COMP 307 Assignment 2 Part 1 Report Fang Zhao (300364061)

In this part I used BPNN as the neural network package. A snippet of the network file "iris.net" is shown here:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09 | lr 0.05  m 0.5  ce 0.01  r 1  percent 100.1  3  4  4  3 |

1. The architecture of the neural network:

|  |  |
| --- | --- |
| Number of layers | 3 |
| Number of neurons in the input layer | 4 |
| Number of neurons in the hidden layer | 4 |
| Number of neurons in the output layer | 3 |

Rationale:

* About the input layer: Since we know the training data, and there is no need to add a bias for this classification problem, 4 nodes in the input layer representing 4 features (sepal length, sepal width, petal length, and petal width) is a reasonable choice.
* About the output layer: There are 3 classes (Iris Setosa, Iris Versicolour, and Iris Virginica) in this data set, so 3 nodes in the output layer is a reasonable choice.
* About the number of hidden layers and the number of nodes in hidden layers: Heaton[[1]](#footnote-1) (2008) has a series of empirically-derived rules-of-thumb for deciding parameters of hidden layers, of these, two are applied in this assignment. *(i).* One hidden layer is sufficient for the large majority of problems. *(ii).* the optimal size of the hidden layer is usually between the size of the input and size of the output layers. So, I added 1 hidden layer with 4 neurons inside.

I experimented some alternative options, e.g. 2 or more hidden layers, and only 2 neurons in hidden layer or more than 4 neurons, etc. Most of results are worse than 1 hidden layer with 4 neurons. A simple table is listed here to show the number of epochs used in different network architecture (assuming other parameters stay same).

|  |  |  |  |
| --- | --- | --- | --- |
| Number of layers | Number of neurons | Epochs | # Note |
| 2 | [4, 3] |  | Reached 100% accuracy, but cannot reach 0.01 critical error |
| 3 | [4, 4, 3] | 517 | Used in final submission |
| 3 | [4, 3, 3] | 529 | Almost same |
| 3 | [4, 2, 3] | 502 | Almost same |
| 4 | [4, 4, 4, 3] | 1057 |  |
| 4 | [4, 6, 8, 3] | 1183 |  |
| 5 | [4, 4, 4, 3, 3] | 5212 | Often does not converge |
| 5 | [4, 2, 2, 2, 3] |  | Does not converge |
| 6 | [4, 3, 3, 3, 3, 3] |  | Does not converge |

*\* Note that the number of epochs does not mean every time. It's the most often case or approximate value of average epochs.*

1. The learning parameters of the neural network:

|  |  |
| --- | --- |
| Learning rate | 0.05 |
| Momentum | 0.5 |
| Initial weight range | 1 |

Rationale:

* About the Learning rate: Learning rate should be adjusted to a balanced value, so that the learning process neither converge too slowly, nor over-compensate too much leading to a divergence of the weights and objective function.

According to my experiment, learning rate set to 1 occasionally goes into infinite loop with no convergence. Learning rate ranging from 0.1 to 0.8 can quickly reach (within 100 epochs) the termination condition (critical error below 0.01, see next section), and the accuracy on training set is usually 100%. However, if I use the test set as a validation set, the performance on it is poor. Number of incorrect classifications sometimes even goes up to 10 out of 75. Learning rate set to below 0.1 generally results in slow convergence, but the performance on test set is getting better.

So, after many times of experiments, I found a balanced value, 0.05, with which the system takes about 500 epochs to learn, and most of the time only has 1 out of 75 incorrect classification.

* About the Momentum: Momentum is used to prevent the system from converging to a local minimum. However, setting the momentum parameter too high can create a risk of overshooting the minimum, which can cause the system to become unstable. A momentum coefficient that is too low cannot reliably avoid local minima, and can also slow down the training of the system.

In my experiment, the momentum doesn't seem to affect the system much. I've tried different combination of parameters, in any case of which the momentum made no obvious difference to the result. My guess is the training set is too small or too simple, it contains no local optima, so the momentum's effect is not obvious.

I set the momentum as 0.5. The system is stable with this momentum value.

* About the initial weight range: The initial weight range provides a range in which the initial weighted are randomly generated. If the initial weight range is set too wide, the search space will become too wide as well, which leads to longer time to find the global optima and hence slower learning. Empirically, [-1, 1] usually is a stable initial range.

