SELPOP-BESTPOP = Number of people selected randomly to exchange
genes within their own population */
#define NEWPOP                  16         /*
NEWPOP-SELPOP = Number of new people created randomly on each
gen. */
#define MUT1         18         /*  MUT1-NEWPOP  =
Number of genes that are severely mutated */
#define MIXGEN                  1          /*
Number of generations between population mixing */

/* result selection */
#define TARGET_UNNORMAL
             400.0
#define MAX_UNNORMAL                          1039.0
```

```
#define MIN_UNNORMAL                                    317.0
#define TARGET_PERCENT_PERMIT                           0.1
#define UNCERTAINTY_PERCENT_PERMIT  0.15

#define CAR          -0.03703703     /* Carbon */
#define MAN          0.28512388      /* Manganese */
#define SIL          0.03913045      /* Silicon */
#define PHO          0.03125002      /* Phosphorus */
#define SUL          -0.3222222      /* Sulphur */
#define CHR          -0.1            /*
Chromium */
#define NIC          0.1             /*
Nickel */
#define MOL          -0.33848798     /* Molybdenum */
#define TIT          -0.42857143     /* Titanium */
#define NIO          -0.48947367     /* Niobium */
#define VAN          -0.5            /*
Vanadium */
#define ALU          -0.5            /*
Aluminium */
#define NIT          -0.11728396     /* Nitrogen */
#define BOR          -0.4333333      /* Boron */
#define COP          -0.01428570     /* Copper */
#define FRT          0.1             /*
Finishin Mill Temperature */
#define CIT          0.1             /*
COiling Temperature */

/* BESTPOP x POPS = no of lines in output file*/

typedef struct
{
        float p[NUM];
} vector;


/* NN related */
vector test[LIMIT], w1,target;
int hits[LIMIT], total;

/* GA related */
float pop[POPS][MAXPOP][SIZE];
double score[POPS][MAXPOP];
double result[POPS][MAXPOP];
double error[POPS][MAXPOP];
double error2[POPS][MAXPOP];

/*----------------------------------------------------------------------*\
   |
|
   |                                                    Randomize
|
   |
|
   \*----------------------------------------------------------------------*/
randomize()
{
        struct timeval tp;
        struct timezone tzp;

        /* Use time of day to feed the random number generator
seed */
        gettimeofday( &tp, &tzp);
        srandom( tp.tv_sec );
}


/*----------------------------------------------------------------------*\
   |
|
   |  irand( range )  - return a random integer in the range 0..(range-1)
|
   |
|
   \*----------------------------------------------------------------------*/
int irand( range ) int range;
{
        return( random() % range );
}
```

```
/*----------------------------------------------------------------------*\
   |
|
   |       scalar_mult    -    multiply    two    vectors
|
   |
|
   \*----------------------------------------------------------------------*/
float scalar_mult( x, y ) vector x, y;
{
        int i;
        float s = 0.0;
        for ( i = 0 ; i < NUM ; i++ )   s += ( x.p[i] * y.p[i] );
        return s;
}


/*----------------------------------------------------------------------*\
   |
|
   |  This function computes the NN's output for a certain input vector.
|
   |  The NN is constructed from 2 layers, first layer has 6 neurons,
|
   |       second       layer       has       1       neuron.
|
   |
|
   \*----------------------------------------------------------------------*/

net(p,i)
        int p,i;


{
        FILE *nmin,*nmin2,*nmin3,*fcount,*ferr;
        int j,k,l,m,sum,e,no_of_columns;
        char c,s[100],minmax_file[50];
        float
res2,res3,res4,res5,res6,res7,res8,res9,res10,res11,res12;
        float
err2,err3,err4,err5,err6,err7,err8,err9,err10,err11,err12;
        float
err_a,err_b,err_c,err_d,err_e,err_f,err_g,err_h,err_i,err_j,err_k;
        float err_1,err_1a;
        float
min[50],max[50],dummy,mean,stdev,dev,generation,k1,k2;
        double check_performance();
        FILE *ifp;



        printf("Calling net with p=%d and i=%d",p,i);
        if ((nmin = fopen("./input/norm_test.in","w")) == NULL )
{
                printf                          ("Can't
write ./input/norm_test.in");
                exit(1);
        } else {

                l=0;
                for (j=0;j<SIZE;j++) {
#if 1
                        fprintf(nmin,"%f
",w1.p[l]);
                        ++l;
#else
                        if (j != 5) {
                                /*
change this line according to which
        variables are wanted to be fixed */

        fprintf(nmin,"%f ",w1.p[l]);
                                ++l;
                        } else {
                                if (j ==
0){

        fprintf(nmin,"%f ",CAR);
                                }
```

```
                                        if (j ==
1) {
        fprintf(nmin,"%f ",MAN);
                                        }
                                        if (j ==
2) {
        fprintf(nmin,"%f ",SIL);
                                        }
                                        if (j ==
3) {
        fprintf(nmin,"%f ",PHO);
                                        }
                                        if (j ==
4) {
        fprintf(nmin,"%f ",SUL);
                                        }
                                        if (j ==
5) {
        fprintf(nmin,"%f ",CHR);
                                        }

                                        if (j ==
6) {
        fprintf(nmin,"%f ",NIC);
                                        }
                                        if (j ==
7) {
        fprintf(nmin,"%f ",MOL);
                                        }

                                        if (j ==
8) {
        fprintf(nmin,"%f ",TIT);
                                        }

                                        if (j ==
9) {
        fprintf(nmin,"%f ",NIO);
                                        }
                                        if (j ==
10) {
        fprintf(nmin,"%f ",VAN);
                                        }

                                        if (j ==
11) {
        fprintf(nmin,"%f ",ALU);
                                        }
                                        if (j ==
12) {
        fprintf(nmin,"%f ",NIT);
                                        }

                                        if (j ==
13) {
        fprintf(nmin,"%f ",BOR);
                                        }
                                        if (j ==
14) {
        fprintf(nmin,"%f ",COP);
                                        }

                                        if (j ==
15) {
        fprintf(nmin,"%f ",FRT);
```

```
                                        }

                                        if (j ==
16) {
        fprintf(nmin,"%f ",CIT);
                                        }
                                }
#endif
                        }
                        fprintf(nmin,"\n");
                        fclose(nmin);
                }
                sprintf(s,      "./generate44       ./input/spec1.t1
6 ./input/_wf1f ./input/_wf1f.lu> /dev/null");
                system(s);
                if ( (nmin2 = fopen("_out","r")) == NULL) {
                        printf ("Can't read _out stage 1");
                        exit(1);
                } else {
                        fscanf(nmin2,"%f %f",&res2,&err2);
                        printf("res2 err2 %f %f\n",res2,err2);
                        printf("Indices %d %d\n",p,i);

                        fclose(nmin2);
                }
                sprintf(s,      "./generate44       ./input/spec1.t1
6 ./input/_wf3f ./input/_wf3f.lu> /dev/null");
                system(s);
                if ( (nmin2 = fopen("_out","r")) == NULL) {
                        printf ("Can't read _out stage 2");
                        exit(1);
                } else {
                        fscanf(nmin2,"%f %f",&res3,&err3);
                        printf("res3 err3 %f %f\n",res3,err3);
                        fclose(nmin2);
                }

                sprintf(s,      "./generate44       ./input/spec1.t1
8 ./input/_wh2f ./input/_wh2f.lu> /dev/null");
                system(s);
                if ( (nmin2 = fopen("_out","r")) == NULL) {
                        printf ("Can't read _out stage 3");
                        exit(1);
                } else {
                        fscanf(nmin2,"%f %f",&res4,&err4);
                        printf("res4 err4 %f %f\n",res4,err4);
                        fclose(nmin2);
                }
                sprintf(s,      "./generate44       ./input/spec1.t1
8 ./input/_wh4f ./input/_wh4f.lu> /dev/null");
                system(s);
                if ( (nmin2 = fopen("_out","r")) == NULL) {
                        printf ("Can't read _out stage 4");
                        exit(1);
                } else {
                        fscanf(nmin2,"%f %f",&res5,&err5);
                        printf("res5 err5 %f %f\n",res5,err5);
                        fclose(nmin2);
                }

                sprintf(s,      "./generate44       ./input/spec1.t1
9 ./input/_wi4f ./input/_wi4f.lu> /dev/null");
                system(s);
                if ( (nmin2 = fopen("_out","r")) == NULL) {
                        printf ("Can't read _out stage 5");
                        exit(1);
                } else {
                        fscanf(nmin2,"%f %f",&res6,&err6);
                        printf("res6 err6 %f %f\n",res6,err6);
                        fclose(nmin2);
                }
                sprintf(s,      "./generate44       ./input/spec1.t1
10 ./input/_wj2f ./input/_wj2f.lu> /dev/null");
                system(s);
                if ( (nmin2 = fopen("_out","r")) == NULL) {
                        printf ("Can't read _out stage 6");
                        exit(1);
                } else {
```

```
			fscanf(nmin2,"%f %f",&res7,&err7);
			printf("res7 err7 %f %f\n",res7,err7);
			fclose(nmin2);
		}

		sprintf(s,		"./generate44		./input/spec1.t1
10 ./input/_wj4f ./input/_wj4f.lu> /dev/null");
		system(s);
		if ( (nmin2 = fopen("_out","r")) == NULL) {
			printf ("Can't read _out stage 7");
			exit(1);
		} else {
			fscanf(nmin2,"%f %f",&res8,&err8);
			printf("res8 err8 %f %f\n",res8,err8);
			fclose(nmin2);
		}

		sprintf(s,		"./generate44		./input/spec1.t1
10 ./input/_wj5f ./input/_wj5f.lu> /dev/null");
		system(s);
		if ( (nmin2 = fopen("_out","r")) == NULL) {
			printf ("Can't read _out stage 8");
			exit(1);
		} else {
			fscanf(nmin2,"%f %f",&res9,&err9);
			printf("res9 err9 %f %f\n",res9,err9);
			fclose(nmin2);
		}

		sprintf(s,		"./generate44		./input/spec1.t1
11 ./input/_wk1f ./input/_wk1f.lu> /dev/null");
		system(s);
		if ( (nmin2 = fopen("_out","r")) == NULL) {
			printf ("Can't read _out stage 9");
			exit(1);
		} else {
			fscanf(nmin2,"%f %f",&res10,&err10);
			printf("res10
err10 %f %f\n",res10,err10);
			fclose(nmin2);
		}

		sprintf(s,		"./generate44		./input/spec1.t1
15 ./input/_wo5f ./input/_wo5f.lu> /dev/null");
		system(s);
		if ( (nmin2 = fopen("_out","r")) == NULL) {
			printf ("Can't read _out stage 10");
			exit(1);
		} else {
			fscanf(nmin2,"%f %f",&res11,&err11);
			printf("res11
err11 %f %f\n",res11,err11);
			fclose(nmin2);
		}

		sprintf(s,		"./generate44		./input/spec1.t1
19 ./input/_ws4f ./input/_ws4f.lu> /dev/null");
		system(s);
		if ( (nmin2 = fopen("_out","r")) == NULL) {
			printf ("Can't read _out stage 11");
			exit(1);
		} else {
			fscanf(nmin2,"%f %f",&res12,&err12);
			printf("res12
err12 %f %f\n",res12,err12);
			fclose(nmin2);
		}

		result[p][i]						=
((res2+res3+res4+res5+res6+res7+res8+res9+res10+res11+res12)/11);
		printf("this is p %d %d\n", p,i);

		nmin3=fopen("./input/nn-input","r");
		fscanf(nmin3,"%f",&target.p[0]);

		err_a = pow ((res2-target.p[0]),2);
		err_b = pow ((res3-target.p[0]),2);
		err_c = pow ((res4-target.p[0]),2);
		err_d = pow ((res5-target.p[0]),2);
		err_e = pow ((res6-target.p[0]),2);
		err_f = pow ((res7-target.p[0]),2);
		err_g = pow ((res8-target.p[0]),2);
		err_h = pow ((res9-target.p[0]),2);
		err_i = pow ((res10-target.p[0]),2);
		err_j = pow ((res11-target.p[0]),2);
		err_k = pow ((res12-target.p[0]),2);

		err_1						=
(err2+err3+err4+err5+err6+err7+err8+err9+err10+err11+err12);
		err_1a = pow((err_1),2);

		printf("err_a is %f\n",err_a);
		printf("err_b is %f\n",err_b);
		printf("err_c is %f\n",err_c);
		printf("err_d is %f\n",err_d);
		printf("err_e is %f\n",err_e);
		printf("err_f is %f\n",err_f);
		printf("err_g is %f\n",err_g);
		printf("err_h is %f\n",err_h);
		printf("err_i is %f\n",err_i);
		printf("err_j is %f\n",err_j);
		printf("err_k is %f\n",err_k);

		printf("err_1 is %f\n",err_1);

		error2[p][i]=sqrt(err_1a);


		error[p][i] = sqrt (((err_1a)/11)+((err_a + err_b + err_c +
err_d + err_e + err_f + err_g + err_h + err_i + err_j + err_k)/11));
		printf("error pi is %f\n\n",error[p][i]);
		fcount=fopen("count","r");
		fscanf(fcount,"%d",&e);
		fclose(fcount);


		e++;
		fcount=fopen("count","w");
		fprintf(fcount,"%d",e);


		fclose(fcount);
		printf("\n-----------> sum %d\n",e);

		printf("In	net()	error	is	%f	res
is %f\n\n",error[p][i],result[p][i]);
		fclose(nmin3);

