Feed Forward Neural Network Entities

Andreas Hadjiprocopis

Ph.D. Thesis

Department of Computer Science
City University

June 21, 2000

This thesis is submitted as part of the requirements for a Ph.D. in Computer Science, in the Department of Computer Science of City University, London.

Contents

List of Figures xiii
List of Tables xvi
Acknowledgements xvii
Declaration xix
Abstract xxiii

I INTRODUCTION 1
1.1 Motivations 1
1.2 Contribution 3
1.2.1 Entities of feed forward neural networks: the model 3
1.2.2 The utility of the model 4
1.2.3 Published work 4
1.3 Structure of the thesis 5

II CONNECTIONISM 7
2.1 Connectionism: raison d'être 7
2.1.1 Symbolic AI 7
2.1.2 What is wrong with symbols? 7
2.1.3 Reductionism versus Emergence 8
2.2 Historical 9
2.3 Connectionist learning 14
2.3.1 Introduction 14
2.3.2 A new computational architecture 15
2.4 Connectionism: what is it really worth? 16

III FEED FORWARD NEURAL NETWORKS 19
3.1 Introduction 19
3.2 FFNN Formalism 20
3.2.1 Operation of the neuron 21
3.2.2 Operation of the FFNN 22
3.3 FFNN as Universal Function Approximators 24
3.4 The back-propagation of error training method 25
3.4.1 Introduction 25
3.4.2 Gradient descent 26
3.4.3 Back-propagation 27

IV CRITIQUE OF FFNN 29
4.1 Introduction . . . 29
4.2 The power of linear classifiers . . . 32
4.2.1 Order of a predicate and the perceptron . . . 32
4.2.2 Linear separability . . . 33
4.3 FFNN: Issues of Computational Complexity . . . 35
4.3.1 Introduction . . . 35
4.3.2 Some complexity classes . . . 35
4.3.3 Known results . . . 36
4.3.4 Conclusion . . . 37
4.4 Learning as optimisation . . . 37
4.4.1 Introduction . . . 37
4.4.2 Back-propagation . . . 38
4.4.3 Local Minima . . . 41
4.4.4 Premature Neuron Saturation . . . 42
4.5 Issues of parallelism and hardware implementation . . . 43
4.5.1 Parallelism . . . 43
4.5.2 Hardware implementation constraints . . . 45
4.6 The non-explicit nature of learning . . . 45
4.7 Summary . . . 46

V FFNN ENTITIES 49
5.1 Introduction . . . 49
5.2 Motivation . . . 51
5.3 Modular neural architectures: state of the art . . . 52
5.3.1 Committees of networks . . . 52
5.3.2 Other ensemble methods: bagging and boosting . . . 52
5.3.3 Mixtures of Experts . . . 53
5.3.4 Summary and margins for improvement . . . 54
5.4 FFNN entities . . . 55
5.4.1 Introduction . . . 55
5.4.2 Class 1 FFNN entities: formalism . . . 57
5.4.3 Class 1 FFNN entities: construction . . . 59
5.4.4 Class 1 FFNN entities: training . . . 60
5.4.5 Class 1 FFNN entities with adjustable connections . . . 60
5.4.6 The derivative of the FFNN transfer function . . . 62
5.4.7 Class 2 FFNN entities: formalism . . . 65

5.4.8 Class 3 FFNN entities 66
5.5 The *np* language and interpreter 67
5.5.1 Introduction 67
5.5.2 Structure 68
5.6 FFNN entities and the Universal Function Approximation property . . . 70
5.7 Single FFNN and C_1 entity: comparison of training times 72
5.8 Benefits from using the entities 76
5.9 Summary 77

VI EMPIRICAL RESULTS **79**
6.1 Introduction 79
6.1.1 Limitations 80
6.1.2 Statistical analysis 80
6.2 Proposed methodology 81
6.2.1 Generalisation Ability 81
6.2.2 Parallelisation of the entities' training procedure 83
6.3 Generalisation Ability 83
6.3.1 Introduction 83
6.3.2 Objectives 84
6.3.3 Statistical significance tests 85
6.3.4 Presentation of the results 86
6.3.5 The Levy data-generating procedure 87
6.3.6 Generalisation Ability: the VARIDIM test 88
6.3.7 Generalisation Ability: the CONSDIM test 122
6.4 Parallelisation of the training process 150
6.4.1 Introduction 150
6.4.2 Results and discussion 151
6.5 Summary and conclusions 154
6.5.1 Generalisation Ability 154
6.5.2 Parallelisation of training 158

VII CONCLUSION **159**
7.1 Recapitulation 159
7.1.1 Motivations 159
7.1.2 FFNN entities: the model 160
7.1.3 Utility of the entities 161
7.1.4 Theoretical results 162

7.1.5 Experimental results 162
7.2 Future work 164

APPENDICES 165

A DERIVATION OF THE BACK-PROPAGATION ALGORITHM **167**
A.1 Introduction 167
A.2 Derivatives with respect to the output layer weights 167
A.3 Derivatives with respect to the hidden layer weights 168
A.4 Final back-propagation equations 169

B THE XOR PROBLEM AND THE PERCEPTRON **171**

C THEORETICAL FRAMEWORKS FOR LEARNING **173**
C.1 Introduction 173
C.2 Formulation of the learning problem 173
C.3 Probably Approximately Correct Learning 174
C.4 The Vapnik-Chervonenkis dimension 175
C.5 Some generalisation bounds 176
C.6 Support Vector Machines 177
C.6.1 Introduction 177
C.6.2 SVM basics 178

D FFNN, THE STONE-WEIERSTRASS THEOREM AND THE UNIVERSAL FUNCTION APPROXIMATION PROPERTY **179**
D.1 Introduction 179
D.2 Metric spaces 179
D.3 The Stone-Weierstrass theorem 180
D.4 FFNN are Universal Function Approximators 181

E TESTING FOR STATISTICAL SIGNIFICANCE **183**
E.1 Introduction 183
E.2 Testing the difference between two populations' means: the *t-test* . . . 184
E.3 Testing the ratio of two populations' variances: the *F-test* 185

F THE *np* SCRIPT LANGUAGE AND INTERPRETER **189**
F.1 Overview of the *np* interpreter 189
F.1.1 Introduction 189
F.1.2 Parallel execution 191

F.1.3 Running *np* . . . 191
F.2 *np* instructions: general object interaction . . . 192
F.2.1 ExtractColumnsFromObject . . . 192
F.2.2 MergeObjects . . . 193
F.2.3 ColumnsArithmetic . . . 193
F.2.4 Deletion of objects . . . 195
F.3 *np* instructions: produce and/or format data sets . . . 195
F.3.1 Vectored data sets . . . 196
F.3.2 Data sets based on time series . . . 198
F.3.3 Data sets for image classification . . . 200
F.4 *np* instructions: single FFNN . . . 202
F.4.1 Creation . . . 203
F.4.2 Training . . . 203
F.4.3 Testing . . . 206
F.5 *np* instructions: Entities . . . 206
F.5.1 Creation . . . 207
F.5.2 Training . . . 209
F.5.3 Entities with connections of adjustable strength . . . 210
F.5.4 Testing . . . 212
F.6 Various other *np* instructions . . . 212
F.6.1 Unlink a file . . . 212
F.6.2 Include an *np* script file . . . 212
F.6.3 Execute a system command . . . 213
F.6.4 Debugging *np* scripts . . . 213
F.6.5 SendInformation . . . 213
F.6.6 SetPath . . . 214
F.7 Files and Channels . . . 214
F.8 The *Lists* specification . . . 215
F.9 The *Configuration Script* format . . . 216
F.10 Alphabetical listing of all *np* instructions . . . 218

G EXAMPLE *np* SCRIPTS **221**
G.1 Some more *np* scripts . . . 224

BIBLIOGRAPHY **229**

INDEX OF NAMES **237**

INDEX OF KEYWORDS 239

List of Figures

2.1 Two basic three-layer neural networks to learn the XOR truth table . . . 11
3.1 A generic FFNN topology . . . 21
3.2 An FFNN with a single hidden layer . . . 23
4.1 Plot of the number of linearly separable dichotomies of N patterns . . . 34
4.2 FFNN training time is proportional to the number of weights . . . 39
5.1 A simple C_1 entity implementation . . . 58
5.2 A C_1 entity with adjustable connections . . . 61
5.3 A C_2 and C_3 entity implementation . . . 65
5.4 Comparison of training times of a single FFNN and an entity . . . 74
6.1 VARIDIM (configuration): **number of weights** as a function of the **number of inputs**, C_1 entity ($\mathcal{C}_1$) . . . 90
6.2 VARIDIM (configuration): **number of weights** as a function of the **number of inputs**, C_1 entity ($\mathcal{C}_{1,big}$) with 66% more weights . . . 91
6.3 VARIDIM (configuration): **number of weights** as a function of the **number of inputs**, C_2 entity ($\mathcal{C}_2$) . . . 91
6.4 VARIDIM (configuration): **number of weights** as a function of the **number of inputs**, C_3 entity ($\mathcal{C}_3$) . . . 92
6.5 VARIDIM (configuration): **number of weights** as a function of the **number of inputs**, single FFNN ($\mathcal{N}_1$) with 35% less weights . . . 92
6.6 VARIDIM (configuration): **number of weights** as a function of the **number of inputs**, single FFNN ($\mathcal{N}_2$) . . . 93
6.7 VARIDIM (configuration): **number of weights** as a function of the **number of inputs**, single FFNN ($\mathcal{N}_3$) with 55% more weights . . . 93
6.8 VARIDIM (configuration): **number of weights** as a function of the **number of inputs**, single FFNN ($\mathcal{N}_4$) with 135% more weights . . . 94
6.9 VARIDIM (time statistics): **training time** as a function of the **number of weights**, C_1 entity ($\mathcal{C}_1$) . . . 95
6.10 VARIDIM (time statistics): **training time** as a function of the **number of weights**, C_1 entity ($\mathcal{C}_{1,big}$) with 66% more weights . . . 95
6.11 VARIDIM (time statistics): **training time** as a function of the **number of weights**, C_2 entity ($\mathcal{C}_2$) . . . 96
6.12 VARIDIM (time statistics): **training time** as a function of the **number of weights**, C_3 entity ($\mathcal{C}_3$) . . . 96

6.13 VARIDIM (time statistics): **training time** as a function of the **number of weights**, single FFNN ($\mathcal{N}_1$) with 35% less weights 97
6.14 VARIDIM (time statistics): **training time** as a function of the **number of weights**, single FFNN ($\mathcal{N}_2$) 97
6.15 VARIDIM (time statistics): **training time** as a function of the **number of weights**, single FFNN ($\mathcal{N}_3$) with 55% more weights 98
6.16 VARIDIM (time statistics): **training time** as a function of the **number of weights**, single FFNN ($\mathcal{N}_4$) with 135% more weights 98
6.17 VARIDIM (sample error statistics): **sample error** as a function of the **number of inputs**, C_1 entity ($\mathcal{C}_1$) 102
6.18 VARIDIM (sample error statistics): **sample error** as a function of the **number of inputs**, C_1 entity ($\mathcal{C}_{1,big}$) with 66% more weights 103
6.19 VARIDIM (sample error statistics): **sample error** as a function of the **number of inputs**, C_2 entity ($\mathcal{C}_2$) 103
6.20 VARIDIM (sample error statistics): **sample error** as a function of the **number of inputs**, C_3 entity ($\mathcal{C}_3$) 104
6.21 VARIDIM (sample error statistics): **sample error** as a function of the **number of inputs**, single FFNN ($\mathcal{N}_1$) with 35% less weights 104
6.22 VARIDIM (sample error statistics): **sample error** as a function of the **number of inputs**, single FFNN ($\mathcal{N}_2$) 105
6.23 VARIDIM (sample error statistics): **sample error** as a function of the **number of inputs**, single FFNN ($\mathcal{N}_3$) with 55% more weights 105
6.24 VARIDIM (sample error statistics): **sample error** as a function of the **number of inputs**, single FFNN ($\mathcal{N}_4$) with 135% more weights 106
6.25 VARIDIM (sample error statistics): least mean squares fit of **sample error** as a function of the **number of inputs**, C_1 entity ($\mathcal{C}_1$) 106
6.26 VARIDIM (sample error statistics): least mean squares fit of **sample error** as a function of the **number of inputs**, C_1 entity ($\mathcal{C}_{1,big}$) with 66% more weights 107
6.27 VARIDIM (sample error statistics): least mean squares fit of **sample error** as a function of the **number of inputs**, C_2 entity ($\mathcal{C}_2$) 107
6.28 VARIDIM (sample error statistics): least mean squares fit of **sample error** as a function of the **number of inputs**, C_3 entity ($\mathcal{C}_3$) 108
6.29 VARIDIM (sample error statistics): least mean squares fit of **sample error** as a function of the **number of inputs**, single FFNN ($\mathcal{N}_1$) with 35% less weights 108

6.30 VARIDIM (sample error statistics): least mean squares fit of **sample error** as a function of the **number of inputs**, single FFNN ($\mathcal{N}_2$) 109

6.31 VARIDIM (sample error statistics): least mean squares fit of **sample error** as a function of the **number of inputs**, single FFNN ($\mathcal{N}_3$) with 55% more weights 109

6.32 VARIDIM (sample error statistics): least mean squares fit of **sample error** as a function of the **number of inputs**, single FFNN ($\mathcal{N}_4$) with 135% more weights 110

6.33 VARIDIM (approximation error statistics): **approximation error** as a function of the **number of weights**, C_1 entity ($\mathcal{C}_1$) 114

6.34 VARIDIM (approximation error statistics): **approximation error** as a function of the **number of weights**, C_1 entity ($\mathcal{C}_{1,big}$) with 66% more weights 114

6.35 VARIDIM (approximation error statistics): **approximation error** as a function of the **number of weights**, C_2 entity ($\mathcal{C}_2$) 115

6.36 VARIDIM (approximation error statistics): **approximation error** as a function of the **number of weights**, C_3 entity ($\mathcal{C}_3$) 115

6.37 VARIDIM (approximation error statistics): **approximation error** as a function of the **number of weights**, single FFNN ($\mathcal{N}_1$) with 35% less weights 116

6.38 VARIDIM (approximation error statistics): **approximation error** as a function of the **number of weights**, single FFNN ($\mathcal{N}_2$) 116

6.39 VARIDIM (approximation error statistics): **approximation error** as a function of the **number of weights**, single FFNN ($\mathcal{N}_3$) with 55% more weights 117

6.40 VARIDIM (approximation error statistics): **approximation error** as a function of the **number of weights**, single FFNN ($\mathcal{N}_4$) with 135% more weights 117

6.41 VARIDIM (approximation error statistics): least mean squares fit of **approximation error** as a function of the **number of inputs**, C_1 entity ($\mathcal{C}_1$) 118

6.42 VARIDIM (approximation error statistics): least mean squares fit of **approximation error** as a function of the **number of inputs**, C_1 entity ($\mathcal{C}_{1,big}$) with 66% more weights 118

6.43 VARIDIM (approximation error statistics): least mean squares fit of **approximation error** as a function of the **number of inputs**, C_2 entity ($\mathcal{C}_2$) 119
6.44 VARIDIM (approximation error statistics): least mean squares fit of **approximation error** as a function of the **number of inputs**, C_3 entity ($\mathcal{C}_3$) 119
6.45 VARIDIM (approximation error statistics): least mean squares fit of **approximation error** as a function of the **number of inputs**, single FFNN ($\mathcal{N}_1$) with 35% less weights 120
6.46 VARIDIM (approximation error statistics): least mean squares fit of **approximation error** as a function of the **number of inputs**, single FFNN ($\mathcal{N}_2$) 120
6.47 VARIDIM (approximation error statistics): least mean squares fit of **approximation error** as a function of the **number of inputs**, single FFNN ($\mathcal{N}_3$) with 55% more weights 121
6.48 VARIDIM (approximation error statistics): least mean squares fit of **approximation error** as a function of the **number of inputs**, single FFNN ($\mathcal{N}_4$) with 135% more weights 121
6.49 CONSDIM (time statistics): **training time** as a function of the number of **training vectors**, C_1 entity ($\mathcal{C}_1$) with 12,625 weights 124
6.50 CONSDIM (time statistics): **training time** as a function of the number of **training vectors**, C_1 entity ($\mathcal{C}_{1,big}$) with 20,010 weights 125
6.51 CONSDIM (time statistics): **training time** as a function of the number of **training vectors**, C_2 entity ($\mathcal{C}_2$) with 12,340 weights 125
6.52 CONSDIM (time statistics): **training time** as a function of the number of **training vectors**, C_3 entity ($\mathcal{C}_3$) with 12,340 weights 126
6.53 CONSDIM (time statistics): **training time** as a function of the number of **training vectors**, single FFNN ($\mathcal{N}_1$) with 12,525 weights 126
6.54 CONSDIM (time statistics): **training time** as a function of the number of **training vectors**, single FFNN ($\mathcal{N}_2$) with 20,040 weights 127
6.55 CONSDIM (time statistics): **training time** as a function of the number of **training vectors**, single FFNN ($\mathcal{N}_3$) with 25,050 weights 127
6.56 CONSDIM (time statistics): **training time** as a function of the number of **training vectors**, single FFNN ($\mathcal{N}_4$) with 30,060 weights 128
6.57 CONSDIM (sample error statistics): **sample error** as a function of the number of **training vectors**, C_1 entity ($\mathcal{C}_1$) with 12,625 weights 131

6.58 ConsDim (sample error statistics): **sample error** as a function of the number of **training vectors**, C_1 entity ($\mathcal{C}_{1,big}$) with 20,010 weights 131

6.59 ConsDim (sample error statistics): **sample error** as a function of the number of **training vectors**, C_2 entity ($\mathcal{C}_2$) with 12,340 weights 132

6.60 ConsDim (sample error statistics): **sample error** as a function of the number of **training vectors**, C_3 entity ($\mathcal{C}_3$) with 12,340 weights 132

6.61 ConsDim (sample error statistics): **sample error** as a function of the number of **training vectors**, single FFNN ($\mathcal{N}_1$) with 12,525 weights . . . 133

6.62 ConsDim (sample error statistics): **sample error** as a function of the number of **training vectors**, single FFNN ($\mathcal{N}_2$) with 20,040 weights . . . 133

6.63 ConsDim (sample error statistics): **sample error** as a function of the number of **training vectors**, single FFNN ($\mathcal{N}_3$) with 25,050 weights . . . 134

6.64 ConsDim (sample error statistics): **sample error** as a function of the number of **training vectors**, single FFNN ($\mathcal{N}_4$) with 30,060 weights . . . 134

6.65 ConsDim (sample error statistics): least means squares fit of **sample error** as a function of the number of **training vectors**, C_1 entity ($\mathcal{C}_1$) with 12,625 weights 135

6.66 ConsDim (sample error statistics): least means squares fit of **sample error** as a function of the number of **training vectors**, C_1 entity ($\mathcal{C}_{1,big}$) with 20,010 weights 135

6.67 ConsDim (sample error statistics): least means squares fit of **sample error** as a function of the number of **training vectors**, C_2 entity ($\mathcal{C}_2$) with 12,340 weights 136

6.68 ConsDim (sample error statistics): least means squares fit of **sample error** as a function of the number of **training vectors**, C_3 entity ($\mathcal{C}_3$) with 12,340 weights 136

6.69 ConsDim (sample error statistics): least means squares fit of **sample error** as a function of the number of **training vectors**, single FFNN ($\mathcal{N}_1$) with 12,525 weights 137

6.70 ConsDim (sample error statistics): least means squares fit of **sample error** as a function of the number of **training vectors**, single FFNN ($\mathcal{N}_2$) with 20,040 weights 137

6.71 ConsDim (sample error statistics): least means squares fit of **sample error** as a function of the number of **training vectors**, single FFNN ($\mathcal{N}_3$) with 25,050 weights 138

6.72 CONSDIM (sample error statistics): least means squares fit of **sample error** as a function of the number of **training vectors**, single FFNN ($\mathcal{N}_4$) with 30,060 weights 138
6.73 CONSDIM (approximation error statistics): **approximation error** as a function of the number of **training vectors**, C_1 entity ($\mathcal{C}_1$) with 12,625 weights 142
6.74 CONSDIM (approximation error statistics): **approximation error** as a function of the number of **training vectors**, C_1 entity ($\mathcal{C}_{1,big}$) with 20,010 weights 142
6.75 CONSDIM (approximation error statistics): **approximation error** as a function of the number of **training vectors**, C_2 entity ($\mathcal{C}_2$) with 12,340 weights 143
6.76 CONSDIM (approximation error statistics): **approximation error** as a function of the number of **training vectors**, C_3 entity ($\mathcal{C}_3$) with 12,340 weights 143
6.77 CONSDIM (approximation error statistics): **approximation error** as a function of the number of **training vectors**, single FFNN ($\mathcal{N}_1$) with 12,525 weights 144
6.78 CONSDIM (approximation error statistics): **approximation error** as a function of the number of **training vectors**, single FFNN ($\mathcal{N}_2$) with 20,040 weights 144
6.79 CONSDIM (approximation error statistics): **approximation error** as a function of the number of **training vectors**, single FFNN ($\mathcal{N}_3$) with 25,050 weights 145
6.80 CONSDIM (approximation error statistics): **approximation error** as a function of the number of **training vectors**, single FFNN ($\mathcal{N}_4$) with 30,060 weights 145
6.81 CONSDIM (approximation error statistics): least means squares fit of **approximation error** as a function of the number of **training vectors**, C_1 entity ($\mathcal{C}_1$) with 12,625 weights 146
6.82 CONSDIM (approximation error statistics): least means squares fit of **approximation error** as a function of the number of **training vectors**, C_1 entity ($\mathcal{C}_{1,big}$) with 20,010 weights 146
6.83 CONSDIM (approximation error statistics): least means squares fit of **approximation error** as a function of the number of **training vectors**, C_2 entity ($\mathcal{C}_2$) with 12,340 weights 147

6.84 CONSDIM (approximation error statistics): least means squares fit of **approximation error** as a function of the number of **training vectors**, C_3 entity (C_3) with 12,340 weights 147
6.85 CONSDIM (approximation error statistics): least means squares fit of **approximation error** as a function of the number of **training vectors**, single FFNN ($\mathcal{N}_1$) with 12,525 weights 148
6.86 CONSDIM (approximation error statistics): least means squares fit of **approximation error** as a function of the number of **training vectors**, single FFNN ($\mathcal{N}_2$) with 20,040 weights 148
6.87 CONSDIM (approximation error statistics): least means squares fit of **approximation error** as a function of the number of **training vectors**, single FFNN ($\mathcal{N}_3$) with 25,050 weights 149
6.88 CONSDIM (approximation error statistics): least means squares fit of **approximation error** as a function of the number of **training vectors**, single FFNN ($\mathcal{N}_4$) with 30,060 weights 149
6.89 Comparison of training times for sequential and parallelised training schemes 152
6.90 The ratio of sequential and parallelised training times against the total number of weights 153
B.1 A geometric representation of a two-variable and two-variable-plus-dummy XOR truth table 171

List of Tables

5.1 Training procedure for the C_3 entities 66
5.2 **Approximation error** results when training time benefits maximise . . . 75
5.3 Maximum **training time** benefits 75
6.1 VARIDIM: description of the evaluated networks 88
6.2 VARIDIM: **sample error** statistics for the entities 99
6.3 VARIDIM: **sample error** statistics for single FFNN 99
6.4 VARIDIM: statistical significance (*t-test*) of the **sample error** results for 100 to 500 input dimensions 100
6.5 VARIDIM: statistical significance (*t-test*) of the **sample error** results for more than 500 input dimensions 100
6.6 VARIDIM: statistical significance (*F-test*) of the **sample error** results for 100 to 500 input dimensions 101
6.7 VARIDIM: statistical significance (*F-test*) of the **sample error** results for more than 500 input dimensions 101
6.8 VARIDIM: **approximation error** statistics for the entities 110
6.9 VARIDIM: **approximation error** statistics for single FFNN 111
6.10 VARIDIM: statistical significance (*t-test*) of the **approximation error** results for 100 to 500 input dimensions 111
6.11 VARIDIM: statistical significance (*t-test*) of the **approximation error** results for more than 500 input dimensions 111
6.12 VARIDIM: statistical significance (*F-test*) of the **approximation error** results for 100 to 500 input dimensions 112
6.13 VARIDIM: statistical significance (*F-test*) of the **approximation error** results for more than 500 input dimensions 112
6.14 CONSDIM: description of the evaluated networks 122
6.15 CONSDIM: **sample error** statistics for the entities 129
6.16 CONSDIM: **sample error** statistics for single FFNN 129
6.17 CONSDIM: statistical significance (*t-test*) of the **sample error** results for 10 to 120 training vectors 130
6.18 CONSDIM: statistical significance (*t-test*) of the **sample error** results for more than 120 training vectors 130
6.19 CONSDIM: statistical significance (*F-test*) of the **sample error** results for 10 to 120 training vectors 130

6.20 ConsDim: statistical significance (*F-test*) of the **sample error** results for more than 120 training vectors 130
6.21 ConsDim: **approximation error** statistics for the entities 139
6.22 ConsDim: **approximation error** statistics for single FFNN 139
6.23 ConsDim: statistical significance (*t-test*) of the **approximation error** results for 10 to 120 training vectors 140
6.24 ConsDim: statistical significance (*t-test*) of the **approximation error** results for more than 120 training vectors 140
6.25 ConsDim: statistical significance (*F-test*) of the **approximation error** results for 10 to 120 training vectors 141
6.26 ConsDim: statistical significance (*F-test*) of the **approximation error** results for more than 120 training vectors 141
6.27 Parallelised training: description of the evaluated networks 150
6.28 Parallelised and sequential training time results 151
B.1 Two-variable and two-variable-plus-dummy XOR truth tables 171

Acknowledgements

This thesis has been a labour of love and at times a tour de force. Now that my task is at an end, I feel I must say a warm thank-you to all those kind people who have helped and encouraged me during the past several years. In particular I would like to pay tribute to the dedicated assistance I received from my parents Prokopis and Eleni, my two supervisors Peter Smith and David Gilbert, my Friends Mimmo Mancio and Diana Deacon, Dionysius Glycopantis, Alan Muir, C.M., Salim Omarouayache, Socrates Mylonas, Chidi Iweha, Herbert Wiklicky, Nicos Angelopoulos, Alessandra Di Pierro, Akmal Chaundri and Linda Coughlan.

I am also greatly indebted to Donald Knuth who created TEX, the program on which LATEX is based and with which this thesis was so easily and beautifully typeset.

Finally, I wish to express my gratitude and admiration to all those who have contributed to the greatness of UNIX – perhaps the only real and useful operating system.

A.H.P.

London
November 1999

Declaration

I hereby grant powers of discretion to the University Librarian to allow this thesis to be copied in whole or in part without further reference to me. This permission covers only single copies made for study purposes, subject to normal conditions of acknowledgement.

I remember perfectly well how it was that I stopped painting. One evening, after I had been for eight hours on end in my studio, painting for five or ten minutes at a time and then throwing myself down on the divan and lying there flat, staring up at the ceiling, for an hour or two – all of a sudden, as though at last, after so many feeble attempts, I had had a genuine inspiration, I stubbed out my last cigarette in an ashtray already full of dead cigarette-ends, leapt cat-like from the armchair into which I had sunk, seized hold of a small palette-knife which I sometimes use for scraping off colours and slashed repeatedly at the canvas on which I had been painting, not content until I had reduced it to ribbons. Then from a corner of the room I took a blank canvas of the same size, threw away the torn canvas and placed the new one on the easel. Immediately afterwards, however, I realised that the whole of my – shall I say creative? – energy had been vented completely in my furious, and fundamentally rational, gesture of destruction. I had been working on that canvas for the last two months, doggedly and without pause; slashing it to ribbons with a knife was equivalent, fundamentally, to finishing it – in a negative manner, perhaps, as regards external results, which in any case had little interest for me, but positively, in relation to my own inspiration. In point of fact, my destruction of the canvas meant that I had reached the conclusion of a long discourse which I had been holding with myself for goodness knows how long. It meant that I had at last planted my foot on solid ground. And so the empty canvas that now stood on the easel was not just an ordinary canvas which had not yet been used; it was a particular canvas that I had placed on the easel at the termination of a long job of work. In effect, I thought, seeking to console myself against the sense of catastrophe that was throttling me, this canvas, similar in appearance to so many other canvases but for me fraught with meaning and consequence, could be the starting-point from which I could now begin all over again, in complete freedom; just as if those ten years of painting had not gone by and I myself were still twenty-six, as I was when I had left my mother's house and had gone to live in the studio in Via Margutta, in order to devote myself, in complete leisure, to painting. However, on the other hand, it might well be – in fact, it was highly probable – that the empty canvas now flaunting itself on the easel was the outward sign of a development no less intimate and no less necessary but entirely negative, a development which might lead me, by imperceptible stages, to complete impotence. And that this second hypothesis might well be the true one appeared to be borne out by the fact that slowly but surely boredom had come to be the companion of my work during the last six months, until finally it had brought it to a full stop on that afternoon when I slashed my canvas to tatters ...

Alberto Moravia, *The Empty Canvas*, 1961

ABSTRACT

THE APPLICATION of *feed forward neural networks* (FFNN) to tasks involving high-dimensional data has always presented problems which emanate from the simple fact that these networks can not be scaled up unreservedly without serious side effects.

Gradient descent optimisation methods, independently of how well they perform in lower dimensions, will reach their limitations as soon as the search space reaches a critical dimension. For certain kernels and training data, this should be expected to happen sooner rather than later because the volume of this space grows *exponentially* with the number of input variables – something also known as *curse of dimensionality.* For complex problems, the optimisation process may be hindered further by the appearance of numerous *local minima* due to the increased complexity and multi-modality of the error surface. In addition, *neurons* saturate and lose sensitivity when an excessively high input signal is received. This results in information being blocked and training impeded.

Parallelisation of FFNN proves extremely difficult in practice due to the exuberant communication overheads. This problem, which relates to the *fine* rather than *coarse-grain* parallelism inherent in the FFNN topology, removes virtually any possibility for *efficient* parallel distributed processing. The prospect of winning over the curse of dimensionality remains, largely, utopian.

In this thesis, a methodology for replacing the monolithic FFNN with an *entity* of simpler and smaller FFNN units is proposed. Our motivation stems from the inability of the single FFNN to deal effectively with all the problems mentioned above. Furthermore, although existing neural network models, be they modular or monolithic, are relatively successful in addressing issues of generalisation, specialisation and confidence of prediction, the problems associated with high-dimensional data and scaling remain basically unanswered. The thesis that the brain is not only characterised by a massively connected network of neurons but also by the existence of different computational systems operating at different levels of abstraction and specialising at different functions is by itself a right justification to replace the single FFNN with the entities. The claim here is that the use of the entities not only eliminates the aforementioned scaling problems, hence, allowing for network implementations with, virtually, no size restrictions, but also improves generalisation ability and training consistency, favours a coarse-grain parallelisation of the training process and promotes a computational model which can be studied with an arbitrary level of abstraction.

The concept of neural network decomposition is materialised with the construction of three different FFNN entity models, namely, classes 1, 2 and 3. A mathematical proof that these models are *universal function approximators* is accomplished with the aid of the *Stone-Weierstrass* theorem.

Finally, the generalisation ability and training consistency of the entities as well as time benefits obtained by parallelising their training procedure, are assessed in practice. These empirical results support the claims about the benefits obtained from the use of the entities and the thesis that they can safely replace single FFNN in applications of prohibitevely high dimensionality.

CHAPTER I

INTRODUCTION

1.1 Motivations

Feed forward neural networks (FFNN) are mathematical techniques based on *connectionist* principles and used in the approximation of general mappings from one finite dimensional space to another. They also present a practical application of the theoretical resolution of *Hilbert's* 13^{th} *problem* by **Kolmogorov**, [Kolmogorov, 1957], and **Lorenz**, and have been used with success in a variety of applications ranging from **pattern recognition**, [Bishop, 1995], to **earthquake prediction**, [Hadjiprocopis et al., 1994], from **medical prognosis** and **survival prediction**, [Ohno-Machado and Musen, 1996], to **financial forecasting**, [Gately, 1996].

The successful utilisation of feed forward neural networks in problems of extremely high dimensionality naturally calls for larger implementations. However, traditional neural networks can not be expanded indefinitely because **scaling problems** arise as a direct result of the *curse of dimensionality* – the exponential growth of the volume of Euclidean space as its dimensions increase.

The existing training algorithms for multi-layer perceptrons – all, one way or another, descendants of *hill climbing* and *gradient descent* – are unable to handle the size of the vast weight search space. They are hindered further by the appearance of innumerable local minima due to the multi-modality and complexity of the error surface.

In addition, *premature neuron saturation*, [Burrows and Niranjan, 1993], occurring at the output of the hidden layer nodes due to the presence of an excessively high input signal, causes neurons to lose their sensitivity – the propagation of information is severely blocked.

The focus of this work was on establishing a methodology for creating large neural networks which are immune to the curse of dimensionality and the other problems which

plague traditional architectures. Our efforts were constrained by the requirements outlined below. The new architecture must:

i. possess the **universal function approximation** property. This will ensure that, at least theoretically, such a network can approximate arbitrarily well any real, continuous function,

ii. be able to scale up unreservedly and without any side effects on performance,

iii. have a structure which favours efficient parallelisation of the training process and feasible hardware implementation,

iv. derive from connectionist principles.

The motivation behind substituting a *monolithic* architecture (and, specifically, the traditional FFNN model) with an equivalent *modular* neural network and, in particular, the proposed entities methodology, emanates from the following facts and observations:

i. *Partitioning* a task into smaller sub-tasks is a very good way to reduce complexity without compromising the fitness of the solution. Similarly, breaking a huge and complicated structure (such as a solid neural network) into an entity of smaller structures (the modular network) will most definitely reduce the complexity of the whole system.

ii. A taxonomy of the components of neural network architectures may be defined in which the neuron is the finest level of classification, a layer is a coarser level and a network is a still coarser level. Solid neural networks (e.g. FFNN) are typically designed to be modular **only** at the neuronal level.

iii. The brain is not only characterised by a massively connected network of neurons but also by the existence of different computational systems operating at different levels of abstraction and specialising in different functions:

> " ... the brain is composed of many different parallel, distributed systems, performing well defined functions [...] To address the issue of scaling, we may need to learn how to combine small networks and to place them under the control of other networks." [Freeman, 1991, pp.29-30]

Although the above statement does not deny the main connectionist principle of the simplicity of the basic building element (the neuron in our case), it does recognise the need for somehow *organising* these elements into various entities/modules/blocks which should exist and operate at *different levels of abstractions* and *specialising in different functions.*

So, with *Connectionism* as our point of departure, we set about to develop a framework for constructing a system which is characterised by diversity in representation, level of abstraction and functionality of its constituent elements. These elements are all *based* on the neuron and taking into consideration the taxonomy described in (ii), above.

The end result was reached by applying the same fundamental principles of *distributed processing* and *knowledge representation* used in the development of a FFNN, on building blocks of a much more composite character than that of the *neuron*. Thus, arriving to the concept of *feed forward neural network entities*: a system of processing units of arbitrary complexity, linked via connections of adjustable strength and optimised using common gradient descent methods.

1.2 Contribution

Two questions must be answered by the author of a thesis:

- *what is novel in this work ?*
- *what is useful about this work ?*

These questions are briefly answered here and, at greater length, in the body of this thesis.

1.2.1 Entities of feed forward neural networks: the model

Connectionism promotes a model of computation based on *emergence*; a principle which claims that complex organisational structures can arise from the agglomeration of simple units which do not individually exhibit any of the properties of the system as a whole.

The behaviour of a connectionist system is determined by independent, local processes in the hope that they will produce the higher level tasks required. Thus, just like an *ant colony* or a *society of bees*, the potential of a connectionist system is expected to *exceed the mere sum of the potential of its parts.*

Feed forward neural networks utilise in full these principles and, indeed, have a long record of successful applications to problems where statistical methods or traditional AI techniques have not been doing so well[1].

Entities of feed forward neural networks arise from the direct extension of these connectionist principles. However, the building block of the system is not the neuron but

[1] For example, in pattern recognition and optical character recognition.

the single FFNN instead. By analogy, more complex entities may be constructed by connecting other, less complex, entities together.

1.2.2 The utility of the model

An entity of FFNN can be scaled up, to deal with data of extremely high dimensions, with, virtually, no restrictions; new FFNN units can be added to the entity to absorb the new inputs without encountering the scaling problems, which haunt the single FFNN, at least not to the same extent. Thus, problems which involve high-dimensional data, such as those in the fields of *finance* and *meteorology*, may now be tackled without resorting to dimensionality reduction techniques.

Secondly, the concept of the entities may be used not only to connect single FFNN together but also other entities. What is more, the connectivity and training methodology of the entities remains largely independent of the type of their elements. The significance of this last point is that the same basic connectivity schemes and training methods may be used with any entity and independently of the nature and type of its building blocks.

Thirdly, the entities' structure promotes *coarse-grain* parallelism. This is a feature which favours efficient parallelisation since the main problem which plagued many parallel implementations of single FFNN was the excessive communication overheads arising from the *fine-grain* model of parallelism they featured. More importantly, the communication needs of a neural network determine its successful and efficient **hardware implementation**. In this respect, the engineering problems associated with transferring an entity to silicon are much reduced compared to those of the single FFNN.

Fourthly, the idea of constructing an entity with blocks of **arbitrary type** and **size** promotes a model of computation which can be studied with an **arbitrary level of abstraction.**

1.2.3 Published work

This research work has yielded four publications:

1. [Hadjiprocopis and Smith, 1998]
2. [Hadjiprocopis and Smith, 1997b]
3. [Hadjiprocopis and Smith, 1997a]
4. [Hadjiprocopis et al., 1994]

1.3 Structure of the thesis

The next chapter outlines the main weaknesses of the symbolic approach to Artificial Intelligence and examines how this approach is *complemented* by the *diametrically opposite* connectionist principles and philosophy. A brief historical note on *Connectionism*, *Hebbian learning* and *neural networks*, in general, is also included.

Chapter 3 serves as a guide to *feed forward neural networks*. It includes a *formal mathematical description* of these networks, their operation and a reference to their universal function approximation capabilities. A proof that FFNN can approximate arbitrarily well any real continuous function is included in appendix D on page 179. Finally, a brief introduction to **gradient-descent** optimisation techniques and, in particular to the **back-propagation** method is given. More details on back-propagation, including derivation and formalism, appendix A on page 167.

Chapter 4 contains a critique of *feed forward neural networks* as far as issues of **computational complexity**, **optimisation** and **scaling problems**, **parallelism** and **hardware implementation** are concerned. Appendix C on page 173 contains a *theoretical framework of learning* including references to *Probably Approximately Correct learning* and the *Vapnik-Chervonenkis dimension* which, we feel, will be useful for any future work dealing with the derivation of worst-case generalisation bounds for the entities.

