1/1/2016

Genetic Algorithm

Vision System for machined surfaces’ roughness prediction

Contents

[Acronyms and Abbreviations 2](#_Toc469417394)

[Background 3](#_Toc469417395)

[1.Introduction 3](#_Toc469417396)

[2.Project Proposal and Aim 4](#_Toc469417397)

[3. Theory 4](#_Toc469417398)

[3.1 Surface Roughness 4](#_Toc469417399)

[Image Processing: 4](#_Toc469417400)

[Cutting Speed: 5](#_Toc469417401)

[Feed 5](#_Toc469417402)

[Depth of cut: 5](#_Toc469417403)

[Interaction between the three parameters: 5](#_Toc469417404)

[3.2 Genetic Algorithm Predicting Roughness 6](#_Toc469417405)

[Fitness Function 6](#_Toc469417406)

[Selection Technique – tournament selection 6](#_Toc469417407)

[Crossover Technique 6](#_Toc469417408)

[Mutation Technique 7](#_Toc469417409)

[3.3 Neural Network Predicting Roughness 7](#_Toc469417410)

[Neural Network Architecture 7](#_Toc469417411)

[Fitness function 7](#_Toc469417412)

[Training 7](#_Toc469417413)

[Value Scaling 8](#_Toc469417414)

[3.3 Genetic Algorithm Predicting Parameters 8](#_Toc469417415)

[4.Physical Aspects and Experiment 9](#_Toc469417416)

[Description 9](#_Toc469417417)

[Experiment 9](#_Toc469417418)

[5.Optimisation 9](#_Toc469417419)

[Population Size 9](#_Toc469417420)

[Tournament Size 9](#_Toc469417421)

[Fitness Function 9](#_Toc469417422)

[End Condition 9](#_Toc469417423)

[6. Results and discussion 9](#_Toc469417424)

[Genetic Algorithm Predicting Parameters 10](#_Toc469417425)

[Genetic Algorithm experiment and Results: 10](#_Toc469417426)

[Fitness Equation 1: 10](#_Toc469417427)

[Fitness Equation 2: 15](#_Toc469417428)

[Comparison between these two fitness functions: 18](#_Toc469417429)

[Predicting Roughness with the selected equation: 19](#_Toc469417430)

[Comparison of NN parameters and their corresponding results 21](#_Toc469417431)

[7. Conclusion 25](#_Toc469417432)

[8.Reference 25](#_Toc469417433)

[Bibliography 25](#_Toc469417434)

[9. Appendix 26](#_Toc469417435)

[Code 26](#_Toc469417436)

# Acronyms and Abbreviations

V Spindle Speed

F Feed Rate

Ra Surface Roughness

Ga Average Grey Scale value of Surface digital image

NN Neural Networks

GA Genetic Algorithm

D Depth of cut

AI Artificial intelligence

# Background

Studies show that, the corresponding surface roughness, measured in micrometres, of a machined surface is dependent on the parameters selected during the machining operation (i.e. Spindle speed, tool cutting speed, feed rate, tool rate of approach and Depth of cut and tool cutting depth). Machining is loosely defined as material removal by use of a cutting tool. It has also emerged from research that the measured surface roughness can be discerned from the grey level content of the machined surface’s digital image, thus making it possible to determine associated surface roughness from image grey level. Making use of these properties a vision system for predicting surface roughness of a machined surface can be formulated.

# 1.Introduction

In the field of artificial intelligence, a genetic algorithm is an evolutionary computer programing representation of nature’s natural selection. The earth’s natural environment has been continually changing ever since its creation and will continue to change until its end. Biological organisms have had to adapt generation after generation to find an optimal biological physical structure and being to survive this change in environment. This process was first described by Charles Darwin as the survival of the fittest, in his book “On the *Origin of the Species*” (1859), and it obtained the term evolution. Evolution can be characterised as a continual optimization solution to the environmental problems faced by the species in context.

Just like Darwin’s model, a genetic algorithm (GA) creates the genetic composition of each individual solution within a population and breeds them repeatedly until the optimum solution is found to a given problem. These solutions, as with biological organisms in nature, share their genetic information with its offspring when “mating” with other solutions during the creation of a new generation. This sharing over genetic information over a vast number of generations allow the solution to search the solution search space and eventually finds the most fit solution to the problem.

The following report presents an optimization problem, to a genetic algorithm, that involves using Neural Networks to train a system to correlate surface roughness to a machined surface’s digital image. The genetic algorithm must consider certain parameters to predict the corresponding surface roughness of the machined surface.

# 2.Project Proposal and Aim

The project sets out to determine if Neural Networks and Genetic Algorithms can be utilized to train a system to correlate surface roughness to a machined surface’s digital image, and the machining (cutting) operation parameters(**V, F** & **D**), to predict the corresponding surface roughness of a machined surface (mild steel), given its image and machining parameters. Other goals is to determine which of the two implementations are better and whether or a Genetic Algorithm could be created that outputs possible parameters values, given the required roughness.

# 3. Theory

## 3.1 Surface Roughness

The definition of roughness is:

“Relatively finely spaced surface irregularities on surfaces produced by machining and abrasive operations, the irregularities produced by the cutting action of the tool edges and abrasive grains, and by the feed of the machine tool are roughness. Roughness may be considered as superposed on a "wavy" surface.” (SUNDARAM)

The surface roughness is dependent on the following parameters:

* Machine tool type that is used
* Characteristics of material being machined (for example its microstructure and hardness)
* Cutting speed, depth of the cut and feed rate
* Shape and smoothness of the tool used and also the amount of time that it would be used
* Tool setting
* Vibration caused by the machine tool

When machining a part that will have to operate under cyclic loads, it is important to consider the surface roughness. The overall performance of the machined part is dependent on its surface roughness. According to Taraman and Lambert (1974), there are three constants that have to be considered when predicting this. The parameters include the cutting speed, feed rate and the depth of the cut. In order to achieve the desired surface roughness, one has to ensure that the correct combinations of these parameters are selected. A part with a smoother surface roughness is less prone to undergo fatigue failures due to a decreased amount of residual stresses on the part. (Taraman, 1974), (SUNDARAM)

It was only until recent years that a fourth variable was included, namely the time of cut. For the purpose of this assignment, this fourth variable has been neglected due to it being insignificant in comparison to the other three. (SUNDARAM)

### Image Processing:

There are three common non-contact methods that could be used to measure the surface roughness. These include image recognition, X-ray and ultrasonic methods. Image recognition has the best speed and accuracy which is why it has been selected. This method is completely dependent on illumination and thus one has to ensure that the direction or luminance does not change in order to ensure accurate readings. (XIAOJUN TANG)

After an image of the surface is obtained, it has to be processed before one can predict the roughness of the part. There are two main reasons why this has to be done. Firstly, the image will always initially contain noise. The amount of noise increases if the image has been taken in a hostile environment, for example, at high temperatures. Secondly, there might be sections of the curve that has several pixels with high values due to the fact that the agleam curve is not thin enough. (XIAOJUN TANG)

### Cutting Speed:

As the cutting speed is increased, the surface roughness also improves. As with anything else, there is a limit to which one can increase this speed. This limit is due to the constant decrease in the magnitude of the built-up edge. Once this edge becomes very small, there is very little improvement of the surface finish. This maximum cutting speed is usually around about 300fpm. In the low speed region, the surface roughness decreased as the cutting speed was increased. In the intermediate speed region, the surface roughness deteriorated and in the high speed region it decreased steadily until it reached a restraining value. One other aspect to note is that the surface roughness will only improve as long as the tool is not blunt. Thus the surface roughness would end up deteriorating once the tool has begun to wear. (Chandramani, 1964), (SUNDARAM)

### Feed

It is possible to relate the surface roughness to the feed rate analytically by using the following equation:

Where:

* is the maximum peak to valley height (μ-inch)
* F is the feed (inch/rev)
* R is the nose radius of the cutting tool (inch) (SUNDARAM)

Thus from this relationship it is clear to see that if the nose radius is kept constant, the surface roughness will increase with an increase in feed. In many situations it has been shown that the surface roughness deteriorates if the feed rate is increased beyond a limit. There is a certain critical point at which a decrease in the feed rate would not improve the surface roughness. (Galloway, 1945), (SUNDARAM) An explanation on this will follow:

Decreasing the feed leads to less feed marks on the work-piece. Thus this leads to an improved surface roughness. Each tool type has its own optimal feed rate. (SUNDARAM)

### Depth of cut:

There are many contrasting theories about whether the depth of the cut increases or decreases the surface roughness.

### Interaction between the three parameters:

The cutting speed is inversely proportionate to the life of the tool. A high feed leads to increased cutting forces as well as a lower tool life. Thus it is best to ensure that the feed is low and cutting speed is very high. The surface roughness increases as the cutting speed and the depth of the cut increases. The major problem is that when both the cutting speed and depth of the cut is increased, a state of self-excited vibration occurs. This then deteriorates the surface roughness. The cutting force increases as the cutting depth is decreased along with an increase in the cutting speed and a small feed. Higher cutting forces tend to deform the machine tool and workpiece. (SUNDARAM)The effect of these parameters on the surface roughness will be examined and discussed in the latter of this report.