}


/*----------------------------------------------------------------------*\
   |
|
   |   pop_swap( p, a, b )   -   swap two vectors and scores in the
population p|
   |
|
   \*----------------------------------------------------------------------*/
pop_swap( p, a, b )
		int p, a, b;

{
		int t, i;
		float tg;
		/* Swap vector */
		for ( i = 0 ; i < SIZE ; i++ )

		{

			t = pop[p][a][i];
			pop[p][a][i] = pop[p][b][i];
			pop[p][b][i] = t;
		}

		/* Swap score */
		tg = score[p][a];
		score[p][a] = score[p][b];
		score[p][b] = tg;

		tg = result[p][a];
		result[p][b] = tg;
```

```
			result[p][a] = result[p][b];
			tg = error[p][a];
			error[p][a] = error[p][b];
			error[p][b] = tg;


}




/*----------------------------------------------------------------------*\
 |
|
 |   apply( p, i ) -   apply the i gene of the population p on the NN
|
 |
|
 \*----------------------------------------------------------------------*/
apply( p, i )
	int p, i;
{
	/* Each component realtes to a gene in the NN */

	w1.p[0] = pop[p][i][0];	w1.p[1] = pop[p][i][1];
w1.p[2] = pop[p][i][2];
	w1.p[3] = pop[p][i][3];	w1.p[4] = pop[p][i][4];
w1.p[5] = pop[p][i][5];
	w1.p[6] = pop[p][i][6];	w1.p[7] = pop[p][i][7];
w1.p[8] = pop[p][i][8];
	w1.p[9] = pop[p][i][9];	w1.p[10] = pop[p][i][10];
w1.p[11] = pop[p][i][11];
	w1.p[12] = pop[p][i][12];	w1.p[13] = pop[p][i][13];
w1.p[14] = pop[p][i][14];
	w1.p[15] = pop[p][i][15]; w1.p[16] = pop[p][i][16];
}



/*----------------------------------------------------------------------*\
 |
|
 |   pop_copy( p1, a, p2, b ) - copy the vector b in the population p2 into
|
 |                          the vector a in the population p1.
|
 |
|
 \*----------------------------------------------------------------------*/
pop_copy( p1, a, p2, b)
	int p1, a, p2, b;
{
	int i;

	for ( i = 0 ; i < SIZE ; i++ )
		pop[p1][a][i] = pop[p2][b][i];
}

	for ( p = 0 ; p < POPS ; p++ ) {
		/* Whole population gets values from -1
to 1 */
		for ( i = 0 ; i < MAXPOP ; i++ ) {
			for ( j = 0 ; j < SIZE ;
j++ ) {
	pop[p][i][j] = ((random()&1048575) / 1000000.0 - 0.5) *
2;
				while
( pop[p][i][j] < (-(min[j]/(max[j]-min[j])+0.5))) {

	pop[p][i][j] = ((random()&1048575) / 1000000.0 - 0.5) *
2;
				}
			}
		}
	}
}
```

```
/*----------------------------------------------------------------------*\
 |
|
 |   Calculate the scores of all the vectors in all the populations
|
 |
|
 \*----------------------------------------------------------------------*/
calc_score()
{

	int p, i, j;
	double check_performance();

	for ( p = 0 ; p < POPS ; p++ ){
		for ( i = 0 ; i < MAXPOP ; i++ ){
			{

	printf("Here is scorecalc for pop %d
gene %d: %f\n\n",p,i,score[p][i]);

	apply( p, i);

	score[p][i] = check_performance(p,i);

	printf("Here is scorecc2 for pop %d
gene %d: %f\n\n",p,i,score[p][i]);

	printf("Read %f %f\n", result[p][i],error[p][i]);

			}
		}
	}
}
/*----------------------------------------------------------------------*\
 |
|
 |                     Sort              the              populations
|
 |
|
 \*----------------------------------------------------------------------*/
sort_population()
{

	int p, i, j, k;
	float best;
	/* Use insert sort */
	for ( p = 0 ; p < POPS ; p++ ){
		for ( i = 0 ; i < MAXPOP ; i++ ){
			{
				best =
score[p][i];
				for ( j =
(i+1) ; j < MAXPOP ; j++ )

	if ( score[p][j] > best )

	{

		best = score[p][j];

		k = j;

	}
				if ( best
> score[p][i] )

	pop_swap( p, i, k );

			}
		}
	}

}
```

```
/*------------------------------------------------------------------------*\
|
|   Show (on the standard output) the best scores of all populations
|
|
|
\*------------------------------------------------------------------------*/
statistics( generation )
        int generation;
{
        int p,i;
        printf("generationi is here: %d\n",generation);
        if ( generation % MIXGEN == 0 )
                printf("-----------------------------\n");
        printf(" %4d) First are: ", generation);
        for ( p = 0 ; p < POPS ; p++ )    printf("%f ", score[p][0] );
        printf(" (from %d)\n",total);
}


/*------------------------------------------------------------------------*\
|
|   Generate the next generation in all populations
|
|
|
\*------------------------------------------------------------------------*/
make_next_generation( generation )
        int generation;
{
        int a, p, i, j, k, k1, k2, m, stack;
        float dev,min[50],max[50],dummy,mean,stdev;
        char minmax_file[50],c;
        FILE *ifp;
        ifp=fopen("./input/MINMAX","r");
        while((c=getc(ifp)) != '\n');
        for (j = 0; j < SIZE+1; j++) {

fscanf(ifp,"%f%f%f%f%f%f",&min[j],&max[j],&dummy,
&dummy,&dummy,&dummy);
        }
        fclose(ifp);
        for ( p = 0 ; p < POPS ; p++ ) {
                /* keep best - BESTPOP */
                /* add another group, randomly -
(SELPOP-BESTPOP) */
                for ( i = BESTPOP ; i < SELPOP ; i++ ) {
                        pop_swap( p, i,
(irand( MAXPOP - i ) + i) );
                }
                /* create new individuals */
                for ( i = MUT1 ; i < MAXPOP ; i++ ) {

                        stack = 0;
                        for ( j = 0 ; j < SIZE ;
j++ ) {

        pop[p][i][j] = ((random()&1048575) / 1000000.0 - 0.5) * 2;
                                while
( pop[p][i][j] < (-(min[j]/(max[j]-min[j])+0.5))) {

        pop[p][i][j] = ((random()&1048575) / 1000000.0 - 0.5) * 2;
                                }
                        }
                }
                /*  SELPOP to MUT1 will be severe
mutations */

                stack = 0;
                for ( i = NEWPOP ; i < MUT1 ; i++ ) {
                        pop_copy( p, i, p,
irand(NEWPOP) );
                        /* 5000 is the nominal
mutation value */
                        dev = 1 + ((irand(2000) -
1000 )/ 5000);
                        j=irand(SIZE);

                        while ( dev < (-
(min[j]/(max[j]-min[j])+0.5))) {
                                dev =
1 + ((irand(2000) - 1000 )/ 5000);
                        }
                        pop[p][i][j] = dev;
                }

                /* MUT2 to MAXPOP will be
crossovers */

                stack = 0;
                for ( i = SELPOP ; i < NEWPOP ; i++ )
{
                        /* Every several
generations (set by MIXGEN) there is a cross-over */
                        /* between different
populations. */
                        pop_copy( p, i,
(((generation%MIXGEN)==0) ? irand(POPS) : p), irand(NEWPOP) );
                        j = irand(NEWPOP);
                        k1 = irand( SIZE - 1);
                        k2 = irand( SIZE - 1 -
k1 ) + k1 + 1;
                        for ( m = k1 ; m <= k2 ;
m++ ) pop[p][i][m] = pop[p][j][m];

                        /* Mutate slightly to
preserve diversity */
                        dev = 1 + ((irand(2000)
- 1000 )/ 50000);
                        j=irand(SIZE);
                        while ( dev < (-
(min[j]/(max[j]-min[j])+0.5))) {
                                dev =
1 + ((irand(2000) - 1000 )/ 50000);
                        }
                        pop[p][i][j] = dev;
                }
        }
        calc_score();
        sort_population();
        statistics( generation );
        printf("Done %d generations\n\n",generation);

}
/*------------------------------------------------------------------------*\
|
|
|   Return the number of cases for which the NN returns the correct
value   |
|
|
\*------------------------------------------------------------------------*/
double check_performance (p,i) int p,i;
{
        vector x;
        int j;
        float score_value=0,score_a,d,f;
        net(p,i);


        printf("score_a is %f\n",score_a);

        score_value = error[p][i];
        printf ("Answer is %f and target
is %f\n",result[p][i],target.p[1]);
        if (score_value < 0) {
                score_value = 0; }
        else {
                score_value = 1.0/(score_value);
        }
        printf("score_value is: %f\n",score_value);
        return score_value;
}
```

```c
/*-----------------------------------------------------------------------*\
   |                                                                      |
   |  Return the number of cases for which the NN returns the correct value |
   |                                                                      |
   \*-----------------------------------------------------------------------*/
calc_limit_err()
{
		vector x;
		int j;
		float err=0;

		err = pow((target.p[2]*target.p[0]),2);
		target.p[1] = 1.0/sqrt(err) ;
		printf ("Norm Target Value is %f\n",target.p[0]);
		printf("Target Score is: %f\n",target.p[1]);
		return err;
}


/*-----------------------------------------------------------------------*\
   |                                                                      |
   |          Get data (reads input file)                                 |
   |                                                                      |
   \*-----------------------------------------------------------------------*/
int get_data()
{
		char* FileName = "./input/nn-input";
		FILE *fd;
		int i, posnum, negnum;
		float x,y,t;

		/* opens the file    */
		if ( (fd = fopen(FileName,"r")) == NULL )
		{
			printf ("no-input-file");
			exit(10);
		}

		/* Total number of input values */
		total = 0;

		fscanf(fd,"%f",&t);
		target.p[0]=t;
		printf("Target is %f",target.p[0]);

		fscanf(fd,"%f",&t);
		target.p[2]=t;
		printf("Wanted accuracy is %f",target.p[2]);

		fclose( fd );
		return (0) ;
}/*-----------------------------------------------------------------------*\
   |                                                                      |
   |   best_pop  -  Find the population with the best solution            |
   |                                                                      |
   \*-----------------------------------------------------------------------*/
int best_pop()
{
		int i, p;
		float best=0;
		for ( i = 0 ; i < POPS ; i++ )
			if ( score[i][0] > best )
			{
				best = score[i][0];
				p = i;
			}
		return(p);
}
```

```c
/*-----------------------------------------------------------------------*\
   |                                                                      |
   |                                                                 Main |
   |                                                                      |
   \*-----------------------------------------------------------------------*/



main() {
		int generation, i, j, l, p, best, done = 0,e=0, err_1a, k,z;
		float px, py, px1, py1, p0,p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,p12,er1,er2;
		FILE *fp,*fb,*fcount, *ferr,*fscore;
		FILE *fd;
		double tempResult, tempUncertainty;

		fcount=fopen("count","w");
		fprintf(fcount,"%d",e);
		fclose(fcount);



		randomize();
		get_data();    /* Read input from file */
		calc_limit_err();
		printf("Target is :%f",target.p[0]);
		make_initial_population();
		calc_score();
		sort_population();

		for (p=0;p<POPS;p++) {
			printf("Score of %d is %f",p,score[p][0]);
		}
		printf("We got here ");
		/* Educate the net */
		generation = 0;

		while ( (done != 1 ) && ( generation++ < SESSIONS ) )
		{

			make_next_generation( generation );
			p = best_pop();


			if ( score[p][0] > target.p[1] ) {
				printf ("Target error was %f and error is %f",target.p[1],score[p][0]);
				done = 1;
			} else {
				printf("Done all !!\n");

			}

			fb=fopen("./unnormalise/nn-output_b","a+");
			for (i=0;i<POPS;i++) {
				for ( j=0;j<BESTPOP;j++) {
					apply (i,j);
					l=0;

		tempResult = (result[i][j] + 0.5) * (MAX_UNNORMAL - MIN_UNNORMAL) + MIN_UNNORMAL;

		tempUncertainty = ((error[i][j] + result[i][j] + 0.5) * (MAX_UNNORMAL - MIN_UNNORMAL) + MIN_UNNORMAL) - tempResult;
					if ((fabs(TARGET_UNNORMAL - tempResult) < (TARGET_PERCENT_PERMIT * TARGET_UNNORMAL))

		&& (tempUncertainty < (2 * UNCERTAINTY_PERCENT_PERMIT * TARGET_UNNORMAL)))
					{
```