Chapter 5 deals with the concept of FFNN entities. After a statement of motivation and a review of the current state of the art in *modular neural network architectures* and *ensemble* methods, a formal description of constructing and training the entity classes 1, 2 and 3 entities is given. This is followed by a proof that FFNN entities are universal function approximators along the lines of the *Stone-Weierstrass* theorem (part of appendix D on page 179 contains a description of this important theorem). Section 5.4.6 on page 62 contains a derivation of the expressions for the partial derivatives of a FFNN with respect to its input vector. These derivatives are used to generalise the back-propagation algorithm in order to optimise not only the connections between neurons (i.e. within a single FFNN) but also connections between *any* feed forward computing element, be it a single FFNN, an entity, an entity of entities and so on.

Finally, as proof of concept, chapter 6 contains the results of experiments carried out in order to assess the generalisation ability and training times of the three entity classes and compare them to those of the single FFNN. Also, this chapter contains a demonstration of the ease with which an entity's training process can be parallelised and distributed among different processors in a very efficient fashion. Appendix F on page 189

contains a reference to the *np* script language[2] in which all simulations and test programs were written. Appendix G on page 221 contains all the scripts written for the purposes of obtaining these empirical results.

In conclusion, chapter 7 recapitulates on the requirements for this work and assesses the degree to which they were met. It also suggests possible directions for future research.

[2] *np* is a scripting language which was specifically designed for the purposes of integrating FFNN code. This consists of constructing, training and testing single FFNN and entities.

CHAPTER II

CONNECTIONISM

Martyrs of the dark ages, partisans of a neural ideology, prophets of a neural religion: they preach neural knowledge, baptise and admit mortals into the depth of neural apocrypha ...

2.1 Connectionism: raison d'être

2.1.1 Symbolic AI

Artificial intelligence (AI) is concerned with how to make machines behave in an intelligent manner. The most popular approach to AI for most of its (early) history has been the thesis that a machine which produces an evolving collection of unambiguous symbol tokens and manipulates them according to precise rules. What Simon and Newell, [Newell, 1980], call the *Physical Symbol System* is the necessary and sufficient means for general intelligence of the kind exhibited by humans.

Soon it became clear in the minds of the people pursuing this kind of intelligence that machines based on the *Physical Symbol System* hypothesis were too rigid and inflexible to function well outside the domains for which they were built. Paradoxically, today there are so many systems and machines that can compete with experts (e.g. expert systems applied to medicine) or highly trained workers (e.g. robots manufacturing automobiles or electronic boards) but none are capable of functioning as a shop assistant or solving puzzles that even a child can.

2.1.2 What is wrong with symbols?

On the one hand, knowledge representation and encoding based on unambiguous symbols and structures such as *frames* [Minsky, 1975], *schemata* [Rumelhart, 1975] and *scripts* [Schank, 1976], provide successful means for storing information but fail to allow for interaction between the fragments of knowledge they represent:

> " ... any theory that tries to account for human knowledge using script-like knowledge structures will have to allow them to interact with each other to capture the generative capacity of human understanding in novel situations. Achieving such interactions has been one of the greatest difficulties associated with implementing models that really think generatively using script- or frame-like representations." [McClelland et al., 1986, p.9]

On the other hand, machines operating on the symbol-oriented principle require the knowledge of the rules that apply **precisely** in the *world* where they are trying to *operate* in[1]. This is fairly straight-forward within small and isolated domains, but when we move to the real world and deal with problems without formally bounded domains, we find that not only it is difficult and even impossible to encode the many exceptions to our rules, but also that the number of these rules grows so large that the method soon proves to be computationally intractable causing processing bottlenecks and finally failure.

Rule-based systems, no matter how good they are in their restricted domains, simply can not deal with tasks which are not specified in precise, mathematically tractable ways and, generally, require the simultaneous consideration of many pieces of information which may be ambiguous, inconsistent and incomplete:

> " These models have rules which reliably work – so long as we stay in that special domain ... Inside such simple 'toy' domains, a rule may seem to be quite 'general' but whenever we broaden those domains, we find more and more exceptions – and the early advantage of context-free rules then mutates into strong limitations." [Minsky, 1990]

2.1.3 Reductionism versus Emergence

Premised on the principle that understanding a complex object requires to break it into component parts which are examined individually and then the results of these examinations are added together, *Reductionism* is a keystone of the Scientific Method:

> " ... to divide each problem I examined into as many parts as was feasible, and as was requisite for its better solution ... to direct my thoughts in an orderly way; beginning with the simplest objects, those most apt to be known, and ascending little by little, in steps as it were, to the knowledge

[1] The marriage of fuzzy logic with the symbol-oriented principle was a first step towards dealing with the weaknesses of rule-based systems.

> of the most complex; and establishing an order in thought even when the objects had no natural priority one to another." [Descartes, 1637]

The symbolic approach to AI is fundamentally based on **reductionist** principles. It looks at things *top-down* and develops its strategies within this *analytical*[2] framework: at the beginning, it assumes an outline of the tasks to be accomplished and, later, detail and function are added. Finally, problems encountered during the execution of these tasks are dealt with by breaking down the original tasks. In effect, "intelligence" is centralised not only because everything has to operate according to a "master plan" but, more importantly, because the responsibility for devising and implementing such a plan is solely assumed by a **supreme authority**, a **final arbiter**, a **homunculus** who puts all the inputs together: the **central processor**.

Emergence, on the other hand, claims that interesting and complex organisational models can arise from the agglomeration of simple units which do not individually exhibit any of the properties of the model as a whole. This may be achieved by implementing low level functions in the hope that, by some kind of alchemical magic, they will produce the higher level tasks required.

Emergence not only constitutes a political statement, a kind of nihilist doctrine which explicitly rejects authoritarian power structures (e.g. reductionism's central control) for the sake of distributed control, but also demolishes fundamental assumptions which have bolstered scientific thought for hundreds of years:

> " If the properties of matter and energy at any given level of organisation can not be explained by the properties of the underlying levels, it follows that biology can not be reduced to physics or anthropology to biology." [Landa, 1992]

This *emergent*, *bottom-up* or *synthesis*[3] methodology was given the name of **Connectionism** and, since its genesis, has been challenging the hegemony of "good old-fashioned" AI.

2.2 Historical

Nowadays, as the limitations of single processor, *von Neumann* architectures are becoming obvious, it is a widespread belief that further development of science and

[2] From the Greek word $\alpha\nu\alpha\lambda\acute{\upsilon}\epsilon\iota\nu$; to break, to untie, to undo.

[3] From the Greek word $\sigma\upsilon\nu\theta\grave{\epsilon}\tau\epsilon\iota\nu$; to put together.

technology will depend on establishing alternative computational models which will overcome these difficulties. Several attempts are being made, for example: molecular computing, quantum computing, DNA computing and neural computing.

In the 1940's, in a similar quest for alternative computational methods, a joint effort of **biology**, **cognitive studies** and **engineering** had laid down the foundations of connectionism.

In 1943, McCulloch and Pitts [McCulloch and Pitts, 1943] proposed a model for a nerve cell using a *threshold device*[4]. They showed that such a collection of artificial neurons[5] was capable of calculating certain logical functions.

In 1949, Donald Hebb [Hebb, 1949] pointed the significance of the connections between synapses in the process of learning and developed a very basic learning rule: the synaptic strength between two neurons is increased if both cells are activated at the same time. His findings were directly related to the *behaviorist* and *associationist* points of view, both of which had formed prominent traditions in the history of psychology. *Associationism* in particular is at least as old as Aristotle who proposed a linkage mechanism between memories effected by temporal succession or by "something similar, or opposite or neighbouring", [Anderson and Rosenfeld, 1988].

Hebb was probably inspired by an experiment conducted by Ivan Pavlov. Following the observation that dogs naturally salivate when they see food, Pavlov was ringing a bell whenever he was feeding his dogs. After some time, he observed that the sound of the bell alone was enough to make the dogs salivate. A possible explanation is that a part of the dog's brain, **F**, becomes active when food is seen and, in turn, stimulates **S**, the part responsible for producing saliva. At the same time, another part of the brain, **B**, is activated by the sounds of a bell. Because **S** and **B** are active at the same time, the synaptic strength between them is reinforced and, hence, the influence of **B** on **S**'s activity increases. If this practice continues for some time, **B**'s activity alone will be enough to activate **S**: the dog salivates with a stimulus other than food, e.g. the sounds of a bell!

Of course, this theory does not exclude the possibility that the synaptic strength between **F** and **S** is increased too and, as a result, the dog, will experience hallucinations of food images induced by the sounds of the bell!

---

[4] A device with a binary output whose state depends on whether the sum of its inputs is above or below a predefined threshold level.

[5] McCulloch and Pitts have also introduced this neuro-euphemistic nomenclature which has, ever since, never ceased to create controversy and passion within the circles of academia due to its certain rhetorical appeal.

In 1958, F. Rosenblatt [Rosenblatt, 1962], putting together the ideas of Hebb, McCulloch and Pitts, described the first operational neural network model, the *Perceptron.* He further demonstrated the ability of perceptrons to calculate logical functions by arranging the neurons in a particular topology and modifying the synaptic connections appropriately. Following this initial success, many wild claims were made by Rosenblatt and others about the potential of perceptrons as all-powerful learning devices.

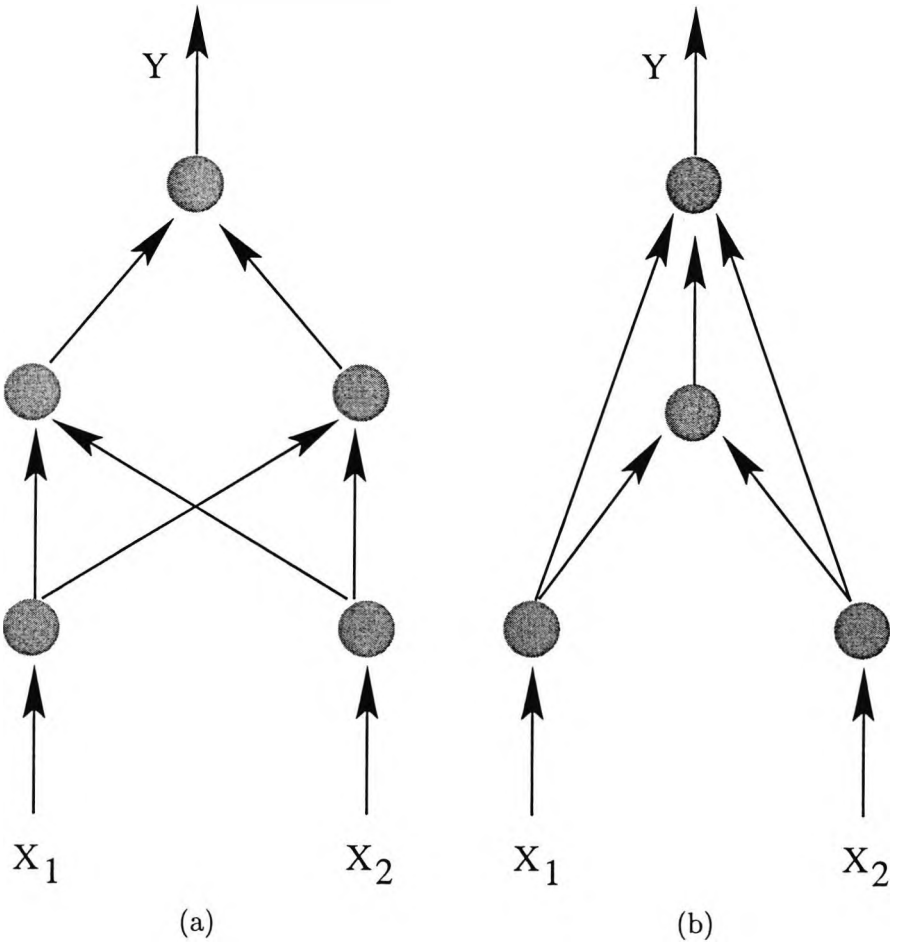

Figure 2.1: These are two basic three-layer neural network configurations which can memorise the XOR truth table

The massive enthusiasm driven by delirious journalists and ambitious scientists did not last long. In 1969, Marvin Minsky and Seymour Papert brought to light the limitations of the perceptron[6]. Their book, entitled *Perceptrons* [Minsky and Papert, 1969], was a neat hatchet job by the leaders of the symbolic-oriented community of AI on the connectionist school, and an excessively pessimistic one too. Further research in

[6] By *perceptron* (also *two-layer perceptron* as opposed to *multi-layer* perceptron) we shall mean a neural network consisting of a single threshold output unit connected to the input layer via a single layer of modifiable connections.

connectionist systems and neural networks was then characterised as *sterile* and as a result almost every activity in the field paused for fifteen years and scientists found it almost impossible to receive funding.

Their critique was unjustifiably cruel because it was mainly based on the inadequacy of the two-layer perceptron to solve the XOR problem[7], a classical problem of linear inseparability. It was known, even then, that this problem could be solved by incorporating another layer of computational units to the two-layer model and adding non-linearities at the output of the neurons. Although a learning rule for the *multi-layer* perceptron had not yet been discovered, Minsky and Papert conjectured that such a rule would be impractical, in principle, due to the combinatorial complexity of the calculations involved. Today, we know, that combinatorial complexity becomes an obstacle when the number of adjustable network parameters becomes very large; in fact four or five orders of magnitude larger than the mere six weights[8] required by a basic three-layer neural network (see for example figures 2.1(a) and 2.1(b)) which successfully solves the XOR problem.

Despite the fact that Minsky and Papert made some points which were impartial and, later, proved to be true, e.g. scaling problems[9], years later, Seymour Papert had admitted that their reasons for writing *Perceptrons* had not been entirely scientific – there was some other secret agenda:

> " Once upon a time two daughter sciences were born to the new science of cybernetics. One sister was natural, with features inherited from the study of the brain, from the way nature does things. The other was artificial, related from the beginning to the use of computers. Each of the sister sciences tried to build models of intelligence, but from very different materials. The natural sister built models (called neural networks) out of mathematically purified neurons. The artificial sister built her models out of computer programs.
>
> In their first bloom of youth the two were equally successful and equally pursued by suitors from other fields of knowledge. They got on very well to-

[7] The XOR problem involves learning a propositional logic relation like *A or B but not both.*

[8] E.g. a fully connected network of two inputs, two hidden units and one output. Alternatively, if direct connections from the input layer to the output layer units are permitted, the XOR truth table can be learned by a three layer network with four units and five weights (two units in the input layer and one unit in the hidden and output layers. The units of the input layer are connected to the hidden unit as well as to the output unit).

[9] A problem which their symbolic models were experiencing too.

> gether. Their relationship changed in the early sixties when a new monarch appeared, one with the largest coffers ever seen in the kingdom of the sciences: Lord DARPA, the Defence Department's Advanced Research Projects Agency. The artificial sister grew jealous and was determined to keep for herself the access to Lord DARPA's research funds. The natural sister would have to be slain.
>
> The bloody work was attempted by two staunch followers of the artificial sister, Marvin Minsky and Seymour Papert, cast in the role of the huntsman sent to slay Snow White and bring back her heart as proof of the dead. Their weapon was not the dagger but the mightier pen, from which came a book – Perceptrons – purporting to prove that neural nets could never fill their promise of building models of mind: only computer programs could do this. Victory seemed assured for the artificial sister. And indeed, for the next decade all the rewards of the kingdom came to her progeny, of which the family of expert systems did best in fame and fortune.
>
> But Snow White was not dead. What Minsky and Papert had shown the world as proof was not the heart of the princess; it was the heart of a pig." [Papert, 1988, p.3]

Fortunately, the rediscovery[10] of the *back propagation* learning rule for multi-layer Perceptrons by Rumelhart and the Parallel Distributed Processing group has initiated new activity in the area of neural networks, [Rumelhart et al., 1986].

As researchers began to realise that there is a significant difference between the capabilities of two-layer and multi-layer perceptrons (with non-linear activations), especially after the establishment of the latter as a *universal function approximator*[11], neural networks were applied to a variety of problems as diverse as *pattern recognition* [Bishop, 1995], *optimisation* and *control* [McKelvey, 1992], *medical prognosis* and *survival prediction* [Ohno-Machado and Musen, 1996], *financial forecasting* [Gately, 1996], *earthquake prediction* [Hadjiprocopis et al., 1994] with success.

The hype starts again! This time as a neural "gold-rush" which severely alters the scientific demography. Suddenly victims of Minsky's and Papert's diatribe appear everywhere; martyrs of the dark ages, partisans of a neural ideology, prophets of a neural religion: they preach neural knowledge, baptise and admit mortals into the depth of neural apocrypha. The golden age of neural empiricism is about to start.

[10] The actual discovery of back propagation is attributed to P. Werbos, [Werbos, 1974].

[11] See section 3.3 on page 24 for more details.

2.3 Connectionist learning

2.3.1 Introduction

Symbolic or rule-based systems use explicit sets of rules and symbol tokens in computations which are, basically, of a high enough level to allow them to be concerned more with algorithms and programs and less with hardware. Connectionism, on the other hand, concerns itself primarily with issues of topology and architecture:

> " ... one answer, perhaps the classic one we might expect from artificial intelligence, is 'software'. If we only had the right computer program, the argument goes, we might be able to capture the fluidity and adaptability of human information processing.
>
> Certainly this answer is partially correct. They have been great breakthroughs in our understanding of cognition as the result of the development of expressive high-level computer languages and powerful algorithms ... However, we do not think that software is the whole story.
>
> In our view, people are smarter than today's computers because the brain employs a basic computational architecture that is more suited to deal with a central aspect of the natural information processing tasks that people are so good at." [McClelland et al., 1986, p.3]

The connectionist approach to AI is based on the idea that intelligence emerges through local interactions of a large number of simple processing units that produce significant global properties – a notion epitomised by the organisation of the brain. Unlike a rule-based system, a connectionist network requires **no final arbiter**, **no central control**, **no homunculus** to put all the inputs together and produce the output. Rather, the output is an **emergent** property of the system as a whole, produced by independent, local decisions/computations and, indeed like an ant colony or a society of bees, *the potential of a connectionist system exceeds the mere sum of the potential of its parts*[12].

[12] Some people would argue that the same apply to other, non-connectionist systems. Let us see if this is true by way of an example. Consider the case of an integrated circuit made up of simple transistors. It is true that the potential of the circuit is greater than the sum of the potential of its parts. However, when one of the billion transistors making up the circuit breaks down, the whole integrated circuit can not operate any more. The potential of the system is now zero! On the other hand, the death of a bee or even the death the queen bee has minimal effect on the society of bees as a whole. Thanks to *decentralisation* the potential potential of the connectionist system still exceeds the potential of its parts.

2.3.2 A new computational architecture

A connectionist, or neural, network consists of a large number of simple and interconnected processing units, the neurons, which send and receive signals amongst themselves as well as with the outside world. The inputs to a neuron are mapped to a single output, usually via a non-linear function and are propagated to other neurons via connections of variable strength, the weights.

They come in many different varieties and flavours, each of which has its own merits and, also, demerits. Sometimes, as in the case of *feed-forward neural networks*[13], learning is *supervised* in the sense that the network expects a "correct" answer from a "teacher" so as to direct its own responses towards it while no feedback is allowed and, therefore, the propagation of signals between the neurons has always one and the same direction. Some other times, as in the case of the *Hopfield* net for example, feedback plays an essential role in its operation, while *self-organised maps* do not require any external supervision during the learning process.

Despite its implementation details, every connectionist system has some interesting properties arising from the **distributive** nature of its architecture, its **massively parallel processing** capabilities and its (*superficial*) resemblance to the human brain:

- **Distributivity**: Knowledge is represented and manipulated not by symbols contained in predetermined structures (frames, schemata, scripts, etc.) but by distributing it to the various units of the system following an internal representation determined by the learning process itself – thus, this process has an implicit character as opposed to the explicit nature of the symbolic approach. "Learning" is not centrally controlled[14] but, instead, is accomplished synergetically by all the units of the network.

- **Robustness**: Damage to a part of a connection machine is, generally, not critical to its entire performance. This quality of plasticity resembles the human brain's ability to recover from damage. In contrast, even a slight damage to the list of instructions of a computer program or excision of an entry from a database tends to be disastrous.

- **Graceful degradation**: Another facet of **robustness**. It refers to the ability of neural networks to perform even when either the input (e.g. partial information)

[13] This thesis is only concerned with *feed-forward neural networks* – we will examine them in more detail in chapter 3 on page 19.

[14] Apart from setting some basic interaction protocol, known as the training algorithm.

or the system are degraded in some manner.

- **General-purpose modelling**: As general-purpose parametrisable devices, neural networks are widely used to express analytically the behaviour of a system (specified by a finite number of observations) and, hence, build a theoretical model which describes this system.

- **Adaptive interpolation, Generalisation, Abstraction**: With reference to the modelling property mentioned above, a neural network is good at abstracting patterns from data. When presented with an unknown input it will perform some kind of interpolation based on the learned abstractions and produce a likely output.

- **Constructivist and Nativist at the same time**: Connectionism generally assumes and operates within a Piagietian/constructivist framework of learning where knowledge is built by the learner, not transmitted by the teacher. For example, before learning begins, the weights are initialised to random values. However, many implementations of connectionist systems do not assume an entirely *tabula rasa*, but instead, they incorporate prior knowledge (nativism) to it, be it in the form of biases to neurons or just by arranging the neurons into a certain topology. The contribution of this innate knowledge to the overall learning process can be controlled and, therefore, its effects can be evaluated in practice.

2.4 Connectionism: what is it really worth?

Karl Marx in his seminal work *Das Capital* argues that commodities are "something twofold, both, objects of utility, and, at the same time, depositories of value", [Marx, 1887, Chapter 1, Section 3]. Perhaps it is not only our personal belief but is shared by many others that Connectionism and neural networks are such "commodities" holding the two-fold property:

- *Objects of utility*, on the one hand, worth no more and no less than what their existence theorems state in a mathematical language which most of the time can be irreversibly precise.

- *Depositories of value*, on the other hand, are worth anything their advocates state.

Minsky and Papert in 1969, claimed that connectionism was an unfortunate dead end, a romantic attempt to mimic the infinitely more complex human brain and, hence,

bound to fail. They wished connectionism's death and so it happened. Minsky and Papert were aware, we claim, of the exact value of connectionism as far as *utility* was concerned, but it was in their interest, or, perhaps in the wider scientific community's interest that connectionism was slain, buried and forgotten. Obviously, they wanted Lord DARPA's research funds all for themselves (the symbolic-oriented AI community). And so they had to show us, people who – unfortunately – are usually concerned with "commodities" as *depositories of value*, that the value of this particular commodity was nil, despite its *utility value.* In Seymour Papert's own words, "What Minsky and Papert had shown the world as proof was not the heart of the princess; it was the heart of a pig".

The excitement in the field of AI, which peaked in the eighties with the advent of expert systems as hundreds of millions of venture capital invested into companies with such emblematic names as *Symbolics* and *LISP machines* or Japan's infamous *Fifth Generation* project, had soon run out of fuel. By 1990 came a shake-out, as even the most ardent proponents of symbolic AI had to admit that their models, for all their Turing equivalence, could not lead us to the holy grail of computer science, the humanly thinking computer, HAL[15].

It was then remembered – **ex machina deo** – that some time in the past there has been an alternative model for intelligence: Connectionism. Although as an *object of utility*, connectionism is not worth much more than it was back in 1969 or 1974, from a *depository of value* point of view, its virtues and potentials are now highly stressed and even exaggerated. Again, there is a lot of vacuous prattle going on, in a motif borrowed from the fifties. This time, however, the Minskys and Paperts of our story are much more and so are the research funds, as Lord DARPA's cousins in every corner of the world are willing to engage in ambitious projects in order not to be left behind in the strategic quest for silicon intelligence.

With one of the sister sciences on early retirement, the second sister will get a chance, for the next five, ten years or, eventually, until its *deposited value* deflates.

[15] In S. Kubrick's *2001*, HAL was activated on January 12, 1992.

CHAPTER III

FEED FORWARD NEURAL NETWORKS

Where an important category of connectionist systems called feed forward neural networks – the main concern of this thesis – is examined.

3.1 Introduction

An important category of connectionist architectures goes under the name of *feed forward neural networks* (FFNN). These are systems where training is exercised through a *supervised* process by which the network is presented with a sequence of input vectors and the respective desired output responses. "Learning" occurs by adjusting the free parameters of the network in a way that the total discrepancy between the desired and obtained outputs is minimised for all the training vectors.

FFNN are made of *neurons* – simple, linear or non-linear[1] computing elements whose basic function is to sum up all their inputs and pass the result to other neurons. They are arranged in layers, one next to the other, and the layers are arranged one after the other. Neurons belonging to the same layer receive inputs from neurons of the previous layer(s) and send their output to neurons of the next layer(s). The strength of the connection between two neurons is called a *weight*. The state of the set of all the weights determines completely the behaviour of the network. A formal description of FFNN and the neuron can be found in section 3.2.

A lot has been said about the brain-like properties of FFNN, for example learning and generalisation. However, the true value of FFNN arises from its *universal function approximation* property: a FFNN with a single hidden layer and employing as many hidden

[1] On deciding between a linear or a non-linear activation function, one must take account of the fact that any superposition and/or composition of linear functions is itself linear and can, therefore, express only linear functions. Thus, a FFNN may not consist entirely of linear activations if it is necessary to approximate non-linear functions.

units (with non-linear activations) as required can approximate any real, continuous function arbitrarily well. Unfortunately, the practical importance of this statement is limited as the requirements for realising this structure, e.g. number of hidden units and methods for selecting optimum values for the set of weights, are not fully quantified.

In section 3.3 we describe Hilbert's 13^{th} problem, its resolution by Kolmogorov and the subsequent improvements by Lorenz and Sprecher leading to a theorem which guarantees (but does not define) the existence of a neural network which can represent **exactly** any real, continuous function in the n-dimensional unit hypercube (e.g. $[0,1]^n$). Finally, we look at **approximate representations** of functions using neural networks since the utilitarian value of Kolmogorov's theorem is minimal.

The process of adjusting the weights, e.g. training, so that the discrepancy between expected and obtained output values is minimised can be very complex, not only because the number of weights can get very large but also because there are several training examples, which in many instances can be contradicting. The most widely used training method, the *back-propagation* or *generalised delta rule*, is based on the original *delta* or *least-mean-squares* rule introduced by Widrow and Hoff, [Widrow and Hoff, 1960], and applied to the *adaptive linear element* (Adaline) perceptron model.

We will have a brief look at back-propagation in section 3.4. Also, appendix A on page 167 contains a derivation of the back-propagation equations.

3.2 FFNN Formalism

Figure 3.1 on the facing page shows an example of a generic FFNN[2] with L layers (including the input and output layers), n inputs and m outputs. This topology is associated with the general mapping $\Phi : \mathbb{R}^n \mapsto \mathbb{R}^m$. The notation we will be using in this and subsequent sections follows:

- the input vector to a FFNN is denoted as $\underline{\mathbf{X}}$, the output vector as $\underline{\mathbf{Y}}$ and the **target** output vector as $\underline{\mathbf{T}}$,
- the input to the i^{th} unit of the l^{th} layer is a vector, denoted by $\underline{\mathbf{x}}_i^l$ while the output of that unit as y_i^l (a scalar),
- the set of the output values of all the units of the l^{th} layer is denoted by the

[2] According to the following definitions, Radial Basis Functions (RBF) classifiers could belong to the family of FFNN too if it were not for definition 3.1 which requires that the kernel functions be **monotonic** – something that RBF's gaussian kernels obviously lack. However, if the requirement of monotonicity is relaxed then RBF neural networks are covered by these definitions too.

vector $\underline{\mathbf{y}}^l$,

- The set of all the weight values of a FFNN is denoted by the matrix $\underline{\mathbf{W}}$,
- the vector $\underline{\mathbf{w}}_j^l$ holds the weights of the connections between the j^{th} unit of the l^{th} layer and all the units of the previous layer, $l-1$,
- w_{ji}^l denotes the weight (a scalar) of the connection between the j^{th} unit of the l^{th} layer and the i^{th} unit of the previous layer, $l-1$,
- the number of units of the l^{th} layer is given by $u(l)$.

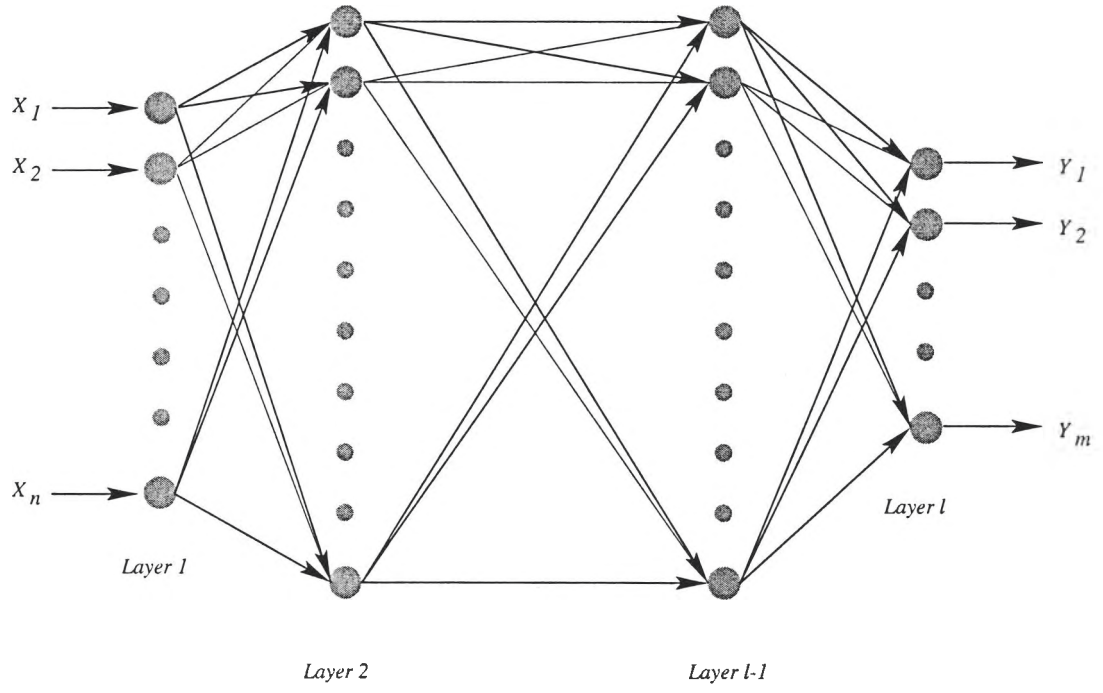

Figure 3.1: A generic FFNN topology with L layers mapping $\mathbb{R}^n \mapsto \mathbb{R}^m$

3.2.1 Operation of the neuron

Firstly, let us define the operation of a single neuron associated with:

- a row vector of adjustable parameters, the weights, $\underline{\mathbf{w}} = (w_1 \;\; w_2 \;\; \ldots \;\; w_p) \in \mathbb{R}^p$,
- a column vector of incoming signals, $\underline{\mathbf{x}} = (x_1 \;\; x_2 \;\; \ldots \;\; x_p) \in \mathbb{R}^p$,
- a bias value, $b \in \mathbb{R}$,
- an activation function, $\sigma(\cdot)$.

The operation of a neuron consists of:

- the weighted sum of the incoming signals is calculated as $\underline{\mathbf{w}} \cdot \underline{\mathbf{x}} = \sum_i w_i x_i$,
- the bias value, b, is added to the above sum,
- the output is calculated as $y = \sigma(\underline{\mathbf{w}} \cdot \underline{\mathbf{x}} + b)$, $\sigma \in \mathcal{S}$ (see definition 3.1, below).

The above procedure applies to all the neurons of a FFNN with the notable exception of the first layer units whose role is limited to distributing the input vector, $\underline{\mathbf{X}}$, to the next layer. The activation function, $\sigma(\cdot)$, is usually chosen to belong to the family of sigmoids $\mathcal{S}$ (see definition 3.1) with, again, the possible exception of the output layer units which might employ any activation function which is compatible with the statistical properties of the output vectors. For the sake of generality we will not assume that the activation for each neuron is the same. Instead, the i^{th} neuron of the l^{th} layer employs the activation function σ_i^l.

DEFINITION 3.1 *Let $\mathcal{S}$ be the family of all functions which are* **monotonic, increasing, differentiable** *and* **bounded** (e.g. $\mathbb{R} \mapsto [0, 1]$) in $\mathbb{R}$.

DEFINITION 3.2 *Let $\mathcal{A}^p$ be the family of all affine transforms in $\mathbb{R}^p$:*

$$\begin{aligned} \mathcal{A}^p = \{A : \mathbb{R}^p \mapsto \mathbb{R} | A(\underline{\mathbf{x}}) &= \underline{\mathbf{w}} \cdot \underline{\mathbf{x}} + b \\ &= \sum_{i=1}^{p} w_i x_i + b, \quad b \in \mathbb{R} \text{ and } \underline{\mathbf{x}}, \underline{\mathbf{w}} \in \mathbb{R}^p\} \end{aligned}$$

DEFINITION 3.3 *Let $\mathcal{N}^p$ be the family of functions implemented by a hidden layer neuron:*

$$\mathcal{N}^p = \{N : \mathbb{R}^p \mapsto \mathbb{R} | N(\underline{\mathbf{x}}) = \sigma(A(\underline{\mathbf{x}})), \ \underline{\mathbf{x}} \in \mathbb{R}^p, \ A \in \mathcal{A}^p \text{ and } \sigma \in \mathcal{S}\}$$

Using the above definition, the output of the j^{th} unit of the l^{th} layer ($1 < l < L$) as a function of the outputs of all the neurons of the previous layer, $\underline{\mathbf{y}}^{l-1}$, and the weights of all the connections with the units of the previous layer, $\underline{\mathbf{w}}_j^l$, is:

$$y_j^l = N(\underline{\mathbf{y}}^{l-1}) = \sigma_j^l(A_j^l(\underline{\mathbf{y}}^{l-1})) = \sigma_j^l(\underline{\mathbf{w}}_j^l \cdot \underline{\mathbf{y}}^{l-1} + b_j^l)$$

where, $A_j^l(\cdot)$ and $\sigma_j^l(\cdot)$ are the affine transform and activation function associated with the j^{th} unit of the l^{th} layer.

3.2.2 Operation of the FFNN

The output of the FFNN corresponding to a given input $\underline{\mathbf{X}}$ is the vector:

$$\underline{\mathbf{Y}} = \{\sigma_1^L(A_1^L(\underline{\mathbf{y}}^{L-1})) \ \ \sigma_2^L(A_2^L(\underline{\mathbf{y}}^{L-1})) \ \ \ldots \ \ \sigma_m^L(A_m^L(\underline{\mathbf{y}}^{L-1}))\} = \underline{\mathbf{y}}^L \tag{3.1}$$

However, for simplification purposes, we can assume a FFNN with a **single, linear output unit** and a **single** hidden layer with **non-linear** units, as depicted in figure 3.2. As we shall see later, such a single hidden layer FFNN with an *arbitrary number of hidden layer units* is sufficiently complex for *universal function approximation.*

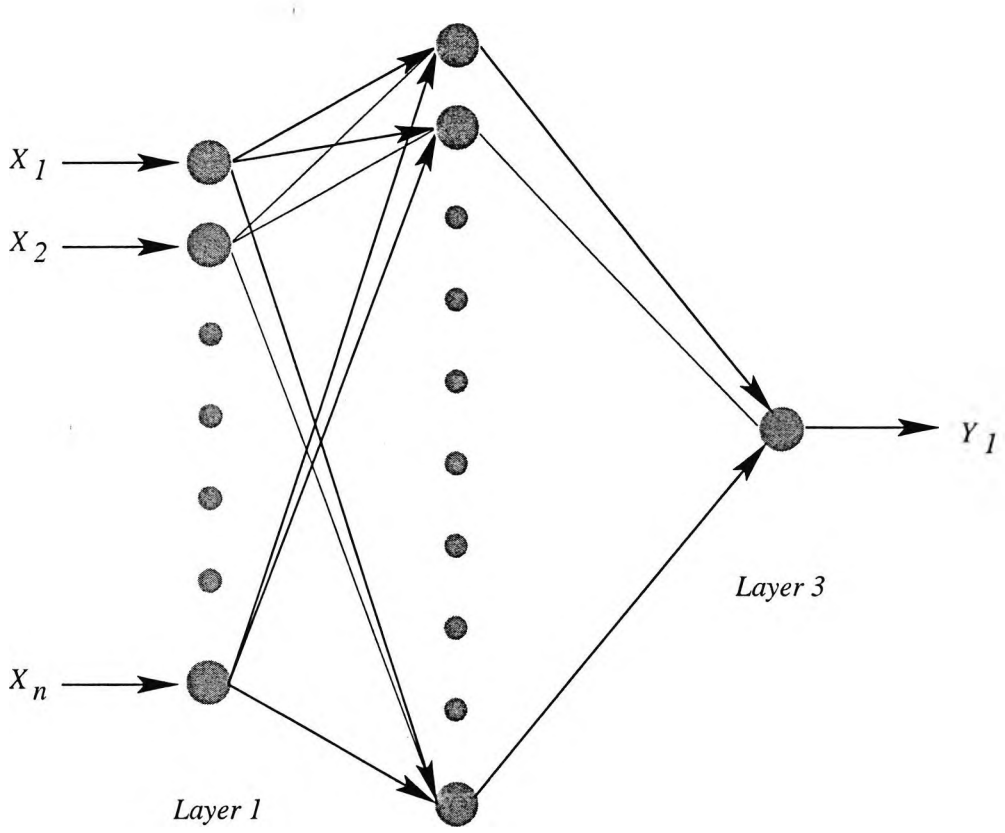

Figure 3.2: An FFNN with a single hidden layer, mapping $\mathbb{R}^n \mapsto \mathbb{R}$

DEFINITION 3.4 *Let $\mathcal{G}^n$ be the family of all the functions that a* FFNN *with n inputs, a single hidden layer of q non-linear units and a single, linear output can implement:*

$$\begin{aligned} \mathcal{G}^n = \{g_n : \mathbb{R}^n \mapsto \mathbb{R} | g_n(\underline{\mathbf{x}}) &= A_1^3(\underline{\mathbf{N}}^2(\underline{\mathbf{x}})) \\ &= b_1^3 + \sum_{i=1}^{q} w_{i1}^3 \, \sigma_i^2(\sum_{j=1}^{n} w_{ji}^2 x_j + b_i^2), \end{aligned} \tag{3.2}$$

where, $\underline{\mathbf{x}} \in \mathbb{R}^n$, $A_1^3 \in \mathcal{A}^q$, $N_i^2 \in \mathcal{N}^n$ and $\underline{\mathbf{N}}^2(\underline{\mathbf{x}}) = (N_1^2(\underline{\mathbf{x}}) \; N_2^2(\underline{\mathbf{x}}) \; \cdots \; N_q^2(\underline{\mathbf{x}}))\}$

3.3 FFNN as Universal Function Approximators

In 1900, David Hilbert as a new Eurystheus had set a number of "labours" to the many young Hercules, the mathematicians of the twentieth century. At a lecture delivered before the International Congress of Mathematicians in Paris he challenged the present and future scientific community with *twenty three* mathematical problems in the hope "... to cast a glance at the next advances of our science and at the secrets of its development during future centuries", [Hilbert, 1902].

