COMP422 - Project 2 - David Mitchell

1 XOR Problem

1.1 XOR with Genetic Programming

For the function set initially I choose the logic operators *OR, NOT, AND* and *EQ*. (the mapping between logic operator and program function is shown in figure 1.1.2) It is known that xor can be solved simply as:

f(a,b) = NOT(a EQ b) or f(a,b) = NOT(a) EQ b when a and b are in [0,1]

Or in words, XOR is true when a is not equal to b or to put it another way, when the opposite of a is equal to b. Using this function set and the terminal values of [[1,1],[0,1],[1,0],[0,0]] the genetic programme was easily able to evolve the below programmes shown in figure 1.1.1.

The method of tree construction chosen was Ramped, in this method half of the initial population is created using the Grow method (nodes are selected from either the function or terminal set until all leaves are from the terminal set) and half using the Full method (where each branch of the tree is fully populated from the function set up to a maximum depth).

	gp_eq(gp_not(a), b) gp_eq(a, gp_not(b)) gp_eq(gp_not(a), b)

FIGURE 1.1.1

Perhaps more interesting however is when we change the function set to not include any logic operators and instead use common arithmetic operations. The operations chosen were, multiplication, addition and subtraction. The top three solved programmes can be seen in figure 1.1.3.

a equal b	gp_eq(a, b)
not a	gp_not(a)
a or b	gp_or(a, b)
a - b	gp_sub(a, b)
a + b	gp_add(a, b)
a * b	gp_mul(a, b)

FIGURE 1.1.2

The fitness function for each of these attempts was the same. Each of the four test examples was evaluated with and its correctness found. If it was correct we incremented a counter (*correct_count*). This counter was subtracted from the number of examples (4) so that if all examples were correctly calculated a score of o (zero) was found. Additionally the number of nodes in the evaluated tree was taken into consideration and that count was multiplied by 0.1 (or some factor which would mean that it had less influence than the number of test cases). This resulting score was then minimized by the genetic programme.

```
gp_mul(gp_sub(b, a), gp_sub(b, a))
gp_sub(gp_add(a, b), gp_mul(gp_add(a, a), b))
gp_add(a, gp_mul(gp_sub(b, gp_add(a, a)), b))
```

FIGURE 1.1.3

1.2 XOR with Neural Network

Initially I constructed a network with input layer, one hidden layer of different amount of tanh neurons and an output layer using the softmax function outputting two values, the probability that the output should be 1 and the probability the output should be 0. It was trained using a learning rate of 0.2 for 1000 iterations with back propagation performed via stochastic gradient descent. The network was initialised with random weights generated from a seed based on my computer's system clock.

Testing of the network's performance was done by running the network to completion (1000 iteration in the case of using the Tanh activation function described above) 30

times. The networks “score” and the mean squared error were tracked and at the end of the 30 runs the mean and standard of each was reported (see figure xxxxxx). Each individual run of the network a different seed to the random weight generator was used (as it was based on the computer's system time).

The scoring of the network's performance was based on a threshold of 0.5, in that any a prediction for 1 with a value over 0.5 means the prediction is 1 and vis versa. If the network predicted all the outputs corrected it was said to have a score of 100%, if only 3 out of the four were predicted correctly it was said to have a score of 75%, and so on.

The tracking of the mean squared error was done by comparing the predicted probabilities of each example ({1,1}, {1,0}, {0,1}, {0,0}) with their expected results in the absolute form (1.0 or 0.0). The difference was squared and averaged and the results can be seen below in figure 1.2.1.

1 Tanh hidden layer

# nodes	score mean	score std	MSE mean	MSE std
2	0.883	0.155	0.059	0.078
4	0.983	0.090	0.008	0.045

FIGURE 1.2.1

Next (mostly out of curiosity) I constructed a 2 node hidden layer with an activation function of Maxout with two pieces each (the number of linear units to use when finding the max). The Maxout activation function was interesting because it is an generalises a Recifier node in it’s leaky version, it was introduced fairly recently (see: http://arxiv.org/pdf/1302.4389v4.pdf Goodfellow et al) and it mostly designed to work with dropout (although it can be used without dropout if the number of pieces are kept at 2). I thought this warranted an experiment. In this case I constructed a network identical to the above Tanh network but used one layer of 2 hidden nodes using the Maxout activation function. Additionally I lowered the number of iterations to 200.

For all intents and purposes this network preformed perfectly with a mean score of 1.0 and a mean squared error of zero down to six significant digits. As to exactly why there is the great increase in performance with this configuration as opposed to the more “traditional” network architecture I can only speculate. I would require a good deal more time to truly understand the ins and out of this type of neuron.

The neural network can only give probabilities whereas the genetic programs results were absolutes. Additionally we can find it much easier to understand the results from a genetic program as it is a simple expression.