```
for
(k=0;k<SIZE;k++) {
#if 1
        fprintf(fb,"%f ",w1.p[l]);
        ++l;
#else
        if (k != 5) {   /* change this line according to which
variables are wanted to be fixed */
                fprintf(nmin,"%f ",w1.p[l]);
                ++l;
        } else {
                if (k == 0){
                        fprintf(nmin,"%f
",CAR);
                }
                if (k == 1) {
                        fprintf(nmin,"%f
",MAN);
                }
                if (k == 2) {
                        fprintf(nmin,"%f ",SIL);
                }
                if (k == 3) {
                        fprintf(nmin,"%f
",PHO);
                }
                if (k == 4) {
                        fprintf(nmin,"%f
",SUL);
                }
                if (k == 5) {
                        fprintf(nmin,"%f
",CHR);
                }
                if (k == 6) {
                        fprintf(nmin,"%f ",NIC);
                }
                if (k == 7) {
                        fprintf(nmin,"%f
",MOL);
                }
                if (k == 8) {
                        fprintf(nmin,"%f ",TIT);
                }
                if (k == 9) {
                        fprintf(nmin,"%f
",NIO);
                }
                if (k == 10) {
                        fprintf(nmin,"%f
",VAN);
                }
                if (k == 11) {
                        fprintf(nmin,"%f
",ALU);
                }
                if (k == 12) {
                        fprintf(nmin,"%f ",NIT);
                }
                if (k == 13) {
                        fprintf(nmin,"%f
",BOR);
                }
                if (k == 14) {
                        fprintf(nmin,"%f
",COP);
                }
                if (k == 15) {
                        fprintf(nmin,"%f
",FRT);
                }
                if (k == 16) {
                        fprintf(nmin,"%f ",CIT);
                }
        }
#endif
                                }
        fprintf(fb,"%f                    %f
\n",result[i][j],(error[i][j]+result[i][j]));
                                }
                        }
                }
                fclose(fb);

        }
        printf("done! \n");
        printf("\n\n****     For     unnormalisation,     goto
/unnormalise and type a.out (no. of lines in output file) => BESTPOP x
POPS = no of lines in output file   (no. of inputs)****\n");
        fscore=fopen("score_output","w");
        for (i=0;i<POPS;i++) {
                for ( j=0;j<MAXPOP;j++) {
                        fprintf(fscore,"Score  for
score_T_298 pop %d gene %d is %f \n",i,j,score[i][j]);
                        printf("scores           for
score_T_298 pop %d chromo %d id %f \n",i,j,score[i][j]);
                }
        }

        printf("\n Scores printed to 'scores'\n\n");
        fclose(fscore);
}
```

Appendix B

This is the documentation for the single objective genetic algorithm program, as described in chapter 2.2.1. The target property is the elongation to failure. This is associated documentation following the MAP format,

http://www.msm.cam.ac.uk/map/mapmain.html.

Program MAP_HOTROLLEDSTEEL_EL

1. Provenance of code.

2. Purpose of code.

3. Specification.

4. Description.

5. References.

6. Parameter descriptions.

7. Error indicators.

8. Accuracy estimate.

9. Any additional information.

10. Example.

11. Auxiliary subroutines required.

12. Keywords.

13. Sources.

1. Provenance of source code.

Min Sung Joo and H. K. D. H. Bhadeshia

Graduate Institute of Ferrous Technology (GIFT)

Pohang University of Science and Technology

Pohang, Kyoungbuk, Republic of Korea

athpimo@postech.ac.kr

2. Purpose of code.

An application of the genetic algorithm (GA) for reaching a solution given a fitting function. This can in theory be applied to any problem, where a database of inputs and outputs has trained a neural network.

3. Specification.

Language: C

Product form: Source code and executable files for UNIX/Linux machines.

Complete program.

4. Description.

The single_EL.tar.gz file, which can be downloaded from here, contains the

following files. A working version of a GA is also included to optimize the ductility of the hot-rolled steels as a function of chemical composition and processing variables.

genetic.c	- A C program used to run the genetic algorithm.
gareadme.doc	- A manual giving further details about running and using the GA program for altering different settings to optimize the process.
gareadme.txt	- A txt version of *gareadme.doc*
single_EL	- An executable program of *genetic.c*.
generate44	- It reads the normalized input data file, *input/norm_test.in*, and uses the weight files in subdicrectory *input*. The results are written to the temporary output file *_ot, _out, _res* and *_sen*.
score_output	- A file that contains the scroes of each chromosome after the program has concluded.
unnormalise/treatout.c	- A C program for un-normalising the output data files.
unnormalise/treatout	- An executable program of *unnormalise/treatout.c*.
unnormalise/nn-output_b	- A file which contains the normalized inputs and outputs of the GA after the program has concluded.
unnormalise/result	- A file which contains the unnormalized values for inputs, outputs and error of each chromosome after unnormalising data in the normalized *nn-output_b* file.

input/labels.txt - A list of input variables.

input /nn-input - Normalised inputs file (target and accuracy values for the GA).

input /norm_test.in - A text file which contains the normalized input variables initialized by the GA to be fed into the neural network.

The following are concerned with the neural network files that work with the GA, but can be changed according to user requirements:

*input /_w*f* - The weights files for the different models.

input /.lu* - Files containing information for calculating the size of the error bars for the different models.

input /spec1.t1 - An altered version of *spec.t1* which is a dynamic file, created by neural network. It is read by the program *generate44*.

input /outran.x - A normalized output file that was created when developing the model. It is accessed by *generate44* via *input/spec1.t1*.

input /MINMAX - Minimum and maximum values for the input variables used in the original database.

The file gareadme.txt contains a manual, which provides a detailed set of instructions for downloading and running the GA program. A summary of the

information in this manual is given here.

5. References.

Ryu, J. H., Model for mechanical properties of Hot-Rolled Steels, Master Thesis, POSTECH, 2008.

Shah, I., Tensile Properties of Austenite Stainless Steel, M. Phil. Thesis, University of Cambridge, 2002.

Delorme, A., Genetic algorithm for optimization of mechanical properties, Technical report, University of Cambridge, 2003.

6. Parameter descriptions.

Input parameters

To specify the target value and accuracy desired, *input/nn-input* must be amended. The normalized target value relates to a real value, according to *the input/MINMAX* file used.

Row1	Normalised target value
Row2	Accuracy (in decimal e.g. 0.1 for 10%)

To specify the permitted range of the output, *genetic.c* must be changed.

#define TARGET_UNNORMAL	Unnormalised target value

#define MAX_UNNORMAL	Maximum of target value in *intput/MINMAX*
#define MIN_UNNORMAL	Minimum of target value in *input/ MINMAX*
#define TARGET_PERCENT_PERMIT	Permitted range for the target (in decimal e.g. 0.1 for 10%)
#define UNCERTAINTY_PERCENT_PERMIT	Permitted range for the uncertainty (in decimal e.g. 0.1 for 10%)

To initiate the GA search, the inputs are randomly generated and placed in *input/norm_test.in*. It should be noted that each chromosome generally relates to a different steel composition, but this could change over the course of optimization. For the current ultimate tensile strength model, the composition and processing variables, totally 17 data items, are specified:

Gene number	Variable
1	Normalised C, wt%
2	Normalised Mn, wt%
3	Normalised Si, wt%
4	Normalised P, wt%
5	Normalised S, wt%
6	Normalised Cr, wt%

7	Normalised Ni, wt%
8	Normalised Mo, wt%
9	Normalised Ti, wt%
10	Normalised Nb, wt%
11	Normalised V, wt%
12	Normalised Al, wt%
13	Normalised N, ppm
14	Normalised B, ppm
15	Normalised Cu, wt%
16	Normalised Finishing rolling temperature, ˚C
17	Normalised Coiling temperature, ˚C

Each input is normalized using the equation:

Normalized value = (value – min)/(max-min) – 0.5

Where the values for min and max are defined as follows:

Gene number	Variable	Min	Max
1	C, wt%	0.0204	0.8684
2	Mn, wt%	0.1670	1.4100
3	Si, wt%	0.0000	0.2170
4	P, wt%	0.0040	0.0220
5	S, wt%	0.0020	0.0150

6	Cr, wt%	0.0000	0.1600
7	Ni, wt%	0.0000	0.0600
8	Mo, wt%	0.0000	0.0200
9	Ti, wt%	0.0000	0.0040
10	Nb, wt%	0.0000	0.0040
11	V, wt%	0.0000	0.0030
12	Al, wt%	0.0000	0.0640
13	N, ppm	0.00	87.00
14	B, ppm	0.00	2.00
15	Cu, wt%	0.0000	0.0300
16	Finishing rolling temperature, ˚C	808.00	925.00
17	Coiling temperature, ˚C	478.00	714.00

Output parameters

Two output files are produced by the GA program: *unnormalise/nn-output_b* and *score-output.*

score-output simply prints out the scores for each chromosome.

unnormalise/nn-output_b contains the inputs, prediction and (prediction + error). This is done for the best chromosomes within all populations:

Column 1-17	The normalized predicted inputs
Column 18	The normalized predicted output

Column 19	A value for the error, which includes the experimental noise of the database, an estimation of the uncertainty in the prediction and the test error. However, this is added to the prediction value so that the *input/MINMAX* file can easily handle the value.

The normalized values in all columns must be un-normalised using the equation:

actual value = (normalized value + 0.5) * (max-min) + min

The C program, *unnormalise/treatout.c*, is used to translate the output files to produce the actual values of inputs and outputs which are written to *unnormalise/result*.

7. Error indicators.

None.

8. Accuracy estimate.

See:

Input parameters, output parameters.

9. Any additional information.

See:

gareadme.txt

10. Example.

1. Program text

Complete program

2. Program data

The input file (*input/nn-input*) is:

0.083333

0.1

The input variables (*genetic.c*) are:

```
#define TARGET_UNNORMAL                    35.0
#define MAX_UNNORMAL                       50.0
#define MIN_UNNORMAL                       14.0
#define TARGET_PERCENT_PERMIT              0.1
#define UNCERTAINTY_PERCENT_PERMIT         0.15
```

3. Program results

The output file unnormalise/nn-output_b, which contains the normalised values for the ultimate tensile strength model including compositions and processing variables (following result is only single selected case):

C	Mn	Si	P	S
-0.489758	-0.225548	-0.250718	0.000000	0.000000

Cr	Ni	Mo	Ti	Nb
-0.296620	0.000000	0.000000	0.000000	0.000000

V	Al	N	B	Cu
1.000000	0.000000	0.000000	-0.050906	1.000000

FRT	CT	Pred	Pred + Err
0.000000	-0.657954	0.087502	0.198706

These are then run with *unnormalise/treatout.c*, where the normalised values are converted into the actual values to *unnormalise/result*:

C	Mn	Si	P	S
0.029085	0.508144	0.054094	0.013000	0.008500

Cr	Ni	Mo	Ti	Nb
0.032541	0.030000	0.010000	0.002000	0.002000

V	Al	N	B	Cu
0.001500	0.032000	43.50000	0.898188	0.045000

FRT	**CT**	**Pred**	**Pred + Err**
866.5000	440.7229	35.1500	4.00335

11. Auxiliary subroutines required.

None.

12. Keywords.

hot-rolled steel, genetic algorithm, elongation, neural network.

13. Sources.

genetic.c

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#include <sys/time.h>

/* NN related */
#define NUM          17          /* Total number of genes
*/
#define LIMIT        150         /* Maximum number of
inputs the system can handle */
#define SESSIONS     1000        /* Number of generations
that we'll put the system through */
/* GA related */
#define POPS         2           /* Number of populations
*/
#define SIZE         17          /* Size of vector in the
genetic algorithms */
#define MAXPOP                   20          /* Size of
population */
#define BESTPOP                  8           /*
Number of individuals taken from the best */
#define SELPOP                   12          /*
SELPOP-BESTPOP = Number of people selected randomly to exchange
genes within their own population */
#define NEWPOP                   16          /*
NEWPOP-SELPOP = Number of new people created randomly on each
gen. */
#define MUT1         18          /*   MUT1-NEWPOP   =
Number of genes that are severely mutated */
```

```
#define MIXGEN                          1              /*
Number of generations between population mixing */
/* result selection */
#define TARGET_UNNORMAL
            35.0
#define MAX_UNNORMAL                                   50.0
#define MIN_UNNORMAL                                   14.0
#define TARGET_PERCENT_PERMIT                          0.1
#define UNCERTAINTY_PERCENT_PERMIT  0.15


#define CAR          -0.03703703    /* Carbon */
#define MAN          0.28512388     /* Manganese */
#define SIL          0.03913045     /* Silicon */
#define PHO          0.03125002     /* Phosphorus */
#define SUL          -0.3222222     /* Sulphur */
#define CHR          -0.1                           /*
Chromium */
#define NIC          0.1                            /*
Nickel */
#define MOL          -0.33848798      /* Molybdenum */
#define TIT          -0.42857143    /* Titanium */
#define NIO          -0.48947367    /* Niobium */
#define VAN          -0.5                           /*
Vanadium */
#define ALU          -0.5                           /*
Aluminium */
#define NIT          -0.11728396    /* Nitrogen */
#define BOR          -0.4333333     /* Boron */
#define COP          -0.01428570    /* Copper */
```

```
#define FRT                0.1                /*
Finishin Mill Temperature */
#define CIT                0.1                /*
COiling Temperature */
/* BESTPOP x POPS = no of lines in output file*/

typedef struct
{
        float p[NUM];
} vector;


/* NN related */
vector test[LIMIT], w1,target;
int hits[LIMIT], total;

/* GA related */
float pop[POPS][MAXPOP][SIZE];
double score[POPS][MAXPOP];
double result[POPS][MAXPOP];
double error[POPS][MAXPOP];
double error2[POPS][MAXPOP];

/*----------------------------------------------------------------------*\
   |
|
   |                                                          Randomize
|
   |
|
   \*----------------------------------------------------------------------*/
randomize()
{
        struct timeval tp;
        struct timezone tzp;

        /* Use time of day to feed the random number generator
seed */
        gettimeofday( &tp, &tzp);
        srandom( tp.tv_sec );
}
/*----------------------------------------------------------------------*\
   |
|
   |  irand( range )   - return a random integer in the range 0..(range-1)
|
   |
|
   \*----------------------------------------------------------------------*/
int irand( range ) int range;
{
        return( random() % range );
}


/*----------------------------------------------------------------------*\
   |
|
   |       scalar_mult     -     multiply     two     vectors
|
   |
|
   \*----------------------------------------------------------------------*/
float scalar_mult( x, y ) vector x, y;
{
        int i;
        float s = 0.0;
        for ( i = 0 ; i < NUM ; i++ )    s += ( x.p[i] * y.p[i] );
        return s;
}
/*----------------------------------------------------------------------*\
   |
|
   |  This function computes the NN's output for a certain input vector.
|
   |  The NN is constructed from 2 layers, first layer has 6 neurons,
|
   |         second        layer        has        1        neuron.
|
   |
|
   \*----------------------------------------------------------------------*/
```

```
net(p,i)
        int p,i;


{
        FILE *nmin,*nmin2,*nmin3,*fcount,*ferr;
        int j,k,l,m,sum,e,no_of_columns;
        char c,s[100],minmax_file[50];
        float res2,res3,res4;
        float err2,err3,err4;
        float err_a,err_b,err_c;
        float err_1,err_1a;
        float
min[50],max[50],dummy,mean,stdev,dev,generation,k1,k2;
        double check_performance();
        FILE *ifp;



        printf("Calling net with p=%d and i=%d",p,i);
        if   ((nmin   =   fopen("./input/norm_test.in","w"))   ==
NULL ) {
                printf                              ("Can't
write ./input/norm_test.in");
                exit(1);
        } else {

                l=0;
                for (j=0;j<SIZE;j++) {
#if 1
                        fprintf(nmin,"%f
",w1.p[l]);
                        ++l;
#else
                        if (j != 5) {
                                /*
change this line according to which

        variables are wanted to be fixed */

        fprintf(nmin,"%f ",w1.p[l]);
                                ++l;
                        } else {
                                if (j ==
0){

        fprintf(nmin,"%f ",CAR);
                                }
                                if (j ==
1) {

        fprintf(nmin,"%f ",MAN);
                                }
                                if (j ==
2) {

        fprintf(nmin,"%f ",SIL);
                                }
                                if (j ==
3) {

        fprintf(nmin,"%f ",PHO);
                                }
                                if (j ==
4) {

        fprintf(nmin,"%f ",SUL);
                                }
                                if (j ==
5) {

        fprintf(nmin,"%f ",CHR);
                                }

                                if (j ==
6) {

        fprintf(nmin,"%f ",NIC);
                                }
                                if (j ==
7) {
```

```
fprintf(nmin,"%f ",MOL);
}
if (j == 8)
{
fprintf(nmin,"%f ",TIT);
}
if (j == 9)
{
fprintf(nmin,"%f ",NIO);
}
if (j ==
10) {
fprintf(nmin,"%f ",VAN);
}
if (j ==
11) {
fprintf(nmin,"%f ",ALU);
}
if (j ==
12) {
fprintf(nmin,"%f ",NIT);
}
if (j ==
13) {
fprintf(nmin,"%f ",BOR);
}
if (j ==
14) {
fprintf(nmin,"%f ",COP);
}
if (j ==
15) {
fprintf(nmin,"%f ",FRT);
}
if (j ==
16) {
fprintf(nmin,"%f ",CIT);
}
}
#endif
}
fprintf(nmin,"\n");
fclose(nmin);
}

sprintf(s, "./generate44 ./input/spec1.t1
8 ./input/_wh1f ./input/_wh1f.lu> /dev/null");
system(s);
if ( (nmin2 = fopen("_out","r")) == NULL) {
printf ("Can't read _out stage 1");
exit(1);
} else {
fscanf(nmin2,"%f %f",&res2,&err2);
printf("res2 err2 %f %f\n",res2,err2);
printf("Indices %d %d\n",p,i);

fclose(nmin2);

}
sprintf(s, "./generate44 ./input/spec1.t1
8 ./input/_wh4f ./input/_wh4f.lu> /dev/null");
system(s);
if ( (nmin2 = fopen("_out","r")) == NULL) {
printf ("Can't read _out stage 2");
```

```
exit(1);
} else {
fscanf(nmin2,"%f %f",&res3,&err3);
printf("res3 err3 %f %f\n",res3,err3);
fclose(nmin2);
}
sprintf(s, "./generate44 ./input/spec1.t1
12 ./input/_wl4f ./input/_wl4f.lu> /dev/null");
system(s);
if ( (nmin2 = fopen("_out","r")) == NULL) {
printf ("Can't read _out stage 3");
exit(1);
} else {
fscanf(nmin2,"%f %f",&res4,&err4);
printf("res4 err4 %f %f\n",res4,err4);
fclose(nmin2);
}

result[p][i] = ((res2+res3+res4)/3);
printf("this is p %d %d\n", p,i);

nmin3=fopen("./input/nn-input","r");
fscanf(nmin3,"%f",&target.p[0]);

err_a = pow ((res2-target.p[0]),2);
err_b = pow ((res3-target.p[0]),2);
err_c = pow ((res4-target.p[0]),2);

err_1 = (err2+err3+err4);
err_1a = pow((err_1),2);

printf("err_a is %f\n",err_a);
printf("err_b is %f\n",err_b);
printf("err_c is %f\n",err_c);

printf("err_1 is %f\n",err_1);

error2[p][i]=sqrt(err_1a);
error[p][i] = sqrt (((err_1a)/3)+((err_a + err_b +
err_c)/3));
printf("error pi is %f\n\n",error[p][i]);
fcount=fopen("count","r");
fscanf(fcount,"%d",&e);
fclose(fcount);


e++;
fcount=fopen("count","w");
fprintf(fcount,"%d",e);


fclose(fcount);
printf("\n-----------> sum %d\n",e);

printf("In net() error is %f res
is %f\n\n",error[p][i],result[p][i]);
fclose(nmin3);

}
/*----------------------------------------------------------------------*\
|
|
| pop_swap( p, a, b ) - swap two vectors and scores in the
population p|
|
|
\*----------------------------------------------------------------------*/
pop_swap( p, a, b )
int p, a, b;

{
int t, i;
float tg;
/* Swap vector */
for ( i = 0 ; i < SIZE ; i++ )

{
t = pop[p][a][i];
pop[p][a][i] = pop[p][b][i];
pop[p][b][i] = t;
}
```

```
			/* Swap score */
			tg = score[p][a];
			score[p][a] = score[p][b];
			score[p][b] = tg;

			tg = result[p][a];
			result[p][a] = result[p][b];
			result[p][b] = tg;

			tg = error[p][a];
			error[p][a] = error[p][b];
			error[p][b] = tg;


}



/*------------------------------------------------------------------------*\
   |
|
   |   apply(  p,  i  )  -   apply  the  i  gene  of  the  population  p  on  the  NN
|
   |
|
   \*------------------------------------------------------------------------*/
apply( p, i )
			int p, i;
{
			/* Each component realtes to a gene in the NN */

			w1.p[0] = pop[p][i][0];    w1.p[1] = pop[p][i][1];    w1.p[2]
= pop[p][i][2];
			w1.p[3] = pop[p][i][3];    w1.p[4] = pop[p][i][4];    w1.p[5]
= pop[p][i][5];
			w1.p[6] = pop[p][i][6];    w1.p[7] = pop[p][i][7];    w1.p[8]
= pop[p][i][8];
			w1.p[9]   =   pop[p][i][9];      w1.p[10]   =   pop[p][i][10];
w1.p[11] = pop[p][i][11];
			w1.p[12]   =   pop[p][i][12];   w1.p[13]   =   pop[p][i][13];
w1.p[14] = pop[p][i][14];
			w1.p[15] = pop[p][i][15]; w1.p[16] = pop[p][i][16];
}

/*------------------------------------------------------------------------*\
   |
|
   |   pop_copy( p1, a, p2, b ) - copy the vector b in the population p2 into |
   |                                        the  vector  a  in  the  population  p1.
|
   |
|
   \*------------------------------------------------------------------------*/
pop_copy( p1, a, p2, b)
			int p1, a, p2, b;
{
			int i;

			for ( i = 0 ; i < SIZE ; i++ )
						pop[p1][a][i] = pop[p2][b][i];
}
/*------------------------------------------------------------------------*\
   |
|
   |                              Initialize                  the                  populations
|
   |
|
   \*------------------------------------------------------------------------*/
make_initial_population()

{
			int p, i, j, k, stack = 0
			float min[50],max[50],dummy,mean,stdev;
			char minmax_file[50],c;
			FILE *ifp;
			ifp=fopen("./input/MINMAX","r");
			while((c=getc(ifp)) != '\n');;
			for (j = 0; j < SIZE+1; j++) {

			fscanf(ifp,"%f%f%f%f%f%f",&min[j],&max[j],&dumm
y,&dummy,&dummy,&dummy);
			}
			fclose(ifp);

			for ( p = 0 ; p < POPS ; p++ ) {
						/* Whole population gets values from -1
to 1 */
						for ( i = 0 ; i < MAXPOP ; i++ ) {
									for  (  j  =  0  ;  j  <  SIZE  ;
j++ ) {

			pop[p][i][j] = ((random()&1048575) / 1000000.0 - 0.5) *
2;
												while
( pop[p][i][j] < (-(min[j]/(max[j]-min[j])+0.5))) {

			pop[p][i][j] = ((random()&1048575) / 1000000.0 - 0.5) *
2;
												}
									}
						}
			}
}



/*------------------------------------------------------------------------*\
   |
|
   |      Calculate  the  scores  of  all  the  vectors  in  all  the  populations
|
   |
|
   \*------------------------------------------------------------------------*/
calc_score()
{

			int p, i, j;
			double check_performance();

			for ( p = 0 ; p < POPS ; p++ ){
						for ( i = 0 ; i < MAXPOP ; i++ ){
									{

			printf("Here        is        scorecalc        for        pop        %d
gene %d: %f\n\n",p,i,score[p][i]);

			apply( p, i);

			score[p][i] = check_performance(p,i);

			printf("Here        is        scorecc2        for        pop        %d
gene %d: %f\n\n",p,i,score[p][i]);

			printf("Read %f %f\n", result[p][i],error[p][i]);

									}
						}
			}
}
/*------------------------------------------------------------------------*\
   |
|
   |                              Sort                  the                  populations
|
   |
|
   \*------------------------------------------------------------------------*/
sort_population()
{

			int p, i, j, k;
			float best;
			/* Use insert sort */
			for ( p = 0 ; p < POPS ; p++ ){
						for ( i = 0 ; i < MAXPOP ; i++ ){
									{
```

```
                                        best =
score[p][i];
                                        for ( j =
(i+1) ; j < MAXPOP ; j++ )

            if ( score[p][j] > best )

            {

                        best = score[p][j];

                        k = j;

            }
                                        if ( best >
score[p][i] )

            pop_swap( p, i, k );

                                    }
                        }
            }

}


/*---------------------------------------------------------------------*\
   |
|
   |    Show (on the standard output) the best scores of all populations
|
   |
|
   \*---------------------------------------------------------------------*/
statistics( generation )
            int generation;
{
            int p,i;
            printf("generationi is here: %d\n",generation);
            if ( generation % MIXGEN == 0 )
                        printf("-----------------------------\n");
            printf(" %4d) First are: ", generation);
            for ( p = 0 ; p < POPS ; p++ )    printf("%f ", score[p][0] );
            printf(" (from %d)\n",total);
}


/*---------------------------------------------------------------------*\
   |
|
   |       Generate the next generation in all populations
|
   |
|
   \*---------------------------------------------------------------------*/
make_next_generation( generation )
            int generation;
{
            int a, p, i, j, k, k1, k2, m, stack;
            float dev,min[50],max[50],dummy,mean,stdev;
            char minmax_file[50],c;
            FILE *ifp;
            ifp=fopen("./input/MINMAX","r");
            while((c=getc(ifp)) != '\n');
            for (j = 0; j < SIZE+1; j++) {

            fscanf(ifp,"%f%f%f%f%f%f",&min[j],&max[j],&dummy,
&dummy,&dummy,&dummy);
            }
            fclose(ifp);
            for ( p = 0 ; p < POPS ; p++ ) {
                        /* keep best - BESTPOP */
                        /* add another group, randomly -
(SELPOP-BESTPOP) */
                        for ( i = BESTPOP ; i < SELPOP ; i++ ) {
                                    pop_swap(    p,    i,
(irand( MAXPOP - i ) + i) );
                        }
                        /* create new individuals */
                        for ( i = MUT1 ; i < MAXPOP ; i++ ) {

                                    stack = 0;
                                    for ( j = 0 ; j < SIZE ;
j++ ) {

            pop[p][i][j] = ((random()&1048575) / 1000000.0 - 0.5) *
2;
                                                while
( pop[p][i][j] < (-(min[j]/(max[j]-min[j])+0.5))) {

            pop[p][i][j] = ((random()&1048575) / 1000000.0 - 0.5) *
2;
                                                }
                                    }
                        }
                        /*  SELPOP to MUT1 will be severe
mutations */

                        stack = 0;
                        for ( i = NEWPOP ; i < MUT1 ; i++ ) {
                                    pop_copy(  p,  i,  p,
irand(NEWPOP) );
                                    /* 5000 is the nominal
mutation value */
                                    dev = 1 + ((irand(2000)
- 1000 )/ 5000);
                                    j=irand(SIZE);

                                    while  (  dev  <  (-
(min[j]/(max[j]-min[j])+0.5))) {
                                                dev  =
1 + ((irand(2000) - 1000 )/ 5000);
                                    }
                                    pop[p][i][j] = dev;
                        }

                        /*  MUT2  to  MAXPOP  will  be
crossovers */

                        stack = 0;
                        for ( i = SELPOP ; i < NEWPOP ; i++ )
{

                                    /*   Every   several
generations (set by MIXGEN) there is a cross-over */
                                    /*   between different
populations. */
                                    pop_copy(   p,   i,
(((generation%MIXGEN)==0) ? irand(POPS) : p), irand(NEWPOP) );
                                    j = irand(NEWPOP);
                                    k1 = irand( SIZE - 1);
                                    k2 = irand( SIZE - 1 -
k1 ) + k1 + 1;
                                    for ( m = k1 ; m <= k2 ;
m++ ) pop[p][i][m] = pop[p][j][m];

                                    /* Mutate slightly to
preserve diversity */
                                    dev = 1 + ((irand(2000)
- 1000 )/ 50000);
                                    j=irand(SIZE);
                                    while  (  dev  <  (-
(min[j]/(max[j]-min[j])+0.5))) {
                                                dev  =
1 + ((irand(2000) - 1000 )/ 50000);
                                    }
                                    pop[p][i][j] = dev;
                        }
            }
            calc_score();
            sort_population();
            statistics( generation );
            printf("Done %d generations\n\n",generation);

}
```

```
/*----------------------------------------------------------------------*\
   |                                                                    |
   |  Return the number of cases for which the NN returns the correct value   |
   |                                                                    |
   \*----------------------------------------------------------------------*/
double check_performance (p,i) int p,i;
{
		vector x;
		int j;
		float score_value=0,score_a,d,f;
		net(p,i);


		printf("score_a is %f\n",score_a);

		score_value = error[p][i];

		printf ("Answer is %f and target is %f\n",result[p][i],target.p[1]);
		if (score_value < 0) {
				score_value = 0; }
		else {
				score_value = 1.0/(score_value);
		}
		printf("score_value is: %f\n",score_value);
		return score_value;
}


/*----------------------------------------------------------------------*\
   |                                                                    |
   |  Return the number of cases for which the NN returns the correct value   |
   |                                                                    |
   \*----------------------------------------------------------------------*/
calc_limit_err()
{
		vector x;
		int j;
		float err=0;

		err = pow((target.p[2]*target.p[0]),2);
		target.p[1] = 1.0/sqrt(err) ;
		printf ("Norm Target Value is %f\n",target.p[0]);
		printf("Target Score is: %f\n",target.p[1]);
		return err;
}
/*----------------------------------------------------------------------*\
   |                                                                    |
   |            Get data (reads input file)                             |
   |                                                                    |
   \*----------------------------------------------------------------------*/
int get_data()
{
		char* FileName = "./input/nn-input";
		FILE *fd;
		int i, posnum, negnum;
		float x,y,t;

		/* opens the file    */
		if ( (fd = fopen(FileName,"r")) == NULL )
		{
				printf ("no-input-file");
				exit(10);
		}

		/* Total number of input values */
		total = 0;

		fscanf(fd,"%f",&t);
		target.p[0]=t;
		printf("Target is %f",target.p[0]);
		fscanf(fd,"%f",&t);
		target.p[2]=t;
		printf("Wanted accuracy is %f",target.p[2]);

		fclose( fd );
		return (0) ;
}
/*----------------------------------------------------------------------*\
   |                                                                    |
   |    best_pop   -   Find the population with the best solution       |
   |                                                                    |
   \*----------------------------------------------------------------------*/
int best_pop()
{
		int i, p;
		float best=0;
		for ( i = 0 ; i < POPS ; i++ )
				if ( score[i][0] > best )
				{
						best = score[i][0];
						p = i;
				}
		return(p);
}



/*----------------------------------------------------------------------*\
   |                                                                    |
   |                                                             Main   |
   |                                                                    |
   \*----------------------------------------------------------------------*/



main() {
		int generation, i, j, l, p, best, done = 0,e=0, err_1a, k,z;
		float px, py, px1, py1, p0,p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,p12,er1,er2;
		FILE *fp,*fb,*fcount, *ferr,*fscore;
		FILE *fd;
		double tempResult, tempUncertainty;

		fcount=fopen("count","w");
		fprintf(fcount,"%d",e);
		fclose(fcount);
		randomize();
		get_data();    /* Read input from file */
		calc_limit_err();
		printf("Target is :%f",target.p[0]);
		make_initial_population();
		calc_score();
		sort_population();

		for (p=0;p<POPS;p++) {
				printf("Score of %d is %f",p,score[p][0]);
		}
		printf("We got here ");
		/* Educate the net */
		generation = 0;

		while ( (done != 1 ) && ( generation++ < SESSIONS ) )

		{
				make_next_generation( generation );
				p = best_pop();
				if ( score[p][0] > target.p[1] ) {
						printf ("Target error was %f and error is %f",target.p[1],score[p][0]);
						done = 1;
				} else {
						printf("Done all !!\n");

				}
```

```
fb=fopen("./unnormalise/nn-
output_b","a+");
for (i=0;i<POPS;i++) {
for
( j=0;j<BESTPOP;j++) {
apply
(i,j);
l=0;

tempResult = (result[i][j] + 0.5) * (MAX_UNNORMAL -
MIN_UNNORMAL) + MIN_UNNORMAL;

tempUncertainty = ((error[i][j] + result[i][j] + 0.5) *
(MAX_UNNORMAL - MIN_UNNORMAL) + MIN_UNNORMAL) -
tempResult;

if
((fabs(TARGET_UNNORMAL - tempResult) <
(TARGET_PERCENT_PERMIT * TARGET_UNNORMAL))

&& (tempUncertainty < (2 *
UNCERTAINTY_PERCENT_PERMIT * TARGET_UNNORMAL)))
{
for
(k=0;k<SIZE;k++) {
#if 1

fprintf(fb,"%f ",w1.p[l]);

++l;
#else

if (k != 5) { /* change this line according to which
variables are wanted to be fixed */

fprintf(nmin,"%f ",w1.p[l]);

++l;

} else {

if (k == 0){

fprintf(nmin,"%f
",CAR);

}

if (k == 1) {

fprintf(nmin,"%f
",MAN);

}

if (k == 2) {

fprintf(nmin,"%f ",SIL);

}

if (k == 3) {

fprintf(nmin,"%f
",PHO);

}

if (k == 4) {

fprintf(nmin,"%f
",SUL);

}

if (k == 5) {

fprintf(nmin,"%f
",CHR);

}

if (k == 6) {

fprintf(nmin,"%f ",NIC);

}

if (k == 7) {

fprintf(nmin,"%f
",MOL);

}

if (k == 8) {

fprintf(nmin,"%f ",TIT);

}

if (k == 9) {

fprintf(nmin,"%f
",NIO);

}

if (k == 10) {

fprintf(nmin,"%f
",VAN);

}

if (k == 11) {

fprintf(nmin,"%f
",ALU);

}

if (k == 12) {

fprintf(nmin,"%f ",NIT);

}

if (k == 13) {

fprintf(nmin,"%f
",BOR);

}

if (k == 14) {

fprintf(nmin,"%f
",COP);

}

if (k == 15) {

fprintf(nmin,"%f
",FRT);

}

if (k == 16) {

fprintf(nmin,"%f ",CIT);

}

}
#endif
}

fprintf(fb,"%f %f
\n",result[i][j],(error[i][j]+result[i][j]));
}
```

```
                                                }
                                        }

                                        fclose(fb);
                        }
                        printf("done! \n");
                        printf("\n\n****        For     unnormalisation,     goto
/unnormalise and type a.out (no. of lines in output file) => BESTPOP x
POPS = no of lines in output file    (no. of inputs)****\n");
                        fscore=fopen("score_output","w");
                        for (i=0;i<POPS;i++) {
                                        for ( j=0;j<MAXPOP;j++) {
                                                fprintf(fscore,"Score  for
score_T_298 pop %d gene %d is %f \n",i,j,score[i][j]);
                                                printf("scores          for
score_T_298 pop %d chromo %d id %f \n",i,j,score[i][j]);
                                        }
                        }

                        printf("\n Scores printed to 'scores'\n\n");
                        fclose(fscore);
}
```