The 13^{th} problem, namely the *Impossibility of the solution of the general equations of the* 7^{th} *degree by means of functions of only two arguments*, can be generalised to the problem of trying to represent any real, continuous function of n arguments by superposition of compositions of as many as required functions of $m \ll n$ arguments.

In the mid to late 1950's A. Kolmogorov and V.I. Arnold in a succession of papers "fought" a battle of who was going to be first to settle Hilbert's 13^{th} challenge. Eventually, Kolmogorov won. In 1957, he published an important theorem concerning the approximation of arbitrary continuous functions, $f : [0,1]^n \mapsto \mathbb{R}$, in terms of functions of a single variable, [Kolmogorov, 1957]:

THEOREM 3.1 *Any continuous function, f, of n variables, $(x_1 x_2 \dots x_n) \in [0,1]^n$, can be characterised completely by finite superposition of compositions of functions of one variable as*:

$$f(x_1, x_2, \dots, x_n) = \sum_{j=1}^{2n+1} g_j(\sum_{i=1}^{n} \psi_{ij}(x_i)) \tag{3.3}$$

where the g_j's and ψ_{ij}'s are continuous functions (of one variable). Furthermore, the ψ_{ij}'s are **fixed, monotonic** and **increasing** functions which **are not dependent** on f.

In 1965, D.A. Sprecher obtained an improvement over Kolmogorov's original theorem in the sense that the g_j's are replaced by a single g which is **real, continuous** and **does depend** on f and the ψ_{ij}'s are replaced by a single ψ which is **real, continuous, monotonic, increasing** and **independent** of f:

$$f(x_1, x_2, \dots, x_n) = \sum_{j=1}^{2n+1} g(\sum_{i=1}^{n} \lambda^i \psi(x_i + \epsilon(j-1)) + j - 1) \tag{3.4}$$

where ϵ is a positive, rational number and λ is a constant independent of f. Observe the presence of the "bias" terms $\epsilon(j-1)$ for the hidden layer units (e.g. the inner sum) and $j-1$ for the output unit (e.g. the outer sum).

Despite the fact that Sprecher's **exact representation** leads to a three-layer neural network, mapping[3], $f : [0,1]^n \mapsto \mathbb{R}$, the important issue of how to construct such a network is not dealt with. In general, Kolmogorov's original theorem, as well as all the improvements followed that, are concerned with the *existence* of representations of a function by different univariate functions but are not addressing at all the practical issues of the *construction* of such a composition. For example, the functions g and ψ are highly non-smooth and virtually unknown. Furthermore, according to Poggio and Girosi, "g is at least as complex, in terms of bits needed to represent it, as f", [Poggio and Girosi, 1989, p. 7].

In view of all the practical difficulties associated with an *exact* decomposition of f, research had been directed towards *approximate* representations which seemed, and indeed were, more promising from a practical point of view. In recent years, several scientists, [Cybenko, 1989], [Funahashi, 1989], [Hornik et al., 1992], [Kůrková, 1991] and others, attempted, successfully, to **approximate**, with arbitrary precision, the general real, continuous n-dimensional function, $f(x_1, x_2, \ldots, x_n)$, by finite linear combinations of non-linearities[4]:

$$\sum_{i=1}^{q} \lambda_i \sigma(\underline{\mathbf{w}}_i \cdot \underline{\mathbf{x}} + b_i) \tag{3.5}$$

where λ_i and b_i are constants $\in \mathbb{R}$, $\underline{\mathbf{w}}_i \in \mathbb{R}^n$, $\underline{\mathbf{x}} = (x_1 \;\; x_2 \;\; \ldots \;\; x_n) \in \mathbb{R}^n$, $\sigma \in \mathcal{S}$ (see definition 3.1 on page 22) and is fixed, in contrast to Kolmogorov's exact representation formula which uses different functions (the g_j's and ψ_{ij}'s).

3.4 The back-propagation of error training method

3.4.1 Introduction

The training of the two-layer perceptron as well as that of its multi-layer successor, the FFNN, is said to be **supervised** in the sense that a "teacher" defines what the network response to a given input stimulus **should be**. The modification of the adjustable parameters of the network aims at making the actual and desired network response coincide and is guided by the training algorithm, which in turn, makes its decisions based on the discrepancy between the obtained and expected responses.

---

[3] Use $\underline{\mathbf{f}} : [0,1]^n \mapsto \mathbb{R}^m$ when dealing with more than one output. In this case, $\underline{\mathbf{f}} = (f_1(x_1, x_2, \ldots, x_n), \ldots, f_m(x_1, x_2, \ldots, x_n)) \in \mathbb{R}^m$

[4] Observe that this is the equation describing a three-layer FFNN with n inputs, q hidden layer units employing sigmoidal activation functions and a single output as in figure 3.2 on page 23 and definition 3.4 on page 23.

The training algorithms employed by supervised learning neural networks are, largely, implementations of the *gradient descent* optimisation method which searches for the best combination of weight values by minimising the *mean squared error*[5] associated with the given network and a set of inputs and expected outputs.

We shall describe what gradient descent is in the next section, but firstly, let us define a measure of the discrepancy between the obtained and expected network responses by using the notion of *mean squared error*:

DEFINITION 3.5 *Given an* FFNN, *the set of its adjustable parameters,* $\underline{\mathbf{W}}$, *a set of* v *training pairs consisting of an input vector,* $\underline{\mathbf{X}} \in \mathbb{R}^n$, *and a target output vector,* $\underline{\mathbf{T}} \in \mathbb{R}^m$ *and the obtained output response,* $\underline{\mathbf{Y}} \in \mathbb{R}^m$, *a measure of the error associated with that* FFNN *is given by:*

$$\begin{aligned} E(\underline{\mathbf{W}}) &= ||\underline{\mathbf{T}} - \underline{\mathbf{Y}}||^2 \\ &= \frac{1}{2mv} \sum_{i=1}^{m} \sum_{j=1}^{v} (T_{ij} - Y_{ij})^2 \end{aligned} \tag{3.6}$$

The function $E(\underline{\mathbf{W}})$ defines a multi-dimensional **error surface**[6], for a given set of input, output and target vectors. For each combination of weight values, $\underline{\mathbf{W}}$, there is an error value proportional to the discrepancy between the obtained and expected (target) FFNN outputs. Training consists of finding the optimum set of weights[7], $\underline{\mathbf{W}}_o$, which gives an acceptably low error.

3.4.2 Gradient descent

Being on the top of a hill, the shortest route to a valley below, is by descending the steepest slope. Similarly, being anywhere on the error surface, $E(\underline{\mathbf{W}})$, the shortest route to a minimum, being either local or global, is in the direction of the steepest slope (gradient). Thus, FFNN learning rules based on the *gradient descent* method require that the weight vector changes in the direction pointed by the **negative gradient** of the mean squared error function. In this way, the error will decrease at the fastest possible

[5] "One problem is that there can not be any standard, universal way to measure errors, because each type of error has different costs in different situations. But let us set this issue aside and do what scientists often do when they can't think of anything better: sum the squares of the differences" [Minsky and Papert, 1988, Epilogue]

[6] This error surface has a dimensionality equal to the number of elements in $\underline{\mathbf{W}}$ plus one, the bias value. The dimensionality is, thus, equal to the number of adjustable network parameters.

[7] The set of adjustable network parameters includes the weights as well as the biases to each neuron's activation. For the sake of clarity we will not refer to the biases separately.

rate. Therefore, the *weight update* or *learning* rule is given by the following expression:

$$\underline{\mathbf{W}}(t+1) = \underline{\mathbf{W}}(t) - \beta \nabla_W E \tag{3.7}$$

where, $\underline{\mathbf{W}}(t)$ is the set of weights at time t, $\beta \in \mathbb{R}$ is a constant controlling the weight change[8] and $\nabla_W E$ is the *gradient* (also known as *curl*) of the error function, $E(\underline{\mathbf{W}})$ with respect to the weights.

3.4.3 Back-propagation

Back-propagation is a training algorithm inspired by the gradient descent optimisation method and adapted to the case of the many layers of weights and computational nodes of an FFNN. In fact, back-propagation is nothing more than just a particularly efficient method (thanks to *chain rule*) to compute $\nabla_W E$.

We have already talked about back-propagation's colourful history in section 2.2 on page 9. It is a method which is constantly modified and enhanced with little embellishments here and there. It is also a highly criticised algorithm because of its many inefficiencies and uncertainty of convergence, or actually, its convergence to minima of uncertain nature (e.g. local or global?). For a more detailed description of these criticisms, refer to sections 4.4.2 on page 38 and 4.4.3 on page 41.

None-the-less, it constitutes a very important discovery in the field of neural networks as it has opened up the avenues for the use of multi-layer networks, in the late 80's, and the revival of neuro-computing. Why it had to be re-invented and popularised by Rumelhart in 1986, when it was once proposed by Werbos, [Werbos, 1974], ten years earlier is another story ...

The main problem with equation 3.7 when applied to multi-layer perceptrons, lies in calculating the gradient of the error with regards to all the weights of the network, $\nabla_W E$. Consider equation 3.2 of definition 3.4 on page 23. The output of a FFNN, $g_n(\underline{\mathbf{X}})$, depends *directly* on the weights of the output layer, $\underline{\mathbf{w}}^3$, but *indirectly* on the weights of the hidden layer, $\underline{\mathbf{w}}^2$. "Indirectly" in the sense that the effect of $\underline{\mathbf{w}}^2$ upon g_n is through the activation function, σ. As a result, the calculation of $\nabla_W E$ becomes complicated.

Back-propagation works in three phases. The first phase consists of propagating the

[8] The role of this constant is crucial in reaching a solution: if it is large, it speeds up the convergence but the weights are changed in big steps and therefore a potential solution might be overlooked. In the scenario of descending the hill, β is equivalent to the size of our footstep. If too small, it will take us a long time to reach the valley, if too big, we might end up ascending the next hill, missing entirely the valley below us!

input vector through all the computational units until the output and, subsequently, the error values are obtained. In the second phase, the errors from the output layer are propagated (e.g. *back-propagated*) to the previous layers and the contribution of each weight to the error is calculated. There the effect of the current layer's weight values in the error is calculated. Finally, during the third phase, the weights are modified according to the findings of the second phase.

For a derivation of the back-propagation equations refer to Appendix A on page 167. Also, section 4.4 on page 37 explains in more depth the problems associated with the use of this training algorithm.

CHAPTER IV

CRITIQUE OF FFNN

That neural networks are a panacea, is a myth. In this chapter we describe the main points of the critique associated with the use of feed forward neural networks and give examples of where and how they are bound to fail.

4.1 Introduction

Ever since their appearance, neural networks have been in the center of heated arguments and controversy essentially because Connectionists had "audaciously" attempted to extend their networks from "processing waveforms or evaluating credit histories", [Anderson and Rosenfeld, 1988, p. 599], to trying to understand how the human mind works.

Following are some of the main criticisms expressed which are directly or indirectly relevant to scope of this work:

1. **Linear classifiers** (the perceptron for example) can implement discriminant functions (decision planes or *hyper-planes* in higher dimensions) which are only *linear*. Because this class of functions is very restricted and forms only a very small subset of the total number of all possible decision boundaries, linear classifiers are simply not powerful enough to be used in applications where it is important that an exact decision boundary is found. A classical demonstration of this is when the perceptron attempts to learn the XOR function (see appendix B on page 171).

   In section 4.2.2 we explain the facts mentioned above in detail and quantify the limitations of the perceptron. Note that these problems are associated only with the task of finding an *exact representation* for a given training set. In applications where we are primarily interested in good *generalisation* these concerns are largely irrelevant and the perceptron with its simple architecture and training process,

proved to be a useful and practical tool.

2. The addition of an *intermediate layer of processing units* (hidden layer) and the use of *non-linear activation functions* in the original perceptron design led to the non-linear, multi-layer perceptron (or FFNN in our terminology). The capabilities of such a multi-layer construction can be appreciated from the implication of a mathematical result, due to Kolmogorov, on the existence of exact representations of continuous functions by superpositions of one variable functions. See section 3.3 on page 24 for more details.

   Even so, the utilitarian value of Kolmogorov's theorem is *weakened* by some yet un-answered questions regarding the construction of the neural network[1] as well as by problems of extreme sensitivity to the input variables. For these reasons, researchers in the field of neural networks are no longer seeking **exact** representations but rather an **approximation** to the training data, with arbitrary accuracy. The latter is guaranteed by the *Stone-Weierstrass* theorem on *universal function approximation*. However, one must note that *universal function approximation* is one thing and *universal computation* (Turing) is another!

   A discussion about using neural networks for *exact representation* of functions as well as the universal function approximation property can be found in section 3.3 on page 24.

3. For the simpler case of linearly separable data, the perceptron training algorithm could find an optimal set of weights relatively fast. With the addition of another layer of processing units, however, many more weights have to be optimised. Consequently, training these networks becomes more difficult, especially as the number of input dimensions increases. Thus, one may ask whether, given a neural network and a set of training examples, there exists a set of weights for which the network produces the correct output for all the examples. This is usually referred to as the *loading problem* and it has been shown that it is NP-hard or NP-complete depending on the network topology and activation functions employed.

   This result is mainly of *theoretical* importance to the problem of efficient training of neural networks. In practice, the implications of the *loading problem* are not as dramatic since we are not as much interested in *memorisation* as we are in *generalisation* of the training set. This issue is discussed in section 4.3.

[1] For example, answering questions such as *how many hidden units* or *what activation functions* without resorting to rules of thumb.

4. The nature of the error surface and, in particular, the existence of local minima presents one of the biggest problems in training a neural network.

   On the one hand, the error surface depends not only on the network topology but also on the particular training set, cost function used and kernel type. On the other hand, back-propagation and, generally, *gradient descent*-based optimisation methods lack simple and effective mechanisms to avoid local minima[2].

   As the number of weights in a neural network increases (e.g. in order to deal with high-dimensional training data associated with "real world" problems – as opposed to "toy" problems) the error-weight space explodes exponentially in the number of dimensions due to the "*curse of dimensionality*"[3]. Thus, the search in this space for optimal weight values, with or without back-propagation, is seriously hindered. It has to be mentioned here that in some occasions the dimensionality of a problem can be reduced without serious loss of information by application of various *dimensionality reduction techniques*. For example, *Principal Components Analysis*, [Bishop, 1995, p. 180], is such a technique which has been widely used in reducing the dimensionality of the problem so that it is solved by a smaller neural network.

   In addition, the error surface becomes very nasty, in terms of *multi-modality*, *complexity*, *dimensionality* and *number of stationary points*, when networks are scaled up to tackle with problems of the "real world".

   Certainly, there are instances of error surfaces with no local minima. Those, however, require *linear separability* of the training patterns (which may be achieved by increasing the number of input variables, see section 4.2.2) or a huge number of hidden layer units which will reduce the generalisation ability of the network and hinder the training process even more.

   These problems are examined in sections 4.4.2 and 4.4.3.

5. A problem which is related to the neuron fan-in is that of *premature neuron saturation*. This refers to situations where the input signals to the hidden layer nodes are so high that the neurons are forced to produce an output response very close to the upper bound of their sigmoid activation. Saturation causes the neurons to lose their sensitivity to input signals, the propagation of information

[2] It is not accidental that most AI courses introduce the technique of *hill climbing* by the example of a *blind* person ascending a hill. Blind, indeed, is gradient descent!

[3] The *exponential* growth of the volume (of Euclidean space) as its dimensions increase is known as *curse of dimensionality*.

is blocked and, thus, sub-optimal solutions arise. This problem is discussed in section 4.4.4.

6. Although a FFNN is an inherently parallel computational system, it is quite difficult to obtain any practical benefits from this feature. The *fine-grain* parallelism involved and the nature of the back-propagation algorithm require intense communication between the processing elements (neurons). Thus, any advantage gained from parallelised operation will be lost in the face of vast communication overhead.

   The feasibility of parallel implementation as well as the main constraints on hardware execution of FFNN are discussed in section 4.5.

7. The current neural network architectures are criticised for their non-explicit nature of learning. The most common and convenient description of a neural network is that of a *black box* where intermediate learning steps are either unavailable or difficult to visualise. Section 4.6 refers to these criticisms.

The above criticisms will be discussed extensively in subsequent sections. The chapter ends with section 4.7 which contains a summary of the main points.

4.2 The power of linear classifiers

4.2.1 Order of a predicate and the perceptron

Among the first "neuro-sceptics" to doubt the universal applicability of neural networks were Minsky and Papert. In 1969 (see section 2.2 on page 9) they drew the line between what can and can not be learned by the neural network models of their time (e.g. the perceptron).

Their main contribution, [Minsky and Papert, 1969], is the concept of *order* associated with a given perceptron configuration. Order limits the predicates that are computable by a perceptron in the sense that a perceptron with a given order O can not be trained to compute any predicate whose order is greater than O. So, whenever a given problem is of low order, the perceptron performs well. However, for tasks with unbounded order, problems of *size* and *scaling* are encountered: in order to increase the likelihood for a perceptron to learn to compute high-order functions, the number of connections must be high.

The basic idea behind Minsky's and Papert's concept of *order* was not new. Essentially, it is a re-formulation of the problem where a classifier lacks the expressive power to im-

plement all decision boundaries a problem might require. A more thorough study of the power of linear classifiers (not only of the perceptron) was done by Cover, [Cover, 1965], in 1965. Later, Vapnik and Chervonenkis introduced the notion of the *growth function* associated with a classifier, while studying the relationship between relative frequencies of events and their probabilities, [Vapnik and Chervonenkis, 1971] (see appendix C on page 173 for more details). Finally, Valiant, [Valiant, 1994], formalised the learning problem, thus, allowing all these ideas to be combined in a unified framework, the *Probably Approximately Correct* (PAC) learning.

The essential difference, however, between Minsky's and Papert's *order* of a perceptron and the various other measures of the power of linear classifiers as described in the previous paragraph as well as in the next section, is that the former is described in a framework which ignores the fact that a perceptron is more a linear classifier than anything else. Had Minsky and Papert made their critique starting from the fact that a perceptron can only implement linear decision surfaces, they would have been far more constructive and, probably, have arrived to the solution of the problem with less controversy and friction.

4.2.2 Linear separability

Another limiting factor of a perceptron's classification ability is the requirement that its input patterns be *linearly separable*. A set of M-dimensional points belonging to two classes, A and B, is said to be *linearly separable* if there exists a hyper-plane (in M dimensions) which can form a decision boundary between points of class A and points of class B. Since a perceptron is basically a *linear threshold device*, e.g. it can implement only linear decision surfaces, linear separability of the input patterns is strictly necessary for successful (e.g. by 100 %) classification.

It is interesting to consider the fraction of the dichotomies[4] of N points in *general positions*[5] in M dimensions which are linearly separable. This is given by the following expression, [Cover, 1965]:

---

[4] The term *dichotomy* refers to each possible binary (e.g. only two classes, A and B) classification (labelling) of points in continuous M-dimensional space. For N such points, the total number of dichotomies is 2^N.

[5] N points are in **general positions** when there is no subset of M or fewer points which are **linearly dependent**. An equivalent definition of the term **general positions** is the following: when $N > M$, N points are in general positions in an M-dimensional space if and only if no $M + 1$ points lie on and $(M - 1)$-dimensional hyperplane. When $N \leq M$, N points are in general positions if no $(M - 2)$-dimensional hyperplane contains all the points. See for example appendix B on page 171.

$$P_{N,M} = \begin{cases} \frac{1}{2^{N-1}} \sum_{i=0}^{M} \binom{N-1}{i} & \text{for } N > M+1 \\ 1 & \text{for } N \leq M+1 \end{cases} \tag{4.1}$$

We can see from the above equation that when $N \leq M+1$ all N patterns are linearly separable and, hence, the perceptron will be able to classify them correctly (it is just a question of finding the correct weight values). On the other hand, if the number of patterns is greater than $M+1$ then the number of linearly separable patterns diminishes as the ratio $\frac{N}{M+1}$ increases. Consequently, the probability that a perceptron will be successful in a classification task becomes minimal when N is 3 or 4 times greater than the number of input dimensions, M.

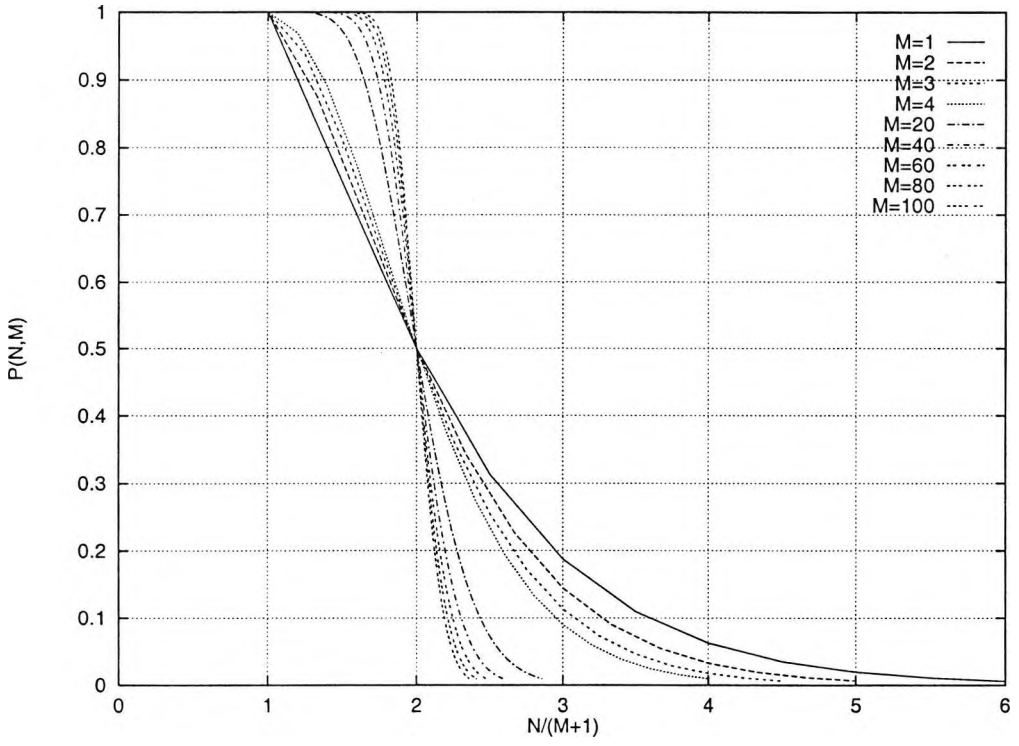

Figure 4.1: Plot of the quantity $P_{N,M}$ for various values of M, the number of input dimensions.

Figure 4.1 shows various plots of equation 4.1 for different M values ($M = 1$ is indicated by a continuous line whereas $M = 100$ lies on the opposite side of the envelope). We can see that as long as the ratio $\frac{N}{M+1}$ is kept below the critical value of 2 and for any N, the linearly separable patterns are dominant. However, as soon as $\frac{N}{M+1}$ becomes greater than 2 (by increasing N and/or decreasing M), the number of linearly separable

patterns falls rapidly. In this case, the perceptron is obviously inadequate[6].

Today, however, the perceptron and, in general, most neural network classifiers are used in tasks which do not involve memorising the complete set of all input patterns. Today, people[7] place more emphasis in *generalisation* than in *memorisation* as they are primarily interested in designing systems which are accurate when presented with previously unseen data even though they may fail to separate the training data *exactly*.

4.3 FFNN: Issues of Computational Complexity

4.3.1 Introduction

One direction to approach theoretical questions regarding learning by neural networks originates with the work of Judd, [Judd, 1988] and [Judd, 1990]: "Given a network architecture (interconnection graph as well as choice of activation function) and a set of training examples, does there exist a set of weights so that the network produces the correct output for all examples?". This question is known as the *loading problem* and is of fundamental *theoretical* importance to artificial neural networks.

We say theoretical importance because the scepticisms regarding neural network convergence to an optimum solution are mainly coming from mathematicians or theoretical computer scientists not working in the area of AI. On the other hand, *users* have reported no serious troubles and have developed a practical feeling on the design and training of neural networks in a somewhat "magic" environment.

4.3.2 Some complexity classes

An informal discussion of some well known structural complexity classes is given below[8]. Basically, this classification is based on the subtle distinction between *solving* a problem using a *deterministic* algorithm and *checking* a solution reached by a *non-deterministic* algorithm.

- **Class P**: a problem is in the class P when there is a *polynomial time*[9], *deterministic* algorithm which solves the problem.

[6] In order to remedy this situation one has to keep the value of $\frac{N}{M+1}$ (well) below 2. This can be achieved by either increasing the number of input dimensions or by decreasing the number of input patterns. The XOR problem (or the parity problem in higher dimensions) can be learned by a perceptron if more inputs are added. See appendix B on page 171 for a reference to the XOR problem.

[7] In pattern recognition and signal processing for example.

[8] A standard text on structural complexity classes is [Garey and Johnson, 1979].

[9] In the length of any reasonable encoding of the inputs.

- **Class NP**: a problem is in NP when a "**guessed**" solution for the problem can be **verified** in polynomial time. This allows for *non-deterministic* algorithms to be used in guessing solutions and then verify them in polynomial time. Thus, the class NP contains those problem for which[10] a deterministic algorithm would yield a solution in *exponential time* whereas a *non-deterministic* algorithm finds a solution in *polynomial time*. Problems in NP are further categorised into:
  - A problem X is **NP-hard** if and only if any problem Y in NP can be **transformed** in polynomial time by f to X, such that given an instance I of Y, I has a solution if and only if $f(I)$ has a solution.
  - A problem is **NP-complete** if and only if it is both NP and NP-hard. The *travelling salesman* and *general satisfiability*[11] problems both belong to the NP-complete class.

It is known that P $\subseteq$ NP, however, whether P $\subset$ NP (i.e. a strict inclusion) is still an open question because no problem has been found, yet, for which one can prove that it is NP but not P.

4.3.3 Known results

The loading problem, as formulated by Judd, seems to be a relevant model (some reservations are expressed in the conclusions, see section 4.3.4) for supervised learning using FFNN. It is known that this problem is generally NP-complete and that many strong restrictions on design parameters do not help to avoid the intractability of loading, [Šíma, 1994]. Wiklicky has extended these results and proved that the loading problem for higher order networks is even *undecidable*, [Wiklicky, 1993].

Blum and Rivest, [Blum and Rivest, 1988], have shown that the loading problem is NP-complete if the neural network (FFNN) contains only three units using the threshold activation function. This result has been generalised for neural networks using semi-linear activation functions, [DasGupta et al., 1995]. Still, the activation function is non-differentiable and, thus, it can not be used with any *gradient descent* training methods.

Finally, in [Šíma, 1996] it was shown that the loading problem for a 3-node neural network with sigmoidal activations and zero bias value for the output node (a condition

[10] Here lies one of the most fundamental and famous problems of Computer Science: is $P = NP$ or not?

[11] 3-SAT and up.

Šima calls *output separation*) is NP-hard. This implies that training FFNN with a single hidden layer which satisfy the output separation condition is *intractable.*

4.3.4 Conclusion

The results mentioned in this section are undoubtedly *pessimistic* because they are derived by imposing several and strict conditions on the architecture and nature of the neural network. Choices, which in practice are – at least – rare[12], are here necessary in order to simplify the loading problem. In addition, the way the problem has been formulated does not exactly coincide with the aims of usual practices of training and testing a neural network. In reality, the purpose of training a neural network is not *memorisation* of the training patterns (that is what the loading problem assumes) but, instead, a good *generalisation* behaviour when the network is presented with unknown patterns (extracted from the same source as the training patterns). This, of course, means that the final set of weights is not, necessarily, required to be selected on the assumption of zero training error. These results are – none-the-less – significant in the sense that they give us an indication of the upper bounds of the computational complexity of training a neural network. Whether these results are to be expected in practice or not is a totally different issue.

4.4 Learning as optimisation

4.4.1 Introduction

Supervised learning with neural networks has always been studied and dealt with as a *problem of optimisation.* The various perceptron's training algorithms are more or less straight-forward implementations of the *gradient descent* method. More complex neural network architectures require more complex learning algorithms; all of which are invariably based on *gradient descent.*

The difference, however, between the training algorithms for the perceptron and *Adaline*[13] and the more complex algorithms for multi-layer neural networks is that the former's error function in the weight space has a *unique* valley – a unique global minimum, [Baldi and Hornik, 1989]. Therefore, reaching this point with *gradient descent* is just a matter of choosing the correct step size (learning rate). This is no longer true

[12] For example [Höffgen, 1993] announces NP-completeness results for the loading problem on the assumption that the weights be restricted to binary values of -1 and 1!

[13] **AD**Aptive **LIN**ear **E**lement is, basically, a single layer network with a linear activation function at its output. See [Widrow and Hoff, 1960] for more details.

for the case of gradient descent applied in *multi-layer*, *non-linear* architectures. In this case the error function has a much more complicated topography with numerous local minima.

4.4.2 Back-propagation

Firstly, a clarification of what is meant by *back-propagation*. *Back-propagation* is an *algorithm* which implements the *gradient descent* optimisation technique on *differentiable* error functions which do not depend directly on their inputs but rather on *functions* of the inputs[14]. In effect, back-propagation is a way to calculate the derivatives of the cost function with respect to each of the free parameters of the network without explicitly working out the analytic expressions for the derivatives. Where these expressions are available, the algorithm is not necessary. Although this is rarely the case because the introduction of non-linear activation functions and hidden layers in the neural network architecture renders the analytic methods fractious.

As Minsky and Papert pointed out:

> " We have the impression that many people in the connectionist community do not understand that this [the back-propagation algorithm] is merely a particular way to compute a gradient and have assumed instead that back-propagation is a new learning scheme that somehow gets around the basic limitation of *hill climbing*." [Minsky and Papert, 1988, p.286]

In addition, issues of computational complexity of the *back-propagation* algorithm become important as the number of free parameters of the network (the weights and biases) as well as the number of inputs and the number of training examples increase. If any forward and backward step costs $O(|\underline{\mathbf{W}}|)$ ($|\underline{\mathbf{W}}|$ being the number of weights and biases) and is performed for all T patterns, P times (e.g. the number of training iterations), an estimation of the learning time on sequential machines is then proportional to

$$T \times P \times |\underline{\mathbf{W}}| \tag{4.2}$$

EXAMPLE 4.1 *The fact that the training time of a* FFNN *for a fixed number of epochs*[15] *is proportional to the number of weights has been confirmed with the following experimental setup. A training data set consisted of* 10 *vectors each of* 50 *input variables and*

[14] The level of *indirection* depends on the number of hidden layers of the neural network; each hidden layer adds one more level.

[15] The cycle of presenting all the training examples to the network once and doing all the necessary weight updates is called an *epoch*.

1 *output and was produced by the procedure* **Levy6** *(for more details on this procedure refer to section 6.3.5 on page 87). A* FFNN *of variable number of weights was trained with this data set for* 1,000 *epochs. The average training time (over* 10 *repeats for each different size) was recorded for different number of weights. The results depicted in figure* 4.2 *show that the mean training time of a fully connected* FFNN *is proportional to the total number of weights, for fixed number of epochs and training size.*

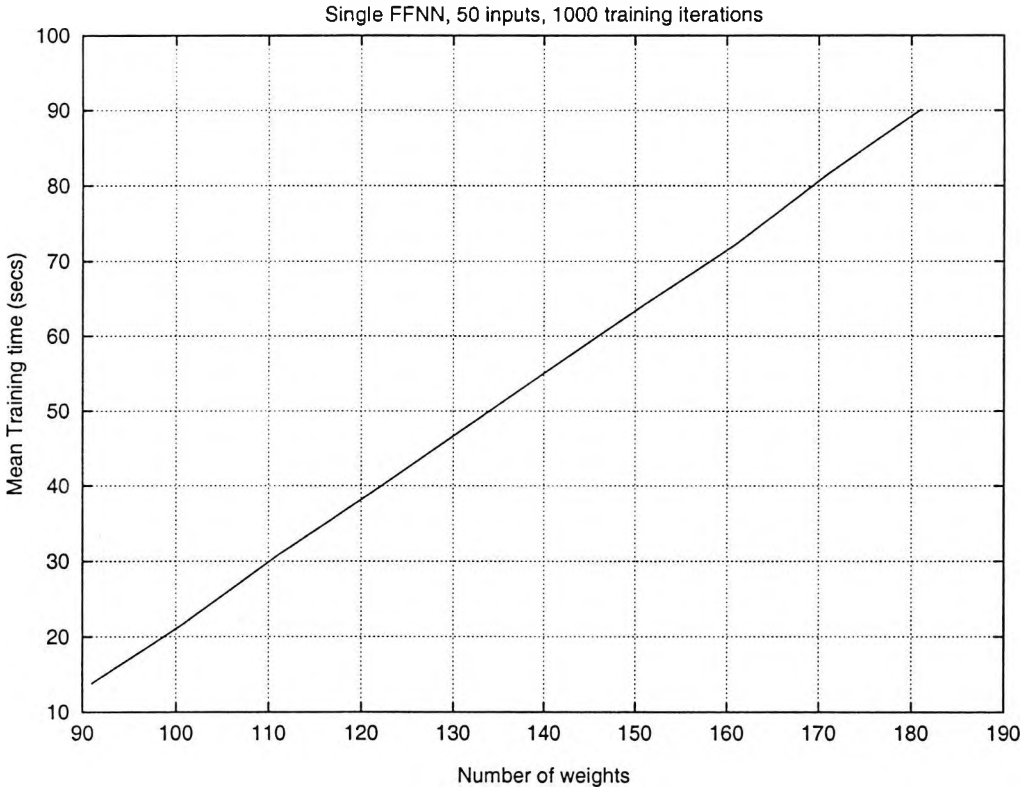

Figure 4.2: FFNN training time is proportional to the number of weights

The number of adjustable parameters in a fully connected, FFNN of l layers[16] with respect to the number of units per layer ($\underline{\mathbf{L}} = \{L_i\}_{i=1}^{l}$) and the number of input dimensions (N) is given by:

$$|\underline{\mathbf{W}}| = \sum_{i=1}^{|\underline{\mathbf{L}}|-1} L_i \times L_{i+1} + \sum_{i=1}^{|\underline{\mathbf{L}}|} L_i \tag{4.3}$$

Thus, the convergence time expression with respect to the number of input dimensions and hidden layer units is proportional to:

$$T \times P \times (\sum_{i=1}^{|\underline{\mathbf{L}}|-1} L_i \times L_{i+1} + \sum_{i=1}^{|\underline{\mathbf{L}}|} L_i) \tag{4.4}$$

[16] The l layers include the input layer, all the hidden layers and the output layer.

The above *polynomial* primarily depends on the number of input dimensions N, $N = L_1$. The number of units of the hidden layers must also, somehow, reflect the size of the input layer. Even with moderate settings and conservative rules of thumb, the number of free parameters that have to be adjusted by back-propagation may reach very high levels causing extremely long training times and a high probability of falling in one of the numerous local minima. The fact that many "real world" problems require networks of several thousands of weights (e.g. in hand-written character recognition or speech recognition tasks) implies that successful training algorithms effectively have to face the *curse of dimensionality*. Section 5.7 on page 72 compares the training times for a single FFNN and an FFNN entity.

One of the advantages of back-propagation is its local nature. Without the need for information on more than the previous and next layer units, the algorithm can be parallelised – though not very efficiently due to communication overheads, as we will see in section 4.5.1 – with all the benefits this implies. Locality, however, is not always an advantageous feature of an algorithm. Gradient descent is a *local optimisation* method and this is the main reason why it is so susceptible to local minima (see also the next section).

A possible remedy to the shortcomings of local training algorithms is to employ *global optimisation* techniques in conjunction with probabilistic methods. Unfortunately, these algorithms (see for example [Torn and Zilinkas, 1987], [Zhigljavsky, 1991] and the *magic hair-brushing* algorithm in [Chao et al., 1991]) require extremely long time to converge to a solution due to their probabilistic nature. Some alternative attempts to use *global optimisation* in a *deterministic* framework (e.g. the *Terminal Attractor Back-Propagation* algorithm proposed in a paper by [Wang and Hsu, 1991]) generally failed in discovering the global minimum. As pointed out in [Bianchini et al., 1997]: "there is no theoretical assurance that the global solution will be reached, unless the starting point lies in the domain of attraction of the global minimum" – this means that the only guarantee for reaching the *global* minimum is to be already in the global minimum valley! In addition, because of instability behaviour in the neighbourhood of a singularity, due to limited numerical precision, random jumps in the weight space – equivalent to injecting noise to the weights in conventional back-propagation – may divert the learning trajectory away from the global minimum (as well as *towards* the global minimum when trapped in a local one. Hence, the term *paradox of global convergence*).

A novel approach to the optimisation of learning machines such as the FFNN has recently

appeared under the name of Support Vector Machines(SVM) [Vapnik and Chervonenkis, 1974], [Vapnik, 1979], [Vapnik, 1995]. SVM are based on the previous work by Vapnik and Chervonenkis on statistical learning theory and the VC dimension, [Vapnik and Lerner, 1963] and [Vapnik and Chervonenkis, 1964]. SVM work by minimising the *upper bound* of the *generalisation error* of the learning machine, instead of the *sample error*. In this way, SVM do not have to face *bias-variance trade-off*, [Geman and Bienenstock, 1992], like conventional optimisation methods.

Appendix C contains a useful introduction to the concepts of VC dimension. Section C.6 in Appendix C describes SVM at a greater length. A very informative tutorial on SVM is given by [Burges, 1998].

4.4.3 Local Minima

As we have seen in the previous section, one of the problems haunting back-propagation is the existence of *minima*, *maxima* and *saddle* points on the error surface defined by the error function, $E(\underline{\mathbf{W}})$, in the weight space, $\underline{\mathbf{W}}$. This complicated topology of the error terrain makes it difficult to find the *one* (global) minimum corresponding to the lowest or, even, zero error.

For networks[17] having a single layer of computational units, linear activation functions at their outputs and employing a sum-of-squares error function, $E(\underline{\mathbf{W}})$ will be a *quadratic* polynomial on the weights and, therefore, the error surface will have a general multi-dimensional *parabolic* form with a single minimum which can be easily reached.

However, for the general FFNN architecture the error function will be a highly non-linear function of the weights suffering all the pathologies mentioned earlier.

Essentially, the presence of local minima derives from two different reasons, as discussed in [Bianchini et al., 1998]:

- **spurious** local minima may arise because of unsuitable choice of activation and error functions,
- **structural** local minima may arise because of the nature of the particular problem.