## 3.2 Genetic Algorithm Predicting Roughness

### Fitness Function

The fitness function for this implementation is the SSE of the actual roughness vs the predicted roughness. The lower the value the fitter the individual. Two equations where tested to determine what mathematical function will model the relation between the parameters and the roughness in order to predict the roughness. Equation one is a simple combination of the different parameters where equation two adds exponents:

Where:

* is the surface roughness
* is the cutting speed
* is the feed rate
* is the depth of cut
* is the average grey scale image value
* C is the weight values

### Selection Technique – tournament selection

Selection is one of the main operators in EAs. Its main function is to select the fittest individuals for mating to create the next generation, therefore emphasizing better solutions. There are many selection models, but for this assignment, the Tournament Selection model was chosen and used to enable the selection of the two best individuals from the population.

Tournament selection involves creating a tournament of a desired size from the random selection of individuals within the population. Once the tournament is created, the fittest individual of that tournament is selected as a parent fit for mating.

Elitism was used to ensure that the best individuals make it through to the next generation without being mutated.

### Crossover Technique

The crossover is simply the process of deciding which parents to inherit a given gene from and produces an offspring made up of genes from its parents. Uniform cross over is a procedure that assigns a random cross over probability to each gene carrying element of the offspring

A uniform crossover was used to ensure that the chromosomes are mutated randomly and not at the same place with every generation. This created a higher diversity within the population. A Convex Combination of the two parent genes where used to determine the new gene of the offspring. Since the genes of individuals are continues values, this crossover method creates a good blend of the two selected parents.

The above equation shows how a new gene is created where is a uniformly random value between 0 and 1.

A parameter is also added that determines the probability that this crossover will occurs, otherwise the genes of either parent is chosen at random.

### Mutation Technique

The aim of mutation is to introduce new genetic material into an existing individual, this adds diversity to the genetic characteristics of the population. It supports crossovers by ensuring that the full range of allele is accessible for each gene.

The Gaussian mutation technique was used and the mutation rates were changed as time progresses. Thus, as the population starts tending towards the correct values, the amount by which the mutation varies the chromosome is varied.

A parameter is also added that determines the probability that mutations will occur. Another parameter controls the probability for a specific gene to mutate.

## 3.3 Neural Network Predicting Roughness

### Neural Network Architecture

The **NN’s** is a Feed Forward Neural Network. It consists of a Input Layer of 4 + 10000 input values, the 4 parameter values and the grey scale image values for a 100x100 image. The Hidden Layer consisted of 5 neurons which could be adjusted before each test run and an Output Layer of just one neuron outputting the predicted roughness. The Sigmoid Activation function was the activation function for all neurons.

### Fitness function

Similarly, to the associated **GA** implementation, the **NN** implementation also uses a SSE to determine the fitness of a set of weights whilst training where the error is determined by the difference between the predicted roughness and actual roughness.

### Training

The **NN** was trained using the Gradient decent method since the Sigmoid Activation function was used. An array of values stores learning rates for each associated weight so that they can be changed independently to increase the performance of the **NN**.

The **NN** will be trained stochastically. In order to increase performance and average out the weight updates due to stochastic training, momentum is added to the weight update. This momentum pushes weight updates in the average direction of all updates.

Two data sets were used during the training of the **NN**, a training set and an evaluation set. The training set was used to train the neural network and detect under fitting, the evaluation set was used to detect overfitting.

### Value Scaling

In order to make good use of the Sigmoid Activation function, input values had to be scaled. The min and max values for all parameters were determine from the collected data and used to scale the values to be between -0.5 and 0.5. This method should increase the speed at which the **NN** is able to learn.

## 3.3 Genetic Algorithm Predicting Parameters

The Genetic Algorithm Predicting Parameters was an idea conceived from the curiosity of whether or not the opposite can be achieved from the above two algorithms. Given a roughness value, the algorithm should output the parameters that will result in the associated parameter. Since many different combinations of parameters can produce a specific roughness theoretically, it was decided to use the above trained **NN** as a fitness measure instead of using the experimental data.

The configuration for this **GA** is similar to that of the Genetic Algorithm Predicting Roughness. The only difference is the fitness function, which was mentioned above and the genes within a chromosome now represents the parameter values instead of coefficient values.

The **GA** was implemented producing not only the four parameters, but also 100x100 pixel values that experimented with to determine what images are associated with which roughness values.

# 4.Physical Aspects and Experiment

## Description

The objective of the vision system is to make use of certain parameters (**V**, **F**, **D** and **Ga**) to predict surface roughness, using Neural networks (even other cutting conditions, that is, for which different combinations of operation parameters (**V**, **F**, and **D**) are selected. This will be achieved through applying the following:

* corresponding surface roughness is measured using a reference contact method, and the corresponding image is taken accordingly.
* The measured contact surface roughness is used as a target during the training. This includes making use of **AI** techniques to predict the roughness of a machined surface through creation of a **NN** and **GA** acting as a surface roughness predictor.
* After training, the **NN** and **GA** will make use of four known attributes of a machined surface and then output the associated surface roughness.
* Training data will be obtained experimentally. Grey level is extracted from the surface image and used as an input to the **AI** algorithms. It was also experimented whether the image pixels can further be used as inputs to improve the accuracy of the training process.
* A **GA** will also be implemented to determine each of the four parameters associated with a certain surface roughness using the aforementioned **NN** to determine an individual’s fitness.
* Additional outputs as many as the image pixels themselves can also be employed to regenerate the image and get a feel of the roughness perceived by the vision system.

## Experiment

# 5.Optimisation

## Population Size

## Tournament Size

## Fitness Function

## End Condition

# 6. Results and discussion

//I’ve got a feeling that the NN predicted the results more accurately than the GA, this could be because that the GA was trying to fit a specific equation which we are not sure perfectly models the roughness function. The NN is able to determine this function on its own. –Comment From Francois

## Genetic Algorithm Predicting Parameters

This GA was relatively easy to determine the configuration. A big performance factor was in the initialization of the genes. If the values initialized were too big or too small, then the algorithm struggled to converge to an answer. The experimental data was inspected and the genes were initialized to be within the ranges found within the experimental data. The algorithm was able to determine the parameters for a given roughness within 500 generations.

Unfortunately looking at the images generated by the algorithm it seems to be just noise and not representative of the actual roughness image. This can be attributed to the fact that the NN uses an average of how the image looks and not specific positions within the image.

The GA’s accuracy increase tremendously with the GA getting fitness values of 0, when it did not have to generate an image as well. This can be explained by the fact that the images used to train the NN have a specific form and it is very difficult for the GA to create the small changes required to build a similar looking image, hence the noisy images. The following table list the parameter values found using this algorithm:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Roughness** | **Speed** | **Feed** | **Depth** | **Average Ga** |
| 1.2 | 1464.62 | 4.38 | 1.01 | 139.77 |
| 1.4 | 789.51 | 3.92 | 1.5 | 137.89 |
| 1.6 | 527.77 | 7.99 | 0.5 | 134.20 |
| 1.8 | 357.10 | 6.46 | 1.5 | 146.86 |

The predicted values can be evaluated by looking at the experimental data with similar values. For instance, one of the experimental data points has the parameters Speed = 750, Feed = 4, Depth = 1.5 and Ga = 135,1.37 with a roughness of 1.37. These values are really close to that predicted for a roughness of 1.5. It should be noted that these parameters should be tested in a real world setting in order to determine their validity.

## Genetic Algorithm experiment and Results:

Two different fitness equations will be investigated to propose a solution to determine the surface roughness accurately. For all of the results obtained, the SSE between the predicted roughness and actual roughness (as found from the surface image) was calculated. The SSE values of all of the individuals in the population were compared and only the best fitness value from each generation was plotted in the plots to follow.

Fitness Equation 1:

Where:

* is the roughness
* is the speed
* is the feed
* is the depth of the cut
* is the surface image

When experimenting with a Genetic Algorithm, there are several parameters that could be varied in order to find an optimal solution to the problem. The parameters that will be changed are the:

* crossover rate for values of 0.2, 0.4, 0.6, 0.8 and 0.9
* mutation rate for values of 0.2, 0.4., 0.6, 0.8, 0.9
* mutation magnitude for values of 0.5, 1, 2 and 10
* chance of a gene in a chromosome to mutate 0.2, 0.4, 0.6, 0.8 and 1
* population size for values of 100, 150, 200 and 250
* tournament size for values of 4, 6, 10, 20 and 50

Figure 1: Influence of crossover rate on the fitness minimum over 10 000 generations

The above figure indicates that the minimum fitness value that was achieved was at a crossover rate of 0.8. Initially, the fitness value starts at approximately 150; however it quickly converged to approximately 120. It can be seen that a higher crossover rate leads to a faster convergence. However, the disadvantage of this is that there is a very low diversity in these cases which leads to less exploration and more exploitation. In summary, this leads to an overall worse performance. It can also be seen that increasing the crossover rate, significantly influences the performance of the fitness minimum. In order to incorporate enough exploration, a crossover rate of 0.6 will be used as the optimal crossover rate as this will provide a sufficient amount of convergence after 2500 generations. One does not want a too high crossover rate as this would decrease the diversity and also not a very low crossover rate as this would be tedious.

Figure 2: Influence of different mutation rates on the fitness minimum over 10 000 generations

After varying the mutation rate, it is clear that it has a significant influence on the fitness of the function. It can be seen that the best results were found at a fitness minimum of approximately 120. The mutation rates that converged to this minimum value were 0.8 and 0.9. Between these two values, a mutation rate of 0.9 is preferred. The reason for this is that it takes longer to converge to its minimum fitness value. As a mutation rate of 0.9 takes longer to achieve the minimum fitness value, it has a larger diversity and leads to an increased overall performance of the GA. For this reason a mutation rate of 0.9 was selected as the optimal value. A higher mutation rate has better odds to achieve the optimal solution at all.