Appendix C

This is the documentation for the multiple objective genetic algorithm program, as described in chapter 2.2.2. The target properties are the ultimate tensile strength and the elongation to failure of hot-rolled steels. This is associated documentation following the MAP format,

http://www.msm.cam.ac.uk/map/mapmain.html.

Program MAP_HOTROLLEDSTEEL_DOMAINS

1. Provenance of code.

2. Purpose of code.

3. Specification.

4. Description.

5. References.

6. Parameter descriptions.

7. Error indicators.

8. Accuracy estimate.

9. Any additional information.

10. Example.

11. Auxiliary subroutines required.

12. Keywords.

13. Sources.

1. Provenance of source code.

Min Sung Joo and H. K. D. H. Bhadeshia

Graduate Institute of Ferrous Technology (GIFT)

Pohang University of Science and Technology

Pohang, Kyoungbuk, Republic of Korea

athpimo@postech.ac.kr

2. Purpose of code.

This program is an implementation of a genetic algorithm which can reach optimum sets of parameters for two target values of the ultimate tensile strength and the elongation to failure of hot-rolled steels. This can in theory be applied to any problem, where a neural network exists.

3. Specification.

Language: C

Product form: Source code and executable files for UNIX/Linux machines.

Complete program.

4. Description.

The multiple_UTS_EL.tar.gz file, which can be downloaded from here, contains the following files. A working version of a GA is also included to optimize the ductility of the hot-rolled steels as a function of chemical composition and processing variables.

genetic.c	- A C program used to run the genetic algorithm.
genetic.c	- A header file for *genetic.c*.
multiple_UTS_EL	- An executable program of *genetic.c*.
generate44	- It reads the normalized input data file, *input/norm_test.in*, and uses the weight files in subdicrectory *input*. The results are written to the temporary output file *_ot, _out, _res* and *_sen*.
unnormalise/treatout.c	- A C program for un-normalising the output data files.
unnormalise/treatout	- An executable program of *unnormalise/treatout.c*.
unnormalise/nn-output_b	- A file which contains the normalized inputs and outputs of the GA after the program has concluded.
unnormalise/result	- A file which contains the unnormalized values for inputs, outputs and error of each chromosome after unnormalising data in the normalized *nn-output_b* file.
input/labels.txt	- A list of input variables.
input /TARGETS	- Normalised inputs file (target and accuracy values for the GA).

input /norm_test.in - A text file which contains the normalized input variables initialized by the GA to be fed into the neural network.

The following are concerned with the neural network files that work with the GA, but can be changed according to user requirements:

*input /11/_w*f* - The weights files for the UTS model.

input /11/.lu* - Files containing information for calculating the size of the error bars for the UTS model.

input /11/spec1.t1 - An altered version of *spec.t1* which is a dynamic file for UTS model, created by neural network. It is read by the program *generate44*.

input /11/outran.x - A normalized output file that was created when developing the UTS model. It is accessed by *generate44* via *input/11/spec1.t1*.

*input /17/_w*f* - The weights files for the EL model.

input /17/.lu* - Files containing information for calculating the size of the error bars for the EL model.

input /17/spec1.t1 - An altered version of *spec.t1* which is a dynamic file for EL model, created by neural network. It is read by the program *generate44*.

input /17/outran.x - A normalized output file that was created when developing the EL model. It is accessed by *generate44*

via *input/11/spec1.t1*.

input /MINMAX - Minimum and maximum values for the input variables used in the original database.