This means that some of these local minima may be eliminated with the right choice of neural network topology and activation and error functions. However, those minima inherent in the problem at hand can not be rid of unless an appropriate reconstruction of the training set or, even, a reformulation of the learning problem is adopted.

[17] For example, the perceptron.

4.4.4 Premature Neuron Saturation

It has been shown that optimal solutions to the problem of training a FFNN for a particular data set are obtained when the majority of the hidden layer units operates predominantly in the linear region of their sigmoid activation, with small excursions into the nonlinear region, [Burrows and Niranjan, 1993].

Premature neuron saturation refers to situations where the input signals to the hidden layer nodes are so high that the neurons are forced to produce an output response very close to the upper bound of their sigmoid activation. Saturation causes the neurons to lose their sensitivity to input signals, the propagation of information is blocked and, thus, sub-optimal solutions arise.

Saturation has been mainly attributed to learning with high initial weight values. However, the input signal to a hidden layer neuron is a summation of the product of a weight value and previous layer outputs *over* all the previous layer units. Therefore, saturation may also arise when the number of units in each layer is large, for example in networks with a lot of inputs (input dimensions).

A possible solution to the problem of *premature neuron saturation* would be to initialise the weights to some very small values and restrict them – during training – within some ranges which will ensure that saturation will not occur. This solution is based on the assumption that if there is a solution to the problem at hand, then this will, most likely, be found there where saturation does not occur; e.g. in the region of small weights. This is partly true because some weights might have large values without necessarily causing the majority of neurons in a network to saturate. Therefore, the solution may also be found in regions where some weights are small and some are large.

Questions such as *how many weights should be restricted*, *which weights* and *in what ranges* should then be answered in order to ensure that optimal solutions are not excluded *a priori*. The answer to these questions, however, is specific to the neural network architecture chosen and the training data available. Some people might be tempted to introduce a random element in determining these answers with a distribution depending on network architecture and training data. Although such methods may be effective at times, they should be used with skepticism because they introduce more uncertainty, in addition to that of gradient descent, and leave open questions. Moreover, restricting the learning process in such a way, would constitute a serious deviation from the doctrines and meaning of connectionism.

Regularisation, [Bishop, 1995, p. 15, 338], is a technique for controlling the smoothness

of a mapping function (the output of a neural network, for example). One way to achieve this is by controlling the change of the weights in a neural network. Thus, instead of updating weight values uniformly by using a universal weight update rule depending only on the error value, regularisation theory suggests the use of a functional which depends on the weight value as well so that smaller weights are updated differently than larger ones.

Weight decay, a subset of regularisation methods, penalises large weights. Other regularization methods may involve not only the weights but various derivatives of the output function.

4.5 Issues of parallelism and hardware implementation

4.5.1 Parallelism

An efficient parallel implementation of any computational task will have to take into account not only how well the task can be *partitioned* and assigned to the different processors (*load balancing*) but also the *communication requirements* between the various processes and processors. The latter is important from a practical point of view because, on the one hand, the *bandwidth* of the communication channels is limited and, on the other hand, the communication process itself consumes computing power (e.g. *routing, error detection and correction*, etc.), thus slowing down the overall effort. As a result, the number of parallel processes is limited by the *communication overhead* they will introduce and, therefore, can not be increased unconditionally.

The basic processing element of a FFNN is the neuron which, for a fully connected network, communicates with all the neurons of the previous and next layers. Its task is simple and basic but requires a lot of information passing:

- In **forward propagation** mode, it is required to compute a sum of products using the outputs of *all* the neurons from the previous layer, pass it through a non-linearity (activation function) and fan-out the end result to *all the neurons of the next layer.*

- In **back-propagation** mode, it is required to compute a sum of products using the derivatives of the output of the activation function from all the neurons of the next layer and distribute the end result to *all the neurons of the previous layer.*

The most natural partitioning of the problem of training a FFNN for parallelised implementation is at the level of the neuron: each neuron belonging to the same layer

be assigned to a separate processor. Considering the simplicity of the neuron's task and the fact that the number of neurons in a given layer can be large, this scheme is, definitely, not practical as it would require a large number of processors doing trivial computations.

An alternative scheme which takes advantage of the fact that modern processors are quite powerful, is to group several neurons together and assign *each group* to one processor, [Kontoravdis et al., 1992]. In this way a reduction in the number of processors as well as a reduction in processor-to-processor communication is achieved. The cost is that each processor being an essentially von-Neumann machine operates sequentially; therefore, the amount of parallelism is reduced too.

Undoubtedly, the *fine-grain* parallelism which is inherent in the nature of a FFNN does not map very well onto current (parallel) computer architectures and may be inefficient in practice due to excessive communication overhead, [Misra, 1992]. This is especially true if the classical approach of parallelism at the neuronal level is taken.

As an alternative, FFNN are engineered as a mixture of parallel and sequential computational schemes. On the same token, some people are expressing the view that as the computing power of modern processors increases, sequential implementations may be the most viable and optimal choice after all.

We believe that parallelism is a valuable feature of neural network models and should be retained in practical implementations, with any modifications and adjustments deemed necessary. Not only because the immense explosion in the development of conventional hardware of the last ten to twenty years[18] *will soon subside in the face of the limit of atomic size, but also because without parallelism at the hardware level, significant advantages gained by adopting a connectionist approach such as* fault tolerance, scalability *and* hardware extendibility *are lost.*

As it has already been mentioned in section 4.4, artificial neural networks have been used successfully on small pattern recognition problems but, in larger applications, *scalability* problems will occur, [Zell et al., 1993]. In biology (e.g. brain physiology), these same problems seem to be solved by the development of modular structures (see chapter 5) of which parallelism is a key element.

Parallelism and *modularisation* at the *hardware level* are also crucial for building *extendible* systems where new modules can be added without fundamental changes to

[18] The statement that circuit density – the capacity of semiconductors – doubles every 18 months is known as *Moore's Law*.

the behaviour of the other modules – something that is very difficult with the current, non-expandable architectures, [Murre, 1993].

This last point leads us to the next section where practical problems of hardware implementation of neural networks are discussed.

4.5.2 Hardware implementation constraints

As neural network models are growing more complex and their applications are becoming more sophisticated, the *plausibility* and *ease* of *hardware implementation* are crucial in their successful utilisation in non-trivial tasks and in environments outside the computer laboratory. In addition, neural network hardware offers the *high-speed* circuits necessary for *real-time* applications.

The main factor restricting the successful and large scale implementation of neural networks in silicon is, perhaps, the large number of fan-in and fan-out connections per neuronal unit which results in increased complexity of the circuit design, excessive current supply and the prohibitively large chip area required for the calculation of the huge sums of products involved.

In addition, analogue neural network hardware designs suffer from low precision computations, dependent on external factors such as temperature and power supply stability, [Hecht-Nielsen, 1990, p.273].

For a review of hardware neural networks see [Lindsey and Lindblad, 1994]. Chapter 8 of [Hecht-Nielsen, 1990] offers an extensive analysis of hardware neural network implementation.

4.6 The non-explicit nature of learning

Current neural network architectures, and in particular the feed-forward neural networks, are conveniently described in terms of a *black box* model. The large number of weights, the non-linearities applied to the output of each neuron and the presence of one or more hidden layers make it very difficult to observe the intermediate steps of the learning process in the level of weights and neurons.

Firstly, the advantages derived from the simplicity and convenience offered by the *black box* approach are nullified when greater interference with the learning process is desirable.

Secondly, without some form of explanation capability, the full potential of trained

neural networks may not be realised. The problem is *an inherent inability to explain in a comprehensible form, the process by which a given decision or output generated by a* FFNN *has been reached.*

Thirdly, the hopes of many biologists that neural networks will constitute a promising method which would help them understand the biological neural system itself are now very remote possibilities.

Finally, it is important to note that if the learning process was more explicit and better understood we could have been more efficient and effective in reaching a good solution. The discrepancy between expected and desired neural network outputs – an utterly *simplistic* indicator of performance – is, currently, the main guide of the optimisation process. Therefore, although some results are obtained when the error surface is relatively simple, in general, total success can not be guaranteed (as it was discussed in section 4.4).

For neural networks to enhance their overall utility as learning and generalisation tools, it is highly desirable if not essential that an "explanation" capability becomes an integral part of their functionality. Such a capability may be realised with *rule-extraction*: "Given a trained neural network and the examples used to train it, produce a concise and accurate symbolic description of the network", [Craven and Shavlik, 1994].

4.7 Summary

In this chapter some of the most important problems associated with *feed forward neural networks* have been examined. A summary of this critique follows:

1. **Linear classifiers** such as the *perceptron*, the predecessor of the multi-layer FFNN, are able to implement discriminant functions which are only *linear*. Classifiers employing the small class of linear functions (decision planes) are not powerful enough to be used in a wide variety of applications. In fact, Minsky and Papert managed to pause research in the field of neural networks by demonstrating the inability of the perceptron to learn the XOR function. More powerful classifiers may be build, at the cost of increased complexity, by either increasing the *order* of the perceptron or using a non-linear neural network architecture such as the multi-layer FFNN.

2. The utilitarian value of Kolmogorov's theorem of representing functions in terms of a finite superposition of compositions of functions of one variable is minimal. On the one hand, these one variable functions are highly non-smooth and virtually

unknown, [Poggio and Girosi, 1989]. On the other hand, this theorem does not offer any suggestions as to how such a decomposition may be implemented in practice. Consequently, further research in the field has been directed towards *approximate* rather than *exact* representations with all the disadvantages that this implies. Feed forward neural networks were the fruit of this research.

3. Studies of the **computational complexity** of training FFNN suggest that the problem of finding the set of weights for which the network produces the *correct* output for *all* training examples – e.g. the *loading problem* – is NP-hard or NP-complete depending on the network topology and activation functions. In theory, this means that it is computationally expensive, if not impossible, to find the set of weights corresponding to *correct* network behaviour. In practice, however, the implications of these results are not as dramatic because more often than not users are satisfied more with *approximate* rather than *exact* solutions[19].

4. **Learning as optimisation** has greatly simplified the problem of training artificial neural networks by transforming it into purely mathematical terms. However, it has also revealed many problems such as *local minima*, the *multi-modality*, *complexity* and *dimensionality* of the error surface, the *exponential growth of the search space*, *premature neuron saturation*, etc. These problems become worse as the number of adjustable network parameters and the complexity of training data increase.

5. The **parallel** and **hardware implementation** of a monolithic FFNN is very difficult in practice because of *excessive communication requirements* and the *fine-grain* parallelism inherent in its nature. A parallel or hardware implementation of a connectionist system may only be feasible after extensive *modularisation* and adoption of a more *coarse-grain* architecture.

6. The **non-explicit nature of learning** which currently characterises neural network modelled on the *black-box* approach removes virtually any possibility for an explanation capability in trained neural networks. Additionally, if the learning process was more explicit and better understood, reaching a good solution could have been more efficient and effective. *Modularisation* is one way to effect the extraction of high-level information from neural networks.

[19] Keep in mind that *exact* learning (see *memorisation*) of the training set does not imply *correct* behaviour when the entire problem distribution is presented.

CHAPTER V

FFNN ENTITIES

A methodology for the decomposition of a single FFNN which allows for dealing fast and effectively with high-dimensional data without degradation of its approximation and generalisation abilities – is herein discussed.

5.1 Introduction

As it has been discussed in chapter 4, multi-layer feed forward neural networks invariably suffer a number of problems when applied to tasks involving *high-dimensional* data:

- *premature neuron saturation*,
- the intractability of the *loading problem*,
- the *exponential growth* of the search space as its dimensions increase,
- *local minima*, *complex error surfaces* and the inability of the back-propagation algorithm to deal with them effectively.

In addition, current neural network models do not favour practical and efficient *parallelisation* due to their fine-grain structure. *Hardware implementation* of neural networks is also difficult and expensive, especially when the number of weights is large. Finally, the larger the number of weights is, the more difficult it is to observe the intermediate steps in the learning process or to explain in a comprehensible form, the process by which a given decision or output has been reached.

In the past, several training algorithms and procedures as well as network architectures, neuronal models, error and activation functions, etc. have been proposed as solutions to some or all of these problems. Naturally, they have their advantages as well as

their side effects and disadvantages but they do not pose as *generalised* solutions[1]; their efficacy is *specific* to the application domain and the nature of the problem at hand. Simply put, most of these solutions are designed to *fine-tune* neural networks to work optimally within a limited domain rather than develop techniques for both better performance as well as wider applicability using simpler procedures.

In our opinion, and in line to the underlying concepts of connectionism, the learning process should remain simple and without a lot of interference. Thus, in order to address the issue of scaling, one has to rely on the existing basic building blocks; we need to learn how to combine small networks and place them under the control of other networks.

The *modular neural network* architectures developed over the last years are based on the philosophy that the computational benefits gained from the vast connectivity of neuronal elements can be improved, when the current, traditionally solid architecture is extended to a kind of *meta neural network* where connectivity exists at higher levels, e.g. between networks, networks of networks and so on.

In this chapter, a novel *modular neural network* architecture, called an *entity* of FFNN, is introduced. Its design has been motivated by the fact that although existing modular networks are successful in addressing issues of *generalisation*, *specialisation* and *confidence* of prediction, all the problems associated with high-dimensional data and scaling of networks, basically, remain unanswered.

In section 5.2, the main reasons for adopting modular connectionist models are identified. Section 5.3 reviews previous work in the field of modular neural networks. The family of FFNN entities is formally described in section 5.4. An account of how to create and train the entities is also given therein.
In section 5.6, it is proved that FFNN entities are *universal function approximators*. Section 5.7 investigates the time benefits obtained by replacement of a single FFNN with an entity of FFNN.

Finally, section 5.8 outlines all the benefits gained by adopting the proposed architecture of the entities.

[1] The last 50 years, technology and software development have delivered plenty of such solutions with *localised* scope. Ephemeral computer languages, user interfaces, enhanced hardware with added features: a Sisyphean effort to patch up the notoriously flawed von Neumann computer architecture. Intelligent gear-boxes, put the engine left, right and center: a Sisyphean effort to patch up the notoriously inefficient internal combustion engine.

5.2 Motivation

The motivation behind substituting the *monolithic* neural architectures (and, specifically, the FFNN model) with equivalent *modular* constructions emanates from the following facts and observations:

- *Partitioning* a task into smaller sub-tasks is a very good way to reduce complexity without compromising the fitness of the solution. The study of human information processing systems reveals that *task decomposition* is the way which humans deal with NP-completeness. It is, probably, the most natural and common problem solving technique known to man.

  On the same token, breaking a huge and complicated structure (such as a solid neural network) into an entity of smaller structures (the modular network) will most definitely reduce the complexity of the whole system. Network decomposition also promotes *coarse-grain* parallelism, and makes the learning process more *explicit.*

- A taxonomy of the components of neural network architectures may be defined by saying that the neuron is the finest level of classification, a layer is a coarser level and a network is a still coarser level. Solid neural networks are typically designed to be modular at the neuronal level whereas entities of neural networks are designed to be modular at the level of networks. Thus, this design philosophy is just a *natural extension* of already existing connectionist models.

- It seems plausible to neural network researchers, for example [Freeman, 1991], and neurophysiologists, for example [Lavine, 1983] and [MacGregor, 1987], that:

  > " ... the brain is composed of many different parallel, distributed system, performing well defined functions [...] To address the issue of scaling, we may need to learn how to combine small networks and to place them under the control of other networks." [Freeman, 1991, pp.29-30]

  Thus, the brain is not only characterised by a massively connected network of neurons but also by the existence of different computational systems operating at different levels of abstraction and specialising at different functions.

5.3 Modular neural architectures: state of the art

5.3.1 Committees of networks

The method of combining several neural networks together in something called a *committee of networks*, and obtaining the overall response as the *average* of the output of the individual networks has been suggested in [Perrone and Cooper, 1993] and [Perrone, 1994].

Mainly, this method aims at overcoming the problems associated with the usual practice of training a network several times (or, alternatively, training a large number of different networks) with the same data but with different initial conditions (initial weight values) and/or different network topology and then selecting the network with the lowest *sample error*.

On the one hand, following these practices is not only inefficient because the computational effort spent in the training of the rejected networks is wasted – do not forget that only one network will be selected. On the other hand, selecting a network using the criterion of the lowest *sample error* can not really guarantee a good network performance with *unknown data* (see appendix C on page 173 for more details on issues regarding sample error and the confidence of prediction).

Theoretically, the mean squared error of a committee of N networks can be reduced by a factor of N compared to the average error of the networks if acting independently. This result is based on the assumption that the errors of the individual networks are *uncorrelated* [Bishop, 1995]. In practice, however and for obvious reasons, this assumption rarely holds and, thus, error reduction is not as dramatic. None-the-less, the error of the committee is guaranteed (see for example [Bishop, 1995, p.366]) to be lower than the average error of the individual networks acting independently.

It is obvious that, although, the idea behind the *committee of networks* is successful in producing a modular connectionist network with enhanced *generalisation* and *approximation* capabilities, fails to address the problems of *scaling* and *curse of dimensionality*. In fact, the effect of these problems is more evident when the number of trained neural networks increases.

5.3.2 Other ensemble methods: bagging and boosting

In this section we will describe some more attempts to implement modular connectionist models based on the idea of using the combined output of a system of neural

networks rather than the output of the best neural network acting individually.

This class of methods for combining neural networks proceeds a step further than the *committee of networks* by training each of the classifiers with a different version of the training data. These different versions are produced by uniformly resampling with replacement the original training set – a procedure known as *bootstrapping*, [Efron, 1982].

Bagging (i.e. boostrap aggregating) is structurally equivalent to the *committees of networks*. Their difference is that instead of each network being trained with the same data set (the case of the *committees of networks*), it is trained with one of the different versions of the training set.

Boosting is a more sophisticated version of *bagging*. In general, for each network in the ensemble, a weight value is assigned to each training vector. The purpose of this weighting scheme is to focus attention to the training vectors associated with high error by decreasing their weight value.

5.3.3 Mixtures of Experts

A *mixture of experts* is a modular connectionist architecture which learns to partition a task into two or more functionally independent tasks and allocates distinct networks to learn each task, [Jacobs et al., 1991]. The assumption here is that, after training, the experts will compute different functions which are useful in different regions of the input space.

The architecture consists of two types of networks: the *experts* and the *gating* networks. Basically, the gating network is trained to select the most appropriate neural network – according to previous performance – from a pool of previously trained candidates (the experts).

A typical example of a simplified application of the mixture of experts model is the absolute value function:

$$f(x) = \begin{cases} -x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

This function can be decomposed in two sub-functions, as it is obvious from the equation above, each relevant to a different region of the input space (e.g. less or greater than zero). After training the two experts to compute their assigned tasks (using standard neural network training practices), we need another network (the gate) to choose the output of only *one* of the two experts for any given input. This gating network operates

on a *winner-takes-all* principle because there exists only one expert for a given region of the input space.

The above problem is quite simple in the sense that the decomposition of the main function is obvious and that the domain of each sub-function is clear and can be revealed by inspection. However, it is not often the case that one is allowed the "convenience" of domain knowledge: when dealing with high-dimensional data a priori decomposition of a task might be difficult as well as the boundaries between the different sub-tasks are, rarely, explicitly marked in the training data.

The main benefit obtained from a mixture of experts model is that it performs a kind of *task decomposition*, induced by the competition at the level of the gating network. Moreover, the process of allocating the experts to subtasks is made part of the learning problem.

However, although networks based on the mixture of experts philosophy implement and utilise some kind of *task decomposition*, promote a framework for better generalisation and incorporate, successfully, properties unknown to the supervised neural networks paradigm such as *specialisation* and *competitive learning*, they are still vulnerable to problems induced by scaling and the curse of dimensionality.

Finally, another model of modular neural networks, the *recursively-defined* mixture of experts in [Jordan and Jacobs, 1992] constitutes just a variation on the theme of the mixture of experts. Its main novelty is that each of the experts can now be a mixture of experts network itself.

5.3.4 Summary and margins for improvement

In the previous section, a review of the current state-of-the-art in the area of modular neural networks was presented. The simplest approach to designing such systems is the *committee of networks*, where several networks are trained on the same data but with different initial conditions and configuration. The output of the committee is then the *average* of all the networks which is guaranteed to be at least lower than the average error of the individual networks acting independently.

The objective behind the idea of combining several networks in a committee is to improve the generalisation ability of the system, make it more stable and less susceptible to the disposition of single networks.

Other examples in this direction are *boosting* and *bagging* which use constructions which are structurally equivalent to the *committees of networks*. Each network is trained with a different version of the data set produced using the method of uniform

resampling with replacement – e.g. *bootstrapping.*

Mixtures of experts, on the other hand, implement a more complex modularisation architecture which utilises the ideas of *competitive learning* and *specialisation* in the hope to achieve some kind of *task decomposition.* Again, *mixtures of experts* are concentrating on improving the *generalisation ability* of the resultant networks. The benefits from such an architecture are, however, doubtful as the boundaries between the different subtasks are rarely explicitly marked in the training data and become more obscure as the number of data dimensions increases.

It is evident that research in the area of modular neural networks **concentrates in improving generalisation ability but neglects to address the scaling problems due to the curse of dimensionality**. In fact, the current modular architectures suffer more from scaling problems than single neural networks. This is so because *ensemble methods* such as the *committees of networks* require that not only one but many single FFNN need to be trained with the same high-dimensional data. On the other hand, *mixtures of experts* will find it increasingly difficult to perform task decomposition on the training data, as its dimensions increase.

Thus, there is a need to consider not only issues of improving the generalisation ability of our neural network models but also, at the same time, to make sure that scaling up such networks when used with high-dimensional data will not render them unusable. *Scaling problems* and the *curse of dimensionality* should be considered when designing alternative neural network architectures. The research described in this thesis is following this direction and, therefore, attempting to fill a gap by considering a problem which, we feel, has been somewhat neglected.

5.4 FFNN entities

5.4.1 Introduction

This section is concerned with presenting practical and theoretical design issues relevant to the construction of FFNN entities. Questions regarding the *nature*, *type* and *complexity* of the underlying components of the entities as well as *interconnection* and *training* schemes, will be answered here.

FFNN entities are designed along the same principles as those of ordinary, monolithic neural networks. After all they, too, belong to the family of connectionist systems; their structure can be described using the same taxonomy where units belong to the finest level, layers to a coarser level and networks to a still coarser level of classification.

Thus, an FFNN entity is composed of not only simple neurons, like the case of single FFNN, but also of units of a more complex character and behaviour. Small *neural networks*, *networks of neural networks* (e.g. other FFNN entities) or their *combinations* can be used in the construction of an entity. The type of underlying entity units needs not be restricted to neuronal. After all what is a neural network (FFNN) other than a function on its inputs which holds the universal function approximation property and whose coefficients need to be finalised through training? In this respect, the family of polynomials are known to be universal function approximator too and, hence, can be used in the construction of an entity.

Obviously, the choice of an entity's basic components is very wide. Whether this is a good thing or just a feature which adds unnecessary complexity to the whole process is an open question. It could be that the choice of components makes little difference to the generalisation ability of the entity and that is only relevant to the nature of the problem at hand. Something similar happens when selecting the type of activation function in the case of neurons. It is known that it should be a non-linear function but whether it should be a logistic, hyperbolic or something else, really depends on the properties of the specific problem's data. For the sake of clarity and simplicity of notation and because of time constraints, this thesis is only concerned with *single output* entities which are composed of single FFNNs.

The units of an entity may be layered and linked with connections of adjustable strength just like ordinary neurons in the case of single FFNN. The connectivity of the elements of an entity, however, is not as strict as that of single FFNN. The restrictions imposed to the connectivity of neurons by *back-propagation*, in the case of single FFNN, only apply to entities whose underlying components are linked with connections of adjustable strength and, thus, will require back-propagation. The existence, however, of adjustable connections between entity units is optional. Thus, the only restriction that governs the topology of these entities is to ensure the absence of any feedback connections. The reason for this is that recurrent neural networks are more complex to train and, thus, were avoided.

Optimising the weights of the neural networks composing the entity or the weights connecting the various entity units is essentially based on gradient descent: back-propagation is used for the former and a modified, generalised, version for the latter. Training an entity is not complicated provided that the many networks existing at different levels are dealt with following a certain sequence so as to avoid any deadlocks.

A possible categorisation criterion for the entities is their *topology*[2] and the *training targets* of each entity component. These two criteria were used to identify three different classes of entities, namely classes 1, 2 and 3. The symbols C_1, C_2 and C_3 shall denote the three entity classes throughout this chapter. The symbols $\mathcal{C}_1$, $\mathcal{C}_2$ and $\mathcal{C}_3$ shall denote the *family of functions* implemented by each of the entity classes.

Class 1 entities follow a partially connected configuration where the units of a given layer (including the inputs) may send their output to any unit which belongs to a layer closer to the output than themselves – e.g. feed-forward signal propagation. Any of the inputs may be sent directly to any non-input layer units as well as it is possible to have connections between units of non-adjacent layers

Class 2 entities are based on a cascaded architecture. Their structure is more regular – a feature which allows the use of some form of feedback. A class 3 entity is a refinement of class 2 with the main inter-connection theme, that of cascaded architecture, being inherited but with training being slightly different.

In effect, the difference between classes 1 and 2 is one between *unordered* and *ordered*, *unstructured* and *structured* topology. Whereas, the difference between classes 1 & 2 and 3 is one of different, more refined training targets. It is interesting to note that the idea of refined training targets, which could have led to a fourth entity class, could not have been applied to C_1 without imposing serious constraints on its architecture and losing the generality offered by its unstructured topology. Thus, the design and implementation of a fourth entity class was not attempted because it was considered to be time consuming and would not add significantly to the novelty of this work. The three entity classes will be described in more detail in the following sections.

5.4.2 Class 1 FFNN entities: formalism

DEFINITION 5.1 *The family of all possible transfer functions implemented by a C_1 entity with n inputs, $\mathbf{x} = \{x_1, x_2, \ldots, x_n\}$, is:*

$$\begin{aligned}\mathcal{C}_1^n = \{ \quad & f_n^m : \mathbb{R}^n \to \mathbb{R} \mid \; f^1_{|\mathbf{x}|}(\mathbf{x}) = g_{|\mathbf{u}_1|+|\mathbf{v}_1|}(\mathbf{u}_1 \cup \mathbf{v}_1) \\ & \textit{where, } \mathbf{u}_i = \left\{ f^j_{|\mathbf{u}_j|+|\mathbf{v}_j|}(\mathbf{u}_j \cup \mathbf{v}_j) \right\}_{j=i+1}^{i+s} \\ & \mathbf{v}_i \subset \{\emptyset, x_1, x_2, \ldots\},\ g_i \in \mathcal{G}^n \textit{ and } f^i_\emptyset = \emptyset \; \}\end{aligned}$$

$\mathcal{G}^i$ represents the family of functions implemented by a feed forward neural network with i inputs according to definition 3.4 on page 23. $\mathbf{u}_i$ is a set of s elements which

[2] Another criterion is the type of their underlying components. Such taxonomy has not been attempted though this possibility may be investigated in the future.

can themselves be either smaller entities (e.g. $f^{j}_{|\mathbf{u}_j|+|\mathbf{v}_j|}(\mathbf{u}_j \cup \mathbf{v}_j)$) or just plain input variables (e.g. x_i). Each entity in the above definition is characterised by a unique identification number – the "m" in f^m_n – and the number of its inputs – the "n" in f^m_n.

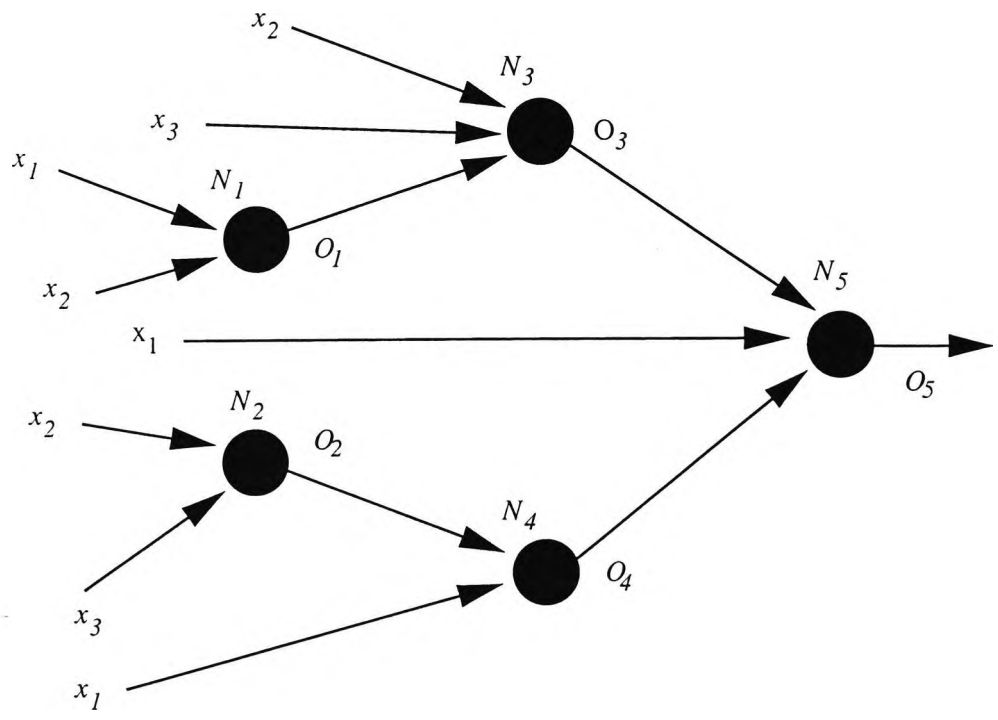

Figure 5.1: A simple C^3_1 entity implementation

EXAMPLE 5.1 *Below is the specification of a* C^n_1 *implementation,* $f^5_3(x_1, x_2, x_3)$, *which consists of five* FFNN: N_1, N_2, N_3, N_4 *and* N_5:

- N_1: *inputs are* $\mathbf{X}_1 = \{x_1, x_2\}$, *output is* $O_1 = f^1_2(x_1, x_2)$,
- N_2: *inputs are* $\mathbf{X}_2 = \{x_2, x_3\}$, *output is* $O_2 = f^2_2(x_2, x_3)$,
- N_3: *inputs are* $\mathbf{X}_3 = \{x_2, x_3, O_1\}$, *output is* $O_3 = f^3_3(x_2, x_3, f^5_2(x_1, x_2))$,
- N_4: *inputs are* $\mathbf{X}_4 = \{x_1, O_2\}$, *output is* $O_4 = f^4_2(x_1, f^3_3(x_2, x_3, f^5_2(x_1, x_2)))$,
- N_5: *inputs are* $\mathbf{X}_5 = \{x_1, O_3, O_4\}$, *output is* O_5, *the final output.*

The implemented neural network transfer function $O_5 = f^5_3(x_1, x_2, x_3)$ *is*:

$$\begin{aligned} O_5 = f^5_3(x_1, x_2, x_3) &= g_3(x_1, O_3, O_4) \\ &= g_3(x_1, g_3(x_2, x_3, O_1), g_2(x_1, O_2)) \\ &= g_3(x_1, g_3(x_2, x_3, g_2(x_1, x_2)), g_2(x_1, g_2(x_2, x_3))) \end{aligned}$$

where, $f^5_3 \in C^3_1$ *and* $g_i \in \mathcal{G}^i$.

5.4.3 Class 1 FFNN entities: construction

Before training a C_1 entity, its architecture, allocation of the set of the input variables to the entity units and connectivity scheme have to be decided.

Class 1 entities conform to a not-necessarily fully connected configuration where a given unit may send its output to any other unit which does not, directly or indirectly, sends its output back to the first unit. This means that the signals must not flow backwards – no feedback connections are allowed. The reasons for this are twofold: firstly, the addition of feedback might seriously destabilise the overall process. Secondly, the training of recurrent neural networks is complex and more difficult than that of feed-forward models.

In order to make it easier to avoid feedback connections, it is preferable that in any entity description or diagram, the units are grouped in **layers** – in the same way as neurons are grouped within single FFNN. This convention will also make it easier to determine the sequence in which entity units should be trained. For example, in the entity implementation of figure 5.2, it is clear that training must take place in three steps:

1. N_1 and N_2,
2. N_3 and N_4,
3. N_5.

In a larger implementation, the benefits arising from the clarity provided by adopting such convention would have been much more obvious. However, the convention of using "layers" is not strictly necessary as the layers in an entity can be identified by a simple partial sort algorithm.

Any of the data inputs can be sent directly to any non-input units as well as it is possible to have connections between units of non-adjacent layers. Apart from the requirements set above, there is no strict methodology to be followed in determining the final architecture and connectivity scheme.

The allocation of the set of the input variables to the entity units is also not strict. Unless there is prior knowledge about the problem domain favouring[3] a particular grouping and allocation to the various FFNN, each entity unit may be allocated as many input variables as it is necessary. It is also possible that any given input variable

[3] Remember the notion of *task decomposition* in section 5.3.3 on page 53.

may be allocated to many input and non-input units (as demonstrated in example 5.1 and shown in figure 5.1).

Another variation of the basic class 1 entity model is when the various entity units are linked by connections of adjustable strength – something equivalent to the weights between neurons of single FFNN. This possibility will be discussed in section 5.4.5.

5.4.4 Class 1 FFNN entities: training

Training a C_1 entity is straightforward. Firstly, all the FFNN which do not receive any input from other FFNN will be trained with all the vectors contained in the training set, the exemplars. Of course, each FFNN will only consider those input variables which have been allocated to it and not any others.

Secondly, we will train those FFNN receiving inputs from the training set directly and/or from already trained FFNN. Notice that the target output to every FFNN in the entity is identical to the target output defined in the training set.

5.4.5 Class 1 FFNN entities with adjustable connections

An additional, though not necessary, step towards optimising a C_1 entity is to introduce adjustable strength connections between each individual FFNN. Figure 5.2 shows the FFNN entity of example 5.1 with added adjustable connections.

As soon as the training of each individual FFNN, is completed in the usual manner, the connections at the level of the entity will be adjusted using gradient descent and back-propagation. This optimisation procedure is done in exactly the same way as it would have been done if we were dealing with a single FFNN.

There are, however, two structural differences which call for minor modifications to the back-propagation algorithm:

1. The architecture of the entity is not as *regular* as that of a fully connected FFNN. For example, connections between units of non-adjacent layers are allowed as well as full connectivity is not a requirement.

2. All the units (neurons) of a FFNN usually employ the same activation function whereas each unit (FFNN) in an entity implements a different function. This means that the derivatives of the transfer functions of each FFNN, with respect to its inputs, must be calculated individually. The formula for the derivative of the FFNN transfer function is given in section 5.4.6.

In appendix A on page 167 the equations of the back-propagation algorithm for a single FFNN are derived. Equation A-9 describes how the update of the weight connecting units i and j is done. This weight update scheme also holds when training the entity. The term δ_j^l was defined in equation A-8 as:

$$\delta_j^l = \begin{cases} (T_j - Y_j) \cdot \sigma_j^{l\prime}(A_j^l(\underline{\mathbf{y}}^{l-1})) & \text{if } l \text{ is the output layer, } L, \\ \sigma_j^{l\prime}(A_j^l(\underline{\mathbf{y}}^{l-1})) \cdot \sum_{k=1}^{u(l+1)} \delta_k^{l+1} \, w_{kj}^{l+1} & \text{if } l \text{ is hidden layer, } 1 < l < L. \end{cases}$$

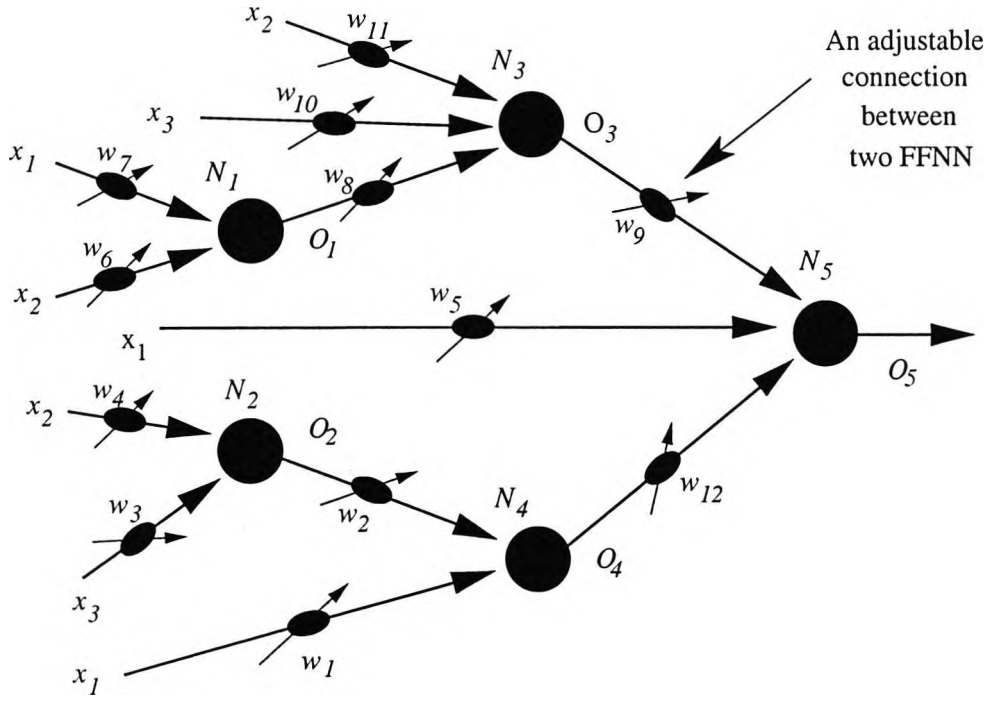

Figure 5.2: A $\mathcal{C}_1^3$ entity with adjustable connections between individual FFNN

The only modifications necessary to the above equation in order to adapt it to the entities' model are the following:

1. The term $\sigma_j^{l\prime}(A_j^l(\underline{\mathbf{y}}^{l-1}))$, e.g. the derivative of the neuronal activation function, should now be replaced by the derivative of the j^{th} FFNN unit in the l^{th} layer, $\nabla_{inp_j^l} F_j^l(\underline{\mathbf{inp}}_j^l)$. The vector $\underline{\mathbf{inp}}_j^l$ holds all the inputs to j^{th} FFNN of the l^{th} layer.

2. Also, the index k in the sum $\sum_{k=1}^{u(l+1)} \delta_k^{l+1} \, w_{kj}^{l+1}$ should now be instantiated over all the items of $\underline{\mathbf{inp}}_j^l$.