Figure 3: Influence of different mutation magnitudes on the fitness minimum over 10 000 generations

From the above figure, it can be seen that the fitness minimum was achieved at approximately 120. It can also be seen that a higher mutation magnitude causes a much slower convergence. Thus it leads to a higher diversity and more exploration. A smaller mutation magnitude starts with a very high fitness minimum, however converges very quickly. The problem with this is that the diversity is very low, which is why it converges so quickly. Increasing the mutation magnitude after approximately 4500 generations, it has no significant influence on the performance of the fitness minimum. Thus if one is considering to run less than 4500 generations, it would be best to use a mutation magnitude of 2 as this converges quite slowly and has a higher diversity than a mutation magnitude of 0.5 and 1. One should not use a high mutation magnitude in this case as it would take too long to converge to the fitness minimum. If one wants to run more than 4500 generations, it is best to use a higher mutation magnitude such as 10. These are directly proportional to one another. A higher mutation magnitude in this case, would increase the amount of exploration whilst still achieving the best possible exploitation.

Figure 4: Influence of tournament size on the fitness minimum over 10 000 generations

Figure 4 illustrates that the tournament size drastically influences the fitness minimum of the population. A lower tournament size, leads to a slower convergence to the minimum fitness value. Thus it creates a larger diversity. From the above figure one can see that it is better to use a lower tournament size in order to achieve the perfect balance between the exploration and exploitation. A tournament size of 6 seems to result in the lowest fitness minimum. It also converges slower and thus has a larger diversity.

Figure 5: Influence of the population size on the fitness minimum over 10 000 generations

From the above figure, one can see that all of the data sets for various population sizes tend to converge at a minimum fitness of approximately 120. This gives an indication that the population size does not have an significant impact on the minimum fitness value.

Figure 6: Influence of the gene mutation rate on the fitness minimum over 10 000 generations

This parameter might be confusing to understand; however it is the probability that a gene in a chromosome will mutate. It can be seen that this probability has no significant influence on the fitness minimum above 4100 generations. For less than 4100 generations, the best gene mutation rate would be 0.2 as it leads to more exploration and a greater diversity.

Fitness Equation 2:

Where:

* is the roughness
* is the speed
* is the feed
* is the depth of the cut
* is the surface image

The parameters that will be changed are the:

* crossover rate for values of 0.2, 0.4, 0.6, 0.8 and 0.9
* mutation rate for values of 0.2, 0.4., 0.6, 0.8, 0.9
* mutation magnitude for values of 0.5, 1, 2 and 10
* chance of a gene in a chromosome to mutate 0.2, 0.4, 0.6, 0.8 and 1
* population size for values of 100, 150, 200 and 250
* tournament size for values of 4, 6, 10, 20 and 50

Figure 7: Influence of crossover rate on the fitness minimum over 10 000 generations

The above figure indicates that a crossover rate of 0.9 achieved the minimum fitness value. The rest of the crossover values reached significantly higher fitness values. The biggest advantage of using a 0.9 crossover rate is that it did not converge immediately. Thus it has a higher diversity and more exploration. It also achieves the best fitness value at approximately 65.

Figure 8: Influence of the mutation magnitude on the fitness minimum over 10 000 generations

From the above figure it can be seen that the best mutation magnitude is 1 as it achieves the lowest fitness minimum at approximately 60. It also converges slowly. A mutation magnitude of 1 does not only have a large diversity and exploration capabilities, but it also reaches the best optimal solution. A mutation rate that is too high (around 10) and too low (around 0.5) leads to a high fitness minimum, which is not desired.

Figure 9: Influence of the mutation rate on the fitness minimum over 10 000 generations

The higher the mutation rate, the better the chances of the GA achieving the optimal solution. For this reason it is clear that the mutation rate has a significant influence on the fitness minimum. It can be seen that the best results were found at a fitness minimum of approximately 75. The mutation rate that converged to this minimum value was 0.9. The mutation rate of 0.9 achieved an overall low fitness minimum whilst converging slowly. Thus it has a balanced amount of exploration and exploitation. The diversity is very high, which is why it had a better probability of achieving the optimal solution. A mutation rate of 0.9 achieved the best performance.

Figure 10: Influence of gene mutation rate on the fitness minimum over 10 000 generations

A lower the gene mutation probability leads to a lower fitness minimum of the GA. All of these gene mutation probabilities, except 1, resulted in a high exploration (high diversity). The overall best gene mutation rate was at 0.2 due to it achieving the lowest fitness minimum. Not only does it have a very high diversity, but it also achieves a more optimal solution.

Figure 11: Influence of the population size on the fitness minimum over 10 000 generations

For this fitness function, the population size had a drastic impact on the fitness minimum of the GA. A population size of 100 achieved the lowest fitness minimum and overall best performance due to the fact that it converged slowly. A population size of 100 is thus sufficient in order to have the correct amount of diversity in the population which is required to achieve the optimal solution. Increasing the population size above 100, would only worsen the fitness minimum.

Figure 12: Influence of the tournament size on the fitness minimum over 10 000 generations

From figure 12, it is seen that a tournament size of 10 was the most sufficient. At a tournament size of 10, the best fitness minimum was achieved as well as a large diversity. Due to the large diversity, the solution did not converge to optimal value quickly. Thus its exploration is high. Tournament sizes lower or higher than 10, only results in worse fitness minimums.

## Comparison between these two fitness functions:

The best results that were determined experimentally provides the following combination of optimal parameters:

|  |  |  |
| --- | --- | --- |
|  | Fitness Equation 1 | Fitness Equation 2 |
| Crossover Rate | 0.6 | 0.9 |
| Mutation rate | 0.9 | 0.9 |
| Mutation Magnitude | 10 | 1 |
| Gene Mutation Probability | 0.2 | 0.2 |
| Population Size | 100 | 100 |
| Tournament Size | 6 | 10 |

Table 1: Comparison between equation 1 and equation 2

Figure 13: Comparison of the minimum fitness values of the two different fitness equations

From figure 13, it is clear that the fitness equation 1 provides more accurate results. Equation 1 will then thus be used to predict the roughness values.

## Predicting Roughness with the selected equation:

|  |  |  |
| --- | --- | --- |
| Actual Roughness | Predicted Roughness | Difference (Error) |
| 7.29 | 5.514011 | 1.775989 |
| 5.54 | 5.477237 | 0.062763 |
| 6.19 | 5.510467 | 0.679533 |
| 7.04 | 5.297885 | 1.742115 |
| 5.02 | 5.199508 | -0.17951 |
| 6.45 | 5.368951 | 1.081049 |
| 6.38 | 5.138984 | 1.241016 |
| 5.92 | 4.993664 | 0.926336 |
| 6.71 | 5.13215 | 1.57785 |
| 2.1 | 2.285936 | -0.18594 |
| 1.14 | 2.229242 | -1.08924 |
| 1.19 | 2.694406 | -1.50441 |
| 2.19 | 2.206255 | -0.01625 |
| 1.52 | 2.292051 | -0.77205 |
| 1.27 | 2.507581 | -1.23758 |
| 0.77 | 2.285275 | -1.51528 |
| 1.38 | 2.277804 | -0.8978 |
| 1 | 2.416787 | -1.41679 |
| 1.15 | 1.993453 | -0.84345 |
| 1.74 | 2.032015 | -0.29202 |
| 1.48 | 2.186138 | -0.70614 |
| 1.14 | 1.911978 | -0.77198 |
| 1.37 | 1.87287 | -0.50287 |
| 1.37 | 2.011291 | -0.64129 |
| 1.1 | 1.724768 | -0.62477 |
| 1.29 | 1.903006 | -0.61301 |
| 1.12 | 2.102371 | -0.98237 |
| 1.47 | 1.845319 | -0.37532 |
| 1.45 | 1.837906 | -0.38791 |
| 1.64 | 2.007051 | -0.36705 |
| 1.49 | 1.728556 | -0.23856 |
| 1.24 | 1.675246 | -0.43525 |
| 1.49 | 1.736123 | -0.24612 |
| 1.42 | 1.490565 | -0.07056 |
| 1.39 | 1.576037 | -0.18604 |
| 1.55 | 1.714715 | -0.16472 |
| 1.65 | 1.280499 | 0.369501 |
| 1.66 | 1.615402 | 0.044598 |
| 1.79 | 1.522787 | 0.267213 |
| 1.61 | 1.298126 | 0.311874 |
| 1.51 | 1.461164 | 0.048836 |
| 1.63 | 1.569324 | 0.060676 |
| 1.52 | 1.237343 | 0.282657 |
| 1.52 | 1.26072 | 0.25928 |
| 1.8 | 1.477261 | 0.322739 |
| 1.62 | 1.034008 | 0.585992 |
| 1.57 | 1.309831 | 0.260169 |
| 1.93 | 1.463954 | 0.466046 |
| 1.61 | 1.121451 | 0.488549 |
| 1.41 | 1.144696 | 0.265304 |
| 1.85 | 1.360727 | 0.489273 |
| 1.62 | 1.050593 | 0.569407 |
| 1.94 | 1.058202 | 0.881798 |
| 1.73 | 1.305189 | 0.424811 |
| 1.65 | 0.934067 | 0.715933 |
| 1.65 | 1.051142 | 0.598858 |
| 2.29 | 1.174312 | 1.115688 |
| 1.52 | 0.944972 | 0.575028 |
| 1.52 | 0.777609 | 0.742391 |
| 2.25 | 1.153039 | 1.096961 |
| 1.61 | 0.955761 | 0.654239 |
| 1.35 | 0.678756 | 0.671244 |
| 0.85 | 3.007637 | -2.15764 |
| 1.08 | 3.110465 | -2.03046 |
| 1.14 | 3.357452 | -2.21745 |
| 1.3 | 0.991356 | 0.308644 |
| 0.95 | 3.086165 | -2.13617 |
| 0.86 | 0.970465 | -0.11046 |
| 1.25 | 1.064146 | 0.185854 |
| 0.88 | 0.741413 | 0.138587 |
| 0.85 | 0.855008 | -0.00501 |
| 1.01 | 0.918466 | 0.091534 |
| 1.01 | 0.534141 | 0.475859 |
| 0.96 | 0.697823 | 0.262177 |
| 1.24 | 0.927646 | 0.312354 |
| 0.93 | 0.660058 | 0.269942 |
| 1.03 | 0.75763 | 0.27237 |
| 1.24 | 0.80659 | 0.43341 |
| 1.05 | 0.691104 | 0.358896 |
| 1.15 | 0.63897 | 0.51103 |
| 1.25 | 0.686266 | 0.563734 |