1. The termination criteria:

|  |  |
| --- | --- |
| Critical error (MSE) | 0.01 |
| Classification of accuracy | 100.1 |

Rationale:

* The termination criteria in this neural network is when the critical error (MSE) is below 0.01. The reason I use critical error instead of accuracy is that the latter is less stable in performance on test set. With accuracy 100% as the termination condition, the MSE sometimes ends at more than 0.09, and the test performance is not good with that neural network. Therefore, I use critical error as the termination condition, and it's set to 0.01.
* When multiple termination conditions are provided, BPNN stops if any of these conditions is met. In order to disable the trigger to stop on accuracy, I set the accuracy to 100.1 (which is an impossible value).

1. The results of 10 independent experiment runs (only showing the summary lines):

|  |
| --- |
| # run no. 1 training:  epoch = 565, mse = 0.010, Percentage = 74/75 = 98.67%  # run no. 1 test:  Mean squared error for training data: 0.013  Number of incorrect classifications: 2/75  Mean squared error for test data: 0.017  Number of incorrect classifications: 1/75  # run no. 2 training:  epoch = 546, mse = 0.010, Percentage = 74/75 = 98.67%  # run no. 2 test:  Mean squared error for training data: 0.013  Number of incorrect classifications: 2/75  Mean squared error for test data: 0.017  Number of incorrect classifications: 1/75  # run no. 3 training:  epoch = 622, mse = 0.010, Percentage = 74/75 = 98.67%  # run no. 3 test:  Mean squared error for training data: 0.013  Number of incorrect classifications: 2/75  Mean squared error for test data: 0.016  Number of incorrect classifications: 1/75  # run no. 4 training:  epoch = 547, mse = 0.010, Percentage = 74/75 = 98.67%  # run no. 4 test:  Mean squared error for training data: 0.013  Number of incorrect classifications: 2/75  Mean squared error for test data: 0.016  Number of incorrect classifications: 1/75  # run no. 5 training:  epoch = 516, mse = 0.010, Percentage = 74/75 = 98.67%  # run no. 5 test:  Mean squared error for training data: 0.013  Number of incorrect classifications: 2/75  Mean squared error for test data: 0.017  Number of incorrect classifications: 1/75  # run no. 6 training:  epoch = 538, mse = 0.010, Percentage = 74/75 = 98.67%  # run no. 6 test:  Mean squared error for training data: 0.012  Number of incorrect classifications: 2/75  Mean squared error for test data: 0.017  Number of incorrect classifications: 1/75  # run no. 7 training:  epoch = 551, mse = 0.010, Percentage = 74/75 = 98.67%  # run no. 7 test:  Mean squared error for training data: 0.013  Number of incorrect classifications: 2/75  Mean squared error for test data: 0.017  Number of incorrect classifications: 1/75  # run no. 8 training:  epoch = 554, mse = 0.010, Percentage = 74/75 = 98.67%  # run no. 8 test:  Mean squared error for training data: 0.013  Number of incorrect classifications: 2/75  Mean squared error for test data: 0.017  Number of incorrect classifications: 1/75  # run no. 9 training:  epoch = 531, mse = 0.010, Percentage = 74/75 = 98.67%  # run no. 9 test:  Mean squared error for training data: 0.013  Number of incorrect classifications: 2/75  Mean squared error for test data: 0.017  Number of incorrect classifications: 1/75  # run no. 10 training:  epoch = 572, mse = 0.010, Percentage = 74/75 = 98.67%  # run no. 10 test:  Mean squared error for training data: 0.013  Number of incorrect classifications: 2/75  Mean squared error for test data: 0.017  Number of incorrect classifications: 1/75 |

The outputs from 10 consecutive runs shows some notable points:

* 10 results are very similar, which indicates that the system with these parameters are quite stable.
* The epochs until termination (usually around 550) is acceptable to me. As discussed earlier, with a sharper learning rate the system can reach the desired termination criteria much faster, but performs worse on the test set. I would consider current system more balanced.
* The accuracy is 74/75 = 98.67%, I would say the performance is not bad. But still, this is an ad-hoc system carefully tuned for the given dataset. Different requirements call for different configurations.
* Each time there is certain misclassified instance, no. 109 (if index start from 1). And this instance is always misclassified under different set of parameters. My guess is that this instance is an outlier or a noise. Most likely the pattern learned from the training set does not cover the features of this outlier.