Summarising, when dealing with FFNN entities and entities of FFNN entities the individual units are trained first using standard back-propagation. If there are connections

of adjustable strength between the various units of the entity, then these may be optimised using back-propagation with the afore-mentioned modifications. The same modifications apply when dealing with entities of FFNN entities with the proviso that $\nabla_{inp_j^l} F_j^l(\underline{\mathbf{inp}}_j^l)$ is the derivative of each of the FFNN entities making up the bigger entity.

5.4.6 The derivative of the FFNN transfer function

We will now proceed to calculate the general expression for the derivative of the transfer function of a FFNN. The results in this section, although derived independently, are related to the work of other researchers in the field of automatic and computational differentiation such as [Lucas, 1997] and [Griewank, 1989].

Recall that the transfer function, $F(\cdot)$, of a *single-output* FFNN with L layers is given by:

$$F(\underline{\mathbf{x}}) = \sigma_1^L(A_1^L(\underline{\mathbf{y}}^{L-1})) \tag{5.1}$$

where $\underline{\mathbf{x}}$ is the input vector to the neural network (of L layers) and $A_i^l(\cdot)$ and $\sigma_i^l(\cdot)$ are the affine transform and activation function associated with the i^{th} unit (neuron) of the l^{th} layer.

In general, $\underline{\mathbf{y}}^l$ is a vector containing the outputs of each neuron of the l^{th} layer:

$$\underline{\mathbf{y}}^l = \begin{pmatrix} \sigma_1^l(A_1^l(\underline{\mathbf{y}}^{l-1})) \\ \sigma_2^l(A_2^l(\underline{\mathbf{y}}^{l-1})) \\ \vdots \\ \sigma_{u(l)}^l(A_{u(l)}^l(\underline{\mathbf{y}}^{l-1})) \end{pmatrix}$$

Note that the incoming signal to the input layer units is $\underline{\mathbf{y}}^0 = \underline{\mathbf{x}}$.

The afore-mentioned affine transform associated with the i^{th} unit of the l^{th} layer, is generally given by:

$$\begin{aligned} A_i^l(\underline{\mathbf{y}}^l) &= b_i^l + \underline{\mathbf{w}}_i^l \cdot \underline{\mathbf{y}}^{l-1} \\ &= b_i^l + (w_{i,1}^l \;\; w_{i,2}^l \;\; \cdots \;\; w_{i,u(l-1)}^l) \cdot \begin{pmatrix} \sigma_1^{l-1}(A_{1-1}^l(\underline{\mathbf{y}}^{l-2})) \\ \sigma_2^{l-1}(A_2^{l-1}(\underline{\mathbf{y}}^{l-2})) \\ \vdots \\ \sigma_{u(l-1)}^{l-1}(A_{u(l-1)}^{l-1}(\underline{\mathbf{y}}^{l-2})) \end{pmatrix} \end{aligned} \tag{5.2}$$

where, $u(l)$ gives the number of units contained in the l^{th} layer, b_i^l is the bias value of the i^{th} neuron in the l^{th} layer, $\underline{\mathbf{w}}_i^l$ is the i^{th} *row* of $\underline{\mathbf{w}}^l$ – e.g. the matrix holding the

weights of the l^{th} layer (of $u(l)$ rows and $u(l-1)$ columns). See section 3.2 on page 20 for more details on notation.

By application of the **chain rule** to equation 5.1 we get the derivative of the FFNN transfer function with respect to the input vector, $\underline{\mathbf{x}}$, as:

$$\nabla_x F = \frac{d\sigma_1^L(A_1^L(\underline{\mathbf{y}}^{L-1}))}{dA_1^L(\underline{\mathbf{y}}^{L-1})} \cdot \nabla_x A_1^L(\underline{\mathbf{y}}^{L-1}) \tag{5.3}$$

The following shorthand notation shall be used throughout the rest of this section:

$$g_i^l = \frac{d\sigma_i^l(A_i^l(\underline{\mathbf{y}}^{l-1}))}{dA_i^l(\underline{\mathbf{y}}^{l-1})} \quad , \quad \underline{\mathbf{g}}^l = \begin{pmatrix} g_1^l \\ g_2^l \\ \vdots \\ g_{u(l)}^l \end{pmatrix} \quad \text{and,} \quad \underline{\mathbf{G}}^l = \begin{pmatrix} g_1^l & 0 & 0 & \cdots & 0 \\ 0 & g_2^l & 0 & \cdots & 0 \\ 0 & 0 & g_3^l & \cdots & 0 \\ \vdots & & & & \\ 0 & 0 & 0 & \cdots & g_{u(l)}^l \end{pmatrix}$$

The calculation of g_i^l is straight forward and specific to the choice of the activation function $\sigma_i^l(\cdot)$. The calculation of the derivative of the affine transform, $A_i^l(\cdot)$, with respect to the input vector is given below.

By differentiation of equation 5.2 we get:

$$\nabla_x A_i^l(\underline{\mathbf{y}}^{l-1}) = \underline{\mathbf{w}}_i^l \cdot \nabla_x \underline{\mathbf{y}}^{l-1} \tag{5.4}$$

$$\text{and,} \quad \frac{\partial A_i^l}{\partial x_j} = \underline{\mathbf{w}}_i^l \cdot \nabla_{x_j} \underline{\mathbf{y}}^{l-1} \tag{5.5}$$

$\nabla_x \underline{\mathbf{y}}^l$ is defined as follows:

$$\begin{pmatrix} y_1^l \\ y_2^l \\ \vdots \\ y_{u(l)}^l \end{pmatrix} \cdot \left(\frac{\partial}{\partial x_1} \quad \frac{\partial}{\partial x_2} \quad \cdots \quad \frac{\partial}{\partial x_{u(1)}}\right) = \begin{pmatrix} \frac{\partial y_1^l}{\partial x_1} & \frac{\partial y_1^l}{\partial x_2} & \cdots & \frac{\partial y_1^l}{\partial x_{u(1)}} \\ \frac{\partial y_2^l}{\partial x_1} & \frac{\partial y_2^l}{\partial x_2} & \cdots & \frac{\partial y_2^l}{\partial x_{u(1)}} \\ \vdots & & & \\ \frac{\partial y_{u(l)}^l}{\partial x_1} & \frac{\partial y_{u(l)}^l}{\partial x_2} & \cdots & \frac{\partial y_{u(l)}^l}{\partial x_{u(1)}} \end{pmatrix} \tag{5.6}$$

whereas the j^{th} column of the above matrix is:

$$\nabla_{x_j} \underline{\mathbf{y}}^l = \begin{pmatrix} \frac{\partial y_1^l}{\partial x_j} \\ \frac{\partial y_2^l}{\partial x_j} \\ \vdots \\ \frac{\partial y_3^l}{\partial x_j} \end{pmatrix} \tag{5.7}$$

By differentiating $y_i^l = \sigma_i^l(A_i^l(\underline{\mathbf{y}}^{l-1}))$ with respect to the j^{th} input using the **chain rule** we have:

$$\frac{\partial y_i^l}{\partial x_j} = \frac{d\sigma_i^l(A_i^l(\underline{\mathbf{y}}^{l-1}))}{dA_i^l(\underline{\mathbf{y}}^{l-1})} \cdot \frac{\partial A_i^l(\underline{\mathbf{y}}^{l-1})}{\partial x_j} = g_i^l \cdot \underline{\mathbf{w}}_i^l \cdot \nabla_{x_j} \underline{\mathbf{y}}^{l-1} \tag{5.8}$$

Substituting equation 5.8 back into 5.6 we get:

$$\nabla_x \underline{\mathbf{y}}^l = \begin{pmatrix} g_1^l \cdot \underline{\mathbf{w}}_1^l \cdot \nabla_{x_1} \underline{\mathbf{y}}^{l-1} & g_1^l \cdot \underline{\mathbf{w}}_1^l \cdot \nabla_{x_2} \underline{\mathbf{y}}^{l-1} & \cdots & g_1^l \cdot \underline{\mathbf{w}}_1^l \cdot \nabla_{x_{u(l)}} \underline{\mathbf{y}}^{l-1} \\ g_2^l \cdot \underline{\mathbf{w}}_2^l \cdot \nabla_{x_1} \underline{\mathbf{y}}^{l-1} & g_2^l \cdot \underline{\mathbf{w}}_2^l \cdot \nabla_{x_2} \underline{\mathbf{y}}^{l-1} & \cdots & g_2^l \cdot \underline{\mathbf{w}}_2^l \cdot \nabla_{x_{u(l)}} \underline{\mathbf{y}}^{l-1} \\ \vdots & & & \\ g_{u(l)}^l \cdot \underline{\mathbf{w}}_{u(l)}^l \cdot \nabla_{x_1} \underline{\mathbf{y}}^{l-1} & g_{u(l)}^l \cdot \underline{\mathbf{w}}_{u(l)}^l \cdot \nabla_{x_2} \underline{\mathbf{y}}^{l-1} & \cdots & g_{u(l)}^l \cdot \underline{\mathbf{w}}_{u(l)}^l \cdot \nabla_{x_{u(l)}} \underline{\mathbf{y}}^{l-1} \end{pmatrix}$$

$$= \begin{pmatrix} g_1^l & 0 & \cdots & 0 \\ 0 & g_2^l & \cdots & 0 \\ \vdots & & & \\ 0 & 0 & \cdots & g_{u(l)}^l \end{pmatrix} \cdot \begin{pmatrix} \underline{\mathbf{w}}_1^l \\ \underline{\mathbf{w}}_2^l \\ \vdots \\ \underline{\mathbf{w}}_{u(l)}^l \end{pmatrix} \cdot \begin{pmatrix} \nabla_{x_1} \underline{\mathbf{y}}^{l-1} & \nabla_{x_2} \underline{\mathbf{y}}^{l-1} & \cdots & \nabla_{x_{u(l)}} \underline{\mathbf{y}}^{l-1} \end{pmatrix}$$

$$= \underline{\mathbf{G}}^l \cdot \underline{\mathbf{w}}^l \cdot \nabla_x \underline{\mathbf{y}}^{l-1} \tag{5.9}$$

Substituting equations 5.4 and 5.9 back into equation 5.3 we get:

$$\begin{aligned} \nabla_x F &= g_i^L \cdot \underline{\mathbf{w}}_i^L \cdot \nabla_x \underline{\mathbf{y}}^{L-1} \\ &= g_i^L \cdot \underline{\mathbf{w}}_i^L \cdot \underline{\mathbf{G}}^{L-1} \cdot \underline{\mathbf{w}}^{L-1} \cdot \nabla_x \underline{\mathbf{y}}^{L-2} \end{aligned}$$

The final derivative expression can be calculated by iteration until the input layer is reached. Thus,

$$\nabla_x F = g_1^L \cdot \underline{\mathbf{w}}_1^L \cdot \underline{\mathbf{G}}^{L-1} \cdot \underline{\mathbf{w}}^{L-1} \cdot \underline{\mathbf{G}}^{L-2} \cdot \underline{\mathbf{w}}^{L-2} \cdots \underline{\mathbf{G}}^2 \cdot \underline{\mathbf{w}}^2 \cdot \underline{\mathbf{G}}^1 \cdot \underline{\mathbf{w}}^1 \cdot \underline{\mathbf{G}}^0 \underline{\mathbf{w}}^0 \nabla_x \underline{\mathbf{y}}^0$$

Since $\underline{\mathbf{y}}^0 = \underline{\mathbf{x}}$ then $\nabla_x \underline{\mathbf{y}}^0 = \underline{\mathbf{1}}$. In addition, the input layer of FFNN will have unit weights, e.g. $\underline{\mathbf{w}}^0 = \underline{\mathbf{1}}$.

Thus, the derivative of the FFNN transfer function becomes:

$$\nabla_x F = g_1^L \cdot \underline{\mathbf{w}}_1^L \cdot \underline{\mathbf{G}}^{L-1} \cdot \underline{\mathbf{w}}^{L-1} \cdot \underline{\mathbf{G}}^{L-2} \cdot \underline{\mathbf{w}}^{L-2} \cdots \underline{\mathbf{G}}^2 \cdot \underline{\mathbf{w}}^2 \cdot \underline{\mathbf{G}}^1 \cdot \underline{\mathbf{w}}^1 \tag{5.10}$$

5.4.7 Class 2 FFNN entities: formalism

A Class 2 (C_2) entity is a special case of the C_1 entity, where interconnections between FFNN are not arbitrary but follow a pattern as shown below.

Firstly, a FFNN (N_1) is trained to implement $g_k(x_1, \ldots, x_k)$ (e.g. using only the first k inputs to train it to produce the expected output given). In most cases this will not be sufficient as the output may depend on some other inputs too. Therefore, a second FFNN (N_2) is trained to implement $g_{l-k}(x_{k+1}, \ldots, x_l)$ (e.g. using the next $l-k$ inputs to train it, again, to produce the expected output given) and so on, until N_i is trained to implement $g_{n-q}(x_{q+1}, \ldots, x_n)$.

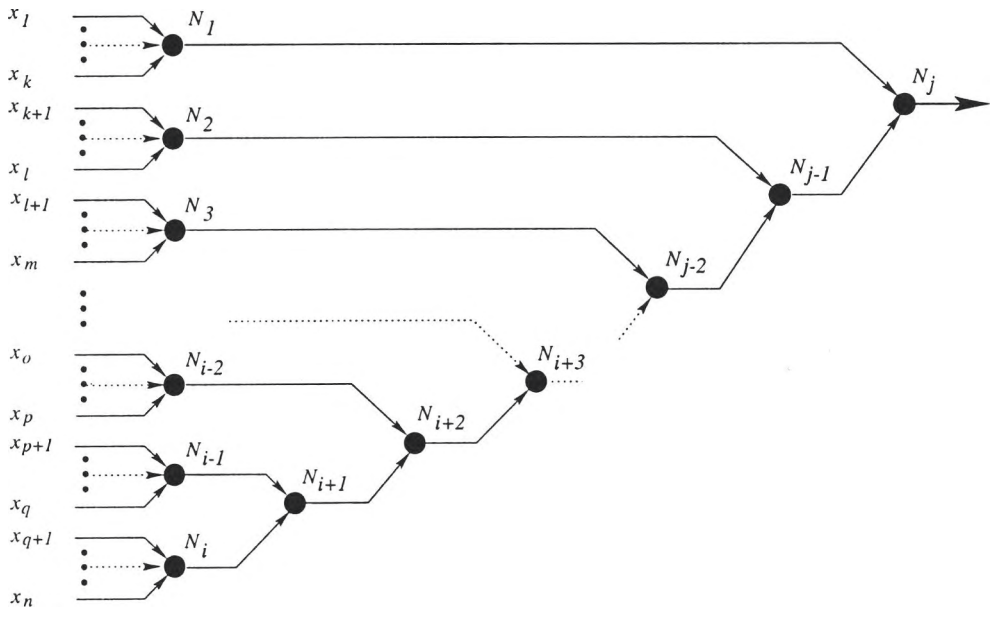

Figure 5.3: A C_2 and C_3 entity implementation

The next FFNN (N_{i+1} in) will be trained with inputs coming from FFNN N_{i-1} and Ni, that means it will attempt to implement $g_2(g_{q-p}(x_{p+1}, \ldots, x_q), g_{n-q}(x_{q+1}, \ldots, x_n))$. Whereas N_{i+2} will be trained with the output of N_{i-2} and N_{i+1} and so on until the final FFNN, N_j is reached.

The family of functions implemented by C_2 entities, $\mathcal{C}_2^n$, is recursively defined in terms of the family of FFNN ($\mathcal{G}^i$) and itself as following:

DEFINITION 5.2

$$\mathcal{C}_2^n = \{f_n : \mathbb{R}^n \to \mathbb{R} \mid f_n(x_1, x_2, \ldots, x_n) = g_2(g_k(x_1, \ldots, x_k), f_{n-k}(x_{k+1}, \ldots, x_n)), \; x_i \in \mathbb{R}, g_i \in \mathcal{G}^i\}$$

5.4.8 Class 3 FFNN entities

Class 3 (C_3) entities have the same interconnection scheme as C_2's. They differ, however, in that the training target (output) of every FFNN is not the expected output, y, as defined in the training data set. Instead, it is a measure ($||\cdot||$) of the discrepancy between *actual* and *desired* outputs of the previous FFNN.

Training commences with N_1. It is trained to implement $g_k(x_1, \ldots, x_k) = y$. In most cases, its actual output, o_1, after training, will not be exactly y. A measure of this discrepancy, given by $e_1 = ||y - o_1||$, will be used as the training target for the next FFNN, N_2 and so on until we cover all the input variables, x_i.

FFNN	Input	Output	Discrepancy
N_1	$x_1 \cdots x_k$	y	$e_1 = \|\|y - o_1\|\|$
N_2	$x_{k+1} \cdots x_l$	e_1	$e_2 = \|\|e_1 - o_2\|\|$
N_3	$x_{l+1} \cdots x_m$	e_2	$e_3 = \|\|e_2 - o_3\|\|$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
N_{i-2}	$x_o \cdots x_p$	e_{i-3}	$e_{i-2} = \|\|e_{i-3} - o_{i-2}\|\|$
N_{i-1}	$x_{p+1} \cdots x_q$	e_{i-2}	$e_{i-1} = \|\|e_{i-2} - o_{i-1}\|\|$
N_i	$x_{q+1} \cdots x_n$	e_{i-1}	$e_i = \|\|e_{i-1} - o_i\|\|$
N_{i+1}	o_{i-1}, o_i	e_{i-2}	$e_{i+1} = \|\|e_{i-2} - o_{i+1}\|\|$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
N_{j-1}	o_{j-2}, o_2	e_1	not required
N_j	o_{j-1}, o_1	y	not required

Table 5.1: Training procedure for the C_3 entities

In effect, the only difference between class 2 and 3 entities is their output target during training. In class 2 entities, this target is just the expected output (y) as defined by the training set. In class 3, however, the target is a measure of how well the previous FFNN managed to meet its own target. This feature may be used to estimate the accuracy of the system's prediction for individual cases and, further more, to assess the confidence of prediction of the different FFNN in the entity. Based on these confidence intervals, fine-tuning of the entity might be possible by replacing those FFNN whose confidence of prediction is inadequate, with other units of different properties and re-training.

Table 5.1 summarises the training procedure for C_3 entities. C_3 entities implement the same transfer functions as those implemented by C_2 and belong to the C_2^n family of functions.

5.5 The *np* language and interpreter

5.5.1 Introduction

np is a simple language which may be used to create, train and test FFNN and entities of FFNN. Its creation has been motivated by the need to have a simple but effective interface to the tedious tasks associated with the preparation of the training and test data as well as obtaining performance measures for the single FFNN and, in particular, the entities.

If training and testing a single FFNN is a humdrum activity, then training and testing an entity consisting of hundreds of FFNN interconnected in many and arbitrary ways is a formidable and unruly undertaking. Just consider that for every FFNN, one has to construct the training and data set by looking for all the outputs of previously trained FFNN and merging them in the right order. Then the same process has to be repeated for the testing stage. The whole thing might take hours in the end. Additionally, the probability of making a mistake by forgetting or mixing the inputs to some FFNN is very high and most likely will go undetected. Most importantly, this procedure has to be repeated every time some change has to be made in the system, for example changing the learning rate or the size of some FFNN.

The real challenge, however, to the enthusiast who chooses to train a neural system manually is when dealing with entities of adjustable connections. As it has already been mentioned in section 5.4.5, the training of entities with adjustable connections is completed in two phases. Firstly, each FFNN unit of the entity has to be trained individually. Secondly, the weights of the connections between these FFNN have to be adjusted using gradient descent. In effect, it is like training a single FFNN with back-propagation by hand but with the additional complication of having to calculate the derivative function of each FFNN unit, which is no longer a simple sigmoid.

Thus, the value and utility of any system which abstracts the process of creating, training and testing single FFNN and entities to a level of automation where a few commands suffice to achieve what it might take hours manually, is self-evident. In this respect, ***np*** and other similar systems are necessary tools for doing any serious and *methodical* experimentation with neural networks. The novelty of ***np*** lies in its ability to deal not only with single FFNN but also with the entities.

The ***np*** system was invaluable to the completion of this work and especially in carrying out a large number of experiments in order to obtain enough empirical results. Its high-level nature made it possible to automate the process of training and testing the entities

as much as possible. It has also helped us to experiment with parallelisation and other ideas. It does not, by all means, represents the best neural network script language and interpreter but, at the time of its making, it was considered to provide an adequate solution to the problem.

The interpreter of ***np*** and all its utilities are in the public domain and free for anybody to copy, use and modify. They can be found at:

http://www.soi.city.ac.uk/homes/livantes/Research.html.

A reference guide to the ***np*** language and interpreter can be found in appendix F on page 189. A selection of ***np*** scripts can be found in appendix G on page 221.

5.5.2 Structure

The interpreter of ***np*** is written in the **perl** language. The system consists of some twenty executables (written in the C language) which are spawned by the interpreter, with appropriate parameters, on parsing the ***np*** program.

The structure of the ***np*** system is as follows:

- At the top level is the ***np*** program which contains instructions and their parameters.
- At the next level we find the ***np*** *interpreter*. It is entirely written in the **perl** language. This language, unlike C, is ideal for string manipulation but considerably slower than C. This is the reason why applications like training a FFNN, which require a lot of processing power and must be efficient and fast, were written in the C language and constitute the lowest level of the ***np*** system. The interpreter parses the ***np*** program and checks that it is syntactically correct. Then, for each instruction, a sequence of executables will be invoked with the supplied parameters. For example a program to create an artificial data set is the following:

```
TRAINING_DATA = ProduceAndFormatVectoredDataSet {
    NumInputs = 500;
    NumOutputs = 1;
    NumLines = 60;
    Y = Levy6;
    Seed = 1974;
}
```

The interpreter will issue the following simple command:

```
unix% Levy6 -inputs 500 -lines 60 -seed 1974 -outputs 1
```

Note that *Levy6* is the executable of a C language program.

Following is a program to calculate the *mean square discrepancy* between actual and expected output values contained in the two single column ASCII files called **actual** and **expected**,

```
ACTUAL_OUTPUT = OpenFileObject {
  Filename = actual;
}
EXPECTED_OUTPUT = OpenFileObject {
  Filename = expected;
}
ERROR = ColumnsArithmetic {
  # mean square error estimate
  RowExpr = 0.5 * ((ACTUAL_OUTPUT[1] - EXPECTED_OUTPUT[1]) ** 2);
  ColExpr = average;
  OutFileName = error;
}
```

This program will be translated into:

```
unix% Merge expected actual | awk '{print 0.5 * (($1-$2) ** 2)'
          | awk '{ s += $1 } END  { print s/NR }' > error
```

- The lowest level of the ***np*** system consists of the C language executables (e.g. *Levy6* etc.) and various built-in **Unix** commands (e.g. *awk*). These applications are small, simple and, more importantly, stand-alone. They are controlled from the command line. In this way, changes can be made to one of these programs without affecting the others. Adding features to the ***np*** system consists of just writing one or two more of these programs. Had the ***np*** system consisted of a single program, maintenance and expansion would have been extremely difficult[4].

[4] These are some of the advantages of systems built on the principles of *emergence* and *connectionism*. These systems are not, of course, restricted only to the field of neural networks. The **Unix** operating system is a good example.

5.6 FFNN entities and the Universal Function Approximation property

THEOREM 5.1 *Any* FFNN *entity implementing the* $\mathcal{C}_1^n$ *family of functions, defined in 5.1 and whose inputs belong to a compact set* $\mathbf{K}$*, is uniformly dense on compacta in the set of all continuous functions in* $\mathbf{K}$.

In order to prove the validity of the above theorem we will resort to the *Stone-Weierstrass theorem*, [Rudin, 1964], and following a procedure similar to the one we used to prove that the family of FFNN holds the universal function approximation property (see appendix D on page 179 and theorem D.3 therein).

Proof

- $\mathcal{C}_1^n$ *is an algebra of functions because it satisfies the three conditions set in definition D.9, namely:*

  1. **addition**: *The sum* $p_{|\underline{\mathbf{x}}|}(\underline{\mathbf{x}})+h_{|\underline{\mathbf{x}}|}(\underline{\mathbf{x}})$ *for* $p_{|\underline{\mathbf{x}}|}, h_{|\underline{\mathbf{x}}|} \in \mathcal{C}_1^{|\underline{\mathbf{x}}|}$, $\underline{\mathbf{x}} \in \underline{\mathbf{K}} \subset \mathbb{R}^n$, $g_i \in \mathcal{G}^i$, *belongs to* $\mathcal{C}_1^{|\underline{\mathbf{x}}|}$ *since:*

$$\begin{aligned} p_{|\underline{\mathbf{x}}|}(\underline{\mathbf{x}}) + h_{|\underline{\mathbf{x}}|}(\underline{\mathbf{x}}) &= g_{|\underline{\mathbf{u}}|+|\underline{\mathbf{v}}|}(f_{|\underline{\mathbf{u}}|}(\underline{\mathbf{u}}) \cup \underline{\mathbf{v}}) + g_{|\underline{\mathbf{u}}'|+|\underline{\mathbf{v}}'|}(f_{|\underline{\mathbf{u}}'|}(\underline{\mathbf{u}}') \cup \underline{\mathbf{v}}') \\ &= \sum_{i=1}^{q} \beta_i \sigma(A_i(f_{|\underline{\mathbf{u}}|}(\underline{\mathbf{u}}) \cup \underline{\mathbf{v}})) + \sum_{j=1}^{q'} \beta_j' \sigma(A_j'(f_{|\underline{\mathbf{u}}'|}(\underline{\mathbf{u}}') \cup \underline{\mathbf{v}}')) \\ &= \sum_{k=1}^{q+q'} \gamma_k \sigma(A_k''(f_{|\underline{\mathbf{u}}''|}(\underline{\mathbf{u}}'') \cup \underline{\mathbf{v}}'')) \end{aligned}$$

  2. **multiplication**: *The product* $p_{|\underline{\mathbf{x}}|}(\underline{\mathbf{x}}) \cdot h_{|\underline{\mathbf{x}}|}(\underline{\mathbf{x}})$ *for* $p_{|\underline{\mathbf{x}}|}, h_{|\underline{\mathbf{x}}|} \in \mathcal{C}_1^{|\underline{\mathbf{x}}|}$, *belongs to* $\mathcal{C}_1^{|\underline{\mathbf{x}}|}$ *since:*

$$\begin{aligned} p_{|\underline{\mathbf{x}}|}(\underline{\mathbf{x}}) \cdot h_{|\underline{\mathbf{x}}|}(\underline{\mathbf{x}}) &= g_{|\underline{\mathbf{u}}|+|\underline{\mathbf{v}}|}(f_{|\underline{\mathbf{u}}|}(\underline{\mathbf{u}}) \cup \underline{\mathbf{v}}) \cdot g_{|\underline{\mathbf{u}}'|+|\underline{\mathbf{v}}'|}(f_{|\underline{\mathbf{u}}'|}(\underline{\mathbf{u}}') \cup \underline{\mathbf{v}}') \\ &= \left(\sum_{i=1}^{q} \beta_i \sigma(A_i(f_{|\underline{\mathbf{u}}|}(\underline{\mathbf{u}}) \cup \underline{\mathbf{v}}))\right) \cdot \left(\sum_{j=1}^{q'} \beta_j' \sigma(A_j'(f_{|\underline{\mathbf{u}}'|}(\underline{\mathbf{u}}') \cup \underline{\mathbf{v}}'))\right) \\ &= \sum_{i,j}^{q \times q'} \beta_i \beta_j' \sigma(A_i(f_{|\underline{\mathbf{u}}|}(\underline{\mathbf{u}}) \cup \underline{\mathbf{v}})) \sigma(A_j'(f_{|\underline{\mathbf{u}}'|}(\underline{\mathbf{u}}') \cup \underline{\mathbf{v}}')) \end{aligned}$$

  Here, σ *is the "cosine squasher" as suggested by* [Gallant and White, 1992]. *See also the relevant sections in appendix D on page 179 and, in particular, equation D-2.*

3. **scalar multiplication**: *The product* $\alpha \cdot p_{|\underline{\mathbf{x}}|}(\underline{\mathbf{x}})$, *for* $p_{|\underline{\mathbf{x}}|} \in C_1^{|\underline{\mathbf{x}}|}$, *and* α *scalar, belongs to* C_1^n *since*:

$$\alpha g_{|\underline{\mathbf{u}}|+|\underline{\mathbf{v}}|}(f_{|\underline{\mathbf{u}}|}(\underline{\mathbf{u}}) \cup \underline{\mathbf{v}}) = \alpha \sum_{i=1}^{q} \beta_i \sigma(A_i(f_{|\underline{\mathbf{u}}|}(\underline{\mathbf{u}}) \cup \underline{\mathbf{v}})) = \sum_{i=1}^{q} \beta_i' \sigma(A_i(f_{|\underline{\mathbf{u}}|}(\underline{\mathbf{u}}) \cup \underline{\mathbf{v}}))$$

- C_1^n *separates points on* $\mathbf{K}$*:*

$$\forall \mathbf{x}_1, \mathbf{x}_2 \in \mathbf{K}; \mathbf{x}_1 \neq \mathbf{x}_2 \; \exists \, f \in C_1^n \mid f_{|\mathbf{x}_1|}(\mathbf{x}_1) \neq f_{|\mathbf{x}_2|}(\mathbf{x}_2)$$

Since $f_{|\mathbf{x}_1|}(\mathbf{x}_1) = g(\mathbf{u} \cup \mathbf{v})$ *(see definition 5.1 on page 57) then, the above statement requires that there is a function* f *in the* C_1^n *family for which:* $f_{|\mathbf{x}_1|}(\mathbf{x}_1) \neq f_{|\mathbf{x}_2|}(\mathbf{x}_2)$*. This implies the following:*

$$\begin{aligned} g(\mathbf{u} \cup \mathbf{v}) &\neq g(\mathbf{u}' \cup \mathbf{v}') \implies \\ \sum_i w_i \sigma(A(\mathbf{u} \cup \mathbf{v})) &\neq \sum_i w_i \sigma(A(\mathbf{u}' \cup \mathbf{v}')) \implies \\ \sigma(A(\mathbf{u} \cup \mathbf{v})) &\neq \sigma(A(\mathbf{u}' \cup \mathbf{v}')) \implies \\ A(\mathbf{u} \cup \mathbf{v}) &\neq A(\mathbf{u}' \cup \mathbf{v}') \end{aligned}$$

In order to show that, choose $A \in \mathcal{A}^n$ *such that* $A(\mathbf{u} \cup \mathbf{v}) \neq A(\mathbf{u}' \cup \mathbf{v}')$*. Because* $\mathbf{u} \cup \mathbf{v} \neq \mathbf{u}' \cup \mathbf{v}'$ *(e.g.* $\mathbf{x}_1 \neq \mathbf{x}_2$*), then* $A(\mathbf{u} \cup \mathbf{v}) \neq A(\mathbf{u}' \cup \mathbf{v}')$ *(remember that* $A(\mathbf{u} \cup \mathbf{v}) = (\mathbf{u} \cup \mathbf{v})\mathbf{w} + b$*) if we choose either* $\mathbf{w}$ *or* b *to be non-zero. This means that* $f_{|\mathbf{x}_1|}(\mathbf{x}_1) \neq f_{|\mathbf{x}_2|}(\mathbf{x}_2)$ *and, therefore,* C_1^n *separates points on* $\mathbf{K}$*.*

- C_1^n *vanishes at no point of* $\mathbf{K}$*:*

$$\forall \mathbf{x} \in \mathbf{K} \; \exists f \in C_1^n \mid f(\mathbf{x}) = c, c \in \mathbb{R}, c \neq 0$$

Since $f_{|\mathbf{x}|}(\mathbf{x}) = g(\mathbf{u} \cup \mathbf{v})$ *(see definition 5.1 on page 57) then, the above statement requires that there is a function* f *in the* C_1^n *family for which* $f_{|\mathbf{x}|}(\mathbf{x}) = g(\mathbf{u} \cup \mathbf{v}) \neq 0$ *This implies the following:*

$$\begin{aligned} g(\mathbf{u} \cup \mathbf{v}) &\neq 0 \implies \\ \sum_i w_i \sigma(A(\mathbf{u} \cup \mathbf{v})) &\neq 0 \implies \\ \sigma(A(\mathbf{u} \cup \mathbf{v})) &\neq 0 \implies \\ A(\mathbf{u} \cup \mathbf{v}) &\neq 0 \end{aligned}$$

In order to show that, choose $A \in \mathcal{A}^n$ *such that* $A(\mathbf{u} \cup \mathbf{v}) \neq 0$ *(remember that* $A(\mathbf{u} \cup \mathbf{v}) = (\mathbf{u} \cup \mathbf{v})\mathbf{w} + b$*). This is easy because even if* $\mathbf{u} \cup \mathbf{v}$ *is zero,* b *can always be chosen to be non-zero, hence* $A(\cdot)$ *is non-zero*[5]*.*

[5] This is one of the reasons why a bias term is needed at each neuron.

Classes 2 and 3 are subsets of the class 1, since the only differences between them are their topology (network structure) and the target values that each FFNN is trained with. Hence, they are, too, universal function approximators.

5.7 Single FFNN and C_1 entity: comparison of training times

In this section we will attempt to quantify the difference in training times between a C_1 entity and a single FFNN. It is the extension to section 4.4.2 on page 38. Here are the main assumptions regarding the architecture of the two models:

i. each FFNN has only one hidden layer,

ii. the number of hidden layer units of a given FFNN is a percentage of the number of its inputs. Call this percentage α,

iii. the number of inputs, n_{inp}, of each of the FFNN units making up the entities is fixed and may be expressed as a percentage of the total number of input dimensions, N. Call this percentage β. Thus,

$$n_{inp} = \beta N \tag{5.11}$$

iv. the total number of FFNN units making up the entity, n_f, should be sufficient to cover all inputs. For example, if there are 100 input dimensions and was decided that each FFNN has a number of inputs which is 20% of the total number of input dimensions (e.g. $100 \times 20\% = 20$ inputs) then there must be at least 5 FFNN in the entity. As a matter of fact, it is good practice to add a few extra FFNN in the entity. Thus,

$$n_f = \gamma \frac{N}{n_{inp}} = \gamma \frac{N}{N\beta} = \frac{\gamma}{\beta} \tag{5.12}$$

where, γ is a percentage (greater than 100% so that no inputs are left out). Note that the connections between the FFNN of the entity are not adjustable.

The time required by a FFNN to do a single training iteration (forward and backward step) over a single input vector of N dimensions is proportional to the number of its adjustable parameters[6]

$$T_S \propto \alpha N(N+2) + 1 \tag{5.13}$$

[6] These are all the weights and biases. Refer to section 4.4.2 on page 38 and equation 4.3 therein for details.

The time required by an entity to do the same is proportional to its total number of adjustable parameters. This is the sum of the number of adjustable parameters of each FFNN making up the entity. Recall that there are n_f identical FFNN, each with n_{inp} inputs. Thus,

$$T_E \propto n_f \cdot \alpha\, n_{inp}(n_{inp}+2)+1 = \frac{\alpha\gamma}{\beta}\beta N(\beta N+2)+1 = \alpha\gamma N(\beta N+2)+1 \tag{5.14}$$

Now, let us compare the training times of the two models in terms of the ratio[7] T_S/T_E:

$$T_S/T_E = \frac{\alpha N(N+2)+1}{\alpha\gamma N(\beta N+2)+1} \tag{5.15}$$

The above expression has a horizontal asymptote[8] given by:

$$Y_a = \lim_{N\to\infty} T_S/T_E = \frac{1}{\beta\gamma} \tag{5.16}$$

Thus, provided that the same α is used, the training time of an entity will be shorter than that of a single FFNN, by a factor which will approach $\beta\gamma$ as the number of input data dimensions (N) increases.

In conclusion,

- the ratio of the training times of the two models is independent of the number of hidden layer units of single FFNN. Thus, these results apply for any size of single FFNN as long as the same α is used for both models,
- the training time of the entities will be less than that of single FFNN as long as $\beta\gamma < 1$. This means that the training time of an entity will be the same even if more FFNN are added to it (in order, perhaps, to improve generalisation) as long as the number of inputs to these FFNN is kept sufficiently small.

EXAMPLE 5.2 *An entity was constructed using* $\gamma = 300\%$ *- e.g. the number of* FFNN *in that entity is three times as much as the minimum number of* FFNN *required to cover all the input dimensions. This means that if there were* 500 *input dimensions in the training data and that each* FFNN *of the entity had* 50 *inputs, then the total number of* FFNN *making up the entity was* $\frac{500}{50} \times 300\% = 30$. *Figure* 5.4 *shows plots of the quantity* $\frac{T_S - T_E}{T_E} \times 100\%$ *as the number of input dimensions varies from* 200 *to* 1,000 *and for different* β. *The top plot corresponds to* $\beta = 1\%$ *(e.g. each* FFNN *had a number of*

[7] The constant of proportionality in the expressions T_S and T_E is the same.

[8] In practical situations $N \to \infty$ may be interpreted as $N > 200$ or $N > 300$ depending on β and γ. This is shown in the plots of figure 5.4.

inputs corresponding to 1% *of the total number of input dimensions) and the bottom plot corresponds to* $\beta = 30\%$. *Remember that the entities will be faster than a single* FFNN *as long as* β *is less than* $1/\gamma$, *e.g.* $1/3$.

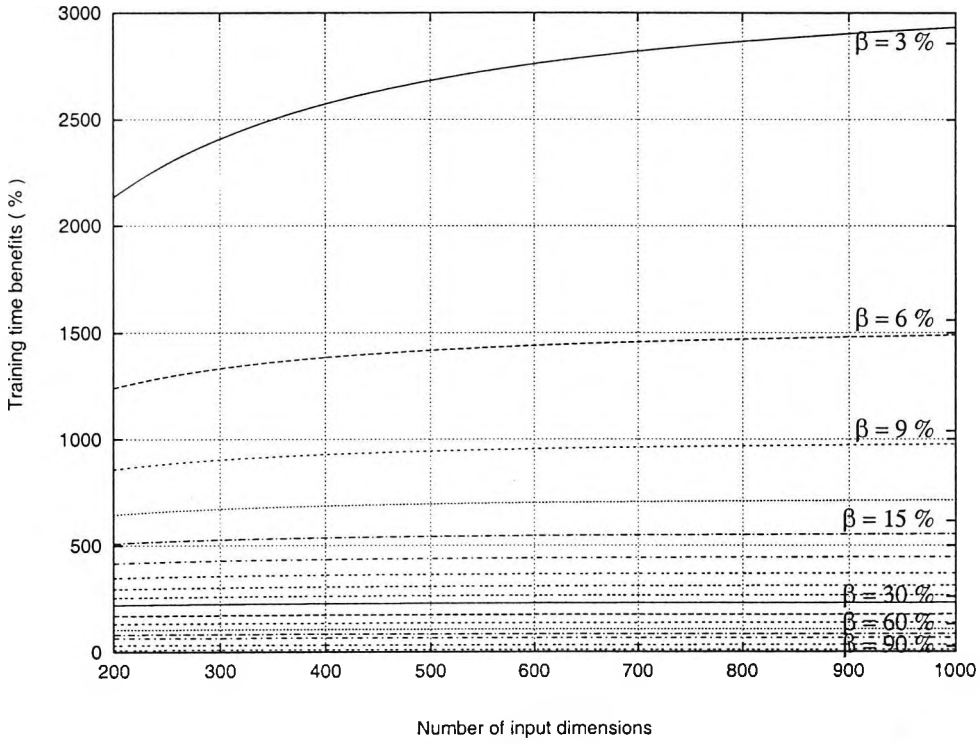

Figure 5.4: Comparison of training times of a single FFNN and an entity: $\frac{T_S - T_E}{T_E} \times 100\%$ versus the number of data dimensions, for $1\% \leq \beta \leq 30\%$

The fastest entity training occurs when each constituent FFNN *has number of inputs which is* 1% *of the data dimensions. That means* 2 *inputs when the dimensions are* 200 *and* 10 *when the dimensions are* 1,000. *In this case the entity is faster than a single* FFNN *by* 25 *to* 30 *times. On the other hand, the training times of the entity and the single* FFNN *are the same when the number of inputs to each entity* FFNN *ranges from* 60 *to* 300, *e.g.* 30% *of the number of data dimensions.*

For the sake of achieving a good generalisation, the number of inputs to each FFNN *making up the entity should neither be too low (e.g.* 1%*) nor too high (e.g.* 30%*). A more wise choice would be something like* 10%.