Figure 14: Roughness Calculations using the desired fitness equation

From the above table it is clear that the error has decreased as the generations progressed as it started off with a difference as high as 1.775989 and eventually decreased to 0.563734. There are different factors that could have contributed towards these errors. These factors include the wearing and tearing of the tool tip. This will be further elaborated on in the conclusion.

# Comparison of NN parameters and their corresponding results

The accuracy of Artificial Neural Networks is often determined by a variety of factors, including: number of neurons in the input and hidden layers, learning rate, momentum factor, number of records in the training dataset, stopping conditions and the various statistical properties of the training records. With the goal of constructing a NN configuration which produces the most accurate results, the following factors were selected to be the varying parameters, since they affected accuracy the most: number of neurons in the hidden layer, learning rate and momentum rate.

As a baseline, a setting of 5 hidden neurons, learning rate of 0.1 and momentum factor of 0.0001 was used. This setting was found to be suitable enough until further investigation into better parameters could be performed. This section documents that phase of this project and concludes with an optimal configuration, based on the findings on several test results.

Firstly, the impact of the momentum factor on the accuracy and training speed of the NN was investigated. The graph below illustrates the performance of the five momentum factor values investigated:

Figure : Test SSE vs. Varying Momentum Rates

Surprisingly, the data obtained indicates that changes in momentum make little to no difference in the accuracy of the NN. In these evaluations, the number of hidden neurons and the learning rate were kept constant so that momentum was the only independent variable. These findings suggest that the input data records do not conflict with each other, that is, that they do not require wildly different NN weight configurations to provide accurate predictions. Since M = 1E-05 made a sudden dip at the end, and no others did, it can be assumed this happened by chance.

The second set of tests focused on determining the optimum learning rate for the NN. This time, the number of hidden neurons and the momentum factor were kept constant, and the NN was trained and tested on a set of four different learning rates.

Figure :SSE vs. different learning rates

Here a similar result is apparent, whereby changes to the learning rate makes little difference on the accuracy or speed at which the NN converges to a specific weight configuration. This indicates that the problem the NN is attempting to solve is likely linear in nature.

The third set of tests looked at how different numbers of neurons within the hidden layer of the NN would affect the NN’s performance. Here the learning and momentum factors were kept constant.

Figure : SSE vs. different numbr of hidden layers

Predictably, when the NN was configured to have more neurons within its hidden layer, it was able to reach a decent accuracy (SSE of ~0.45) in fewer iterations (3 & 5 neurons plateaued at 70-80 iterations whereas 10 & 15 managed it in 15-30 iterations). What was really surprising was that after 50-100 iterations, the NNs using 10 and 15 hidden layer neurons began to recommence improving their accuracy. To determine whether or not this was due to the NN memorizing the training patterns, the NN was set to run a further 600 iterations and the SEE on the test and evaluation patterns was compared. As you can see in the graph below, overfitting only started occurring around iteration 380, meaning that all the different NN configurations likely got caught in a local minimum at the beginning, but managed to move to far better minimum after a few hundred iterations. Further testing showed that given enough iterations, neural networks with 3 and 5 hidden neurons would also begin to re-converge on a new minimum.

Figure : SSE vs. various learning rates

# 7. Conclusion

# 8.Reference

# Bibliography

Chandramani, K. L. (1964). Investigations on the nature \ of surface finish and its variation with curting speed. *Journal of Engineering for Industry, 86-88*, 134-140.

Galloway, D. F. (1945). Recent Research in Metal Machining. *Proceedings of the Institute of Mechanical Engineers, 153*, 113-127.

Jithin Babu. R, A. R. (2014). *CORRELATION AMONG THE CUTTING PARAMETERS, SURFACE ROUGHNESS AND CUTTING FORCES IN TURNING PROCESS BY EXPERIMENTAL STUDIES.* Guwahati: All India Manufacturing Technology, Design and Research Conference.

John W.Jewett Jr., R. A. (2010). *Physics for Scientists and Engineers with Modern Physics* (8th ed.). Brooks/Cole.

SUNDARAM, R. M. (n.d.). *A STATISTICAL ANALYSIS CP SURFACE FINISH IN FINE TURNING OF STEEL .* Retrieved 11 30, 2016, from https://ttu-ir.tdl.org/ttu-ir/bitstream/handle/2346/17190/31295000656479.pdf?sequence=1

Taraman, K. ,. (1974). A surface roughness model for a turning operatio. *International Journal of Production Research, 12*(6), 694-703.

XIAOJUN TANG, X. L. (n.d.). *Surface Roughness Measurement Based on Image Processing and Image Recognition.* Retrieved 12 12, 2016, from http://www.wseas.us/e-library/conferences/2009/book3/Papers/CSMS13.pdf

# 9. Appendix

## Code

Code:

***Code of a GA predicting the roughness is included below:***

* ***Main Program***

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.IO;

namespace GAPredictingRougthness

{

class Program

{

static GreyImageList greyImageList;

static GeneticAlgo GA;

static int MaxNumberOfGenerations = 10000; //Parameter used to determine when the algorithm should stop.

static void Main(string[] args)

{

greyImageList = new GreyImageList();

GA = new GeneticAlgo();

RunGA();

Console.ReadLine();

}

static void RunGA()

{

int counter = 1;

double prevError = Double.MaxValue;

int changeCounter = 0;

FileStream actualRough = new FileStream("actualRough.txt", FileMode.Create, FileAccess.Write);

StreamWriter writer0 = new StreamWriter(actualRough);

FileStream fittestInd = new FileStream("fittestInd.txt", FileMode.Create, FileAccess.Write);

StreamWriter writer1 = new StreamWriter(fittestInd);

FileStream generation = new FileStream("generation.txt", FileMode.Create, FileAccess.Write);

StreamWriter writer2 = new StreamWriter(generation);

FileStream predictedRough = new FileStream("predictedRough.txt", FileMode.Create, FileAccess.Write);

StreamWriter writer3 = new StreamWriter(predictedRough);

FileStream diff = new FileStream("diff.txt", FileMode.Create, FileAccess.Write);

StreamWriter writer4 = new StreamWriter(diff);

do

{

foreach (RougthnessChromosone curModel in GA.population)

{

double totalFitness = 0;

foreach (GreyImage greyImage in greyImageList.GetTestGreyImages())

{

double predictedRougthness = curModel.CalculateRougthness(greyImage);

totalFitness += Math.Pow(greyImage.surface.getRa() - predictedRougthness,2);

}

curModel.SetFitness(totalFitness);

}

GA.NextGeneration();

if (counter % 100 == 0)

{

Console.WriteLine(counter + " | Current Fittest: " + GA.population[0].GetFitness());

writer1.WriteLine(GA.population[0].GetFitness());

writer2.WriteLine(counter);

}

counter++;

} while (counter < MaxNumberOfGenerations);

foreach (GreyImage greyImage in greyImageList.GetTestGreyImages())

{

double predictedRougthness = GA.population[0].CalculateRougthness(greyImage);

Console.WriteLine("Actual ra: {0} | Predicted ra: {1} | Difference: {2}", greyImage.surface.getRa() , predictedRougthness, greyImage.surface.getRa() - predictedRougthness);

writer0.WriteLine(greyImage.surface.getRa());

writer3.WriteLine(predictedRougthness);

writer4.WriteLine(greyImage.surface.getRa() - predictedRougthness);

}

writer0.Close();

writer1.Close();

writer2.Close();

writer3.Close();

writer4.Close();

}

}

}

* ***Genetic Algorithm:***

namespace GAPredictingRougthness

{

class GeneticAlgo

{

public List<RougthnessChromosone> population;

public int populationSize = 100; //Parameter Population Size

public static int TOURNAMENT\_SIZE = 10; //Parameter Tournament Size

public static Random random = new Random();

public GeneticAlgo()

{

population = new List<RougthnessChromosone>();

for (int x = 0; x < populationSize; x++)

{

population.Add(new RougthnessChromosone());

}

}

public void NextGeneration()

{

population.Sort(delegate(RougthnessChromosone o1, RougthnessChromosone o2)

{

if (o1.GetFitness() < o2.GetFitness())

{

return -1;

}

else if (o1.GetFitness() > o2.GetFitness())

{

return 1;

}

return 0;

});

List<RougthnessChromosone> newPopulation = new List<RougthnessChromosone>();

newPopulation.Add(population[0]);

while (newPopulation.Count < populationSize)