2 Digit Recognition

The four tasks I chose where the 10, 15, 30 and 60 percent noise ratio files. In addition to a standard neural network and the nearest neighbor classifiers I also used a maxout neural network (with no dropout) and a network with a shared weight layer.

Of the four the maxout network preformed equal to or better on all four tasks. The results are summarised in figure 2.1. These results represent the mean and standard of classification accuracy after 30 runs. Each run used a seed to the random number generator which was initiated from the computer's system time to ensure randomness during initial weight generation.

Task	neighbor	standard nn	maxout nn	shared weight nn
10% noise	0.963/0.009	0.956/0.011	0.963/0.008	0.960/0.008
15% noise	0.944/0.008	0.927/0.013	0.940/0.007	0.930/0.026
30% noise	0.839/0.018	0.836/0.012	0.840/0.014	0.085/0.018
60% noise	0.534/0.017	0.538/0.024	0.560/0.018	0.560/0.024

FIGURE 2.1

For the nearest neighbor classifier I used a k value of 10 (big surprise there) and Euclidean distance measure. For the neural networks I used a learning rate of 0.02 and a validation size of 10%. The network was considered trained when 10 epochs produced no variation in the validation sets error greater than 0.001.

The network construction is outlined below:

- Standard
    - Input layer
    - Tanh with 10 nodes
    - Tanh with 10 nodes
    - Softmax output

- Maxout
    - Input layer
    - Maxout with 10 nodes with 2 pieces
    - Softmax output
- Convolutional
    - Input layer
    - Rectifier with 8 output channels and a 3 x 3 kernel
    - A fully connected Rectifier layer with 64 nodes
    - Softmax output

Data loading and test split

The data was loaded using a Python library called Pandas which makes file parsing and data parsing quite easy. Using this library I was able to load the data and split into examples and classes by plucking the last column as the class and leaving the rest as the example data. The example data was converted to float values to ensure calculation had the appropriate precision.

The data was then split randomly in half while ensuring an equal distribution of classes in each of the training and test sets.

Conclusions

The simple nearest neighbor approach can be said to be the most preformant of the group because not only did it rank near the top in classification accuracy it was much faster to train and provide test results

3 Symbolic Regression

$$1/x + sin(x), x > 0$$
$$2x + x^2 + 3.0, x \leq 0$$

FIGURE 3.1

Terminal Set

The terminal set chosen for this symbolic regression problem was a number in the range of -10.0 to 10.0 from a uniform distribution. This was done in order to represent the sine wave and the exponential parts of the graph equally (as the crossover from one to the other happens at 0).

Function Set

The function set chosen was the standard arithmetic operators { +, -, * }, the less than, the greater than and the equal to logic operators { <, >, == }, the exponential operator { e^x }, the square root { $\sqrt{x}$ (safe) }, a negative operator {-}, a random integer generator which produces integers from 0-20 and finally the trigonometric operators cosine and sine.

The logic operators were included in hopes of being able handle the area around zero where the equation switches from a wave to an exponential expression. I included the random integers to be able to scale any curves produced by the exponential expression and the square root expression. The arithmetic operators were included to help move the produced graph either up or down in an attempt to get the best fit.

I did notice that the graph could be closely approximated with the simple equation listed in figure 3.2. This equation however misses out of the problem areas of between -1 and 0 and the other between 0 and about +3 caused by the exponential part and the 1/x part of the equations in the questions extended regression problem (figure 3.1).

$$\left(\sqrt{e^{-1.2(x)}}\right) - 0.2$$

FIGURE 3.2

It was therefore thought that the introduction of the logic operators would help cause different, non-smooth, behaviour to the produced graph which would assist in filling in this problem area.

Fitness function and cases

The fitness cases where a set of 200 uniformly chosen random numbers within range of -10 to 10. And the fitness function was the mean squared error the for these 200 cases against the questions extended regression problem.