In this end, an experiment has been conducted in order to verify these results in practice and, also, to investigate the generalisation ability of the two models when the training time benefits maximise. The experiment consisted of training and testing an entity and a single FFNN *with data of* 500 *input dimensions. This data was obtained artificially*

via the **Levy6** *function (see section 6.3.5 on page 87 for more details). The two models were constructed with the following parameters:*

- $\alpha = 10\%$*: the number of hidden layer units of each* FFNN *is just* 10% *of the number of its inputs*[9]*. Thus, the architecture of the main single* FFNN *was* $500 \times 50 \times 1$ *and, therefore, contained* 25,050 *weights,*
- $\beta = 10\%$*: each single* FFNN *making up the entity had* 50 *inputs (e.g.* 10% *of* 500 *input dimensions). Thus the architecture of each single* FFNN *in the entity was:* $50 \times 5 \times 1$ *and, therefore, contained* 255 *weights,*
- $\gamma = 300\%$*: the total number of single* FFNN *in the entity must be three times as much as the number required to cover all* 500 *inputs. This means that the entity consisted of* $500/50 \times 300\% = 30$ *single* FFNN *and contained a total of* 7,650 *weights.*

With these construction parameters, the training time of the entity is predicted to be $1/\beta\gamma = 1/(10\% \times 300\%) = 3$ *times shorter than that of the main single* FFNN*.*

The training data consisted of 70 *vectors while the test data consisted of* 2,000 *vectors. The training/test procedure was repeated for* 50 *times. Each network was trained for* 1,000 *iterations. Tables* 5.2 *and* 5.3 *show the* **minimum, maximum, mean** *and* **standard deviation** *of the* **training time** *and* **approximation error** *of the single* FFNN *and the entity.*

	MINIMUM	MAXIMUM	MEAN	STD. DEV.
Entity	0.0475	0.1067	0.0629	0.0109
single FFNN	0.0722	0.2138	0.1109	0.0196

Table 5.2: **Approximation error** results

(seconds)	MINIMUM	MAXIMUM	MEAN	STD. DEV.
Entity	786	805	792.9	3.49
single FFNN	2,577	2,905	2,699.3	56.22

Table 5.3: **Training time** results

A comparison of **mean** *training times shown in table* 5.3 *indicates that the single* FFNN *takes more than three times longer to train than the entity:* $2699.3/792.9 = 3.4$*. As far as the approximation capability of the two models is concerned (table* 5.2*), the entity*

[9] Note that all single FFNN consisted of a single hidden layer.

generalises much better than the single FFNN *with a* **mean** *error of* 0.0629 *compared to* 0.1109. *Thus, not only the entity is trained significantly faster than the single* FFNN *but also its generalisation ability is better. It is also comparable to the generalisation ability of the entities in test* VARIDIM *of section 6.3.6.5 on page 110 which, according to figure* 6.1 *on page 90 contained twice as many weights.*

5.8 Benefits from using the entities

The adoption of the entities as an alternative to a single, solid FFNN solves the following problems:

1. The *coarse-grain* parallelism which characterises the entities favours a more practical and efficient implementation. By increasing the complexity of the basic computational element (e.g. this is now a FFNN rather than a neuron), a better allocation of resources can be achieved; the mapping of processes to processors is now more balanced. Furthermore, by reducing the number of basic computational elements, the processor to processor communication needs are minimised.

2. The dimensionality of input data no longer governs the number of inputs and size of each FFNN. The entities can deal with high-dimensional data by increasing the number of basic elements (FFNN) and not the number of inputs to each FFNN. Thus, FFNN can be created at any convenient size and added to the entity, virtually, without any serious restrictions – given the coarse-grain parallelism advantage.

3. The relatively small size of the basic processing element of the entity not only allows for successfully dealing with high-dimensional data – which for the case of a solid FFNN was extremely difficult due to the *curse of dimensionality* – but also the training of each FFNN is more effective in the absence of pathologies such as *premature neuron saturation*, *complex error surfaces*, *local minima*, etc.

4. The effects of the NP-completeness results for the loading problem are not as dramatic for the case of the entities as they are for the case of solid neural networks. The difference is that instead of having a single neural network with a large number of inputs, we have a large number of smaller FFNN with significantly less number of inputs.

5. The study, visualisation and interpretation of the learning process are ameliorated with the adoption of the FFNN as the basic building block of the entities – not

too large and complex as in the case of a solid FFNN, not too small and simple as in the case of a neuron – they are now effected within a framework which approaches that of traditional AI, e.g. where information is manipulated at the level of *symbols*.

Still, information is distributed and stored implicitly in the weights of each FFNN and the weights of their connections with other FFNN. But it also exists on a higher level, that of the numerous single FFNN, which can identify more readily, meaningful information.

Thus, the entity learns and process information in a more explicit way – something which brings us a little bit closer to symbolic systems and all the benefits that implies without departing from connectionist principles.

Perhaps a system exhibiting a dual character featuring a connectionist and at the same time symbolic self will be more successful than the unadulterated alternatives of the past.

5.9 Summary

A review of past and present trends in the area of *modular neural network architectures* has revealed that although there has been much effort in improving the generalisation ability of these networks, the aspect of scaling up and the subsequent problems due to the *curse of dimensionality* have received little or no attention. The main motivation of this research was to investigate how the effects of the *curse of dimensionality* on neural networks can be minimised while generalisation ability is not compromised.

This chapter was dedicated in describing the concept of *feed forward neural network entities*, a methodology which uses the ideas of *modularisation* and *function decomposition* in tackling problems associated with *scaling up* single neural networks. The structure of the entities can be described using the same taxonomy as with ordinary neural networks, where units belong to the finest level, layers to a coarser level and networks to a still coarser level of classification, but not at the end of the scale!

An entity is composed of units which are themselves neural networks rather than simple neurons. These units may be layered and linked with connections of adjustable strength just like ordinary neurons in the case of single FFNN. The connectivity of the elements of an entity as well as their training targets were two criteria we used to distinguish between three different entity classes, namely class 1, 2 and 3. The connectivity of

classes 2 and 3 is more ordered than that of class 1 whereas the training targets for classes 1 and 2 are simpler than those of class 3.

Currently, all three entity classes may implement only *single output functions*. Designing entities implementing functions with higher output dimensions can be a direction for future work.

The *universal function approximation* capability of the entities was proved using the *Stone-Weierstrass theorem*. Finally, a simple language, ***np***, which may be used to create, train and test FFNN and entities of FFNN has been introduced.

The main benefit from using the entities instead of single feed forward neural networks is that they can cope well with problems of extremely high dimensions because of their distributed nature. Also, parallel implementation of the entities is much more practical than single FFNN because they favour a *coarser grain* parallelism. In this end, the effects of the NP-completeness results for the loading problem can be reduced. Also, in sequential mode, there are enormous benefits obtained by the reduced training times required by the entities.

Additionally, the entity – because of the adoption of the FFNN as the basic building block – learns and process information in a more explicit way while it promotes a computational model which can be studied with an arbitrary level of abstraction.

The next chapter will investigate, at a practical level, the generalisation ability and training time of the entities and how they compare to those of single FFNN.

CHAPTER VI

EMPIRICAL RESULTS

Various tests comparing the performance of a single FFNN and a FFNN entity were carried out. The results are presented and discussed herein.

6.1 Introduction

In previous sections, it was shown that the families of functions represented by the FFNN entity classes C_1, C_2 and C_3 are *universal function approximators* and, therefore, can approximate arbitrarily well any real, continuous function. The theoretical importance of this result is great because it supports the claim that the expressive power of the entities neural network architecture is equivalent to that of a single FFNN[1].

Moreover, it was argued that the application of FFNN entities in optimisation and classification tasks involving a large number of input parameters will alleviate a significant number of *practical* problems which are currently hindering the learning process of single FFNN when applied to the same tasks. In particular, it is claimed that the entities are largely immune to problems relating to the *curse of dimensionality* such as *extremely complex error surfaces*, *large number of local minima*, *premature neuron saturation* etc. These problems are not only responsible for *unstable* and *inconsistent* training[2] but also for *much longer training times* and *cumbersome architectures.*

Finally, training times can be reduced even further when parallelised entity implementations are used. The efficiency of these implementations is a prominent feature of the entities and emanates from the fact that their structure promotes a *coarse-grain* type

[1] Recall that a similar result, regarding the universal function approximation property, holds for single FFNN too – see section 3.3 on page 24 for details.

[2] Instability because of *premature neuron saturation*, as it was pointed out in section 4.4.4 on page 42, and inconsistency because of the *large number of local minima*, as it was explained in sections 4.4.2 on page 38 and 4.4.3 on page 41.

of parallelism, as opposed to the *fine-grain* parallelism model of the single FFNN.

In this chapter we would like to present the empirical results of the various tests and experiments we have conducted with the purpose of comparing some aspects of the entities and single FFNN.

6.1.1 Limitations

We would like to point out that the experimental procedure presented in this chapter compares a single FFNN with the three entity models on the basis of only *one* problem – the **Levy6** function. Naturally, the results would have been more conclusive if the comparison had been based on a wider choice of data and neural network models.

For example, it is very likely that entities will not perform well on the *n-bit parity* problem because of the fact that each FFNN in an entity is trained with only a partial set of the data inputs. On the other hand, a single neural network which deals with the full set of the inputs will surely perform better.

Thus, we would like to stress the fact that although the entities do perform better than single FFNN on the basis of the **Levy6** data sets, more experiments, based on a wider choice of problems and training data, must be carried out in order to obtain more conclusive results.

6.1.2 Statistical analysis

So why is there a need for an experimental evaluation of neural networks and proper statistical analysis of these results?

> " We do need experiments in neural network research because the methods we employ and the data we want to analyse are too complex for a complete formal treatment. I.e. for a given data analysis problem we do not have the formal instruments to decide which of the methods is the optimal one ... the last word in the decision is always spoken by an empirical check, an experiment, as in any other science that needs empirical evaluation of its theories." [Flexer, 1996]

The tests described in this chapter are divided in two main categories:

1. **Generalisation Ability**

2. **Parallelisation**

The first category, **Generalisation Ability**, consists of two tests, VARIDIM and CONSDIM, whereas the **Parallelisation** category consists of only one test. Section 6.2 outlines the methodology followed during the various experiments of each category.

Section 6.3 deals with the two tests under the **Generalisation Ability** category. Below is the layout to be followed in explaining the procedures and presenting the results of the two tests VARIDIM (in section 6.3.6) and CONSDIM (in section 6.3.7):

- **Procedure/Methodology**: contains a description of all the participating networks, data sets and the training and test procedures.
- **Network sizes** (only for VARIDIM): explains how the participating networks' sizes (in terms of their number of weights) relates to the number of input dimensions.
- **Training time results**: presents and analyses results regarding the training time of the participating networks.
- **Sample error results**: presents and analyses results regarding the **sample error** (e.g. error during training) of the various networks.
- **Approximation error results**: presents and analyses results regarding the **approximation error** (e.g. error during testing) of the various networks.

Section 6.4 deals with the one test under the **Parallelisation** category. It presents and analyses the results regarding the time required by four class 1 FFNN entities trained in *sequential* and *parallelised* modes.

Finally, section 6.5 concludes the chapter with a summary of all the experiments carried out and an outline of the obtained results.

6.2 Proposed methodology

6.2.1 Generalisation Ability

Two tests will be carried out using a total of eight network configurations: four single FFNN, two class 1 entities, one class 2 and one class 3 entity.

In VARIDIM, **the number of training vectors will be kept constant as the number of input dimensions of each network increases.** This test aims at demonstrating how the different networks cope with increasing data dimensionality. Ideally, the dimensionality of the training data should have no effect on the generalisation ability of a neural network. In practice, however, the generalisation ability of a single FFNN is usually

reduced as the number of input dimensions increases. With this test we wish to investigate and compare the effect of increasing data dimensionality on the performance of the participating networks.

For each different network and for each different number of input dimensions we will repeat the training process with different initial conditions (e.g. starting with random weights) for 50 times and measure, each time, the **sample** and **approximation errors** and **training time**.

In ConsDim, **we will keep the number of input dimensions and size of each network fixed while the number of training vectors varies.** The aim of this test is to investigate the effect of the increasing number of training vectors on the **sample** and **approximation error** of the different networks.

For each different network and for each different number of training vectors we will repeat the training process with different initial conditions (e.g. starting with random weights) for 50 times and measure, each time, the **sample** and **approximation errors** and **training time**.

The **training data** for the networks of both tests is obtained artificially using the **Levy** data generating function. This procedure is described in section 6.3.5.

The *means* of the **sample** and **approximation error** results of both experiments will be tested for statistical significance using the *one-tailed t-test* at a 5 % significance level. Why is it necessary to test for statistical significance the obtained experimental results? [Flexer, 1996] emphasises the fact that statistical evaluation is necessary for neural network experiments and advises to use the *t-test* which "should be computed to test the significance of the difference between means".

The *variance*[3] of the **sample** and **approximation error** results of the two experiments will be tested for statistical significance using the *one-tailed F-test* at, again, a 5 % significance level. More details about the statistical significance tests can be found in section 6.3.3 and also in appendix E on page 183.

A more detailed description of the **procedure** followed for VariDim can be found in section 6.3.6.1. Section 6.3.6.2 outlines the relationship between network size (in terms of weights) and the dimensionality of the input data. Then, in sections 6.3.6.3, 6.3.6.4 and 6.3.6.5 the obtained results (**training time**, **sample error** and **approximation error** respectively) are described and discussed.

[3] *Variance* is the square of *standard deviation*. The two terms are directly related and will be used interchangeably throughout this section.

CONSDIM's procedure is described in section 6.3.7.1, in sections 6.3.7.2, 6.3.7.3 and 6.3.7.4 the obtained results (**training time**, **sample error** and **approximation error** respectively) are described, analysed and discussed.

6.2.2 Parallelisation of the entities' training procedure

In this test, the time benefits gained from parallelising the training procedure of a C_1 entity – as opposed to the conventional, sequential training – will be assessed. The parallelisation scheme used was a very simple one. At first, the network's units are grouped according to their respective layers. Secondly, the units of the first layer are divided evenly among **four** networked workstations and their parallel processing starts. When all first layer units are trained, the same procedure is repeated for the units of the second layer and so on, until all the units of the entity are trained. When the training of all units in a given layer is completed, the weights are transfered to a central store where the test procedure will take place sequentially.

The evaluation consisted of training four different C_1 entity networks of various sizes, first *sequentially* and then in *parallel* with the same data set and for the same number of iterations (1,000). This procedure was repeated for 50 times. At the end of each run, the **training time** was recorded. The *minimum*, *maximum*, *mean* and *standard deviation* of the **training time** were then calculated over the 50 runs.

6.3 Generalisation Ability

6.3.1 Introduction

Two tests were carried out. The aim of the first test, VARIDIM, was to investigate the *performance* of the entities and single FFNN when the **number of input dimensions** varied from 100 to 1,000 while the **number of training vectors** was kept constant. The aim of the *second* test, CONSDIM, was to investigate the *performance* of the entities and single FFNN when the **number of training vectors** varied from 20 to 220 while the **number of input dimensions** was kept constant.

Performance is characterised by the triplet:

- the **time** required to complete a fix number of training iterations,
- the **sample error**, e.g. the error over the *training set*,
- the **approximation error**, e.g. the error over the *test set*.

A problem which has been encountered during this evaluation, is that there are structural differences not only between entities and single FFNN networks but also between the different entity classes (e.g. C_1 is structurally different to C_2 and C_3). Thus, it is quite difficult to say whether two network configurations of different architectures are of the same "*capacity*". This translates to the following problem: "*given a training data set of so many dimensions, what should the structural parameters of each evaluated network be in order not to favour a particular network just because it has a larger capacity than the others*"[4]. The *total number of weights* was chosen as the categorisation criterion. Essentially, the number of weights in a network, either being an entity or a single FFNN, is proportional to the cost – e.g. time and computer resources – one has to pay to train that network. Therefore, for a given test, the networks were created so that they had *approximately*[5] the same number of weights.

6.3.2 Objectives

In the two tests of the **Generalisation** category we are interested in comparing the performance (**training time**, **sample error** and **approximation error**) of the participating networks. In particular, we sought an answer to the following questions:

1. Do the networks **consistently** converge to a low enough **sample error** level?
2. Is the corresponding **approximation error** low enough?
3. Do the **sample** and **approximation errors** depend on the number of **input dimensions** (for VARIDIM) or the number of **training vectors** (for CONSDIM)?

The **first** question refers to the complexity of the error surface for the different networks, in terms of the number of local minima it contains. In previous discussions (for example, in Chapter 4 on page 29), we argued that if the *complexity* of the error surface is high, the training procedure will not be consistent – sometimes it will converge to one local minimum, sometimes to another – thus yielding very different sample errors. Therefore, it is necessary to identify those network configurations which, for the same training

[4] A similar problem is, perhaps, the categorisation of boxers; there are many parameters which might be used to compare and categorise boxers (such as, perhaps, weight, height, arms' length, or I.Q.) but there is no absolute criterion.

[5] Here, we use the adverb *approximately* because even the number of weights can not be controlled exactly. For example, take the equation $L_1 = \frac{W}{L_0+1}$ which governs the number of weights, W, of a single FFNN of a single hidden layer of L_1 units, L_0 inputs, one output. It does not have **integer solutions** for all W and L_0. This problem becomes more complicated when dealing with the entities which are composed of a lot of single FFNN.

data, are systematically associated with complex error surfaces. These networks are obviously inadequate.

The **second** question refers to the utility of the given network configuration since the **approximation error** is a measure of how well the trained networks will perform on the whole input data domain. Given that the test set represents accurately the input data distribution, the most useful network will be the one with the *lowest approximation error.*

With the **third** question, for VARIDIM, we are investigating the effect of two problems: the *curse of dimensionality* and *premature neuron saturation.* These problems are known to hinder neural networks when the *dimensionality* of the input data becomes large. For CONSDIM, we are interested to see how the number of training vectors affects the overall performance of each of the participating networks.

6.3.3 Statistical significance tests

When using a *statistical significance test* to compare the *mean* or *variance* of the **sample** and **approximation errors** of an entity and a single FFNN (in any of the two tests VARIDIM and CONSDIM), we start with three hypotheses:

1. The *null hypothesis*, H_0, which says that the *mean* / *variance* of the **sample** / **approximation errors** of an entity and a single FFNN **do not differ significantly** at the 5 % significance level.
2. The H_1 hypothesis which says that the *mean* / *variance* of the **sample** / **approximation error** of the entity is **significantly higher** than that of the single FFNN.
3. The H_2 hypothesis that the *mean* / *variance* of the **sample** / **approximation error** of the entity is **significantly lower** than that of the single FFNN.

Eventually, one of the three hypotheses will be accepted and the other two rejected. The *t-test* and the *F-test* will be performed for each entity / single FFNN pair and for each number of input dimensions (if in VARIDIM) or each number of training vectors (if in CONSDIM). Then, the percentage of acceptance of each hypothesis over some range of input dimensions / number of training vectors will be calculated. In order to make it easier to draw any conclusions associating performance with the number of input dimensions / number of training vectors we have decided to perform the two statistical tests over two ranges:

- the **lower range**: for 100 to 500 input dimensions or 10 to 120 training vectors,

- the **upper range**: for 500 to 1,000 input dimensions or 120 to 200 training vectors.

Thus, for each test (VARIDIM or CONSDIM) and for each **sample / approximation error** result, two tables, corresponding to the **lower** and **upper range** of input dimensions / number of training vectors will be constructed as follows:

- each *row* of the table corresponds to an entity network,
- each *column* of the table corresponds to a single FFNN,
- each entry of the table contains the triplet (α, β, γ), associated with the results of performing the *t-test* or the *F-test* on the entity network which corresponds to the row of the entry and the single FFNN which corresponds to the column of this entry,
- in the triplet (α, β, γ), α is the percentage of acceptance of the *null hypothesis* or H_0. β is the percentage of acceptance of the H_1 hypothesis. γ is the percentage of acceptance of the hypothesis H_2.

For example, take table 6.4 on page 100. Its top, left-hand entry is (33, 5, 62) and corresponds to the comparison of the means (that was a *t-test*) of the **sample error** of the two networks $\mathcal{C}_1$ (row) and $\mathcal{N}_1$ (column), over the lower range of input dimensions (e.g. from 100 to 500). According to this entry, the *null hypothesis* was accepted 33 % of the times, the H_1 hypothesis was accepted 5 % of the times and the H_2 hypothesis was accepted 62 % of the times.

6.3.4 Presentation of the results

The *mean*, *standard deviation*, *minimum* and *maximum* of the three quantities which determined the performance of each network, e.g. **training time**, **sample error** and **approximation error**, are calculated for a given number of input dimensions or training vectors and over the 50 repeats of the training / test procedure. These data were then plotted against the number of **input dimensions** (for VARIDIM) and the number of **training vectors** (for CONSDIM). Additionally, the **sample** and **approximation error** were plotted as scattered points for each different number of **input dimensions** and number of **training vectors**.

An absolute criterion for the comparison of the performance of the different networks is the **lowest**, **highest** and **average** of the *mean* and *standard deviation* of the **sample** and **approximation errors** calculated according to the procedure outlined in section 6.3.6.1. These data are presented in various tables throughout this chapter.

6.3.5 The Levy data-generating procedure

In [Levy and Montalvo, 1985] a procedure for generating multi-modal continuous functions with arbitrary number of inputs is outlined. This procedure was followed in order to create datasets with a variable number of **input dimensions** to be used in the tests following. Certainly there are other procedures which may be used in such tasks. The choice of this particular function was based on the complexity of its surface and the large number of its roots. Furthermore, this same function was used by Levy and Montalvo to test their *Tunnelling Algorithm*, a global optimisation technique.

The original function with variable number of inputs is the following:

$$\begin{aligned}\mathcal{F}_2(x_1,\cdots,x_r) = &\sin(3\pi x_1)^2 + \\ &+\sum_{i=1}^{r-1}(x_i-1)^2(1+10\sin(3\pi x_{i+1})^2) + \\ &+(x_r-1)(1+\sin(2\pi x_r)), \quad \text{for } -1<x_i<1\end{aligned} \tag{6.1}$$

The following **modifications** were made in order to adjust the *range* and *domain* of the original function to the requirements of our neural network models:

- normalisation coefficients were introduced in order to limit the output within the range of, approximately, -1 and 1,
- the transformation $x_i \rightarrow 2x_i - 1$ was applied to each of the input variables so that the function is adjusted to the new domain: $0 < x_i < 1$.

The final form of the data-generating function is the following:

$$\begin{aligned}\mathcal{F}_2(x_1,\cdots,x_r) = &\frac{1}{10}\sin(3\pi(2x_1-1))^2 + \\ &+\frac{1}{1.2\sqrt{r}}\sum_{i=1}^{r-1}(2x_i-2)^2(1+10\sin(3\pi(2x_{i+1}-1))^2) + \\ &+\frac{1}{10}(2x_r-2)(1+\sin(2\pi(2x_r-1))), \quad \text{for } 0<x_i<1\end{aligned} \tag{6.2}$$

The inputs to the Levy function (and, thus, the inputs to the neural networks) are generated by a pseudo-random number generator (C-language's **lrand48()** function which returns long integers *uniformly* distributed over the interval $[0, 2^{31})$. Refer to the **Unix manual pages**, section **3C**, for more details) and normalised to the interval (0.0, 1.0). Typically, a representative set of inputs, produced by this pseudo-random number generator, will have a *standard deviation* of 0.28 and be centred around 0.5.

6.3.6 Generalisation Ability: the VARIDIM test

6.3.6.1 VARIDIM: Methodology

The methodology followed for VARIDIM is detailed below:

- The networks to be tested are described in the following table:

NAME	DESCRIPTION
$\mathcal{C}_1$	a C_1 **entity** model
$\mathcal{C}_{1,big}$	a C_1 **entity** with 66% more weights than $\mathcal{C}_1$
$\mathcal{C}_2$	a C_2 **entity** model with as many weights as $\mathcal{C}_1$
$\mathcal{C}_3$	a C_3 **entity** model with as many weights as $\mathcal{C}_1$
$\mathcal{N}_1$	a **single** FFNN with 35% less weights than $\mathcal{C}_1$
$\mathcal{N}_2$	a **single** FFNN with as many weights as $\mathcal{C}_1$
$\mathcal{N}_3$	a **single** FFNN with 55% more weights than $\mathcal{C}_1$
$\mathcal{N}_4$	a **single** FFNN with 135% more weights than $\mathcal{C}_1$

Table 6.1: VARIDIM, description of the evaluated networks

  The relationship between the **total number of weights** and the **number of input dimensions** for each of the evaluated networks $\mathcal{C}_1$, $\mathcal{C}_{1,big}$, $\mathcal{C}_2$, $\mathcal{C}_3$, $\mathcal{N}_1$, $\mathcal{N}_2$, $\mathcal{N}_3$ and $\mathcal{N}_4$ is depicted in figures 6.1, 6.2, 6.3, 6.4, 6.5. 6.6, 6.7 and 6.8, respectively.

- each FFNN unit of the entity models had 12 to 35 inputs, one hidden layer of 10 to 20 units, and a single output,
- the training data consisted of 70 vectors[6] and was produced by the Levy function, equation 6.2,
- the number of input parameters varied from 100 to 1,000 (step 20),
- the number of weights was increased as the number of input data dimensions increased and according to the percentages of table 6.1. For the case of the single FFNN, this meant increasing the units of the hidden layer. For the case of the entities, it just meant increasing the number of FFNN while their size (e.g. number of inputs as well as number of weights) was unchanged,

[6] This particular number of training data vectors was chosen after experimenting with 50, 70 and 100 vectors. The number of 50 vectors was too small for extracting reliably any conclusions regarding the generalisation performance of the networks. Whereas, the number 100 was too large, not only because of the danger of *memorisation* (as opposed to *generalisation*) but also because the training times were going to be larger. Also, note that the results of these tests will be studied *comparatively rather than in absolute terms.*

- for each different training data set, both the entities and the single FFNN were trained for 1,000 iterations[7],
- the test data consisted of 2,000 vectors. None[8] of these vectors was used for training. The final approximation error was calculated using the *mean squared error measure* (see equation 3.6 on page 26),
- for a given number of inputs, the training/test procedure was repeated for 50 times with the same data and network architecture but with different starting weights. The **training time** and **sample** and **approximation errors** were recorded for each of these training attempts and then the *maximum*, *minimum*, *mean* and *standard deviation* were calculated as follows:

```
procedure VARIDIM:
  for number of input dimensions i := 100 to 1,000 step 20 do
    produce training data set
    produce test data set
    produce neural network/entity
    calculate total number of weights, numW
    initialise sum and sumsum to zero
    reset minimum error_i and maximum error_i
    for training attempts j := 1 to 50 do
      initialise weights to random
      train neural network/entity for 1,000 iterations
      test neural network/entity
      calculate error_ij using equation for mean squared error
      store error_ij
      update minimum error_i := minimum of (error_ij and minimum error_i)
      update maximum error_i := maximum of (error_ij and maximum error_i)
      sum := sum + error_ij
      sumsum := sumsum + error_ij^2
    end
    mean error_i := sum / 50
    standard deviation of error_i := sqrt( sumsum / 50 - (mean error_i^2) )
    store minimum error_i and maximum error_i
    store mean error_i and standard deviation of error_i
  end
end VARIDIM.
```

[7] This particular number of training iterations was considered after experimenting with smaller and larger numbers. In these preliminary tests, the *rate of change* of the training error, for most of the networks, was approaching zero after about 1,000 iterations. Using a larger number of iterations would not only have resulted in longer training times but, also, would have risked *over-fitting*.

[8] Remember that both training and test input vectors are generated randomly with different seed.

- the quantities *minimum error*$_i$, *maximum error*$_i$, *mean error*$_i$ and *standard deviation of error*$_i$ were, then, plotted against the total number of inputs, i. See, for example, figure 6.17 which refers to the **sample error** results of the $\mathcal{C}_1$ entity network.
- the values of *error*$_{ij}$ were plotted (as scattered points) against the total number of inputs, i. See, for example, figure 6.25 which refers to the **sample error** results of the $\mathcal{C}_1$ network. On each of the **scatter plots** a line has been fitted using the *least mean squares* method. These lines indicate the general linear trend, assuming there is one, followed by the **sample** and **approximation error** for each different network – something which is difficult to see from the scattered points alone,
- the **average**, **lowest** and **highest** values of *mean error*$_i$ and *standard deviation of error*$_i$ over the whole range of input dimensions are shown in various tables throughout the next sections. See, for example, table 6.2 which refers to the **sample error** results of the entities.

6.3.6.2 VARIDIM: network sizes

The following figures depict the relationship between the **number of weights** and **number of inputs** of each of the networks used in this evaluation (see also table 6.1).