{

//newPopulation.add(TournamentSelection());

newPopulation.Add(TournamentSelection2());

}

population = newPopulation;

}

public RougthnessChromosone TournamentSelection2()

{

List<RougthnessChromosone> tournament = new List<RougthnessChromosone>();

for (int x = 0; x < TOURNAMENT\_SIZE; x++)

{

tournament.Add(population[random.Next(populationSize)]);

}

tournament.Sort(delegate(RougthnessChromosone o1, RougthnessChromosone o2)

{

if (o1.GetFitness() < o2.GetFitness())

{

return -1;

}

else if (o1.GetFitness() > o2.GetFitness())

{

return 1;

}

return 0;

});

RougthnessChromosone parent1 = tournament[0];

tournament = new List<RougthnessChromosone>();

for (int x = 0; x < TOURNAMENT\_SIZE; x++)

{

tournament.Add(population[random.Next(populationSize)]);

}

tournament.Sort(delegate(RougthnessChromosone o1, RougthnessChromosone o2)

{

if (o1.GetFitness() < o2.GetFitness())

{

return -1;

}

else if (o1.GetFitness() > o2.GetFitness())

{

return 1;

}

return 0;

});

RougthnessChromosone parent2 = tournament[0];

return new RougthnessChromosone((RougthnessChromosone)parent1, (RougthnessChromosone)parent2);

}

}

* ***Chromosome of the Roughness***

namespace GAPredictingRougthness

{

class RougthnessChromosone

{

List<Double> coeffs;

private double NumberOfGenes = 9;

public double fitness = double.MaxValue;

public static double mutationMagnitude =1; //Parameter Maximum value that a mutation will decrease or increase

public static double mutationChancePerValue = 0.2; //Parameter Chance that a gene in a chromosome will mutate [0;1].

public static double EquationNumber = 1; //Parameter 0 uses the equation "a + b\*speed + c\*feed + d\*depth + e\*Ga" and 1 uses the equation "a + b\*speed^f + c\*feed^g + d\*depth^h + e\*Ga^i"

public double mutateChance = 0.9; //Parameter Chance that a chromosome will be mutated [0;1].

public double crossoverRate = 0.9; //Parameter Chance that a chromosome will be crossovered [0;1].

public RougthnessChromosone()

{

coeffs = new List<Double>();

for(int x = 0; x < 5; x++)

{

coeffs.Add(initializeValue());

}

for (int x = 5; x < NumberOfGenes; x++)

{

coeffs.Add(1 + GeneticAlgo.random.NextDouble() \* 0.2 - 0.1);

}

}

public RougthnessChromosone(List<Double> coefficients)

{

coeffs = new List<Double>();

foreach (Double curD in coefficients)

{

coeffs.Add(curD);

}

}

public double initializeValue()

{

return GeneticAlgo.random.NextDouble() \*2 - 1 ;

}

public RougthnessChromosone(RougthnessChromosone parent1, RougthnessChromosone parent2)

{

coeffs = new List<Double>();

double r = GeneticAlgo.random.NextDouble();

if (r < crossoverRate)

{

for (int x = 0; x < NumberOfGenes; x++)

{

r = GeneticAlgo.random.NextDouble();

// if(r < 0.5){

// coeffs.add(parent1.coeffs.get(x));

// }else{

// coeffs.add(parent2.coeffs.get(x));

// }

coeffs.Add(r \* parent1.coeffs[x] + (1 - r) \* parent2.coeffs[x]);

}

}

else {

for (int x = 0; x < NumberOfGenes; x++)

{

coeffs.Add(parent1.coeffs[x]);

}

}

r = GeneticAlgo.random.NextDouble();

if (r < mutateChance)

{

MUTATE();

}

}

public void MUTATE()

{

for (int x = 0; x < NumberOfGenes; x++)

{

double r = GeneticAlgo.random.NextDouble();

if (r < mutationChancePerValue)

{

double curD = coeffs[x];

curD = (curD - (GeneticAlgo.random.NextDouble() \* mutationMagnitude - (mutationMagnitude / 2.0)));

coeffs[x] = curD;

}

}

}

public double CalculateRougthness(GreyImage gI)

{

Surface curSurface = gI.surface;

if (EquationNumber == 1)

{

return coeffs[0] + coeffs[1] \* Math.Pow(curSurface.getSpeed(), coeffs[5]) + coeffs[2] \* Math.Pow(curSurface.getFeed(), coeffs[6]) + coeffs[3] \* Math.Pow(curSurface.getDepth(), coeffs[7]) + coeffs[4] \* Math.Pow(curSurface.getGa(), coeffs[8]);

}

else

{

return coeffs[0] + (coeffs[1] \* curSurface.getSpeed()) + (coeffs[2] \* curSurface.getFeed()) + (coeffs[3] \* curSurface.getDepth()) + (coeffs[4] \* curSurface.getGa());

}

}

public RougthnessChromosone Clone()

{

return new RougthnessChromosone(this.coeffs);

}

public List<Double> GetCoefs()

{

return coeffs;

}

public double GetFitness()

{

return fitness;

}

public void SetFitness(double fitness)

{

this.fitness = fitness;

}

public void DrawImage()

{

byte[] pixels = new byte[10000];

for(int x = 0; x < 10000; x++)

{

pixels[x] = Convert.ToByte(Math.Ceiling((coeffs[x + 4])\*255));

}

GreyImageList.DrawImage(pixels, @"C:\Users\Francois\Desktop\TestImage.bmp");

}

}

* Grey Image Class:

namespace GAPredictingRougthness

{

class GreyImage

{

private int width; // Width of image

private int height; // Height of image

private byte Ga; // Mean grey level content of image

private string pathName; // File path of image

public List<Double> scaledPixelArray;

public Surface surface;

public GreyImage(BitmapSource bitmap, byte ga, string nem)

{

if (bitmap.Format != PixelFormats.Gray8) // Convert image format to greyscale

{

bitmap = new FormatConvertedBitmap(bitmap, PixelFormats.Gray8, null, 0);

}

width = bitmap.PixelWidth; // Width of image

height = bitmap.PixelHeight; // height of image

pathName = nem; // File path of image

Ga = ga; // mean grey level content of image

}

public GreyImage(BitmapSource bitmap, byte ga, string nem,Surface surface)

{

if (bitmap.Format != PixelFormats.Gray8) // Convert image format to greyscale

{

bitmap = new FormatConvertedBitmap(bitmap, PixelFormats.Gray8, null, 0);

}

byte[] bytePixelArray = new byte[bitmap.PixelHeight \* bitmap.PixelWidth];

bitmap.CopyPixels(bytePixelArray, bitmap.PixelWidth, 0);

scaledPixelArray = new List<double>();

foreach (byte curByte in bytePixelArray)

{

scaledPixelArray.Add(System.Convert.ToDouble(curByte)/255.0);

}

width = bitmap.PixelWidth; // Width of image

height = bitmap.PixelHeight; // height of image

pathName = nem; // File path of image

Ga = ga; // mean grey level content of image

this.surface = surface;

}

public int getWidth()

{

return width;

}

public int getHeight()

{

return height;

}

public byte getGa()

{

return Ga;

}

}

* Grey Image List Class:

namespace GAPredictingRougthness

{

class GreyImageList

{

private List<GreyImage> roughList; // roughness data image list

private List<GreyImage> evalList;

private BitmapSource bitmapIn;

private byte[] pixelArray;

private byte Ga; // Grey level content of a greyscale image

static double maxSpeed = double.MinValue;

static double maxFeed = double.MinValue;

static double maxDepth = double.MinValue;

static double maxGa = double.MinValue;

static double maxRa = double.MinValue;

static double minSpeed = double.MaxValue;

static double minFeed = double.MaxValue;

static double minDepth = double.MaxValue;

static double minGa = double.MaxValue;

static double minRa = double.MaxValue;

public GreyImageList() // Grey images list class

{

roughList = new List<GreyImage>();

evalList = new List<GreyImage>();

Ga = 0;

pixelArray = null;

LoadData(@"..\..\..\..\Images\Roughness Data\SmallerImages", "RoughnessDataSmall.txt", roughList, false);

LoadData(@"..\..\..\..\Images\Evaluation Data\SmallerImages", "EvaluationSmall.txt", evalList, false);

// File directory of the images

//Display(image, bitmapIn, roughList);

}

private void LoadData(string filesPath, string fileName, List<GreyImage> List, bool writeToFile)

// Load "image files" to List from filesPath folder and write details to a text file

{

string[] imageFiles = Directory.GetFiles(filesPath, "\*.jpg");

if (imageFiles == null)

{

return;

}

if (writeToFile)

{

StreamWriter SW = new StreamWriter(fileName);

SW.WriteLine("Speed (V), Feed (F), Depth (D), Grey Level (Ga), Roughness (Ra)");

for (int x = 0; x < imageFiles.Length; x++)

{

bitmapIn = new BitmapImage(new Uri(Path.GetFullPath(imageFiles[x])));

if (bitmapIn.Format != PixelFormats.Gray8) // Convert image format to greyscale

{

bitmapIn = new FormatConvertedBitmap(bitmapIn, PixelFormats.Gray8, null, 0);

}

pixelArray = new byte[bitmapIn.PixelHeight \* bitmapIn.PixelWidth];

getPixels(bitmapIn); // Load image pixel data into an array

Ga = meanGrey(); // compute mean grey level content of an image

imageFiles[x] = FileName(imageFiles[x], filesPath + @"\"); // Modify file name

GreyImage greyCopy = new GreyImage(bitmapIn, Ga, imageFiles[x]);