But having pointed out no. 109, I do not claim that 100% accuracy is always better. Over-fitting is always on the check list when the accuracy is very high.

1. The accuracy of K-nearest neighbour was 96% when k=3, whereas the accuracy of Neural network is 98.67%. They are both quite good performance. The difference between them is not enough to tell which is better in performance. Also, the iris problem is too simple to be the judgement of who has better performance. In fact, we could say that the Neural network method here is an overkill to a simple problem, because Neural network method can tackle with much more complicated problem than K-nearest neighbour method.

K-nearest neighbours, on the other hand, is easy to use. It has fewer parameters (only the value of k in the simple implementation from assignment 1), does not need to learn a model out of training set, and is much easier to implement.

COMP 307 Assignment 2 Part 2 Report Fang Zhao (300364061)

In this part I used JGAP as the genetic programming package. Before I implemented the coding part, I used Microsoft Excel to observe the data and help me deciding terminal set and function set. A plot of data is shown here:

A preliminary analysis: this should be a polynomial function with 4 or more degrees, and the degree must be an even number (4, 6, 8, …). the mathematical function should be like (assuming 4 degrees):

where *a*, *b*, *c*, *d*, and *e* are the coefficients. Here the question we need GP to solve is: *(i)*. whether those operators are plus or minus, and *(ii)*. the value of *a*, *b*, *c*, *d*, and *e.*

1. The terminal set I used in the final submitted version is one mutable constant and one feature. In JGAP the terms used for them are *Terminal* and *Variable* respectively.

In the terminal set, the feature represents the input variable *x*, and the mutable constant represents the bias on *y* axis, which is *e* in the hypothetical equation. Also by observation we should be able to deduct that *e* is a small value between 0 to 2, so we can limit the range of it.

About *a*, *b*, *c*, and *d*, there were five mutable constants representing *a*, *b*, *c*, *d*, and *e* in the terminal set at first. After a series of experiments, the error rate became smaller and smaller, and I became aware that *a*, *b*, *c*, and *d* were less needed. In other words, only one mutable constant is enough for this task. And after I dropped *a*, *b*, *c*, and *d*, the system became much faster to get an error-free solution.

However, after I did a quick research on literature I found it debatable that whether we should adjust the node set according to result. By customising the candidate node set according to the true solution programme, it's almost same as feeding the initial population with the true solution. This will make the system less general to other equations.

1. The function set I used in this task is *[Add, Subtract, Multiply*]. JGAP provides many types of function nodes, but we only want the necessary ones. If the preliminary analysis is correct, we only need *Add*, *Subtract*, and *Power*. However, if we use *Power*, we'll need more constants to represent the different exponent values. So I decide to use *Multiply* to replace *Power* (simply because they are interchangeable in the way like ). This will reduce the size of node set, and leave smaller search space for GP to evolve.

Having chosen [*constant, variable x, add, subtract, multiply*] as the node set, after all, this is just a bold guess. If GP can't evolve out a good solution, we can always enlarge the search place and add more terminal / function nodes. Luckily the later experiment did get a good enough solution.

1. The fitness function I used in this task is:

where is the difference between the programme-generated value and the actual value when the input variable *x* is (as in the given *x-y* pairs). In plain English, this function evaluates the sum of the absolute difference between the programme-generated *y* and the expected *y*. Because it's measuring the total error, the smaller the result is, the fitter (better) the programme is.

1. JGAP has a variety of parameter settings. Their default value and the value I used in this task are listed in the table:

|  |  |  |  |
| --- | --- | --- | --- |
| Parameter | Default Value | Value Used | Description |
| Crossover Probability | 0.9 | 0.9 | The probability that a crossover operation is chosen during evolution |
| Reproduction Probability | 0.1 | 0.1 | The probability that a reproduction operation is chosen during evolution |
| Mutation Probability | 0.1 | 0.1 | The probability that a node is mutated during growing a program |
| Dynamize Arity Probability | 0.08 | 0.08 | The probability that the arity of a node is changed during growing a program |
| New Chromosome Percent | 0.3 | 0.3 | Percentage of the population that will be filled with new individuals during evolution |
| Choosing Function Probability | 0.9 | 0.9 | In crossover: If random number (0..1) < this value, then choose a function otherwise a terminal |
| Max Crossover Depth | 17 | 10 | The maximum depth of an individual resulting from crossover |
| Max Initial Depth | 7 | 5 | The maximum depth of an individual when the Genotype is created |
| Min Initial Depth | 2 | 2 | The minimum depth of an individual when the Genotype is created |
| Population Size | Not set | 800 | The number of chromosomes that will be stored in the Genotype |
| Generation Size | Not set | 500 | The number of generations to evolve |