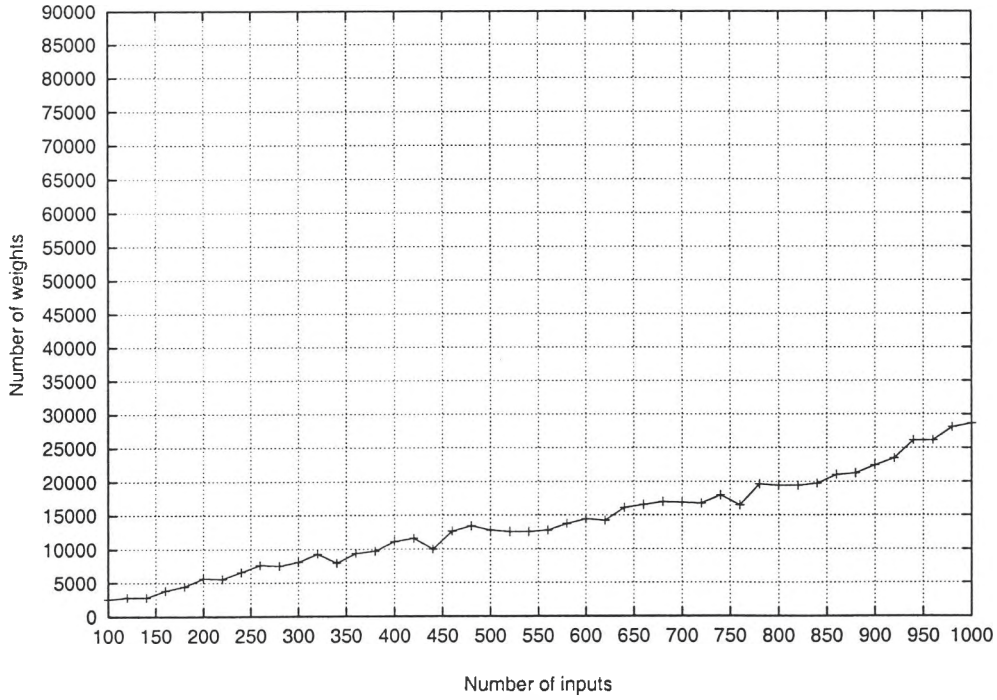

Figure 6.1: $\mathcal{C}_1$, **number of weights** against **number of inputs**

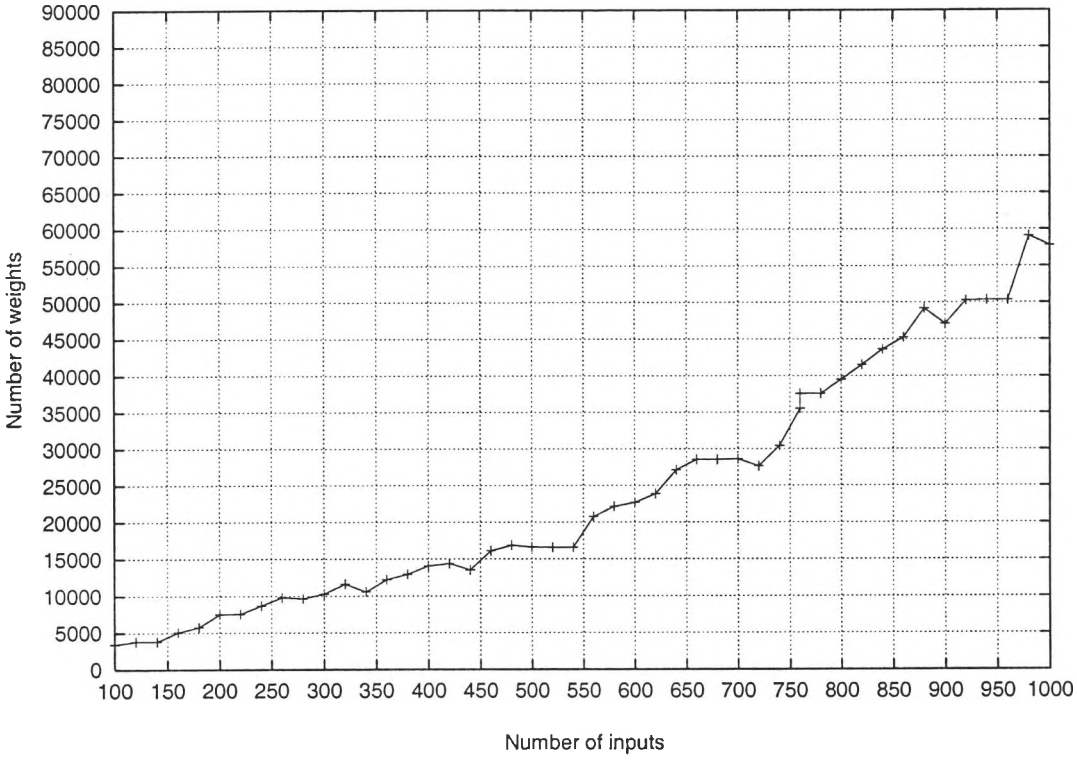

Figure 6.2: $C_{1,big}$, **number of weights against number of inputs**

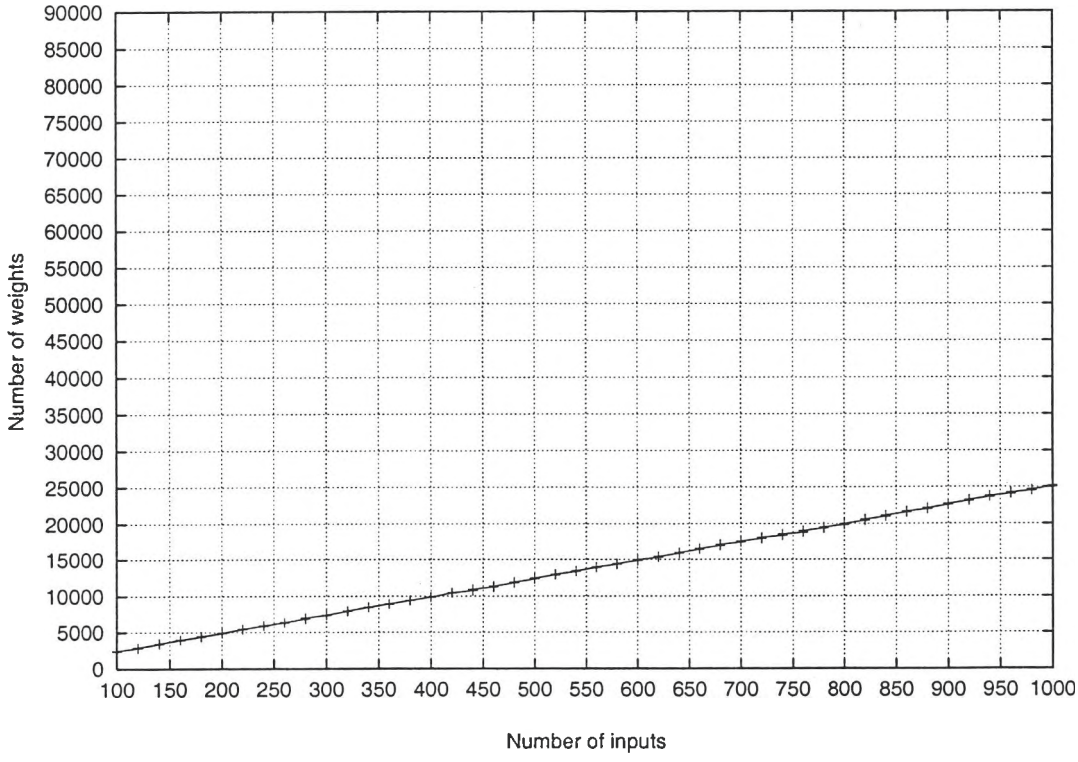

Figure 6.3: C_2, **number of weights against number of inputs**

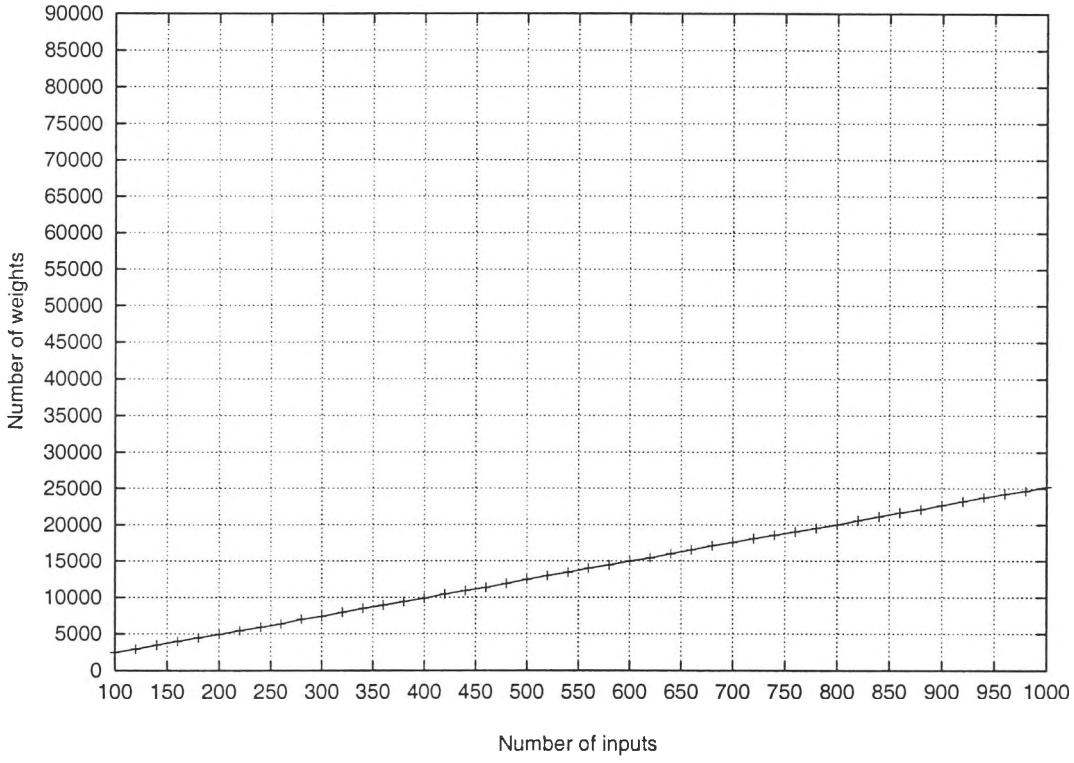

Figure 6.4: $\mathcal{C}_3$, **number of weights** against **number of inputs**

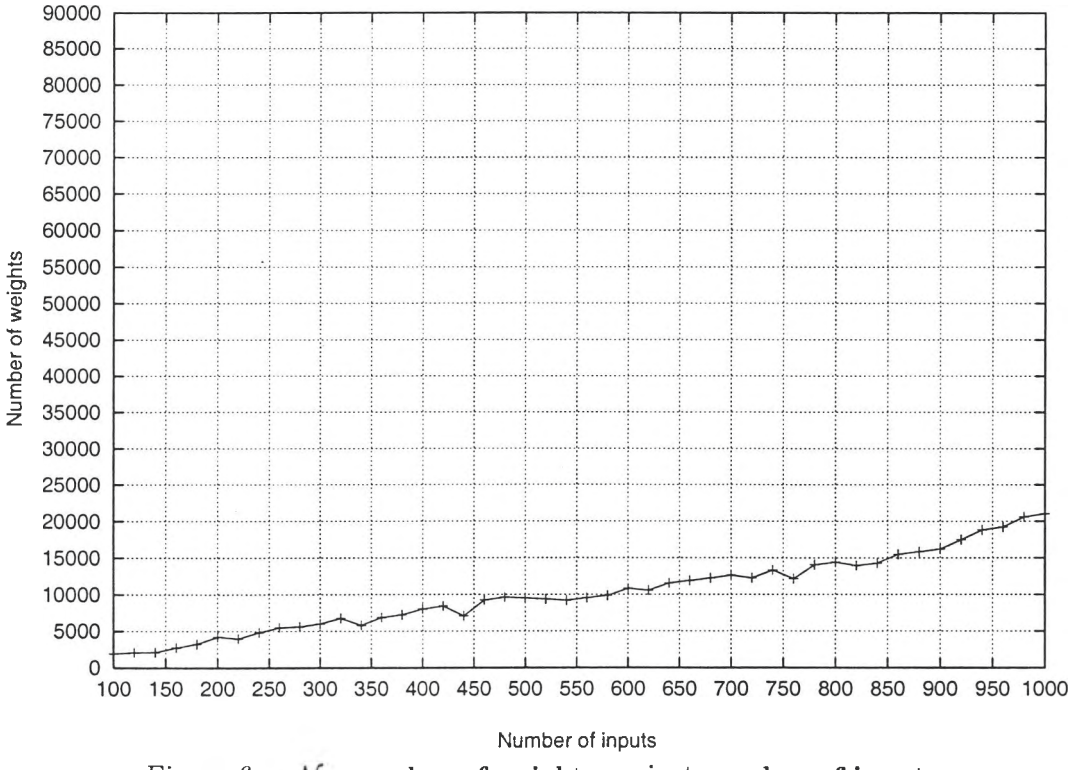

Figure 6.5: $\mathcal{N}_1$, **number of weights** against **number of inputs**

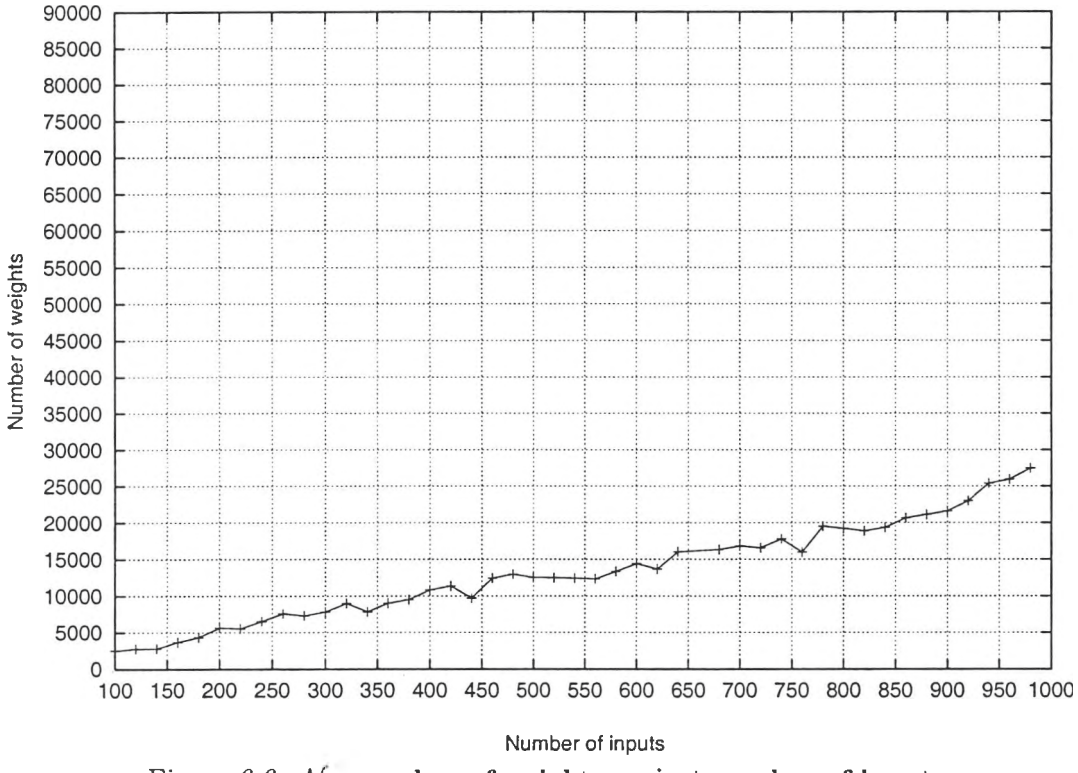

Figure 6.6: $\mathcal{N}_2$, **number of weights** against **number of inputs**

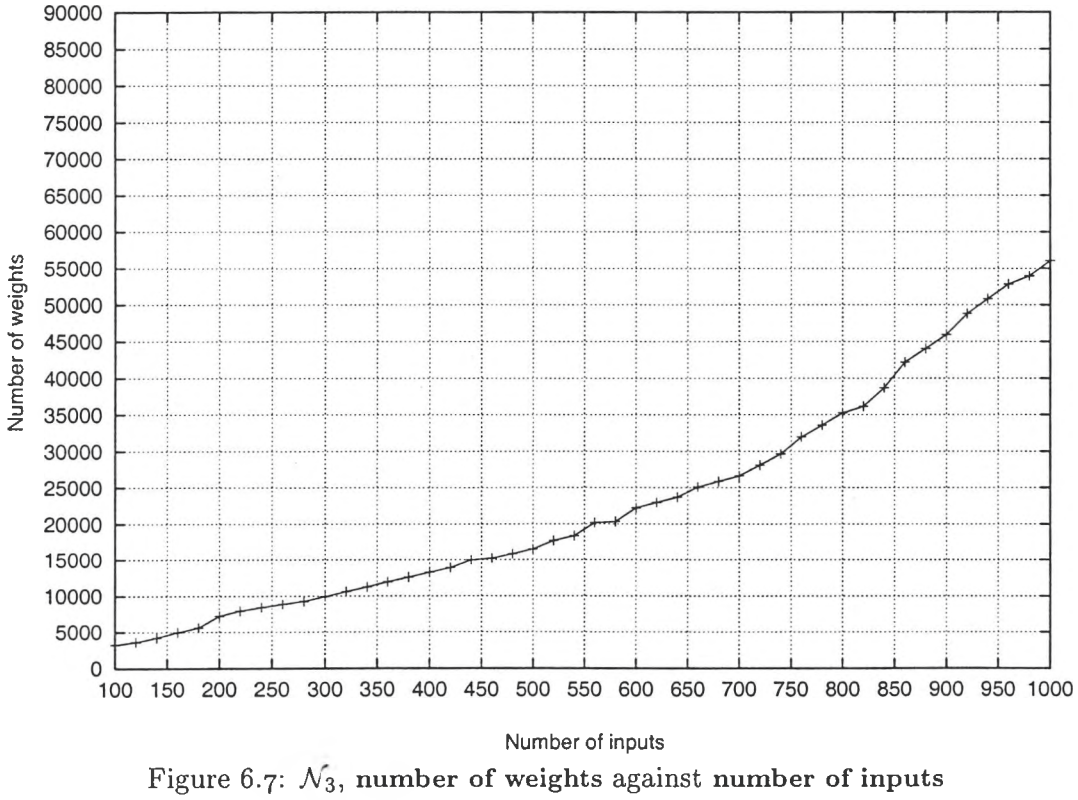

Figure 6.7: $\mathcal{N}_3$, **number of weights** against **number of inputs**

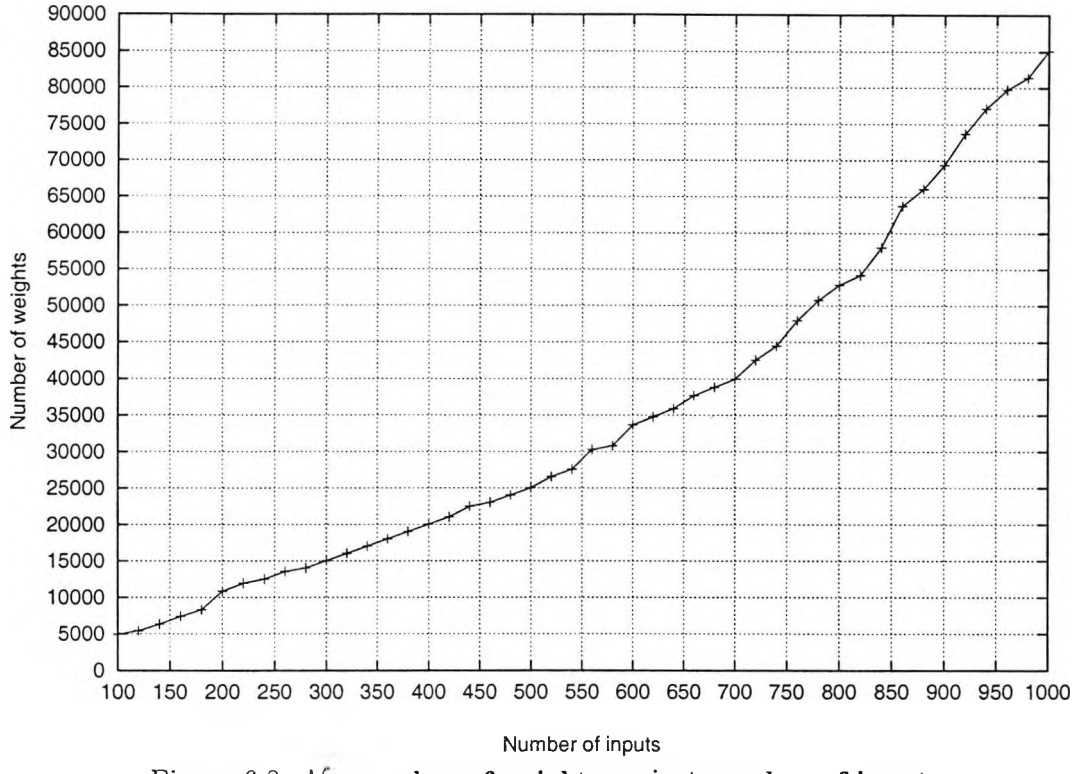

Figure 6.8: $\mathcal{N}_4$, **number of weights** against **number of inputs**

6.3.6.3 VARIDIM: training time results

The fact that the **training time** of a single FFNN (for a fixed number of training iterations) is proportional to the **total number of weights** is, again, confirmed in practice (see also section 4.4.2 on page 38 and figure 4.2 therein). Figures 6.13, 6.14, 6.15 and 6.16 show the linear relationship between **training time** and **number of weights** for the single FFNN $\mathcal{N}_1$, $\mathcal{N}_2$, $\mathcal{N}_3$ and $\mathcal{N}_4$, respectively. Moreover, the same relationship of proportionality applies for the entities too. Figures 6.9, 6.10, 6.11 and 6.12 depict this linear relationship for the four entity networks $\mathcal{C}_1$, $\mathcal{C}_{1,big}$, $\mathcal{C}_2$ and $\mathcal{C}_3$, respectively.

However, the fact that the entities' and single FFNN's training times are directly proportional to the number of their weights does not equate them as far as performance is concerned, simply because performance consists of three quantities rather than just one – time. Thus, one should not isolate **training time** but link it, at least, to **approximation error**. As we will see later, when the **approximation error** results will be discussed (in section 6.3.6.4), an entity yields a much lower **approximation error** than a single FFNN with the same number of weights (compare, for example, $\mathcal{C}_1$ to $\mathcal{N}_2$, tables 6.8 and 6.9 on page 111) Although both require the same training time. *In spite of the fact that all experiments were carried out on computers of equal CPU power, some occasional variations in training time occurred because, most likely, of network traffic.*

The following figures show the **time** taken for the networks to complete 1,000 training iterations as a function of the **total number of their weights.**

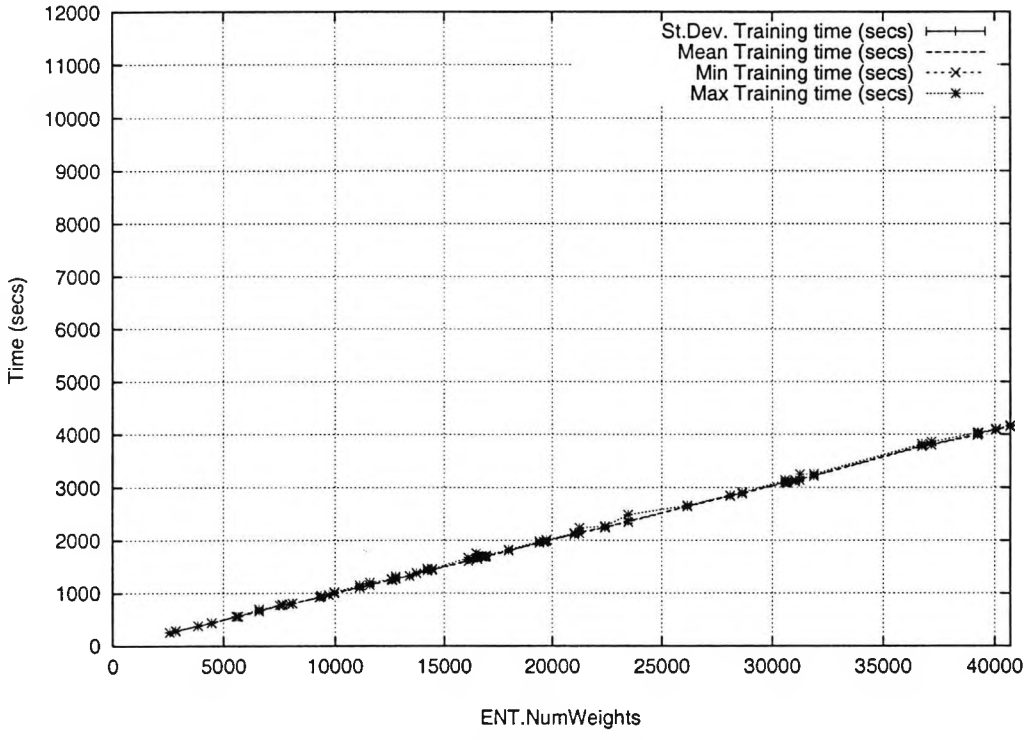

Figure 6.9: $\mathcal{C}_1$, **training time against number of weights**

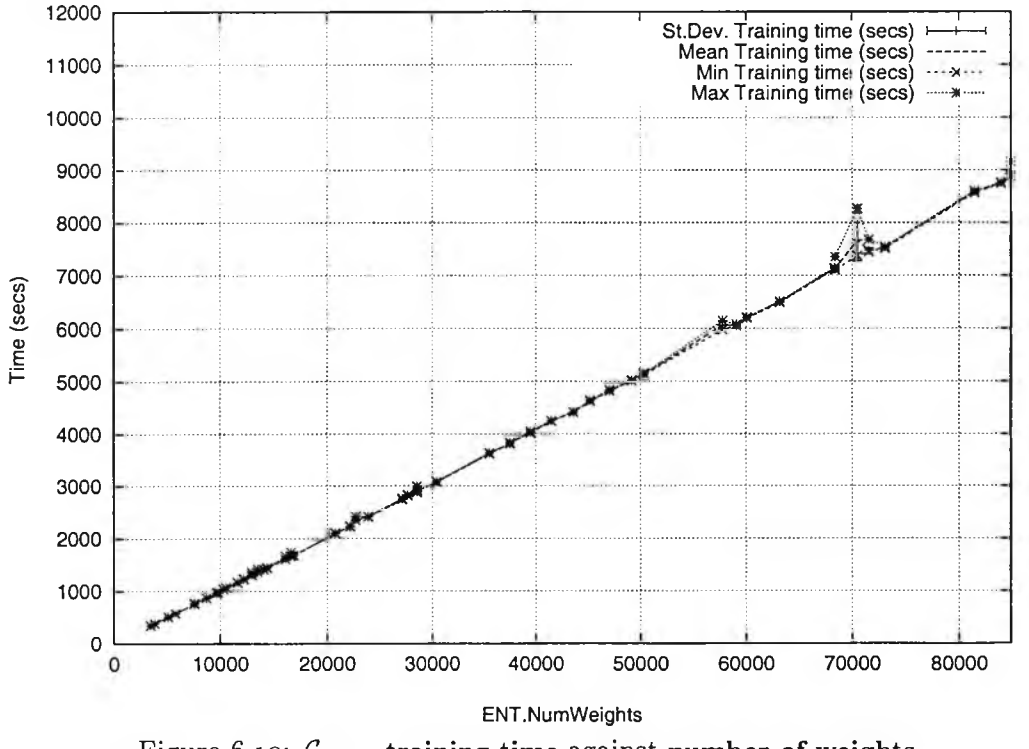

Figure 6.10: $\mathcal{C}_{1,big}$, **training time against number of weights**

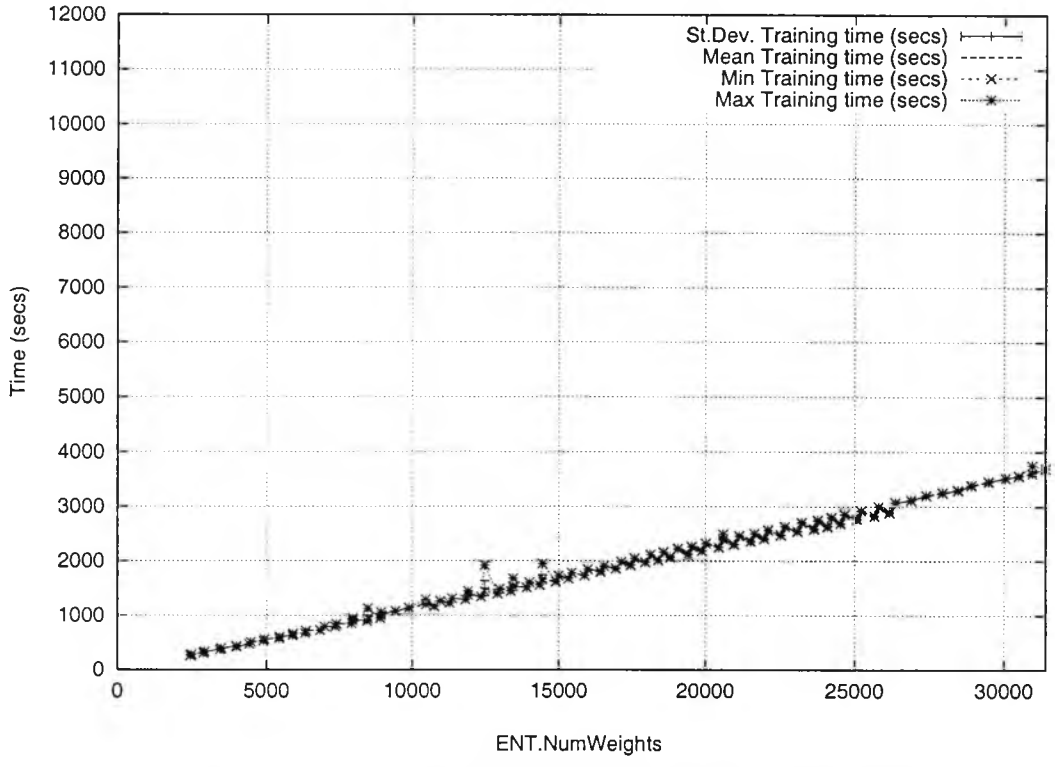

Figure 6.11: C_2, **training time** against **number of weights**

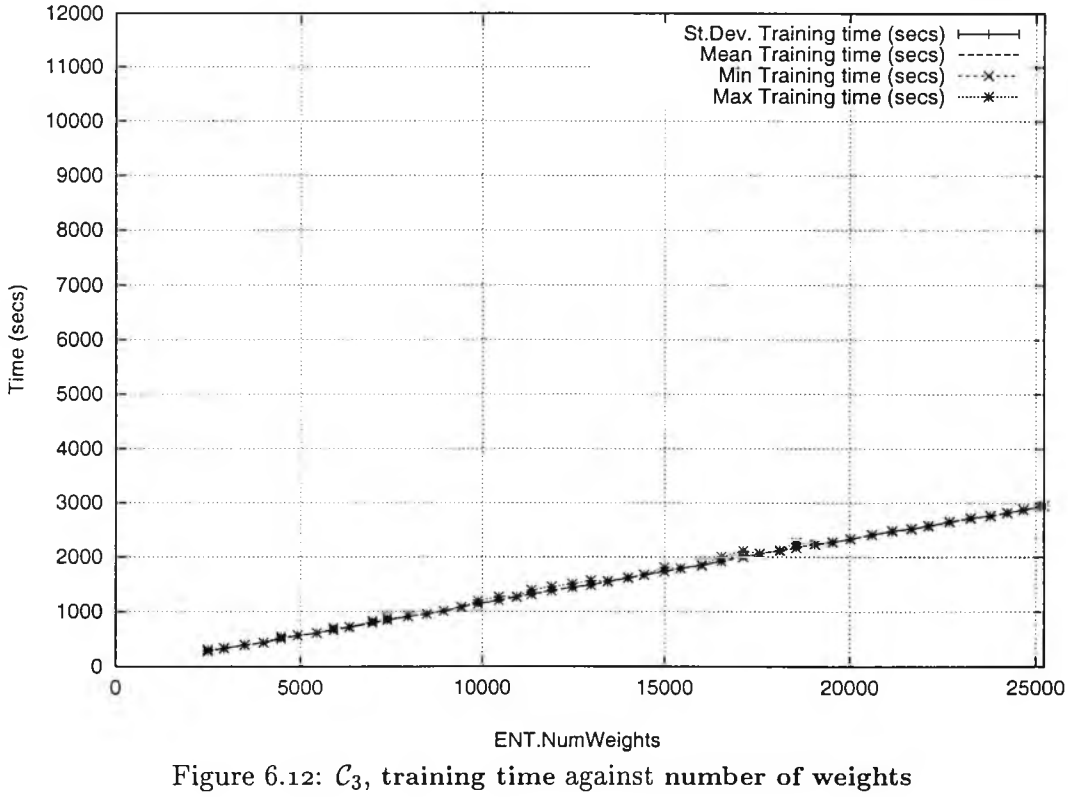

Figure 6.12: C_3, **training time** against **number of weights**

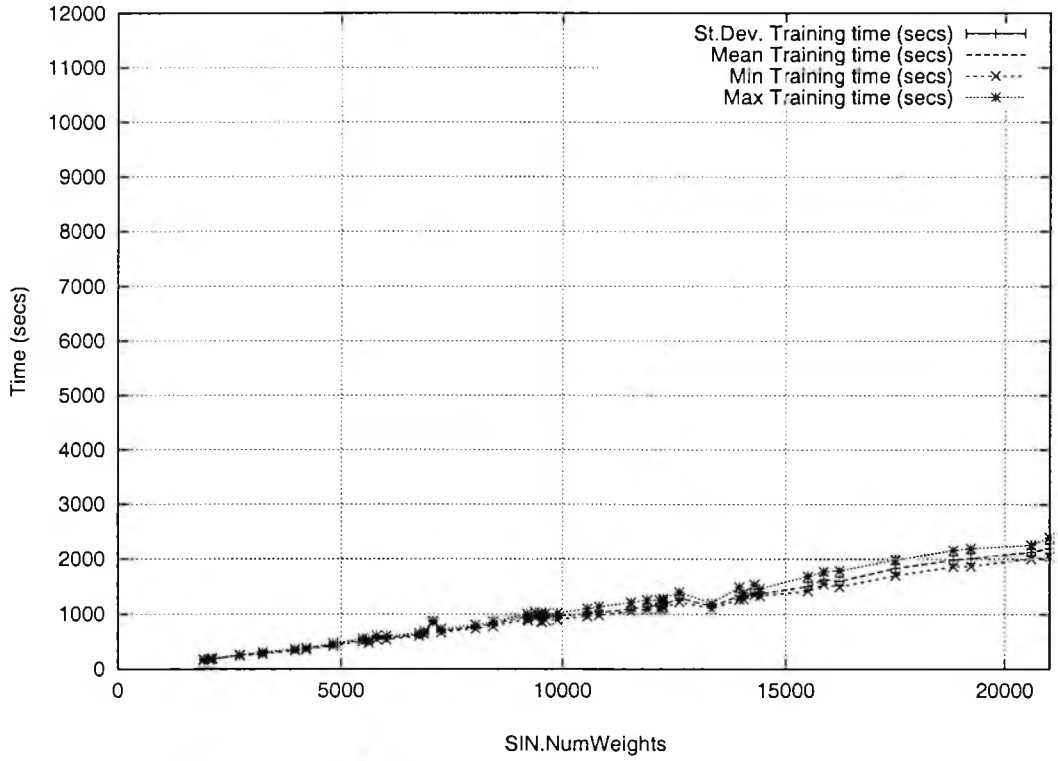

Figure 6.13: $\mathcal{N}_1$, **training time** against **number of weights**

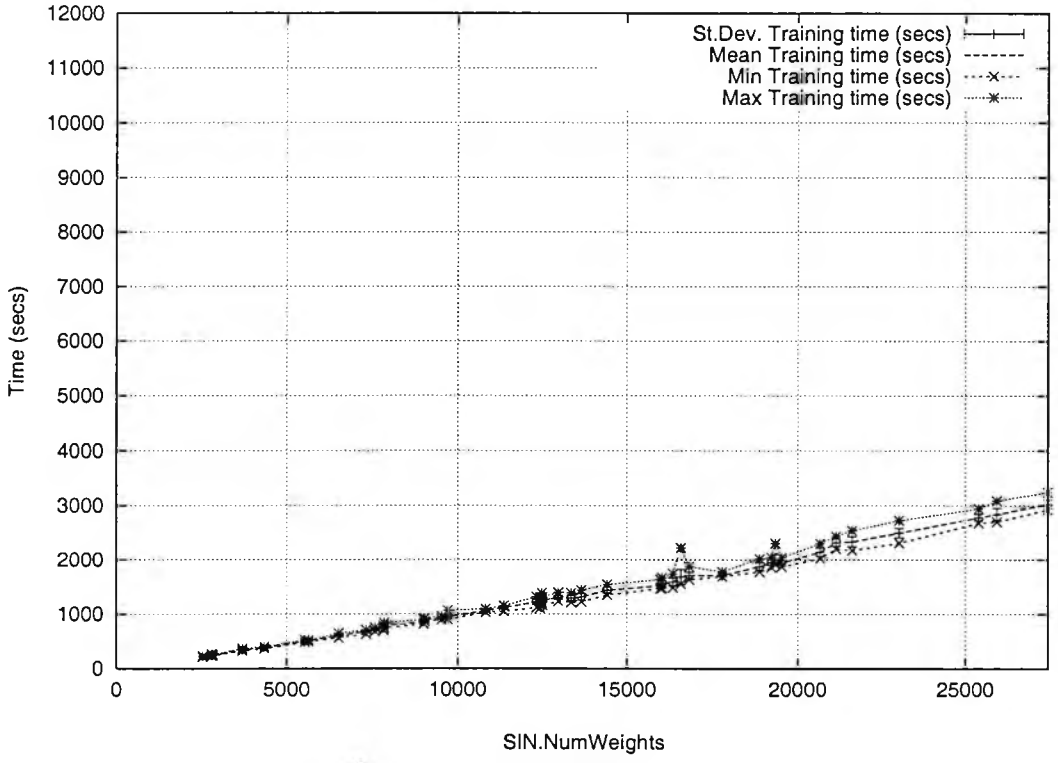

Figure 6.14: $\mathcal{N}_2$, **training time** against **number of weights**

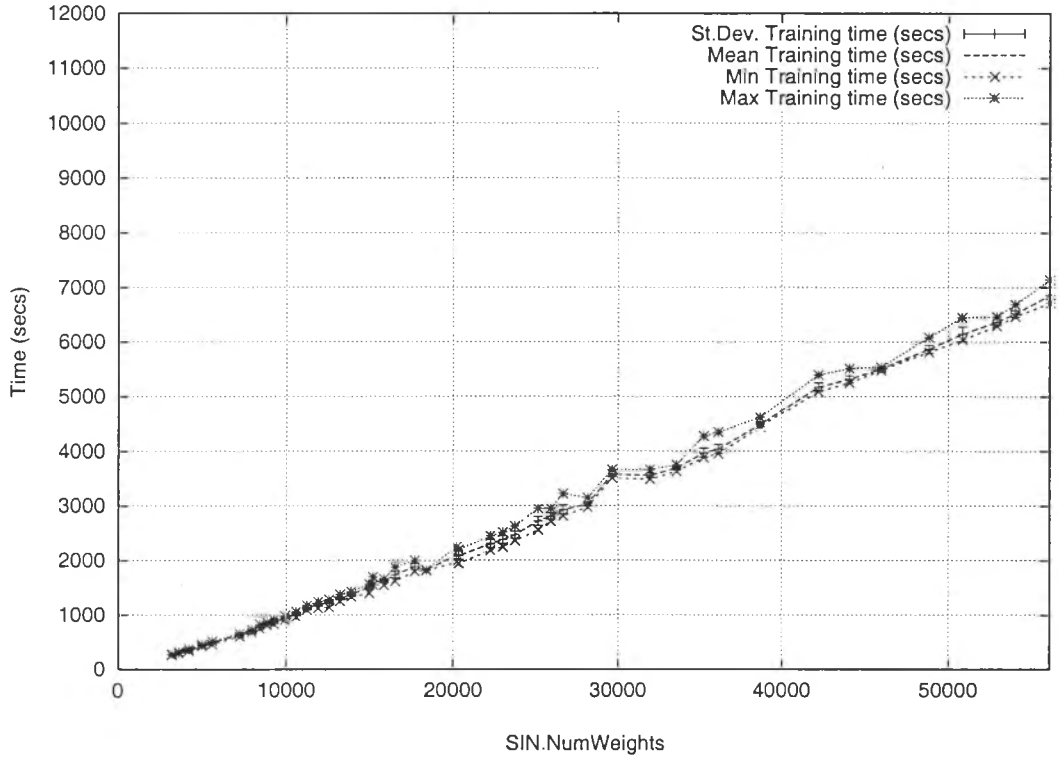

Figure 6.15: $\mathcal{N}_3$, **training time** against **number of weights**

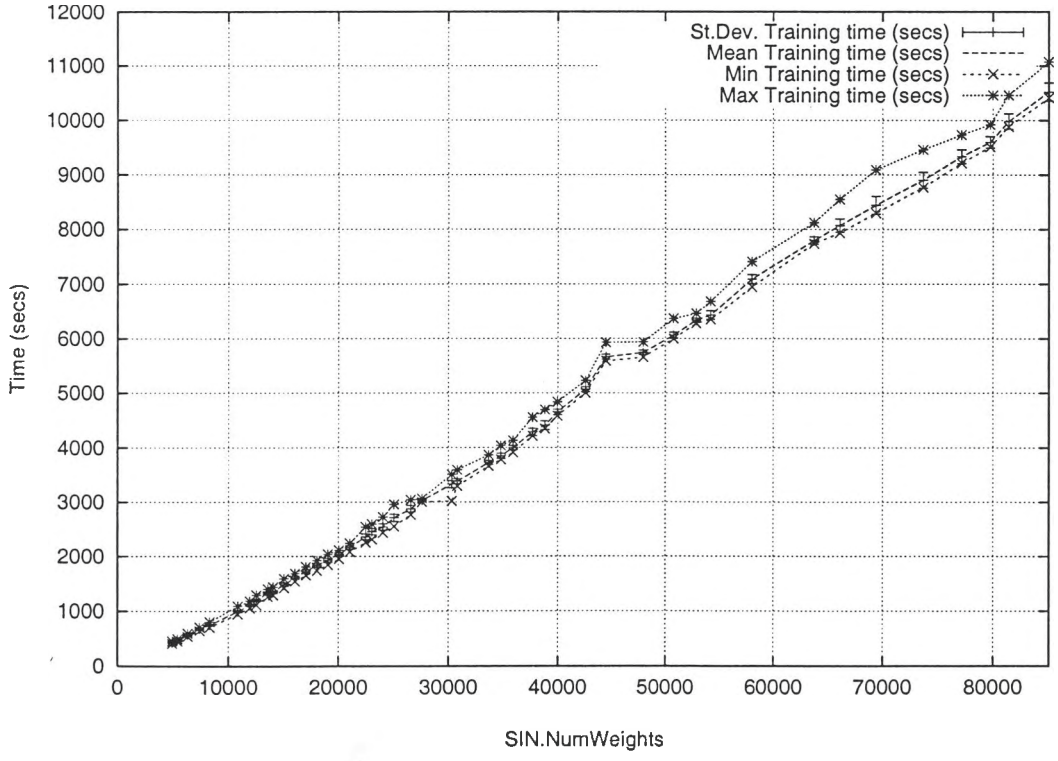

Figure 6.16: $\mathcal{N}_4$, **training time** against **number of weights**

6.3.6.4 VARIDIM: sample error results

Figures 6.17, 6.18, 6.19 and 6.20 show that all entity models reach a very low **sample error**. In particular, $\mathcal{C}_1$ reaches the lowest error, followed by $\mathcal{C}_{1,big}$, $\mathcal{C}_2$ and $\mathcal{C}_3$. The single FFNN also reach a low **sample error** which, most of the time, does not differ significantly from that of the entities (see figures 6.21, 6.22, 6.23 and 6.24). For example, the statistical significance tests comparing the *mean* **sample error** of $\mathcal{C}_1$ and $\mathcal{N}_1$ (*t-test*[9], see tables 6.4 and 6.5) indicate that for both below and over 500 input dimensions, the two errors either do not differ significantly or the **sample error** of $\mathcal{C}_1$ is lower than that of $\mathcal{N}_1$. The cases where the $\mathcal{N}_1$ error is lower than $\mathcal{C}_1$'s are marginal. Of all the entity and single FFNN networks, $\mathcal{C}_2$ and $\mathcal{C}_3$ often[10] exhibit higher **sample errors** than $\mathcal{N}_1$ and $\mathcal{N}_2$, but lower than $\mathcal{N}_3$ and $\mathcal{N}_4$.

	$\mathcal{C}_1$		$\mathcal{C}_{1,big}$		$\mathcal{C}_2$		$\mathcal{C}_3$	
	$\mathcal{C}_1$ entity		$\mathcal{C}_1$ (66% more weights)		$\mathcal{C}_2$ entity		$\mathcal{C}_3$ entity	
$\times 10^{-03}$	MEAN	STD.DEV	MEAN	STD.DEV	MEAN	STD.DEV	MEAN	STD.DEV
lowest	0.176	0.053	0.161	0.028	0.298	0.174	0.636	0.052
average	0.501	0.398	0.557	0.455	1.577	1.486	2.011	1.686
highest	2.292	2.035	3.035	2.710	4.791	6.764	10.747	9.554

Table 6.2: VARIDIM, **sample error** statistics for the entities

	$\mathcal{N}_1$		$\mathcal{N}_2$		$\mathcal{N}_3$		$\mathcal{N}_4$	
	35% less weights		standard		55% more weights		135% more weights	
$\times 10^{-03}$	MEAN	STD.DEV	MEAN	STD.DEV	MEAN	STD.DEV	MEAN	STD.DEV
lowest	0.229	0.307	0.558	0.760	1.817	1.171	1.849	2.148
average	1.488	3.067	1.677	2.502	8.539	11.517	22.986	28.893
highest	4.503	8.523	3.508	9.004	94.157	121.035	84.978	163.099

Table 6.3: VARIDIM, **sample error** statistics for single FFNN

The value of the **sample error**, however, is not necessarily associated with the generalisation ability of a learning machine[11]. What is of interest to us is whether the training process is *consistent*. The term *consistency* refers to the variation of the observed **sample error** when training a network for several times and for a given number of inputs. A measure of this variation is the *standard deviation* of the **sample error** of

[9] For an explanation on how to read these tables please refer to section 6.3.

[10] This becomes rarer as the number of input dimensions increases.

[11] For example, a low sample error for a single FFNN might be an indication of *over-fitting* resulting to very bad generalisation.

the same network over a number of training attempts, for a given **number of inputs**. A consistent training process is indicated by a relatively small *standard deviation* value and is the result – among other factors – of a smooth error surface and a small number of local minima. On the other hand, a large *standard deviation* could be the result of an inconsistent training process which would – most likely – mean that the error surface is so complex that the probability that two runs converge to the same local minimum is small.