How to run the GA to find optimum sets of parameters:

First, you have to define in the "values" of the inputs you wish to vary or those you wish to fix and the corresponding fixed value. Then, you must normalise the desired target value of the ultimate tensile strength and the elongation and enter them in the "*input/nn-input*" file, as well as the wanted accuracy. For example,

```
2
0.4 0.1
0.2 0.1
```

First line means that the number of target is 2.

Second line means that 0.4 of normalised target for UTS with 0.1 (10%) accuracy.

Third line means that 0.2 of normalised target for EL with 0.1 (10%) accuracy.

Secondly, you need to set the permitted values for targets in genetic.h

```
#define MAX_UNNORMAL_UTS                1039.0
#define MIN_UNNORMAL_UTS                317.0
#define MAX_UNNORMAL_EL                 50.0
#define MIN_UNNORMAL_EL                 14.0
#define TARGET_UNNORMAL_UTS             400.0
#define TARGET_UNNORMAL_EL              35.0
#define TARGET_PERCENT_PERMIT           0.1
#define UNCERTAINTY_PERCENT_PERMIT      0.15
```

MAX_UNNORMAL_UTS is maximum UTS, MIN_UNNORMAL_UTS is

minimum UTS, MAX_UNNORMAL_EL is maximum elongation, MIN_UNNORMAL_EL is minimum elongation, TARGET_UNNORMAL_UTS is target UTS and TARGET_UNNORMAL_EL is target EL in MINMAX.
TARGET_PERCENT_PERMIT is permitted range for the targets and UNCERTAINTY_PERCENT_PERMIT is permitted range for the uncertainties.

Finally, compile the C program "genetic.c" and execute it.

How to change GA parameters

The efficiency of the GA depend on the values of parameters such as the number of population, the number of generations ... which are not defined for any problem but must be adapted for each GA. The values are entered directly in the "genetic.h" file so you have to edit it and change the desired values in the header of the file.

5. References.

Ryu, J. H., Model for mechanical properties of Hot-Rolled Steels, Master Thesis, POSTECH, 2008.

Shah, I., Tensile Properties of Austenite Stainless Steel, M. Phil. Thesis, University of Cambridge, 2002.

Delorme, A., Genetic algorithm for optimization of mechanical properties, Technical report, University of Cambridge, 2003.

6. Parameter descriptions.

Input parameters

To specify the target value and accuracy desired, *input/TARGETS* must be amended. The normalized target value relates to a real value, according to *the input/MINMAX* file used.

Row1	The number of targets
Row2	Normalised target value of UTS and its accuracy (in decimal e.g. 0.1 for 10%)
Row3	Normalised target value of EL and its accuracy (in decimal e.g. 0.1 for 10%)

To specify the permitted range of the output, *genetic.h* must be changed.

#define MAX_UNNORMAL_UTS	The maximum UTS in MINMAX
#define MIN_UNNORMAL_UTS	The minimum UTS in MINMAX
#define MAX_UNNORMAL_EL	The maximum elongation in MINMAX
#define MIN_UNNORMAL_EL	The minimum elongation in MINMAX
#define TARGET_UNNORMAL_UTS	The target of UTS
#define TARGET_UNNORMAL_EL	The target of elongation
#define TARGET_PERCENT_PERMIT	The permitted range for the targets
#define UNCERTAINTY_PERCENT_PERMIT	The permitted range for the uncertainties

To initiate the GA search, the inputs are randomly generated and placed in *input/norm_test.in*. It should be noted that each chromosome generally relates to a different steel composition, but this could change over the course of optimization. For the current multi-objective model, the composition and processing variables, totally 17 data items, are specified:

Gene number	Variable
1	Normalised C, wt%
2	Normalised Mn, wt%
3	Normalised Si, wt%
4	Normalised P, wt%
5	Normalised S, wt%
6	Normalised Cr, wt%
7	Normalised Ni, wt%
8	Normalised Mo, wt%
9	Normalised Ti, wt%
10	Normalised Nb, wt%
11	Normalised V, wt%
12	Normalised Al, wt%
13	Normalised N, ppm
14	Normalised B, ppm
15	Normalised Cu, wt%

16	Normalised Finishing rolling temperature, ˚C
17	Normalised Coiling temperature, ˚C

Each input is normalized using the equation:

Normalized value = (value – min)/(max-min) – 0.5

Where the values for min and max are defined as follows:

Gene number	Variable	Min	Max
1	C, wt%	0.0204	0.8684
2	Mn, wt%	0.1670	1.4100
3	Si, wt%	0.0000	0.2170
4	P, wt%	0.0040	0.0220
5	S, wt%	0.0020	0.0150
6	Cr, wt%	0.0000	0.1600
7	Ni, wt%	0.0000	0.0600
8	Mo, wt%	0.0000	0.0200
9	Ti, wt%	0.0000	0.0040
10	Nb, wt%	0.0000	0.0040
11	V, wt%	0.0000	0.0030
12	Al, wt%	0.0000	0.0640
13	N, ppm	0.00	87.00

14	B, ppm	0.00	2.00
15	Cu, wt%	0.0000	0.0300
16	Finishing rolling temperature, ˚C	808.00	925.00
17	Coiling temperature, ˚C	478.00	714.00

Output parameters

Two output files are produced by the GA program: *unnormalise/nn-output_b* and *score-output.*

score-output simply prints out the scores for each chromosome.

unnormalise/nn-output_b contains the inputs, prediction and (prediction + error). This is done for the best chromosomes within all populations:

Column 1-17	The normalized predicted inputs
Column 18	The normalized predicted UTS
Column 19	The normalized uncertainty for UTS
Column 20	The normalized predicted EL
Column 21	The normalized uncertainty for EL

The normalized values in all columns must be un-normalised using the equation:

$$\text{actual value} = (\text{normalized value} + 0.5) * (\text{max-min}) + \text{min}$$

The C program, *unnormalise/treatout.c*, is used to translate the output files to produce the actual values of inputs and outputs which are written to

unnormalise/result.

7. Error indicators.

None.

8. Accuracy estimate.

See:

Input parameters, output parameters.

9. Any additional information.

See:

MAP_HOTROLLEDSTEEL_UTS, MAP_HOTROLLEDSTEEL_EL

10. Example.

1. Program text

Complete program

2. Program data

The input file (*input/nn-input*) is:

2

-0.385042 0.1
0.083333 0.1

The input variables (*genetic.h*) are:

```
#define MAX_UNNORMAL_UTS                    1039.0
#define MIN_UNNORMAL_UTS                    317.0
#define MAX_UNNORMAL_EL                     50.0
#define MIN_UNNORMAL_EL                     14.0
#define TARGET_UNNORMAL_UTS                 400.0
#define TARGET_UNNORMAL_EL                  35.0
#define TARGET_PERCENT_PERMIT               0.1
#define UNCERTAINTY_PERCENT_PERMIT          0.15
```

3. Program results

The output file unnormalise/nn-output_b, which contains the normalised values for the ultimate tensile strength model including compositions and processing variables (following result is only single selected case):

C	**Mn**	**Si**	**P**	**S**
-0.467796	-0.247790	-0.287128	0.103290	-0.404806

Cr	**Ni**	**Mo**	**Ti**	**Nb**
-0.431536	-0.128436	0.153316	-0.200750	-0.233212

V	**Al**	**N**	**B**	**Cu**
1.000000	0.206896	-0.148574	-0.157592	-0.194496

FRT	**CT**	**Predicted UTS**	**Uncertainty for UTS**
0.001974	-0.414350	-0.381139	-0.323613

Predicted EL	**Uncertainty for EL**
0.133929	0.246229

These are then run with *unnormalise/treatout.c*, where the normalised values are converted into the actual values to *unnormalise/result*:

C	**Mn**	**Si**	**P**	**S**
0.047709	0.480497	0.046193	0.014859	0.003238

Cr	**Ni**	**Mo**	**Ti**	**Nb**
0.010954	0.022294	0.013066	0.001197	0.001067

V	**Al**	**N**	**B**	**Cu**
0.004500	0.045241	30.57406	0.684816	0.009165

FRT	**CT**	**Predicted UTS**	**Uncertainty for UTS**
866.7310	498.2134	402.82	41.53

Predicted EL	**Uncertainty for EL**
36.82	4.04

11. Auxiliary subroutines required.

None.

12. Keywords.

hot-rolled steel, genetic algorithm, ultimate tensile strength, elongation, neural network.

13. Sources.

genetic.h

```
#ifndef __GENETIC_H__
#define __GENETIC_H__

#include <stdio.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <math.h>
#include <sys/time.h>

/* NN related */
#define NUM                17         /* Total number of genes
*/
#define LIMIT              150        /* Maximum number of
inputs the system can handle */
#define SESSIONS           1000       /* Number of
generations that we'll put the system through */

/* GA related */
#define POPS               3          /* Number of
populations */
#define SIZE               17         /* Size of inputs in the
genetic algorithms */
#define MAXPOP                        20         /* Size
of population, number of chromosomes */
#define BESTPOP                       3          /*
Number of individuals taken from the best */
#define SELPOP                        12         /*
SELPOP-BESTPOP = Number of people selected randomly to exchange
genes within their own population */
#define NEWPOP                        16         /*
NEWPOP-SELPOP = Number of new people created randomly on each
gen. */
#define MUT1               18         /* MUT1-NEWPOP =
Number of genes that are severely mutated */
#define MIXGEN                        1          /*
Number of generations between population mixing */
/* for Fixed Values of inputs */
#define CAR                -0.03703703   /* Carbon */
#define MAN                0.28512388    /* Manganese */
#define SIL                0.03913045    /* Silicon */
#define PHO                0.03125002    /* Phosphorus */
#define SUL                -0.3222222    /* Sulphur */
#define CHR                -0.1                     /*
Chromium */
#define NIC                0.1                      /*
Nickel */
#define MOL                -0.33848798     /* Molybdenum */
#define TIT                -0.42857143   /* Titanium */
#define NIO                -0.48947367   /* Niobium */

#define VAN                -0.5                     /*
Vanadium */
#define ALU                -0.5                     /*
Aluminium */
#define NIT                -0.11728396   /* Nitrogen */
#define BOR                -0.4333333    /* Boron */
#define COP                -0.01428570   /* Copper */
#define FRT                0.1                      /*
Finish Deformation Temperature */
#define CIT                0.1                      /*
Coiling Temperature */

/* OBJECTIVES */
#define OBJECTIVE_FILE_NAME          "./input/TARGETS"
#define MINMAX_FILE_NAME             "./input/MINMAX"
#define INPUT_VAR_FILE_NAME          "./input/norm_test.in"

#define RESULT_FILE_NAME "./unnormalise/nn-output_b"

#define MAX_TARGET_NUM               10
#define MAX_SMALL_MODEL_NUM          25

/* For two model, UTS, EL */
#define MAX_UNNORMAL_UTS                        1039.0
#define MIN_UNNORMAL_UTS                        317.0
#define MAX_UNNORMAL_EL
        50.0
#define MIN_UNNORMAL_EL                         14.0

#define TARGET_UNNORMAL_UTS                     400.0
#define TARGET_UNNORMAL_EL                      35.0
#define TARGET_PERCENT_PERMIT                   0.1
#define UNCERTAINTY_PERCENT_PERMIT  0.15

typedef struct _targets
{
        int num_targets;
        float targets[MAX_TARGET_NUM];
        float targets_accuracy[MAX_TARGET_NUM];
        float targets_limit_err[MAX_TARGET_NUM];
        float total_limit_err;
} targets_strt;

typedef struct _populations
{
        float pop[POPS][MAXPOP][SIZE];
        float total_score[POPS][MAXPOP];
        float score[MAX_TARGET_NUM][POPS][MAXPOP];
        float result[MAX_TARGET_NUM][POPS][MAXPOP];
        float error[MAX_TARGET_NUM][POPS][MAXPOP];
} populations_strt;

#define         MAX_SYSTEM_STRING       255
```

```
// Computation of NN models with the inputs
void check_performance();

// initialize the populations
void make_initial_population();

// generate the next generation in all populations
void make_next_generation(int generation);

// sort the populations
void sort_population();

// set the error limitation
void calc_limit_err();

// score calculation
void calc_score();

// get data
void get_data();

#endif /* __GENETIC_H__ */
```