The default setting looks mostly reasonable for this task. These settings accord to many recommended / typical values in many articles I searched. The only two parameters I changed are the max crossover depth and the max initial depth. I reduced them a little in hope of improving the performance, as from the preliminary analysis the expected function should not be too complicated hence the syntax tree should not be too deep.

As for the population size, according to my experiment, even with a population of 20, the system can sometimes produce a correct maths function within 500 generations. As the population increases, the chance of successfully producing a correct function increases as well. If the population is set to 600, 20 out of 20 runs the system can produce a correct function within 500 generations. If the population continue to increase, no obvious positive effect can be seen, but the performance continues to drop because of the large number of chromosomes to calculate. Therefore, my conclusion is that the value of population size is like a threshold function. The system needs a large enough population to initialise with a rich set of solutions, but once the population size passes the threshold value, increasing it stops to add value.

The population is set to 800 for this task.

As for the stopping criteria, due to the time constraint, I couldn't find a way to set any termination condition except for the number of generations. In other words, the evolution will run to the end until the specified number of generations is reached. From my experiment, JGAP usually can find the best solution before the 25th generation. (I also didn't have enough time to learn about *log4j* and to change the way JGAP outputs logs, so I'm not sure about in which exact generation GP found the best solution. I can only be sure that the best solution came out from which 25 generations.) But in large amount of repeating runs, very occasionally the best solution is found after the 300th generation. So in hope of more certainty for the system, I set the generation size to 500. I believe this setting does not compromise too much performance.

1. 3 best programmes evolved by GP:

Fitness value: 0.00

* Fitness value: 0.00
* Fitness value: 0.00

These three functions can be transformed to

1. As discussed in section 5, the three best programmes can all be transformed to . To validate these three best programmes we can simply validate on . If we put the desired *y* value and the equation-calculated *y* value together and compare, we can see the difference between them are very small (might just be rounding difference):

|  |  |  |  |
| --- | --- | --- | --- |
| Input x | Desired y | Equation-calculated y | Difference |
| -2.00 | 37.00000 | 37.000000 | 0 |
| -1.75 | 24.16016 | 24.160156 | 0.000004 |
| -1.50 | 15.06250 | 15.062500 | 0 |
| -1.25 | 8.91016 | 8.910156 | 0.000004 |
| -1.00 | 5.00000 | 5.000000 | 0 |
| -0.75 | 2.72266 | 2.722656 | 0.000004 |
| -0.50 | 1.56250 | 1.562500 | 0 |
| -0.25 | 1.09766 | 1.097656 | 0.000004 |
| 0.00 | 1.00000 | 1.000000 | 0 |
| 0.25 | 1.03516 | 1.035156 | 0.000004 |
| 0.50 | 1.06250 | 1.062500 | 0 |
| 0.75 | 1.03516 | 1.035156 | 0.000004 |
| 1.00 | 1.00000 | 1.000000 | 0 |
| 1.25 | 1.09766 | 1.097656 | 0.000004 |
| 1.50 | 1.56250 | 1.562500 | 0 |
| 1.75 | 2.72266 | 2.722656 | 0.000004 |
| 2.00 | 5.00000 | 5.000000 | 0 |
| 2.25 | 8.91016 | 8.910156 | 0.000004 |
| 2.50 | 15.06250 | 15.062500 | 0 |
| 2.75 | 24.16016 | 24.160156 | 0.000004 |

For this task, we don't have a test set or validation set to validate the regression function. But for the available data, 20 pairs of *x-y*, GP has done a fine job and found a matching function.

In this task, I would say that the analysis of the data before running genetic programming helps to narrow down the search space significantly. With a smaller set of nodes to evolve, and with a rich set of candidate solutions, GP is able to find the best programme faster. If we don't assume the form of the equation, it would be much more difficult to do the job.