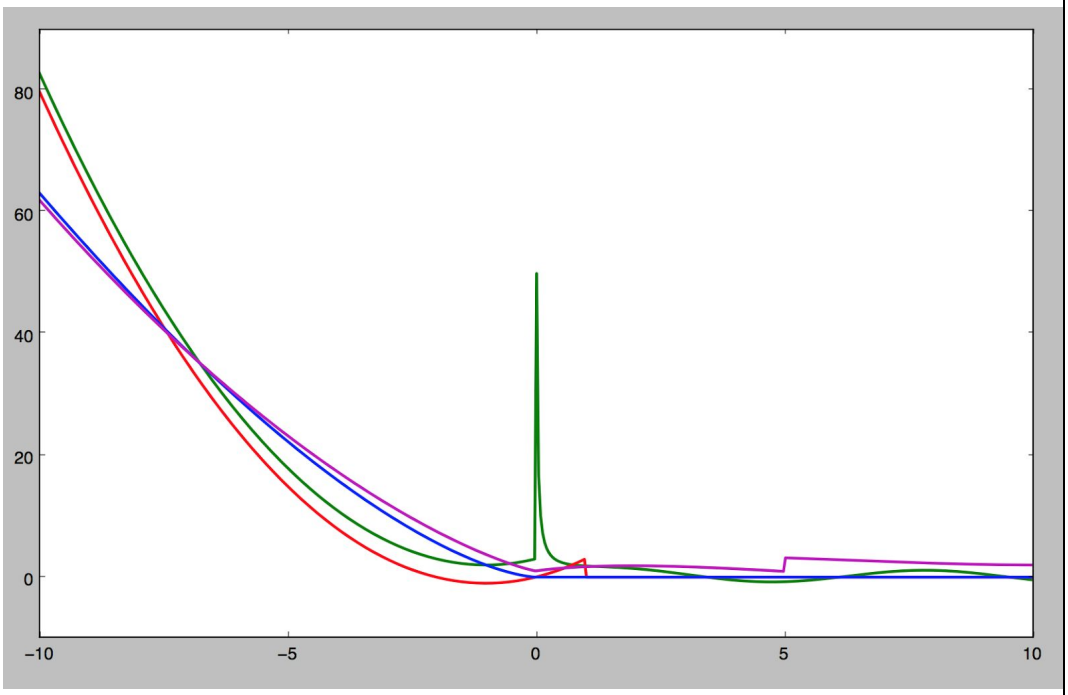

FIGURE 3.3

Other parameters

Other parameters needed to be taken into consideration where the mutation rate, the max depth, the crossover rate, the type of initialisation or the population, the size of the population and the max number of generations. For those settings see figure 3.4 below.

mutation rate	0.25
max depth	4
crossover rate	0.90
initialisation method	half and half
population size	100
number of generations	200
tournament size	10

FIGURE 3.4

Through experimentation it was found that these settings lead to a good balance between speed and accuracy. The selection method was chosen to be tournament size as this method showed the ability to introduce some diversity into the population.

Example produced 'programmes'

sub(sub(mul(mul(x, x), lt(x, 2)), neg(add(x, 2))), safe_sqrt(mul(add(x, 1), add(x, 1))))

This programme produced the best score with an mean squared error of 6.322. It's graph is listed as the red curve in figure 3.3. The two programmes below also performed well with mean squared errors of 10.837 and 12.595 with their graphs in figure 3.3 shown as blue and purple respectively.

mul(gt(safe_sqrt(x), x), mul(add(sub(x, 19), add(3, 18)), x))
mul(neg(sub(neg(x), mul(x, x))), lt(x, gt(safe_exp(x), gt(10, 19))))

Findings

In 30 runs the mean squared error was 65.17 with a standard deviation of 33.50. That is quite a bit of variability and I am sure that in increasing the max number of generations that deviation would come down. Or by setting up other stopping criteria, for example when x percent of the population is within a certain error rate or when a certain percentage of the population is the same programme.

4 Function Optimisation

The stopping criteria for both functions was chosen to be either when the average hamming distance between the solutions in the population is under 0.01, or when the average fitness of the population is under 5.0.

For evaluation of the solution the candidate solution was run through each of the functions (Griewank/Rosenbrock). The result of the function execution was the fitness of the candidate solution.

4.1 Rosenbrock's function

The Rosenbrock function, or as it is sometimes called, the banana function is a difficult problem to optimise as it is easy to fall into a local minimum and as the dimensions increase harder and harder to find the global minimum. The global minimum will always be all ones.

For both the 20 dimension and the 50 dimensions version the settings chosen are listed below

- Population size: 50

- Max number of generations: 2000
- Cognitive rate: 1.0
- Social rate: 1.3
- Inertia: 0.5
- Topology: star

For 20 dimensions the mean after 30 runs was found to be 316.45 with a (very large) standard deviation of 600.72. The best solution would produce an answer of 0. The best solution of the 30 runs was around 97. Each time a local minimum was fallen into which was not able to be gotten out of. As the hamming distance stopping criteria was triggered each time. Perhaps a larger social rate would have aided in this.

For 50 dimensions it took a good deal longer to reach the stopping criteria and again local minimums were fallen into.

Another thing which would have potentially help getting out of the local minimum would be to change the topology to ring and adjust the neighborhood size.

4.2 Griewank function

The optimal solution of the Griewank function is all zeros. This function has many local minimum but they are easier to get out of than the Rosenbrock minimums.