List.Add(greyCopy); // Add grey image object to grey image ArrayList

writeGaData(SW, imageFiles[x]); // Write Image pixel data in text file

}

SW.Close();

}

else

{

for (int x = 0; x < imageFiles.Length; x++)

{

bitmapIn = new BitmapImage(new Uri(Path.GetFullPath(imageFiles[x])));

if (bitmapIn.Format != PixelFormats.Gray8) // Convert image format to greyscale

{

bitmapIn = new FormatConvertedBitmap(bitmapIn, PixelFormats.Gray8, null, 0);

}

pixelArray = new byte[bitmapIn.PixelHeight \* bitmapIn.PixelWidth];

getPixels(bitmapIn); // Load image pixel data into an array

Ga = meanGrey(); // compute mean grey level content of an image

imageFiles[x] = FileName(imageFiles[x], filesPath + @"\"); // Modify file name

GreyImage greyCopy = new GreyImage(bitmapIn, Ga, imageFiles[x], GetSurfaceFromFileName(imageFiles[x]));

List.Add(greyCopy); // Add grey image object to grey image ArrayList

}

}

}

private Surface GetSurfaceFromFileName(string fileName)

{

string[] VFDRa = fileName.Split('\_');

Surface newSurface = new Surface(double.Parse(VFDRa[0]), double.Parse(VFDRa[1]), double.Parse(VFDRa[2]), System.Convert.ToDouble(Ga), double.Parse(VFDRa[3]));

if (newSurface.getSpeed() > maxSpeed)

{

maxSpeed = newSurface.getSpeed();

}

if (newSurface.getSpeed() < minSpeed)

{

minSpeed = newSurface.getSpeed();

}

if (newSurface.getFeed() > maxFeed)

{

maxFeed = newSurface.getFeed();

}

if (newSurface.getFeed() < minFeed)

{

minFeed = newSurface.getFeed();

}

if (newSurface.getDepth() > maxDepth)

{

maxDepth = newSurface.getDepth();

}

if (newSurface.getDepth() < minDepth)

{

minDepth = newSurface.getDepth();

}

if (newSurface.getGa() > maxGa)

{

maxGa = newSurface.getGa();

}

if (newSurface.getGa() < minGa)

{

minGa = newSurface.getGa();

}

if (newSurface.getRa() > maxRa)

{

maxRa = newSurface.getRa();

}

if (newSurface.getRa() < minRa)

{

minRa = newSurface.getRa();

}

return newSurface;

}

private byte meanGrey() // Mean grey level content of the image (Ga)

{

long ga = 0;

int dim = pixelArray.Length;

for (int x = 0; x < dim; x++)

{

ga = ga + pixelArray[x];

}

return Convert.ToByte(ga / dim);

}

public List<GreyImage> GetTestGreyImages()

{

return roughList;

}

public List<GreyImage> GetEvalGreyImages()

{

return evalList;

}

private void writeGaData(StreamWriter SW, string fileName)

{

string[] VFDRa = fileName.Split('\_');

SW.WriteLine("{0},{1},{2},{3},{4}", VFDRa[0], VFDRa[1], VFDRa[2], Ga, VFDRa[3]);

//writeArray(pixelArray, SW);

}

private void writeArray(byte[] array, StreamWriter SW)

{

for (int x = 0; x < array.Length; x++)

{

SW.Write(array[x] + " ");

}

SW.WriteLine("\n\n\n\n\n\n\n");

}

private void getPixels(BitmapSource bitmap)

{

bitmap.CopyPixels(pixelArray, bitmap.PixelWidth, 0); //Load pixel data into an array

}

private string FileName(string path, string mask)

{

string fileName = null;

fileName = path.Replace(mask, "");

fileName = fileName.Replace(".JPG", "");

return fileName;

}

private WriteableBitmap Write(BitmapSource bitmap)

{

WriteableBitmap bitmapInGrey;

bitmap.CopyPixels(pixelArray, bitmap.PixelWidth, 0); //Load pixel data into an array

bitmapInGrey = new WriteableBitmap(bitmap.PixelWidth, bitmap.PixelHeight, 96, 96, PixelFormats.Gray8, null);

bitmapInGrey.WritePixels(new Int32Rect(0, 0, bitmap.PixelWidth, bitmap.PixelHeight), pixelArray, bitmap.PixelWidth, 0);

return bitmapInGrey;

}

private void Display(Image image, BitmapSource temp, ArrayList List)

{

try

{

if (List == null)

{

return;

}

WriteableBitmap bitmap = Write(temp);

image.Source = bitmap;

}

catch (Exception ex)

{

Console.WriteLine("An error has occured | " + ex.Message);

}

}

public static double scaleValue(double actualValue, double actualMin, double actualMax, double scaleMin, double scaleMax)

{

return (actualValue - actualMin) / (actualMax - actualMin) \* (scaleMax - scaleMin) + scaleMin;

}

public static double descaleValue(double scaleValue, double actualMin, double actualMax, double scaleMin, double scaleMax)

{

return ((scaleValue - scaleMin) \* (actualMax - actualMin)) / (scaleMax - scaleMin) + actualMin;

}

public static double scaleSpeed(double speed)

{

return scaleValue(speed, minSpeed, maxSpeed, -0.5, 0.5);

}

public static double scaleFeed(double feed)

{

return scaleValue(feed, minFeed, maxFeed, -0.5, 0.5);

}

public static double scaleDepth(double depth)

{

return scaleValue(depth, minDepth, maxDepth, -0.5, 0.5);

}

public static double scaleGa(double Ga)

{

return scaleValue(Ga, minGa, maxGa, -0.5, 0.5);

}

public static double descaleSpeed(double speed)

{

return descaleValue(speed, minSpeed, maxSpeed, -0.5, 0.5);

}

public static double descaleFeed(double feed)

{

return descaleValue(feed, minFeed, maxFeed, -0.5, 0.5);

}

public static double descaleDepth(double depth)

{

return descaleValue(depth, minDepth, maxDepth, -0.5, 0.5);

}

public static double descaleGa(double Ga)

{

return descaleValue(Ga, minGa, maxGa, -0.5, 0.5);

}

public static double scaleRa(double Ra)

{

return scaleValue(Ra, minRa, maxRa, 0, 1);

}

public static double descaleRa(double scaleRa)

{

return descaleValue(scaleRa, minRa, maxRa, 0, 1);

}

public static double initeSpeed()

{

return descaleValue(GeneticAlgo.random.NextDouble() - 0.5, minSpeed, maxSpeed, -0.5, 0.5);

}

public static double initFeed()

{

return descaleValue(GeneticAlgo.random.NextDouble() - 0.5, minFeed, maxFeed, -0.5, 0.5);

}

public static double initDepth()

{

return descaleValue(GeneticAlgo.random.NextDouble() - 0.5, minDepth, maxDepth, -0.5, 0.5);

}

public static double initGa()

{

return descaleValue(GeneticAlgo.random.NextDouble() - 0.5, minGa, maxGa, -0.5, 0.5);

}

public static void DrawImage(byte[] pixelArray, String filename)

{

System.Drawing.Image xy = byteArrayToImage(pixelArray);

xy.Save(filename, System.Drawing.Imaging.ImageFormat.Bmp);

}

public static System.Drawing.Image byteArrayToImage(byte[] byteArrayIn)

{

int size = (int)Math.Sqrt(byteArrayIn.Length); // Some bytes will not be used as we round down here

System.Drawing.Bitmap bitmap = new System.Drawing.Bitmap(size, size, System.Drawing.Imaging.PixelFormat.Format8bppIndexed);

System.Drawing.Imaging.BitmapData bitmapData = bitmap.LockBits(new System.Drawing.Rectangle(0, 0, bitmap.Width, bitmap.Height), System.Drawing.Imaging.ImageLockMode.WriteOnly, bitmap.PixelFormat);

try

{

// Copy byteArrayIn to bitmapData row by row (to account for the case

// where bitmapData.Stride != bitmap.Width)

for (int rowIndex = 0; rowIndex < bitmapData.Height; ++rowIndex)

Marshal.Copy(byteArrayIn, rowIndex \* bitmap.Width, bitmapData.Scan0 + rowIndex \* bitmapData.Stride, bitmap.Width);

}

finally

{

bitmap.UnlockBits(bitmapData);

}

var newPalette = bitmap.Palette;

for (int index = 0; index < bitmap.Palette.Entries.Length; ++index)

{

var entry = bitmap.Palette.Entries[index];

var gray = (int)(0.30 \* entry.R + 0.59 \* entry.G + 0.11 \* entry.B);

newPalette.Entries[index] = System.Drawing.Color.FromArgb(gray, gray, gray);

}

bitmap.Palette = newPalette;

return bitmap;

}

}

* Surface Class

namespace GAPredictingRougthness

{

class Surface

{

private double speed;

private double feed;

private double depth;

private double Ga;

private double Ra;

public List<Double> inputList;

public Surface(double spd, double fd, double dpt, double ga, double ra)

{

speed = spd;

feed = fd;

depth = dpt;

Ga = ga;

Ra = ra;

inputList = GetInputs();

}

private List<Double> GetInputs()

{

List<Double> inputList = new List<double>();

inputList.Add(speed);

inputList.Add(feed);

inputList.Add(depth);

inputList.Add(Ga);

return inputList;

}

public List<Double> GetScaleInputs()