genetic.c

```
#include "genetic.h"

static targets_strt objectives;
static populations_strt populations;
static float input_to_nn[NUM];
static float input_to_nn_all[POPS][MAXPOP][NUM];

/* initialize this program */
static void init() {
		memset(&objectives, 0x00, sizeof(targets_strt));
		memset(&populations, 0x00, sizeof(populations_strt));
}

/* init random */
static void randomize() {
		struct timeval tp;
		struct timezone tzp;

		/* Use time of day to feed the random number generator
seed */
		gettimeofday( &tp, &tzp);
		srandom( tp.tv_sec );
}

/* input apply for NN */
static void apply(int p,int i) {
		int j;
		/* Each component realtes to a gene in the NN */
		for(j=0;	j<SIZE;	j++)	input_to_nn[j]	=
populations.pop[p][i][j];
}

static void apply_all() {
		int p, i, j;
		for(p=0; p<POPS; p++) {
				for(i=0; i<MAXPOP; i++) {
						for(j=0;j<SIZE;j++) {

		input_to_nn_all[p][i][j] = populations.pop[p][i][j];
						}
				}
		}
}
// swap two vectors and scores in the population p
static void pop_swap(int p,int a,int b) {
		int i;
		float tg;

		/* Swap vector */
		for ( i = 0 ; i < SIZE ; i++ )
		{
				tg = populations.pop[p][a][i];
				populations.pop[p][a][i]		=
populations.pop[p][b][i];
				populations.pop[p][b][i] = tg;
		}

		tg = populations.total_score[p][a];
		populations.total_score[p][a]		=
populations.total_score[p][b];
		populations.total_score[p][b] = tg;

		/* Swap score */
		for(i = 0; i<objectives.num_targets; i++) {
				tg = populations.score[i][p][a];
				populations.score[i][p][a]		=
populations.score[i][p][b];
				populations.score[i][p][b] = tg;
				tg = populations.result[i][p][a];

				populations.result[i][p][a]		=
populations.result[i][p][b];
				populations.result[i][p][b] = tg;

				tg = populations.error[i][p][a];
				populations.error[i][p][a]		=
populations.error[i][p][b];
				populations.error[i][p][b] = tg;
		}
}

// find the population with the best solution
static int best_pop() {
		int m, i, p;
		float best=0;
		for ( i = 0 ; i < POPS ; i++ ) {
				if ( populations.total_score[i][0] > best )
{
						best			=
populations.total_score[i][0];
						p = i;
				}
		}
		return p;
}
```

```
// return a random integer in the range 0..(range-1)
static int irand(int range)
{
        return( random() % range );
}
// copy the vector b in the population p2 into the vector a in the population p1
static void pop_copy(int p1,int a,int p2,int b)
{
        int i;
        for ( i = 0 ; i < SIZE ; i++ )
                populations.pop[p1][a][i] = populations.pop[p2][b][i];
}

/* for NN model */
static void model(int model_num) {
        FILE *fp = NULL;

        int num_res_err = 0, p = 0, i = 0, j = 0;
        float res[POPS][MAXPOP][MAX_SMALL_MODEL_NUM] = {'\0'};
        float err[POPS][MAXPOP][MAX_SMALL_MODEL_NUM] = {'\0'};
        float sum_res[POPS][MAXPOP] = {'\0'}, sum_err[POPS][MAXPOP] = {'\0'};
        float sum_diff_err[POPS][MAXPOP] = {'\0'}, dump;

        if(model_num == 0) {
                /* small model is doing */
                system("./generate44 ./input/11/spec1.t16 ./input/11/_wf1f ./input/11/_wf1f.lu &> /dev/null");
                if ( (fp = fopen("_out","r")) == NULL) {
                        printf ("Cannot open _out stage 1 of model %d", model_num);
                        exit(1);
                }
                for(p=0;p<POPS;p++) {
                        for(i=0;i<MAXPOP;i++) {
        fscanf(fp,"%f %f %f %f",
        &res[p][i][num_res_err],
        &err[p][i][num_res_err],
        &dump, &dump);
                        }
                }
                fclose(fp);
                fp = NULL;
                num_res_err++;

                system("./generate44 ./input/11/spec1.t16 ./input/11/_wf3f ./input/11/_wf3f.lu &> /dev/null");
                if ( (fp = fopen("_out","r")) == NULL) {
                        printf ("Cannot open _out stage 1 of model %d", model_num);
                        exit(1);
                }
                for(p=0;p<POPS;p++) {
                        for(i=0;i<MAXPOP;i++) {
        fscanf(fp,"%f %f %f %f",
        &res[p][i][num_res_err],
        &err[p][i][num_res_err],
        &dump, &dump);
                        }
                }
                fclose(fp);
                fp = NULL;
                num_res_err++;

                system("./generate44 ./input/11/spec1.t18 ./input/11/_wh2f ./input/11/_wh2f.lu &> /dev/null");
                if ( (fp = fopen("_out","r")) == NULL) {
                        printf ("Cannot open _out stage 1 of model %d", model_num);
                        exit(1);
                }
                for(p=0;p<POPS;p++) {
        for(i=0;i<MAXPOP;i++) {
        fscanf(fp,"%f %f %f %f",
        &res[p][i][num_res_err],
        &err[p][i][num_res_err],
        &dump, &dump);
                        }
                }
                fclose(fp);
                fp = NULL;
                num_res_err++;
                system("./generate44 ./input/11/spec1.t18 ./input/11/_wh4f ./input/11/_wh4f.lu &> /dev/null");
                if ( (fp = fopen("_out","r")) == NULL) {
                        printf ("Cannot open _out stage 1 of model %d", model_num);
                        exit(1);
                }
                for(p=0;p<POPS;p++) {
        for(i=0;i<MAXPOP;i++) {
        fscanf(fp,"%f %f %f %f",
        &res[p][i][num_res_err],
        &err[p][i][num_res_err],
        &dump, &dump);
                        }
                }
                fclose(fp);
                fp = NULL;
                num_res_err++;

                system("./generate44 ./input/11/spec1.t19 ./input/11/_wi4f ./input/11/_wi4f.lu &> /dev/null");
                if ( (fp = fopen("_out","r")) == NULL) {
                        printf ("Cannot open _out stage 1 of model %d", model_num);
                        exit(1);
                }
                for(p=0;p<POPS;p++) {
        for(i=0;i<MAXPOP;i++) {
        fscanf(fp,"%f %f %f %f",
        &res[p][i][num_res_err],
        &err[p][i][num_res_err],
        &dump, &dump);
                        }
                }
                fclose(fp);
                fp = NULL;
                num_res_err++;

                system("./generate44 ./input/11/spec1.t110 ./input/11/_wj2f ./input/11/_wj2f.lu &> /dev/null");
                if ( (fp = fopen("_out","r")) == NULL) {
                        printf ("Cannot open _out stage 1 of model %d", model_num);
                        exit(1);
                }
                for(p=0;p<POPS;p++) {
```

```
for(i=0;i<MAXPOP;i++)
{
    fscanf(fp,"%f %f %f %f",
        &res[p][i][num_res_err],
        &err[p][i][num_res_err],
        &dump, &dump);
    }
}
fclose(fp);
fp = NULL;
num_res_err++;

system("./generate44 ./input/11/spec1.t1
10 ./input/11/_wj4f ./input/11/_wj4f.lu &> /dev/null");
if ( (fp = fopen("_out","r")) == NULL) {
    printf ("Cannot open _out
stage 1 of model %d", model_num);
    exit(1);
}
for(p=0;p<POPS;p++) {
    for(i=0;i<MAXPOP;i++)
{
    fscanf(fp,"%f %f %f %f",
        &res[p][i][num_res_err],
        &err[p][i][num_res_err],
        &dump, &dump);
    }
}
fclose(fp);
fp = NULL;
num_res_err++;

system("./generate44 ./input/11/spec1.t1
10 ./input/11/_wj5f ./input/11/_wj5f.lu &> /dev/null");
if ( (fp = fopen("_out","r")) == NULL) {
    printf ("Cannot open _out
stage 1 of model %d", model_num);
    exit(1);
}
for(p=0;p<POPS;p++) {
    for(i=0;i<MAXPOP;i++)
{
    fscanf(fp,"%f %f %f %f",
        &res[p][i][num_res_err],
        &err[p][i][num_res_err],
        &dump, &dump);
    }
}
fclose(fp);
fp = NULL;
num_res_err++;

system("./generate44 ./input/11/spec1.t1
15 ./input/11/_wo5f ./input/11/_wo5f.lu &> /dev/null");
if ( (fp = fopen("_out","r")) == NULL) {
    printf ("Cannot open _out
stage 1 of model %d", model_num);
    exit(1);
}
for(p=0;p<POPS;p++) {
    for(i=0;i<MAXPOP;i++)
{
    fscanf(fp,"%f %f %f %f",
        &res[p][i][num_res_err],
        &err[p][i][num_res_err],
        &dump, &dump);
```

```
    }
}
fclose(fp);
fp = NULL;
num_res_err++;
system("./generate44 ./input/11/spec1.t1
19 ./input/11/_ws4f ./input/11/_ws4f.lu &> /dev/null");
if ( (fp = fopen("_out","r")) == NULL)
{
    printf ("Cannot open
_out stage 1 of model %d", model_num);
    exit(1);
}
for(p=0;p<POPS;p++) {
    for(i=0;i<MAXPOP;i++) {
    fscanf(fp,"%f %f %f %f",
        &res[p][i][num_res_err],
        &err[p][i][num_res_err],
        &dump, &dump);
    }
}
fclose(fp);
fp = NULL;
num_res_err++;

/* results */
for (p=0;p<POPS;p++) {
    for(i=0;i<MAXPOP;i++) {
    sum_res[p][i] = 0.0;
    sum_err[p][i] = 0.0;

    for(j=0;j<num_res_err;j++) {
    sum_res[p][i] += res[p][i][j];
    sum_err[p][i] += err[p][i][j];
    }
    populations.result[model_num][p][i] =
        sum_res[p][i]/num_res_err;

    sum_diff_err[p][i] = 0.0;
    for(j=0;j<num_res_err;j++) {
    sum_diff_err[p][i] +=
        pow(res[p][i][j]-
        objectives.targets[model_num],
        2);
    }
    populations.error[model_num][p][i] =
    sqrt((pow(sum_err[p][i],2)/num_res_err)+
    (sum_diff_err[p][i]/num_res_err));
    if
(populations.error[model_num][p][i] < 0) {
    populations.score[model_num][p][i] =

    0.0;
    } else
{
```

```
populations.score[model_num][p][i] =

1.0/populations.error[model_num][p][i];
                                        }
                                }
                        }
                }
                if(model_num == 1) {
                        /* small model is doing */
                        system("./generate44    ./input/17/spec1.t1
8 ./input/17/_wh1f ./input/17/_wh1f.lu &> /dev/null");
                        if ( (fp = fopen("_out","r")) == NULL) {
                                printf ("Cannot open _out
stage 1 of model %d", model_num);
                                exit(1);
                        }
                        for(p=0;p<POPS;p++) {
                                for(i=0;i<MAXPOP;i++)
{

                fscanf(fp,"%f %f %f %f",

                &res[p][i][num_res_err],

                &err[p][i][num_res_err],

                &dump, &dump);
                                }
                        }
                        fclose(fp);
                        fp = NULL;
                        num_res_err++;

                        system("./generate44    ./input/17/spec1.t1
8 ./input/17/_wh4f ./input/17/_wh4f.lu &> /dev/null");
                        if ( (fp = fopen("_out","r")) == NULL) {
                                printf ("Cannot open _out
stage 1 of model %d", model_num);
                                exit(1);
                        }
                        for(p=0;p<POPS;p++) {
                                for(i=0;i<MAXPOP;i++)
{

                fscanf(fp,"%f %f %f %f",

                &res[p][i][num_res_err],

                &err[p][i][num_res_err],

                &dump, &dump);
                                }
                        }
                        fclose(fp);
                        fp = NULL;
                        num_res_err++;

                        system("./generate44    ./input/17/spec1.t1
12 ./input/17/_wl4f ./input/17/_wl4f.lu &> /dev/null");
                        if ( (fp = fopen("_out","r")) == NULL) {
                                printf ("Cannot open _out
stage 1 of model %d", model_num);
                                exit(1);
                        }
                        for(p=0;p<POPS;p++) {
                                for(i=0;i<MAXPOP;i++)
{

                fscanf(fp,"%f %f %f %f",

                &res[p][i][num_res_err],

                &err[p][i][num_res_err],

                &dump, &dump);
                                }
                        }
                        fclose(fp);
                        fp = NULL;
                        num_res_err++;

                        /* results */
```

```
                        for (p=0;p<POPS;p++) {