- Population size: 100
- Number of evaluations: 20000
- Cognitive rate: 1.0
- Social rate: 1.5
- Inertia: 1.0
- Topology: star

For the 20 dimension runs a convergence on a minimum was reached rather quickly with the above described stopping criteria. I believe that both of these functions deserved different stopping criteria and with more time I would have experimented with this. I would have lowered the average fitness which defined a stopping situation for the Griewank function. A mean of 1.029 with a standard deviation of 0.015 was obtained after 30 runs.

For the 50 dimension runs a convergence was also reached quickly but again I would have liked to adjust the stopping criteria. A mean of 1.534 with a standard deviation of 0.082 was reached after 30 runs.

5 Feature Construction

When using all features in their raw form with no further feature construction the performance was scored using 10 fold stratified cross validation using a Bayesian classifier.

The set of functions chosen were just simple arithmetic functions, sans the division operation. For the terminal set the feature sets vectors were used. The number of generations was chosen to be 250, with a population size of 50, a crossover rate of 0.9 and a mutation rate of 50%.

For each fold in a 10 fold cross validation a single new feature was constructed using an the performance of a further 10 fold cross validation Bayesian classifier. This was done in order to avoid any feature construction bias. The result of this further cross validation was used to determine the fineness of the generated programme which produced the newly constructed feature used.

It was therefore likely to have a different generated programme for each outer fold during the cross validation.

Balance data

It was noted during the testing of on the Balance data set that one newly generated feature was produced using a programme which was used during 7 of the 10 folds.

gp_sub (gp_mul (f2() f1())gp_mul (f4() f3()))

This programme, in different forms, produced a test score of 100% accuracy. This lead to an overall score of 95.8% accuracy when using feature construction on the Balance data set. This compared to a score of 84.1% when using all features.

Wine data

The overall classification accuracy when using all features of the wine data was 96.2%. When using the single constructed feature this accuracy was found to be no better. I was not able to generate a programme which produced a single feature that performed better than all the features by themselves. A typical result was around 94.4%. This is instructive in itself and shows that perhaps there is no relationship between these features. I would have liked to have seen what feature selection would have produced in this situation. Additionally I would have like to attempt constructing a feature for each class, so three features in total.

6 Feature Selection

6.1 All features verses feature selection

In order to judge the performance of all features verses only the top 4 selected features a Bayesian classifier was chosen as it was the quickest and easiest to get off the ground and running. During performance testing a k-fold cross validation method was chosen to get some empirical scores as to how each method performed.

When selecting ranking features care was taken to avoid feature selection bias. The data was split into 10 stratified folds. Each folds' training and test data was sent to the feature selection method. Then feature select was done only on that folds' training data. This data set was then used to do a further 10 fold cross validation whereby each of these inner loops produced training and test data. These inner loop's training data was used to select features and it's test data was used to garner a score using these selected features. The average of this score was then used to judge the features selections performance..

All features

The performance of the all features was judged on the scores returned from a k-fold cross validation method using k as 10. For the Sonar dataset the classification accuracy was at 60.3%, for the Wisconsin Breast Cancer dataset (WBCD) the classification accuracy was at 93.2%

Feature selection methods

I chose to use a Chi2 and a Pearson single feature ranking methods to chose to top 4 features for each dataset. In order to perform wrapper feature selection I chose to wrap a Decision Tree as it is very easy to extract a feature racking from. The findings are outlined below in figure 6.1

	Sonar	WBCD
Pearsons		
score	**0.710/0.026**	0.944/0.005
top features	'feature11', 'feature12', 'feature49', 'feature45'	'feature21', 'feature28', 'feature23', 'feature8'
Chi2		
score/std	0.688/0.0317	0.915/0.003
top features	'feature45', 'feature11', 'feature12', 'feature36'	'feature14', 'feature24', 'feature4', 'feature23'

Wrapper		
score/std	0.676/0.039	**0.955/0.009**
top features	'feature11','feature4', 'feature45', 'feature16'	'feature28','feature22', 'feature23', 'feature12'
All features score	0.603	0.932

FIGURE 3.2

As can be seen the wrapper method had the best performance over the WBCD dataset and the features selected using the Pearson's feature ranking performed the best on the Sonar dataset. It is to be noted that the selected features list above (top features) are just a summary of the most chosen features during the 10 fold cross validation performed within each inner loop of the feature selection.

Conclusions

It was found that in each run of the k-fold cross validation for the inner loop of the wrapper method a highly variable set of features was chosen. But for the other two methods the variability was much less. I wonder how much of this has to do with the way Decision trees work and if it would have been more consistent if another classifier was chosen for the wrapper method.

It was also noticed during testing that the different methods would end up with different values for the number of features which performed the best if all different n values were tested. But it is to be noted that this testing was done without considering the feature selection bias. It would be quite time consuming to dynamically find that value for each fold.