{

List<Double> inputList = new List<double>();

inputList.Add(GreyImageList.scaleSpeed(speed));

inputList.Add(GreyImageList.scaleFeed(feed));

inputList.Add(GreyImageList.scaleDepth(depth));

inputList.Add(GreyImageList.scaleGa(Ga));

return inputList;

}

public double getSpeed()

{

return speed;

}

public double getFeed()

{

return feed;

}

public double getDepth()

{

return depth;

}

public double getGa()

{

return Ga;

}

public double getRa()

{

return Ra;

}

public double getScaledRa()

{

return GreyImageList.scaleRa(Ra);

}

public void Display(StreamWriter SW)

{

SW.Write("{0} {1} {2} {3} | {4}", speed, feed, depth, Ga, Ra);

}

}

* Surface List Class

namespace GAPredictingRougthness

{

class SurfaceList

{

private List<Surface> optiData;

private List<Surface> evalData;

static double maxSpeed = double.MinValue;

static double maxFeed = double.MinValue;

static double maxDepth = double.MinValue;

static double maxGa = double.MinValue;

static double maxRa = double.MinValue;

static double minSpeed = double.MaxValue;

static double minFeed = double.MaxValue;

static double minDepth = double.MaxValue;

static double minGa = double.MaxValue;

static double minRa = double.MaxValue;

public SurfaceList()

{

optiData = new List<Surface>();

evalData = new List<Surface>();

ReadData("RoughnessData.txt", "Evaluation.txt");

getMinMaxValues();

}

private void getMinMaxValues()

{

foreach(Surface curSurface in optiData)

{

if(curSurface.getSpeed() > maxSpeed)

{

maxSpeed = curSurface.getSpeed();

}

if (curSurface.getSpeed() < minSpeed)

{

minSpeed = curSurface.getSpeed();

}

if (curSurface.getFeed() > maxFeed)

{

maxFeed = curSurface.getFeed();

}

if (curSurface.getFeed() < minFeed)

{

minFeed = curSurface.getFeed();

}

if (curSurface.getDepth() > maxDepth)

{

maxDepth = curSurface.getDepth();

}

if (curSurface.getDepth() < minDepth)

{

minDepth = curSurface.getDepth();

}

if (curSurface.getGa() > maxGa)

{

maxGa = curSurface.getGa();

}

if (curSurface.getGa() < minGa)

{

minGa = curSurface.getGa();

}

if (curSurface.getRa() > maxRa)

{

maxRa = curSurface.getRa();

}

if (curSurface.getRa() < minRa)

{

minRa = curSurface.getRa();

}

}

}

public void ReadData(string fileName1, string fileName2) //Reading optimisation and evaluation Data from a file

{

StreamReader SR1 = new StreamReader(fileName1);

string[] dataTray;

SR1.ReadLine(); //Skip Title Row

while (!SR1.EndOfStream)

{

dataTray = SR1.ReadLine().Split(',');

double spd = Double.Parse(dataTray[0]);

double fd = Double.Parse(dataTray[1]);

double dpt = Double.Parse(dataTray[2]);

double ga = Double.Parse(dataTray[3]);

double ra = Double.Parse(dataTray[4]);

Surface temp1 = new Surface(spd, fd, dpt, ga, ra);

optiData.Add(temp1);

}

SR1.Close();

StreamReader SR2 = new StreamReader(fileName2);

SR2.ReadLine(); //Skip Title Row

while (!SR2.EndOfStream)

{

dataTray = SR2.ReadLine().Split(',');

double spd = Double.Parse(dataTray[0]);

double fd = Double.Parse(dataTray[1]);

double dpt = Double.Parse(dataTray[2]);

double ga = Double.Parse(dataTray[3]);

double ra = Double.Parse(dataTray[4]);

Surface temp2 = new Surface(spd, fd, dpt, ga, ra);

evalData.Add(temp2);

}

SR2.Close();

}

public List<Surface> getOptiData()

{

return optiData;

}

public List<Surface> getEvalData()

{

return evalData;

}

public static double scaleValue(double actualValue, double actualMin, double actualMax, double scaleMin, double scaleMax)

{

return (actualValue - actualMin) / (actualMax - actualMin) \* (scaleMax - scaleMin) + scaleMin;

}

public static double descaleValue(double scaleValue, double actualMin, double actualMax, double scaleMin, double scaleMax)

{

return ((scaleValue - scaleMin) \* (actualMax - actualMin)) / (scaleMax - scaleMin) + actualMin;

}

public static double scaleSpeed(double speed)

{

return scaleValue(speed, minSpeed, maxSpeed, -0.5, 0.5);

}

public static double scaleFeed(double feed)

{

return scaleValue(feed, minFeed, maxFeed, -0.5, 0.5);

}

public static double scaleDepth(double depth)

{

return scaleValue(depth, minDepth, maxDepth, -0.5, 0.5);

}

public static double scaleGa(double Ga)

{

return scaleValue(Ga, minGa, maxGa, -0.5, 0.5);

}

public static double scaleRa(double Ra)

{

return scaleValue(Ra, minRa, maxRa, 0, 1);

}

public static double descaleRa(double scaleRa)

{

return descaleValue(scaleRa, minRa, maxRa, 0, 1);

}

}

***Code of a GA using a NN to predict the roughness is included below:***

* ***Main Program:***

namespace NNPredictingRougthness

{

class Program

{

static GreyImageList greyImageList;

static NeuralNetwork NN;

static GeneticAlgo GA;

static void Main(string[] args)

{

greyImageList = new GreyImageList();

//surfaceList = new SurfaceList();

NN = new NeuralNetwork();

TrainNN();

GA = new GeneticAlgo();

DetermineParameters();

Console.ReadLine();

}

static void TrainNN()

{

int counter = 0;

while (counter < 200)

{

counter++;

double totalSSE = 0;

double totalEvalSSE = 0;

foreach (GreyImage greyImage in greyImageList.GetTestGreyImages())

{

Surface surface = greyImage.surface;

List<double> predictedRougthness = NN.Predict(greyImage);

List<double> actualRougthness = new List<double>();

actualRougthness.Add(surface.getScaledRa());

NN.TrainNeuron(0.1, actualRougthness);

totalSSE = Math.Pow(GreyImageList.descaleRa(predictedRougthness[0]) - surface.getRa(), 2);

}

foreach (GreyImage greyImage in greyImageList.GetEvalGreyImages())

{

Surface surface = greyImage.surface;

List<double> predictedRougthness = NN.Predict(greyImage);

List<double> actualRougthness = new List<double>();

actualRougthness.Add(surface.getScaledRa());

totalEvalSSE = Math.Pow(GreyImageList.descaleRa(predictedRougthness[0]) - surface.getRa(), 2);

}

if (counter%1 == 0)

{

Console.WriteLine("{0} | TestSSE: {1} | EvalSSE: {2}",counter,totalSSE,totalEvalSSE);

}

}

foreach (GreyImage greyImage in greyImageList.GetTestGreyImages())

{

Surface surface = greyImage.surface;

List<double> predictedRougthness = NN.Predict(greyImage);

Console.WriteLine("Actual RA: {0} | PredictedRA: {1} | Difference: {2}", surface.getRa(), GreyImageList.descaleRa(predictedRougthness[0]), surface.getRa()- GreyImageList.descaleRa(predictedRougthness[0]));

}

foreach (GreyImage greyImage in greyImageList.GetEvalGreyImages())

{

Surface surface = greyImage.surface;

List<double> predictedRougthness = NN.Predict(greyImage);

Console.WriteLine("Actual RA: {0} | PredictedRA: {1} | Difference: {2}", surface.getRa(), GreyImageList.descaleRa(predictedRougthness[0]), surface.getRa() - GreyImageList.descaleRa(predictedRougthness[0]));

}

}

static void DetermineParameters()

{

double GivenRoughtness = 1.69;

int counter = 1;

double prevError = Double.MaxValue;

int changeCounter = 0;

do

{

foreach (RougthnessChromosone curModel in GA.population)

{

double totalFitness = 0;

Surface surface = new Surface(curModel.GetCoefs()[0], curModel.GetCoefs()[1], curModel.GetCoefs()[2], curModel.GetCoefs()[3],-1);

GreyImage curPattern = new GreyImage("noname", surface, curModel.GetCoefs().GetRange(4, curModel.GetCoefs().Count - 4));

curModel.bestRoughtness = GreyImageList.descaleRa(NN.Predict(curPattern)[0]);

curModel.SetFitness(Math.Pow(curModel.bestRoughtness - GivenRoughtness, 2));

totalFitness += curModel.GetFitness();

curModel.SetFitness(totalFitness);

}

//if (counter != 0)

//{

// prevError = genetic.population.get(0).GetCost();

//}

GA.NextGeneration();

//if ((genetic.population.get(0).GetCost() - prevError) / genetic.population.get(0).GetCost() >= 0.0)

//{

// changeCounter++;

//}

//else {

// changeCounter = 1;

//}

//if (counter % 1000 == 0)

//{

// if (counter % 3000 == 0)

// {

// NutritionProblemModel.mutationMagnitude \*= 2.0;

// //Genetic.TOURNAMENT\_SIZE += 2;

// }

// else {

// NutritionProblemModel.mutationMagnitude /= 2; //DEFUALT 2.0

// }

// if (counter % 10000 == 0)

// {

// Genetic.TOURNAMENT\_SIZE += 5;

// }

// //GeneticModel.mutateChance -= 0.01;

//}

if (counter % 1 == 0)

{

Console.WriteLine(counter + " | Current Fittest: " + GA.population[0].GetFitness() + " | NN Rougness: " + GA.population[0].bestRoughtness);

}

counter++;