Purely out of curiosity, I created another dataset, in which the *x-y* pairs were generated by , and tuned the GP system again and again in hope of getting an acceptable solution. Just like suspected in section 1, the system that solves the task from assignment cannot solve the new problem. The best solution I could get is , which can be transformed into . The difference between this equation and the desired one is not that big. The total error of 20 *x-y* pairs is 11.39, which means the average error per pair is 0.57. I had to raise the mutation probability and reproduction probability to let GP seek for new solutions more progressively, and I also had to increase the maximum depth to allow more complexed solutions. The population size and generation size were also greatly increased to allow more possibilities. It was a lot harder to get a good enough solution.

In summary, this out-of-subject experiment confirmed my theory that an over-tuned system is less useful in solving more general problems.

COMP 307 Assignment 2 Part 3 Report Fang Zhao (300364061)

In this part I used JGAP as the genetic programming package.

1. The terminal set I used in this task is 9 attributes (*Variable*), which represents [*CT, USz, UShp, MA, SESz, BN, BC, NN, M*], and one mutable constant (*Terminal*), which represents an unknown bias.

While the reason of using 9 *Variable*s to represents 9 numerically valued attributes is obvious, the reason of using one mutable constant is purely empirically derived. I have tried using more than one mutable constants, but no visible improvement can be seen. I have tried with zero mutable constant, but the classification accuracy drops drastically. I have also considered using a non-mutable constant, however it seems meaningless without any justification to decide the value of it. So at last, using one mutable constant seems to be a reasonable choice for this task.

The terminal set I used in this task is one constant and one feature. The terms used for "constant" and "feature" in JGAP are Terminal and Variable respectively.

According to the graph, the mathematical function should be like:

where *a*, *b*, *c*, *d*, and *e* are the coefficients. Here the question we need GP to solve is: *(i)*. whether those operators are plus or minus, and *(ii)*. the value of *a*, *b*, *c*, *d*, and *e*. So, a feature represents the input variable *x*, and the constant represents the bias on *y* axis, which is *e* are added into the terminal set. Also by observation we should be able to deduct that *e* is a small value between 0 to 2.

1. T

The function set I used in this task is *[Add, Subtract, Multiply*]. JGAP provides many types of function nodes, but we only want the necessary ones. If the preliminary analysis is correct, we only need *Add*, *Subtract*, and *Power*. However, if we use *Power*, we'll need more constants to represent the different exponent values. So I decide to use *Multiply* to replace *Power* (simply because they are interchangeable in the way like ). This will reduce the size of node set, and leave smaller search space for GP to evolve.

Having chosen [*constant, variable x, add, subtract, multiply*] as the node set, after all, this is just a bold guess. If GP can't evolve out a good solution, we can always enlarge the search place and add more terminals / functions. Luckily the later experiment did get a good enough solution.

## only add and subtract can generate not-so-bad programmes, accuracy can reach about 90+.

## including Multiply and Divide does increase the accuracy by about 2 or 3 percent from observation of multiple runs. Accuracy can climb up to about 95

## I have tried more functions, like exponential, logarithm, power, and trigonometric functions, but none of them showed significant improvement on the final accuracy.

## There is one function I really wanted to implement for this task, the sigmoid function. However

1. T

The fitness function I used in this task is:

where is the difference between the programme-generated value and the actual value when the input variable *x* is (as in the given *x-y* pairs). In plain English, this function evaluates the sum of the absolute difference between the programme-generated *y* and the expected *y*. Because it's measuring the total error, the smaller the result is, the fitter (better) the programme is.

1. J

JGAP has a variety of parameter settings. Their default value and the value I used in this task are listed in the table:

|  |  |  |  |
| --- | --- | --- | --- |
| Parameter | Default Value | Value Used | Description |
| Crossover Probability | 0.9 | 0.9 | The probability that a crossover operation is chosen during evolution |
| Reproduction Probability | 0.1 | 0.1 | The probability that a reproduction operation is chosen during evolution |
| Mutation Probability | 0.1 | 0.1 | The probability that a node is mutated during growing a program |
| Dynamize Arity Probability | 0.08 | 0.08 | The probability that the arity of a node is changed during growing a program |
| New Chromosome Percent | 0.3 | 0.3 | Percentage of the population that will be filled with new individuals during evolution |
| Choosing Function Probability | 0.9 | 0.9 | In crossover: If random number (0..1) < this value, then choose a function otherwise a terminal |
| Max Crossover Depth | 17 | 10 | The maximum depth of an individual resulting from crossover |
| Max Initial Depth | 7 | 5 | The maximum depth of an individual when the Genotype is created |
| Min Initial Depth | 2 | 2 | The minimum depth of an individual when the Genotype is created |
| Population Size | Not set | 800 | The number of chromosomes that will be stored in the Genotype |
| Generation Size | Not set | 500 | The number of generations to evolve |