                for(i=0;i<MAXPOP;i++) {

                sum_res[p][i] = 0.0;

                sum_err[p][i] = 0.0;


                for(j=0;j<num_res_err;j++) {

                sum_res[p][i] += res[p][i][j];

                sum_err[p][i] += err[p][i][j];
                                        }

                populations.result[model_num][p][i] =

                        sum_res[p][i]/num_res_err;


                sum_diff_err[p][i] = 0.0;

                for(j=0;j<num_res_err;j++) {

                sum_diff_err[p][i] +=

                        pow(res[p][i][j]-

                        objectives.targets[model_num],

                        2);
                                        }

                populations.error[model_num][p][i] =

                sqrt((pow(sum_err[p][i],2)/num_res_err)+

                (sum_diff_err[p][i]/num_res_err));

                                        if
(populations.error[model_num][p][i] < 0) {

                populations.score[model_num][p][i] =


                0.0;
                                        }   else
{

                populations.score[model_num][p][i] =

                1.0/populations.error[model_num][p][i];
                                        }
                                }
                        }
                }
}

/* Main Function */
int main() {
                int generation, done;
                int m, p, i, j, k, l;
                double      tempResultUTS,      tempUncertaintyUTS,
tempResultEL, tempUncertaintyEL;

                FILE *fp = NULL, *fcount = NULL;

                // initialize the program
                init();
                randomize();

                // read input from file
                get_data();

                // make initial populations
                make_initial_population();

                // calculate the initial score with NN model
                calc_score();
                sort_population();
```

```
generation = 0;
done = 0;

if ((fp=fopen(RESULT_FILE_NAME,"a+")) == NULL) {
        printf("Cannot open %s\n", RESULT_FILE_NAME);
        exit(1);
}

while ( (done != 1 ) && ( generation++ < SESSIONS ) ) {
        if((fcount = fopen("count", "w")) == NULL){
                printf("Cannot open count\n");
                exit(1);
        }
        fprintf(fcount, "%d", generation);
        fclose(fcount);

        make_next_generation( generation );
        p = best_pop();

        if ( populations.total_score[p][0] > objectives.total_limit_err ) {
                printf ("Target error was %f and error is %f",
        objectives.total_limit_err,
        populations.total_score[p][0]);
                done = 1;
        } else {
                printf("Done all, target : %f, score : %f!!\n", objectives.total_limit_err, populations.total_score[p][0]);
        }

        for (i=0;i<POPS;i++) {
                for (j=0;j<BESTPOP;j++) {
                        apply (i,j);
                        l=0;
                        /* for only two model, UTS, EL */

        tempResultUTS = (populations.result[0][i][j] + 0.5) * (MAX_UNNORMAL_UTS - MIN_UNNORMAL_UTS) + MIN_UNNORMAL_UTS;

        tempUncertaintyUTS = ((populations.error[0][i][j] + populations.result[0][i][j] + 0.5) * (MAX_UNNORMAL_UTS - MIN_UNNORMAL_UTS) + MIN_UNNORMAL_UTS) - tempResultUTS;


        tempResultEL = (populations.result[1][i][j] + 0.5) * (MAX_UNNORMAL_EL - MIN_UNNORMAL_EL) + MIN_UNNORMAL_EL;

        tempUncertaintyEL = ((populations.error[1][i][j] + populations.result[1][i][j] + 0.5) * (MAX_UNNORMAL_EL - MIN_UNNORMAL_EL) + MIN_UNNORMAL_EL) - tempResultEL;

                        if ((fabs(TARGET_UNNORMAL_UTS - tempResultUTS) < (TARGET_PERCENT_PERMIT * TARGET_UNNORMAL_UTS))

        && (fabs(TARGET_UNNORMAL_EL - tempResultEL) < (TARGET_PERCENT_PERMIT * TARGET_UNNORMAL_EL))

        && (tempUncertaintyUTS < (2 * UNCERTAINTY_PERCENT_PERMIT * TARGET_UNNORMAL_UTS))

        && (tempUncertaintyEL < (2 * UNCERTAINTY_PERCENT_PERMIT * TARGET_UNNORMAL_EL))) {
```

```
                                for (k=0;k<SIZE;k++) {
#if 1

        fprintf(fp,"%f ",input_to_nn[l]);

        ++l;
#else

        if (k != 19) {

                fprintf(fp,"%f ",input_to_nn[l]);

                ++l;

        } else {

        /* change this line according to which

        variables are wanted to be fixed */

                if (k == 19) fprintf(fp,"%f ",pwhtT);

        }
#endif
                                }
                                for(m = 0; m < objectives.num_targets; m++) {

        fprintf(fp,"%f %f ",populations.result[m][i][j],(populations.error[m][i][j]+populations.result[m][i][j]));
                                }

        fprintf(fp, "\n");
                                }
                        }
                }
        }

        printf("done! \n");
        fclose(fp);
        return 1;
}

void get_data() {
        FILE * fp = NULL;

        int num_targets = 0, i = 0;
        float temp_value = 0.0;

        if ((fp = fopen(OBJECTIVE_FILE_NAME, "r")) == NULL) {
                printf("Cannot open the %s\n", OBJECTIVE_FILE_NAME);
                exit(1);
        }
        fscanf(fp, "%d", &num_targets); // get model(target) numbers

        objectives.num_targets = num_targets;
        for(i = 0; i < num_targets; i++) {
                fscanf(fp, "%f%f",
                        &(objectives.targets[i]),
        &(objectives.targets_accuracy[i]));
                temp_value = pow(((objectives.targets[i]) * (objectives.targets_accuracy[i])), 2);
                objectives.targets_limit_err[i] = 1.0/sqrt(temp_value);
        }
        for (i = 0; i < num_targets; i++) {
                objectives.total_limit_err += objectives.targets_limit_err[i];
        }
        fclose(fp);
        return;
}
```

```
void make_initial_population() {
	int p, i, j;
	char c;
	FILE *fp = NULL;
	float min[NUM+MAX_TARGET_NUM]={'\0'}, max[NUM+MAX_TARGET_NUM]={'\0'};
	float dummy = 0;

	if((fp=fopen(MINMAX_FILE_NAME,"r")) == NULL) {
		printf("Cannot open %s\n", MINMAX_FILE_NAME);
		exit(1);
	}

	while((c=getc(fp)) != '\n');
	for (j = 0; j < SIZE+objectives.num_targets; j++) {

	fscanf(fp,"%f%f%f%f%f%f",&min[j],&max[j],

	&dummy,&dummy,&dummy,&dummy);
	}

	for ( p = 0 ; p < POPS ; p++ ) {
		/* Whole population gets values from -1 to 1 */
		for ( i = 0 ; i < MAXPOP ; i++ ) {
			for ( j = 0 ; j < SIZE ; j++ ) {

	populations.pop[p][i][j] = ((random()&1048575) / 1000000.0 - 0.5) * 2;
				while ( populations.pop[p][i][j] < (-(min[j]/(max[j]-min[j])+0.5))) {

	populations.pop[p][i][j] = ((random()&1048575) / 1000000.0 - 0.5) * 2;
				}
			}
		}
	}
	fclose(fp);
	return;
}

void calc_score() {
	int p, i;

#if 0
	for ( p = 0 ; p < POPS ; p++ ){
		for ( i = 0 ; i < MAXPOP ; i++ ){
			apply(p, i);
			check_performance(p,i);
		}
	}
#else
	apply_all();
	check_performance();
#endif
	return;
}

void check_performance()
{
	FILE *fp_input_var = NULL,*fp_output = NULL;
	char system_string[MAX_SYSTEM_STRING] = {'\0'};

	int p,i,j,l,r;

	/* set the input values */
	if ((fp_input_var = fopen(INPUT_VAR_FILE_NAME,"w")) == NULL ) {
		printf ("Cannot open %s\n", INPUT_VAR_FILE_NAME);
		exit(1);
	}
	for(p=0;p<POPS;p++){
		for(i=0;i<MAXPOP;i++){
			l=0;
			for (j=0;j<SIZE;j++) {
```

```
#if 1

	fprintf(fp_input_var,"%f ",input_to_nn_all[p][i][l]);
						++l;
#else
						if (j != 19) {

	/* change this line according to which

	   variables are wanted to be fixed */

	fprintf(fp_input_var,"%f ",input_to_nn_all[p][i][l]);

	++l;
						} else {

	if (j == 19) fprintf(fp_input_var,"%f ",pwhtT);
						}
#endif
				}

	fprintf(fp_input_var,"\n");
			}
	}
	fclose(fp_input_var);
	for(j = 0; j<objectives.num_targets;j++) {
		model(j);
	}

	for(p=0;p<POPS;p++) {
		for(i=0;i<MAXPOP;i++) {

	populations.total_score[p][i] = 0.0;
		}
	}

	for(p=0;p<POPS;p++) {
		for(i=0;i<MAXPOP;i++) {
#if 0
				for(j = 0; j<objectives.num_targets;j++) {

	populations.total_score[p][i] += populations.score[j][p][i];
				}
#else
				// Two models
				r = irand(1000);

	populations.total_score[p][i] =

	(((float)r/1000) * populations.score[0][p][i]) + ((1 - ((float)r/1000)) *populations.score[1][p][i]);
#endif
		}
	}

	return;
}

void sort_population()
{
	int m, p, i, j, k;
	float best;
	/* Use insert sort */

	for ( p = 0 ; p < POPS ; p++ ){
		/* best total_score */
		for ( i = 0 ; i < MAXPOP ; i++ ){
				best = populations.total_score[p][i];
				k = i;
				for ( j = (i+1) ; j < MAXPOP ; j++ )
						if ( populations.total_score[p][j] >= best )
						{

	best = populations.total_score[p][j];
```

```
			k = j;
					}
				if ( best > populations.total_score[p][i] )
			pop_swap( p, i, k );
		}

		/* best for model i */
		for (m = 0; m<objectives.num_targets; m++) {
				best = populations.score[m][p][m+1];
				k = m+1;
				for ( i = m+2 ; i < MAXPOP ; i++ ){
					if ( populations.score[m][p][i] >= best ) {
			best = populations.score[m][p][i];
			k = i;
					}
				}
				pop_swap(p, m+1, k);
		}
	}
}

void make_next_generation(int generation)
{
	int p, i, j, k1, k2, l, stack;
	float dev = 0.0, min[NUM+MAX_TARGET_NUM] = {'\0'}, max[NUM+MAX_TARGET_NUM] = {'\0'}, dummy = 0.0;
	char c;
	FILE *fp = NULL;

	if((fp=fopen(MINMAX_FILE_NAME,"r")) == NULL) {
		printf("Cannot open file %s\n", MINMAX_FILE_NAME);
		exit(1);
	}

	while((c=getc(fp)) != '\n');
	for (j = 0; j < SIZE+objectives.num_targets; j++) {
	fscanf(fp,"%f%f%f%f%f%f",&min[j],&max[j],
	&dummy,&dummy,&dummy,&dummy);
	}

	for ( p = 0 ; p < POPS ; p++ ) {
		/* keep best - BESTPOP */
		/* add another group, randomly - (SELPOP-BESTPOP) */
		for ( i = BESTPOP ; i < SELPOP ; i++ ) {
				pop_swap( p, i, (irand( MAXPOP - i ) + i) );
		}
		/* create new individuals */
		for ( i = MUT1 ; i < MAXPOP ; i++ ) {

				stack = 0;
				for ( j = 0 ; j < SIZE ; j++ ) {

	populations.pop[p][i][j] = ((random()&1048575) / 1000000.0 - 0.5) * 2;
					while ( populations.pop[p][i][j] < (-(min[j]/(max[j]-min[j])+0.5))) {

	populations.pop[p][i][j] = ((random()&1048575) / 1000000.0 - 0.5) * 2;
					}
				}
		}
		/* SELPOP to MUT1 will be severe mutations */
				stack = 0;
				for ( i = NEWPOP ; i < MUT1 ; i++ ) {
					pop_copy( p, i, p, irand(NEWPOP) );
					/* 5000 is the nominal mutation value */
					dev = 1 + ((irand(2000) - 1000 )/ 5000);
					j=irand(SIZE);

					while ( dev < (-(min[j]/(max[j]-min[j])+0.5))) {
						dev = 1 + ((irand(2000) - 1000 )/ 5000);
					}
					populations.pop[p][i][j] = dev;
				}

				/* MUT2 to MAXPOP will be crossovers */

				stack = 0;
				for ( i = SELPOP ; i < NEWPOP ; i++ ) {

					/* Every several generations (set by MIXGEN) there is a cross-over */
					/* between different populations. */
					pop_copy( p, i, (((generation%MIXGEN)==0) ? irand(POPS) : p), irand(NEWPOP) );
					j = irand(NEWPOP);
					k1 = irand( SIZE - 1);
					k2 = irand( SIZE - 1 - k1 ) + k1 + 1;
					for ( l = k1 ; l <= k2 ; l++ ) populations.pop[p][i][l] = populations.pop[p][j][l];

					/* Mutate slightly to preserve diversity */
					dev = 1 + ((irand(2000) - 1000 )/ 50000);
					j=irand(SIZE);

					while ( dev < (-(min[j]/(max[j]-min[j])+0.5))) {
						dev = 1 + ((irand(2000) - 1000 )/ 50000);
					}
					populations.pop[p][i][j] = dev;
				}
		}
		calc_score();
		sort_population();
		printf("Done %d generations\n\n",generation);

		fclose(fp);
		return;
}
```

Curriculum Vitae

Name: Joo, Min Sung

E-mail: athpimo@postech.ac.kr

Date of birth: 23th June, 1980

Place of birth: Seoul, South Korea

Education

M. S. 2008, POSTECH (Pohang, Korea), Graduate Institute of Ferrous Technology, Computational Metallurgy Group

B. S. 2006, POSTECH (Pohang, Korea), Department of Computer Science and Engineering

Publications

Minsung Joo, Joohyun Ryu and H. K. D. H. Bhadeshia: Domains of Steels with Identical Properties, submitted to *Materials and Manufacturing Processes.*