} while (counter < 1000);

String output = "";

Console.WriteLine(GA.population[0].GetCoefs()[0] + "; " + GA.population[0].GetCoefs()[1] + "; " + GA.population[0].GetCoefs()[2] + "; " + GA.population[0].GetCoefs()[3]);

GA.population[0].DrawImage();

}

}

* ***Roughness Chromosome:***

namespace NNPredictingRougthness

{

class RougthnessChromosone

{

List<Double> coeffs;

private double NumberOfGenes = 4 + 10000;

public double fitness = double.MaxValue;

public static double mutationMagnitude = 0.1;

public static double mutationChancePerValue = 0.5;

public double mutateChance = 0.5;

public double crossoverRate = 0.5;

public double bestRoughtness = -1;

public RougthnessChromosone()

{

coeffs = new List<Double>();

coeffs.Add(GreyImageList.initeSpeed());

coeffs.Add(GreyImageList.initFeed());

coeffs.Add(GreyImageList.initDepth());

coeffs.Add(GreyImageList.initGa());

for (int x = 4; x < NumberOfGenes; x++)

{

coeffs.Add(initializeValue());

}

}

public RougthnessChromosone(List<Double> coefficients)

{

coeffs = new List<Double>();

foreach (Double curD in coefficients)

{

coeffs.Add(curD);

}

}

public double initializeValue()

{

return 0.8 + GeneticAlgo.random.NextDouble() \* 0.2 ;

}

public RougthnessChromosone(RougthnessChromosone parent1, RougthnessChromosone parent2)

{

coeffs = new List<Double>();

double r = GeneticAlgo.random.NextDouble();

if (r < crossoverRate)

{

for (int x = 0; x < NumberOfGenes; x++)

{

r = GeneticAlgo.random.NextDouble();

// if(r < 0.5){

// coeffs.add(parent1.coeffs.get(x));

// }else{

// coeffs.add(parent2.coeffs.get(x));

// }

coeffs.Add(r \* parent1.coeffs[x] + (1 - r) \* parent2.coeffs[x]);

}

}

else {

for (int x = 0; x < NumberOfGenes; x++)

{

coeffs.Add(parent1.coeffs[x]);

}

}

r = GeneticAlgo.random.NextDouble();

if (r < mutateChance)

{

MUTATE();

}

}

public void MUTATE()

{

for (int x = 0; x < 4; x++)

{

double r = GeneticAlgo.random.NextDouble();

if (r < mutationChancePerValue)

{

double curD = coeffs[x];

curD = Math.Max(0, (curD - (GeneticAlgo.random.NextDouble() \* 1 - (1 / 2.0))));

coeffs[x] = curD;

}

}

for (int x = 4; x < NumberOfGenes; x++)

{

double r = GeneticAlgo.random.NextDouble();

if (r < mutationChancePerValue)

{

double curD = coeffs[x];

curD = Math.Min(Math.Max(0, (curD - (GeneticAlgo.random.NextDouble() \* 0.1 - (0.1 / 2.0)))),1);

coeffs[x] = curD;

}

}

}

public RougthnessChromosone Clone()

{

return new RougthnessChromosone(this.coeffs);

}

public List<Double> GetCoefs()

{

return coeffs;

}

public double GetFitness()

{

return fitness;

}

public void SetFitness(double fitness)

{

this.fitness = fitness;

}

public void DrawImage()

{

byte[] pixels = new byte[10000];

for(int x = 0; x < 10000; x++)

{

pixels[x] = Convert.ToByte(Math.Ceiling((coeffs[x + 4])\*255));

}

GreyImageList.DrawImage(pixels, @"C:\Users\Francois\Desktop\TestImage.bmp");

}

}

* Neural Network Class:

namespace NNPredictingRougthness

{

class NeuralNetwork

{

private readonly int J = 5; //Number of neurons in hidden layer

private readonly int I = 4 + 10000; //Number of inputs

private readonly int K = 1; //Number of neurons in output layer

private List<Double> InputN;

private List<Double> HiddenN;

private List<Double> OutputN;

private double momentumConstant = 0.0001;

private double[,] LearningConstantsV;

private double[,] LearningConstatnsW;

private double[,] V;

private double[,] W;

private double[,] dV;

private double[,] dW;

private double[,] prevdV;

private double[,] prevdW;

private Random random = new Random();

public NeuralNetwork()

{

V = new double[J,I + 1];

W = new double[K,J + 1];

dV = new double[J,I + 1];

dW = new double[K,J + 1];

prevdV = new double[J,I + 1];

prevdW = new double[K,J + 1];

LearningConstantsV = new double[J,I + 1];

LearningConstatnsW = new double[K,J + 1];

InputN = new List<Double>();

HiddenN = new List<Double>();

OutputN = new List<Double>();

initializeWeights();

initializeNeurons();

}

private void initializeWeights()

{

for (int j = 0; j < J; j++)

{

for (int i = 0; i < I + 1; i++)

{

V[j,i] = calcRandomWeight();

dV[j,i] = 0;

prevdV[j,i] = 0;

LearningConstantsV[j,i] = 0.01;

}

}

for (int k = 0; k < K; k++)

{

for (int j = 0; j < J + 1; j++)

{

W[k,j] = calcRandomWeight();

dW[k,j] = 0;

prevdW[k,j] = 0;

LearningConstatnsW[k,j] = 0.01;

}

}

}

public void SetWeights(double[,] newInputWeights, double[,] newOutputWeights)

{

V = newInputWeights;

W = newOutputWeights;

}

private void initializeNeurons()

{

for (int j = 0; j < J; j++)

{

HiddenN.Add(0.0);

}

for (int k = 0; k < K; k++)

{

OutputN.Add(0.0);

}

}

// private double calcRandomWeight(){

// return (random.nextDouble()\*2.0 - 1.0);

// }

private double calcRandomWeight()

{

return (random.NextDouble() \* (2.0 / Math.Sqrt(I + 1)) - (1.0 / Math.Sqrt(I + 1)));

}

public List<Double> Predict(GreyImage image)

{

InputN = new List<Double>();

InputN.AddRange(image.surface.GetScaleInputs());

InputN.AddRange(image.scaledPixelArray);

CalculateHiddenLayer();

CalculateOutputLayer();

return OutputN;

}

private void CalculateHiddenLayer()

{

for (int j = 0; j < J; j++)

{

double total = 0;

for (int i = 0; i < I; i++)

{

total += V[j,i] \* InputN[i];

}

total += V[j,I] \* (-1); // hidden bias

HiddenN[j] = HiddenActivationFunction(total);

}

}

private void CalculateOutputLayer()

{

for (int k = 0; k < K; k++)

{

double total = 0;

for (int j = 0; j < J; j++)

{

total += W[k,j] \* HiddenN[j];

}

total += W[k,J] \* (-1); // output bias

OutputN[k] = OutputActivationFunction(total);

}

}

private double HiddenActivationFunction(double net)

{

return SigmoidFunction(net);

}

private double OutputActivationFunction(double net)

{

return SigmoidFunction(net);

}

private double SigmoidFunction(double net)

{

return 1 / (1 + Math.Exp(-1 \* net));

}

private double LinearFunction(double net)

{

return net;

}

public void TrainNeuron(double learningRate, List<Double> actualValues)

{

for (int j = 0; j < J; j++)

{

double Yj = HiddenN[j];

for (int i = 0; i < I; i++)

{

for (int k = 0; k < K; k++)

{

dV[j,i] += LearningConstantsV[j,i] \* (actualValues[k] - OutputN[k]) \* OutputN[k] \* (1 - OutputN[k]) \* W[k,j] \* Yj \* (1 - Yj) \* InputN[i];

}

}

for (int k = 0; k < K; k++)

{

dV[j,I] += LearningConstantsV[j,I] \* (actualValues[k] - OutputN[k]) \* OutputN[k] \* (1 - OutputN[k]) \* W[k,j] \* Yj \* (1 - Yj) \* (-1.0); //output bias

}

}

for (int k = 0; k < K; k++)

{

for (int j = 0; j < J; j++)

{

dW[k,j] += LearningConstatnsW[k,j] \* (actualValues[k] - OutputN[k]) \* OutputN[k] \* (1 - OutputN[k]) \* HiddenN[j];

}

dW[k,J] += LearningConstatnsW[k,J] \* (actualValues[k] - OutputN[k]) \* OutputN[k] \* (1 - OutputN[k]) \* (-1.0); //Hidden bias

}

LearnNeuron();

}

public void LearnNeuron()

{

for (int j = 0; j < J; j++)

{

for (int i = 0; i < I + 1; i++)

{

V[j,i] += (dV[j,i] + momentumConstant \* prevdV[j,i]);

if (dV[j,i] \* prevdV[j,i] == Math.Abs(dV[j,i] \* prevdV[j,i]))

{

LearningConstantsV[j,i] += 0.00001;

}

else {

LearningConstantsV[j,i] -= 0.00001;

}

prevdV[j,i] = dV[j,i];

dV[j,i] = 0;

}

}

for (int k = 0; k < K; k++)

{

for (int j = 0; j < J + 1; j++)

{

W[k,j] += (dW[k,j] + momentumConstant \* prevdW[k,j]);

if (dW[k,j] \* prevdW[k,j] == Math.Abs(dW[k,j] \* prevdW[k,j]))

{

LearningConstatnsW[k,j] += 0.000001;

}

else {

LearningConstatnsW[k,j] -= 0.000001;

}

prevdW[k,j] = dW[k,j];

dW[k,j] = 0;

}

}

}

}