The default setting looks mostly reasonable for this task. These settings accord to many recommended / typical values in many articles I searched. The only two parameters I changed are the max crossover depth and the max initial depth. I reduced them a little in hope of improving the performance, as from the preliminary analysis the expected function should not be too complicated hence the syntax tree should not be too deep.

As for the population size, according to my experiment, even with a population of 20, the system can sometimes produce a correct maths function within 500 generations. As the population increases, the chance of successfully producing a correct function increases as well. If the population is set to 600, 20 out of 20 runs the system can produce a correct function within 500 generations. If the population continue to increase, no obvious positive effect can be seen, but the performance continues to drop because of the large number of chromosomes to calculate. Therefore, my conclusion is that the value of population size is like a threshold function. The system needs a large enough population to initialise with a rich set of solutions, but once the population size passes the threshold value, increasing it stops to add value.

The population is set to 800 for this task.

As for the stopping criteria, due to the time constraint, I couldn't find a way to set any termination condition except for the number of generations. In other words, the evolution will run to the end until the specified number of generations is reached. From my experiment, JGAP usually can find the best solution before the 25th generation. (I also didn't have enough time to learn about *log4j* and to change the way JGAP outputs logs, so I'm not sure about in which exact generation GP found the best solution. I can only be sure that the best solution came out from which 25 generations.) But in large amount of repeating runs, very occasionally the best solution is found after the 300th generation. So in hope of more certainty for the system, I set the generation size to 500. I believe this setting does not compromise too much performance.

1. 3

Three best programmes evolved by GP:

Fitness value: 0.00

* Fitness value: 0.00
* Fitness value: 0.00

These three functions can be transformed to

1. A

As discussed in section 5, the three best programmes can all be transformed to . To validate these three best programmes we can simply validate on . If we put the desired *y* value and the equation-calculated *y* value together and compare, we can see the difference between them are very small (might just be rounding difference):

|  |  |  |  |
| --- | --- | --- | --- |
| Input x | Desired y | Equation-calculated y | Difference |
| -2.00 | 37.00000 | 37.000000 | 0 |
| -1.75 | 24.16016 | 24.160156 | 0.000004 |
| -1.50 | 15.06250 | 15.062500 | 0 |
| -1.25 | 8.91016 | 8.910156 | 0.000004 |
| -1.00 | 5.00000 | 5.000000 | 0 |
| -0.75 | 2.72266 | 2.722656 | 0.000004 |
| -0.50 | 1.56250 | 1.562500 | 0 |
| -0.25 | 1.09766 | 1.097656 | 0.000004 |
| 0.00 | 1.00000 | 1.000000 | 0 |
| 0.25 | 1.03516 | 1.035156 | 0.000004 |
| 0.50 | 1.06250 | 1.062500 | 0 |
| 0.75 | 1.03516 | 1.035156 | 0.000004 |
| 1.00 | 1.00000 | 1.000000 | 0 |
| 1.25 | 1.09766 | 1.097656 | 0.000004 |
| 1.50 | 1.56250 | 1.562500 | 0 |
| 1.75 | 2.72266 | 2.722656 | 0.000004 |
| 2.00 | 5.00000 | 5.000000 | 0 |
| 2.25 | 8.91016 | 8.910156 | 0.000004 |
| 2.50 | 15.06250 | 15.062500 | 0 |
| 2.75 | 24.16016 | 24.160156 | 0.000004 |

For this task, we don't have a test set or validation set to validate the regression function. But for the available data, 20 pairs of *x-y*, GP has done a fine job and found the regression function.

In this task, I would say that the analysis of the data before running genetic programming helps to narrow down the search space significantly. With a smaller set of nodes to evolve, and with a rich set of candidate solutions, GP is able to find the best programme faster.

1. Heaton, J., 2008. *Introduction to neural networks with Java*. Heaton Research, Inc.. [↑](#footnote-ref-1)