%, %, %	$\mathcal{N}_1$	$\mathcal{N}_2$	$\mathcal{N}_3$	$\mathcal{N}_4$
$\mathcal{C}_1$	33 , 5 , 62	14 , 0 , 86	14 , 0 , 86	14 , 0 , 86
$\mathcal{C}_{1,big}$	43 , 14 , 43	14 , 10 , 76	10 , 0 , 90	10 , 4 , 86
$\mathcal{C}_2$	52 , 48 , 0	62 , 19 , 19	57 , 5 , 38	33 , 5 , 62
$\mathcal{C}_3$	29 , 71 , 0	76 , 19 , 5	71 , 10 , 19	43 , 9 , 48

Table 6.4: VARIDIM, statistical significance (*t-test*) of the **sample error** results for 100 to 500 input dimensions

%, %, %	$\mathcal{N}_1$	$\mathcal{N}_2$	$\mathcal{N}_3$	$\mathcal{N}_4$
$\mathcal{C}_1$	56 , 4 , 40	52 , 0 , 48	4 , 0 , 96	0 , 0 , 100
$\mathcal{C}_{1,big}$	56 , 4 , 40	48 , 0 , 52	4 , 0 , 96	0 , 0 , 100
$\mathcal{C}_2$	72 , 12 , 16	91 , 0 , 9	12 , 0 , 88	0 , 0 , 100
$\mathcal{C}_3$	84 , 12 , 4	83 , 13 , 4	28 , 0 , 72	0 , 0 , 100

Table 6.5: VARIDIM, statistical significance (*t-test*) of the **sample error** results for more than 500 input dimensions

The training process of the entities, for a given number of inputs, is *fairly consistent* as there is not so much variation. In particular, the *standard deviation* of sample error of the $\mathcal{C}_1$ network is around 0.4×10^{-03}, on average, and not exceeding 2×10^{-03}. $\mathcal{C}_{1,big}$'s error is around 0.5×10^{-03}, on average, and not exceeding 2.7×10^{-03}. For the $\mathcal{C}_2$ network, it is around 1.5×10^{-03}, on average, and not exceeding 7×10^{-03}, whereas for $\mathcal{C}_3$, the *standard deviation* is around 1.7×10^{-03}, on average, and not exceeding 10×10^{-03} (see table 6.2). On the other hand, the *standard deviation* of the **sample error** for $\mathcal{N}_2$ is 2.5×10^{-03} and not exceeding 9×10^{-03} (see table 6.3). This is approximately, *five times higher than that of* $\mathcal{C}_1$ *and* $\mathcal{C}_{1,big}$. The larger single FFNN, $\mathcal{N}_3$ and $\mathcal{N}_4$, are even more inconsistent. The smallest single FFNN, $\mathcal{N}_1$ is not as inconsistent as $\mathcal{N}_3$ or $\mathcal{N}_4$, but *just a little bit more* than $\mathcal{N}_2$.

The above conclusions are also supported by the scatter plots of the **sample error** and the least mean squares lines fitted on them, as shown in figures 6.25, 6.26, 6.27, 6.28

corresponding to $\mathcal{C}_1$, $\mathcal{C}_{1,big}$, $\mathcal{C}_2$, $\mathcal{C}_3$, respectively. In particular, it can be seen that the **sample error** of $\mathcal{C}_1$ and $\mathcal{C}_{1,big}$ deviates very little from the mean with only a few outliers below 300 inputs; as the number of input dimensions increases, the outliers disappear. $\mathcal{C}_2$ shows the same behaviour but with *higher* **sample error** levels. Its outliers are also more. Of the entities, $\mathcal{C}_3$ has the *widest error variation with a significant number of outliers, especially when the number of input dimensions is less than* 300.

%, %, %	$\mathcal{N}_1$	$\mathcal{N}_2$	$\mathcal{N}_3$	$\mathcal{N}_4$
$\mathcal{C}_1$	10, 4, 86	5, 0, 95	10, 0, 90	0, 0, 100
$\mathcal{C}_{1,big}$	10, 9, 81	14, 5, 81	0, 5, 95	5, 0, 95
$\mathcal{C}_2$	29, 33, 38	33, 19, 48	14, 13, 71	19, 10, 71
$\mathcal{C}_3$	43, 52, 5	43, 24, 33	24, 9, 67	24, 5, 71

Table 6.6: VARIDIM, statistical significance (*F-test*) of the **sample error** results for 100 to 500 input dimensions

%, %, %	$\mathcal{N}_1$	$\mathcal{N}_2$	$\mathcal{N}_3$	$\mathcal{N}_4$
$\mathcal{C}_1$	0, 0, 100	4, 0, 96	0, 0, 100	0, 0, 100
$\mathcal{C}_{1,big}$	4, 0, 96	0, 0, 100	0, 0, 100	0, 0, 100
$\mathcal{C}_2$	12, 8, 80	26, 0, 74	4, 0, 96	0, 0, 100
$\mathcal{C}_3$	12, 8, 80	30, 9, 61	16, 0, 84	0, 0, 100

Table 6.7: VARIDIM, statistical significance (*F-test*) of the **sample error** results for more than 500 input dimensions

For a small number of input dimensions, the single FFNN's, $\mathcal{N}_1$ and $\mathcal{N}_2$, error variation is approximately the same as, if not less than, that of $\mathcal{C}_2$ and $\mathcal{C}_3$ but much higher than $\mathcal{C}_1$ and $\mathcal{C}_{1,big}$ (see figures 6.29, 6.30, 6.31, 6.32 corresponding to $\mathcal{N}_1$, $\mathcal{N}_2$, $\mathcal{N}_3$ and $\mathcal{N}_4$, respectively). However, as the number of input dimensions increases, the outliers in the scatter plots of the single FFNN become more, whereas the outliers in the entities' plots either disappear ($\mathcal{C}_1$ and $\mathcal{C}_{1,big}$) or are significantly reduced ($\mathcal{C}_2$ and $\mathcal{C}_3$). $\mathcal{N}_4$ shows the same behaviour as $\mathcal{N}_1$ and $\mathcal{N}_2$, with the difference that after 600 inputs, there is no trend associated with the sample error but, instead, the points are scattered all over! As far as the statistical significance test comparing the variances of the various networks (*F-test*, see tables 6.6 and 6.7) is concerned, it is clear that the variance of the entities' **sample error** is consistently lower than that of the single FFNN. In particular, for less than 500 input dimensions, only $\mathcal{C}_2$ and $\mathcal{C}_3$'s variances do not differ significantly from those of $\mathcal{N}_1$ and $\mathcal{N}_2$. The rest of the entities have lower variances than any single FFNN. When the number of input dimensions exceeds 500, the hypothesis that an entity's

variance is significantly lower than that of a single FFNN is true for **all** entity / single FFNN pairs. The above observations support the view that *the training process*[12] *of the entities is more consistent than that of the single* FFNN and that *as the number of input dimensions exceeds the critical value of* 500, *the training of single* FFNN *is very inconsistent.*

The following figures show the *minimum*, *maximum*, *mean* and *standard deviation* of the **sample error** reached by each of the evaluated networks after a 1,000 iterations as a function of the number of their **inputs**.

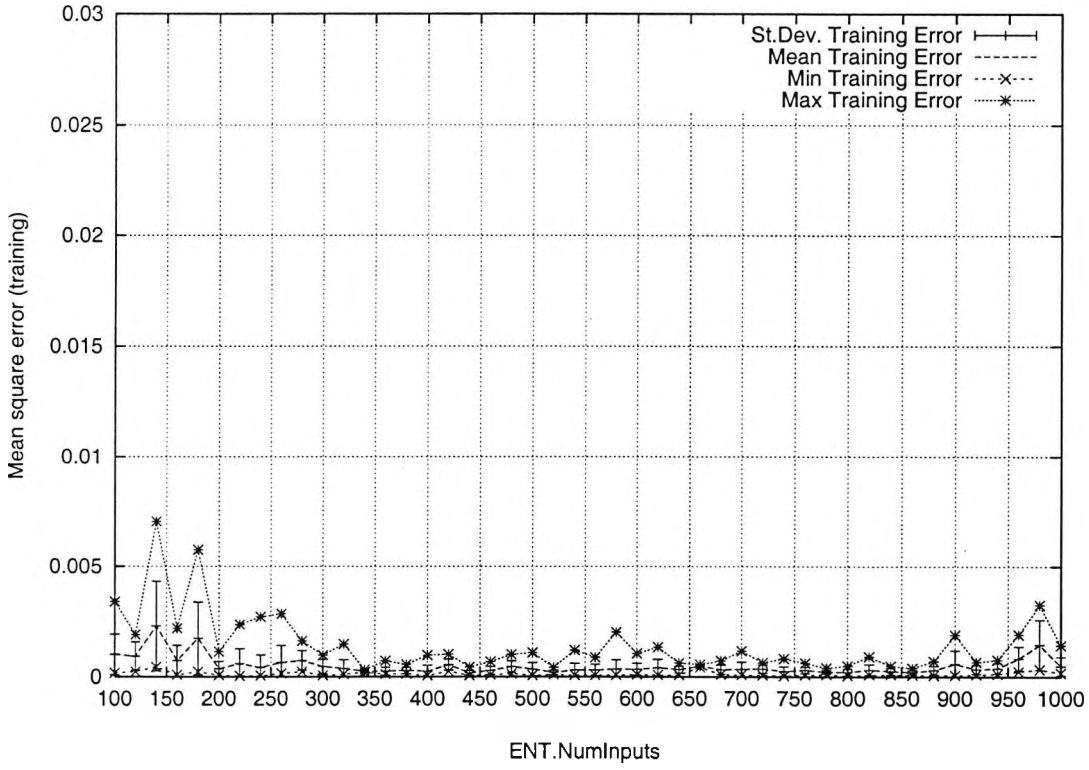

Figure 6.17: C_1, **sample error** against **number of inputs**

[12] At least for this particular benchmark.

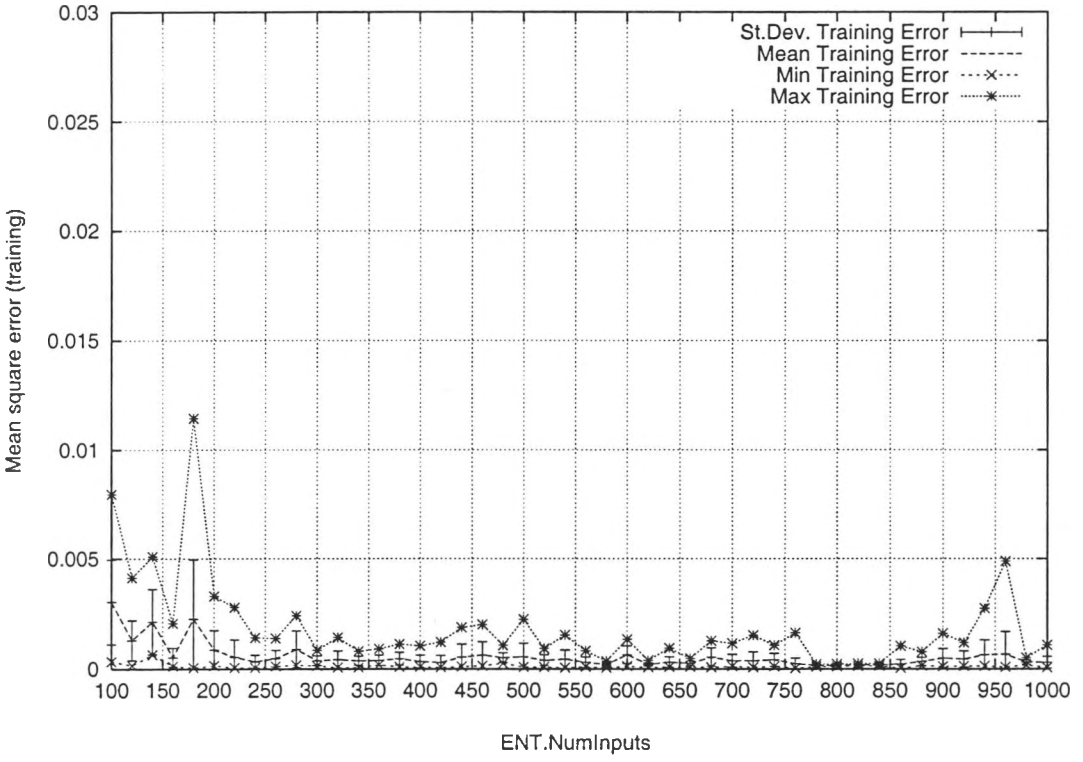

Figure 6.18: $C_{1,big}$, **sample error against number of inputs**

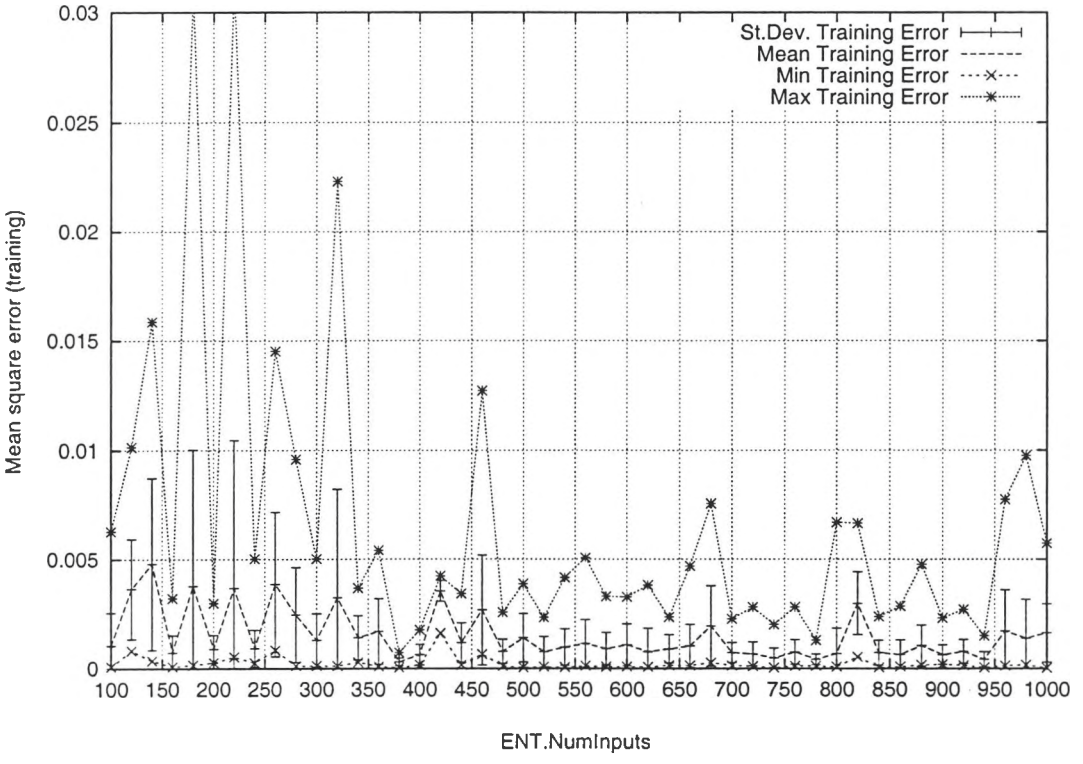

Figure 6.19: C_2, **sample error against number of inputs**

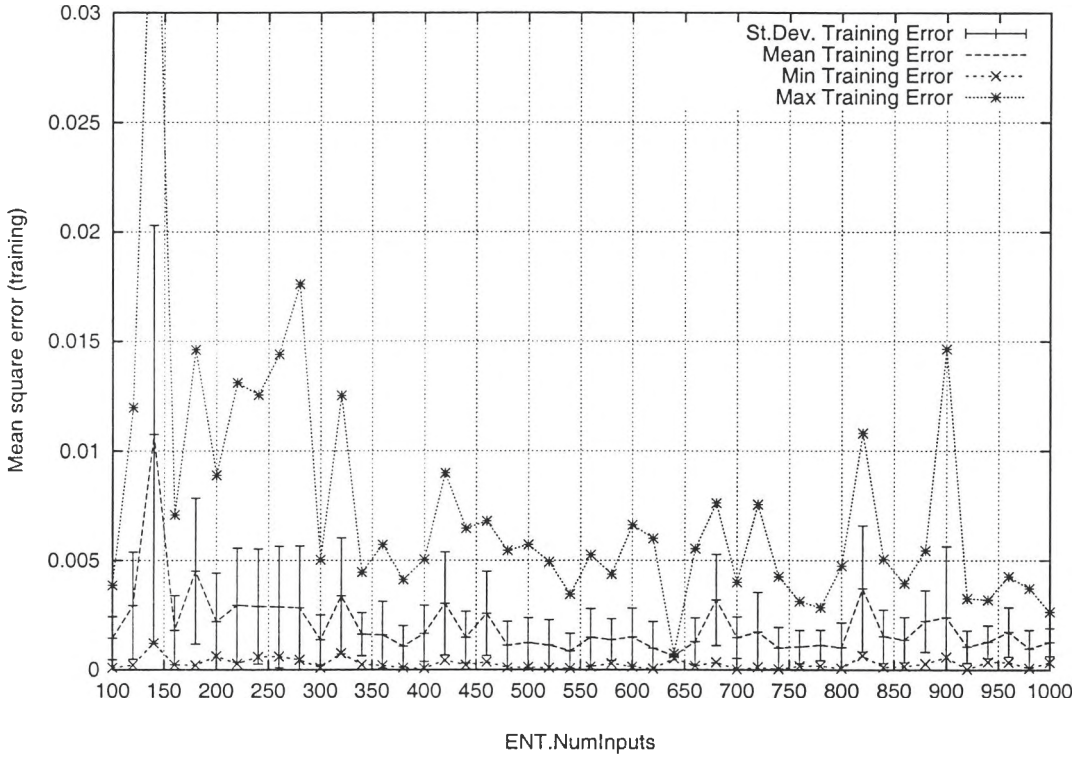

Figure 6.20: $\mathcal{C}_3$, **sample error** against **number of inputs**

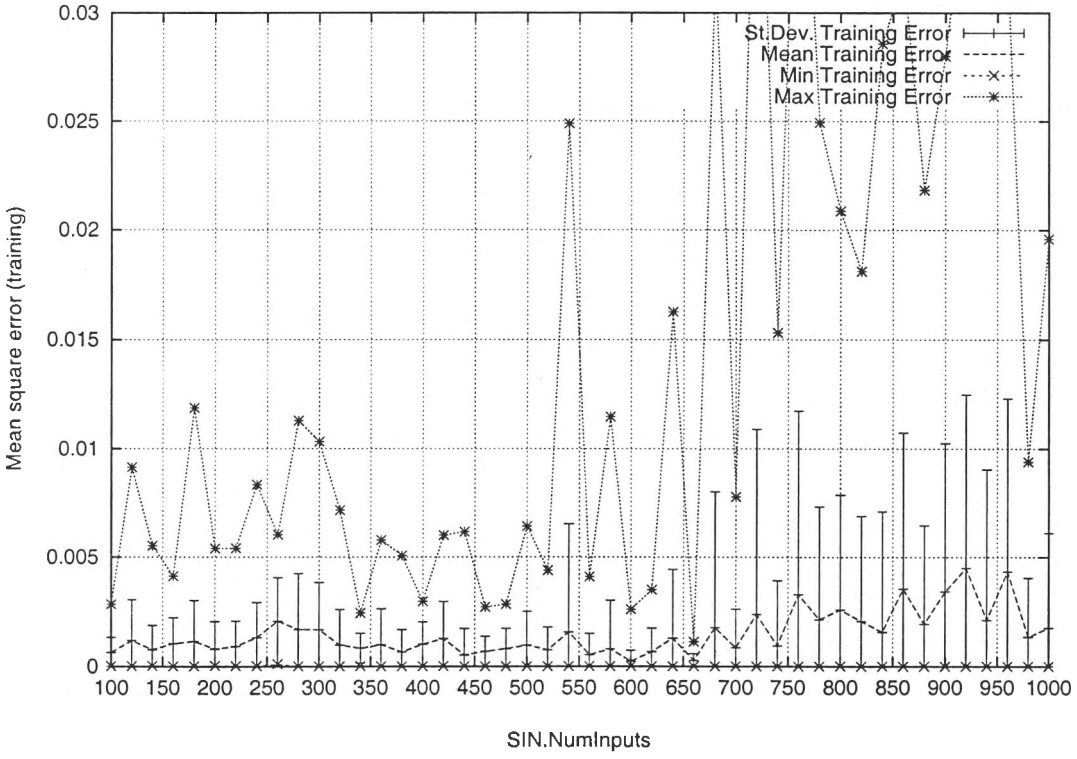

Figure 6.21: $\mathcal{N}_1$, **sample error** against **number of inputs**

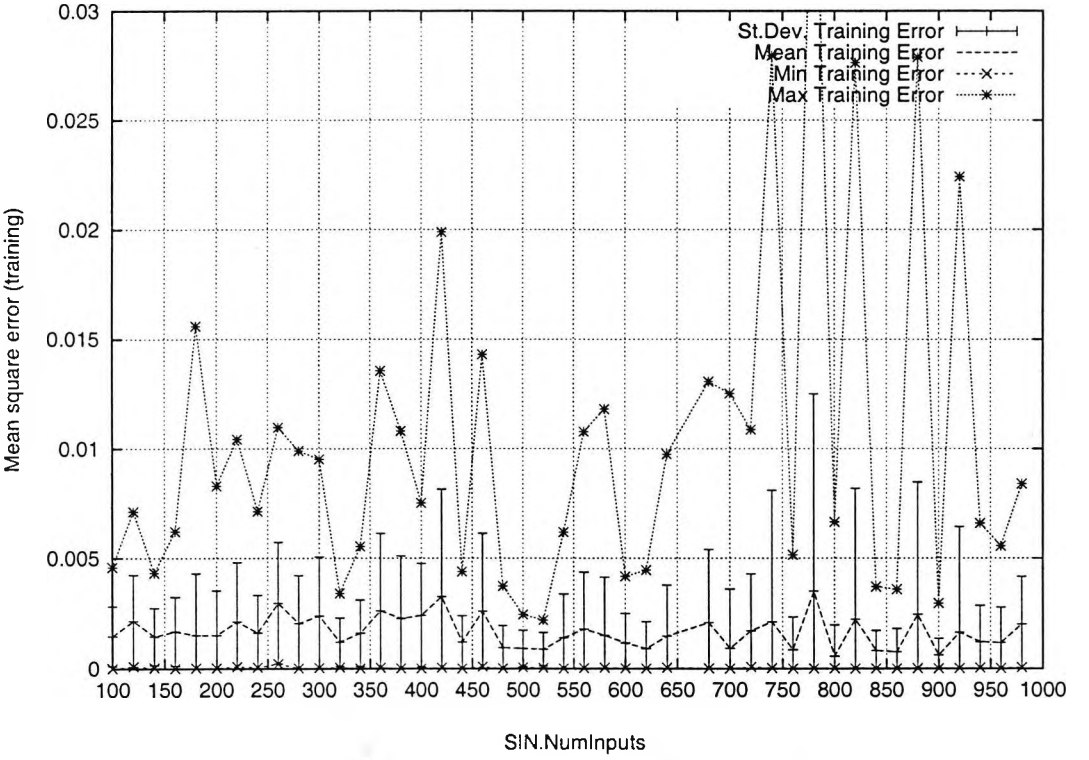

Figure 6.22: $\mathcal{N}_2$, **sample error against number of inputs**

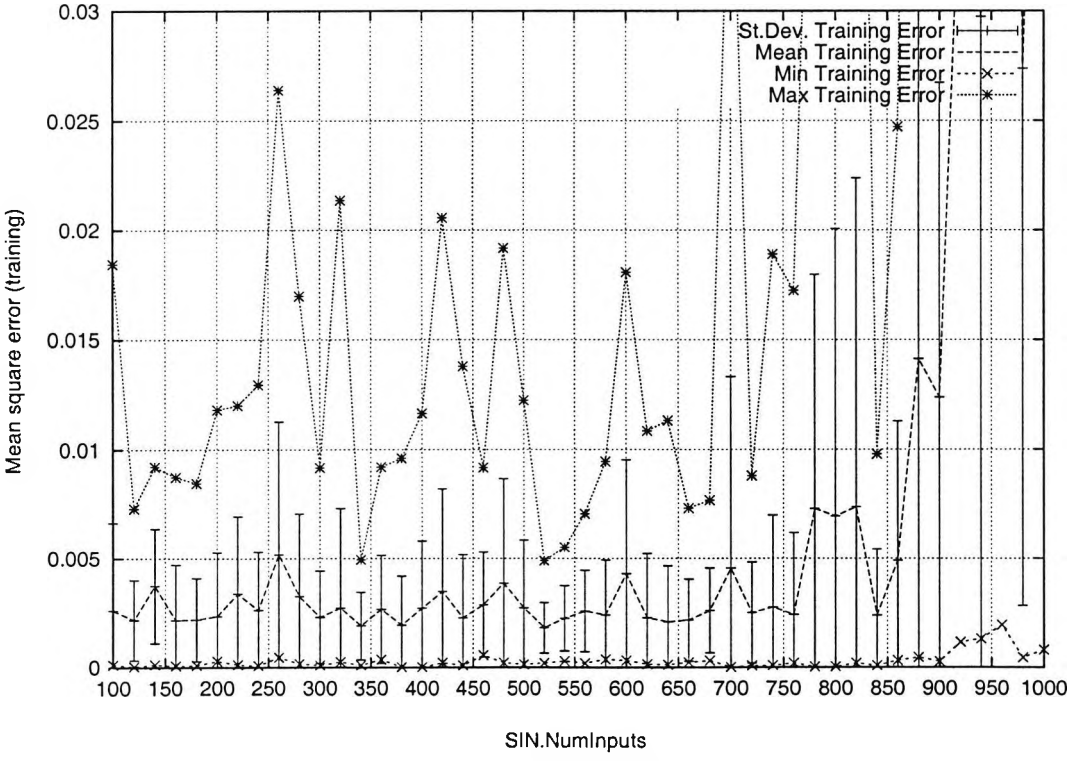

Figure 6.23: $\mathcal{N}_3$, **sample error against number of inputs**

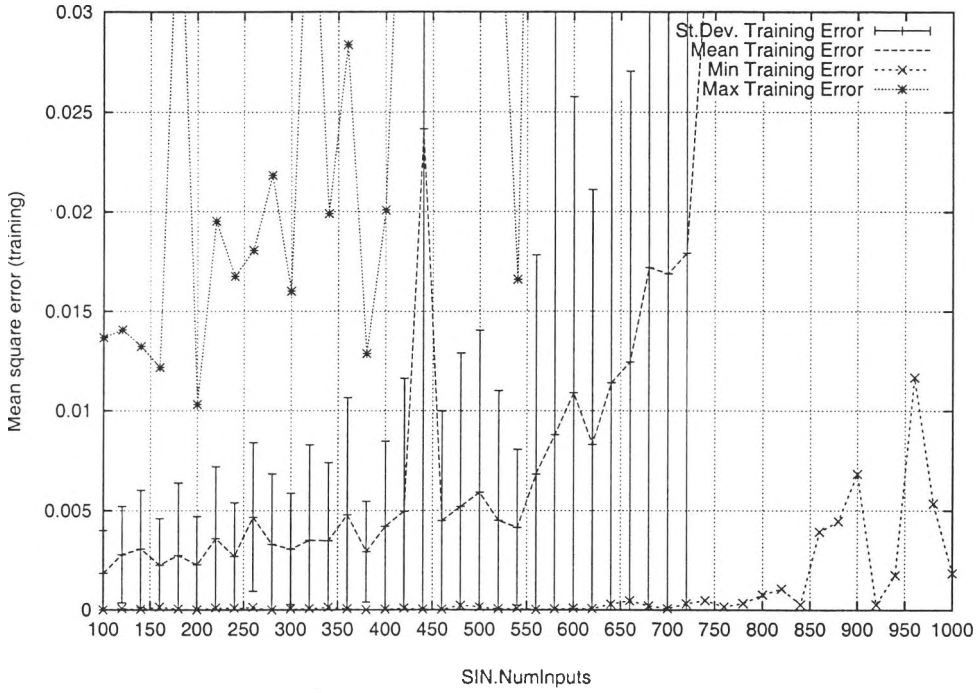

Figure 6.24: $\mathcal{N}_4$, **sample error** against **number of inputs**

The following figures contain scatter plots of the **sample error**, and least mean squares lines fitted on them, for all evaluated networks, as the number of their **inputs** increases.

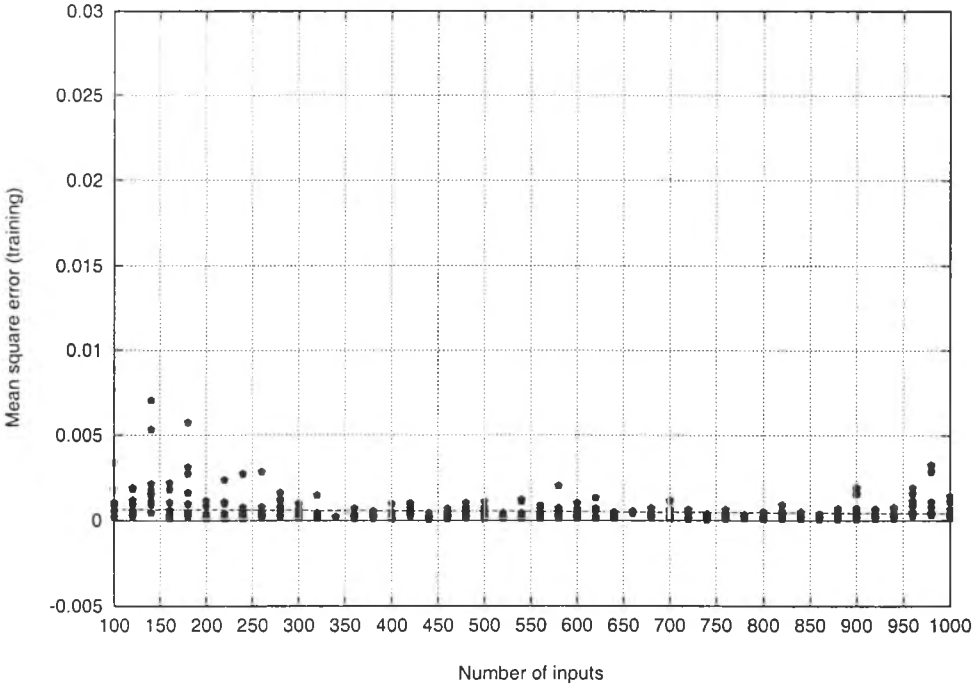

Figure 6.25: $\mathcal{C}_1$, **sample error** against **number of inputs**

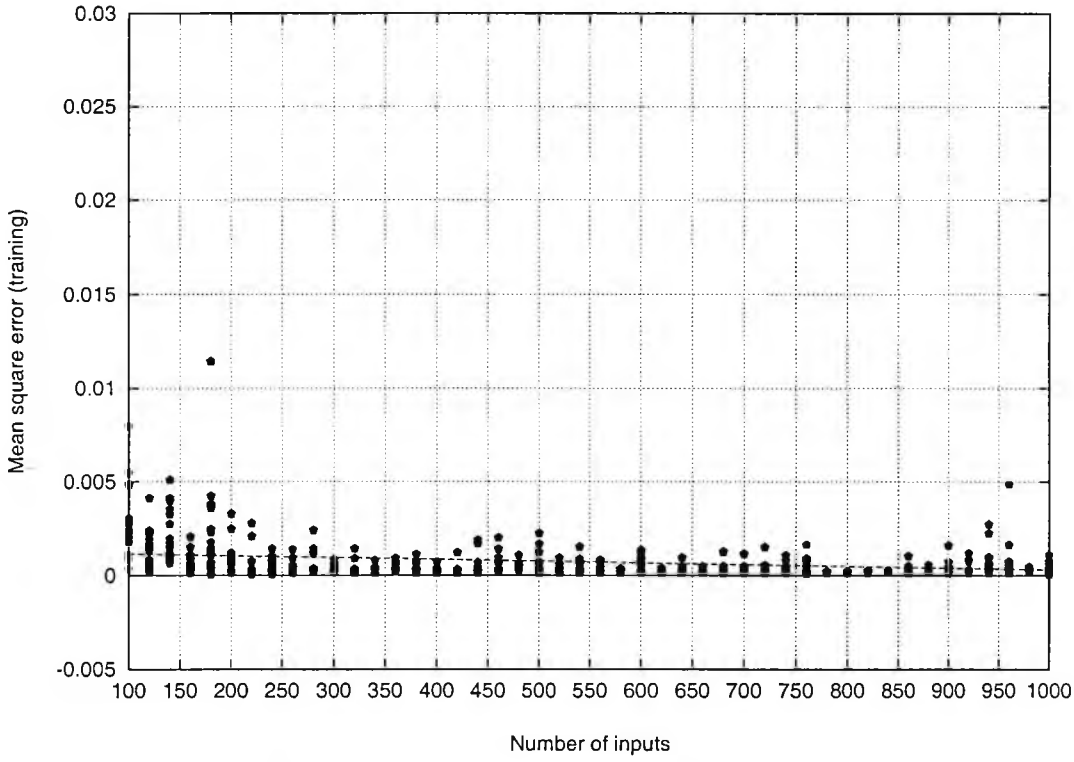

Figure 6.26: $C_{1,big}$, **sample error against number of inputs**

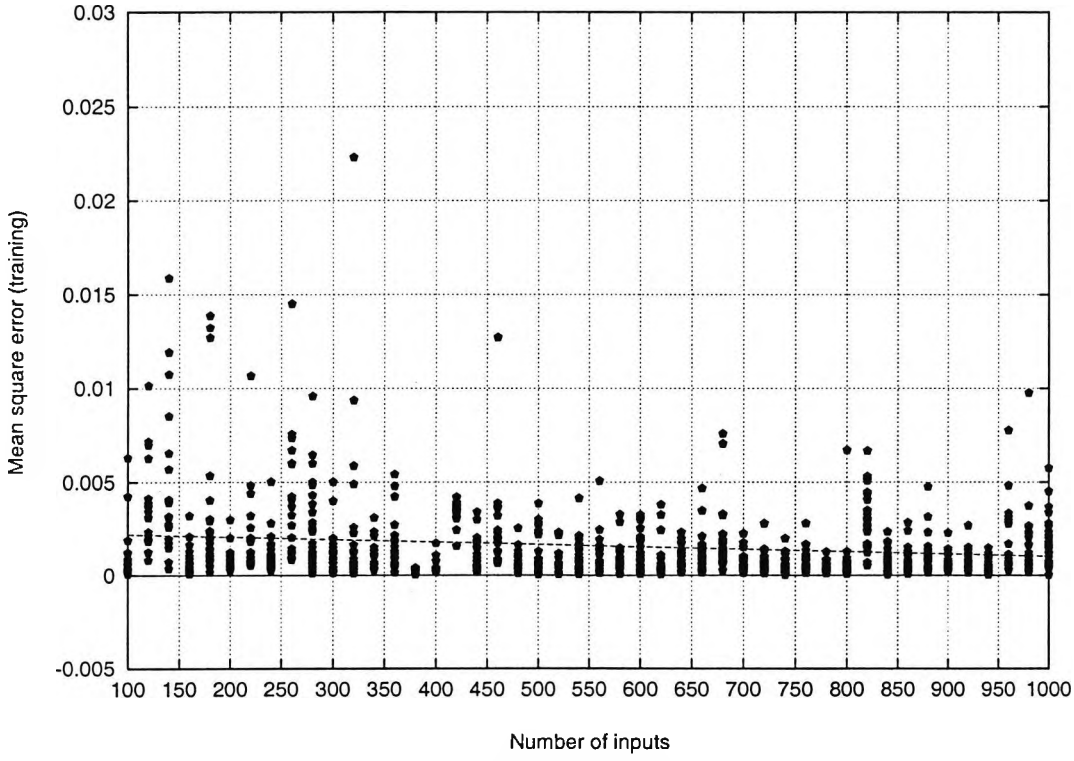

Figure 6.27: C_2, **sample error against number of inputs**

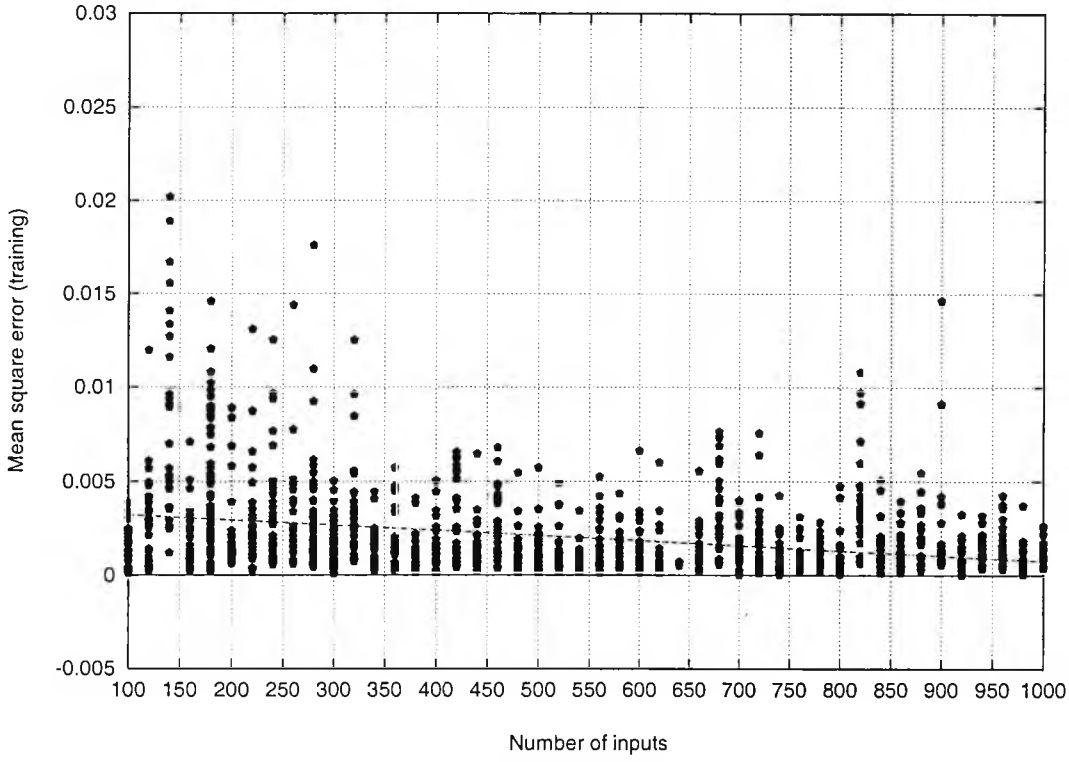

Figure 6.28: $\mathcal{C}_3$, **sample error** against **number of inputs**

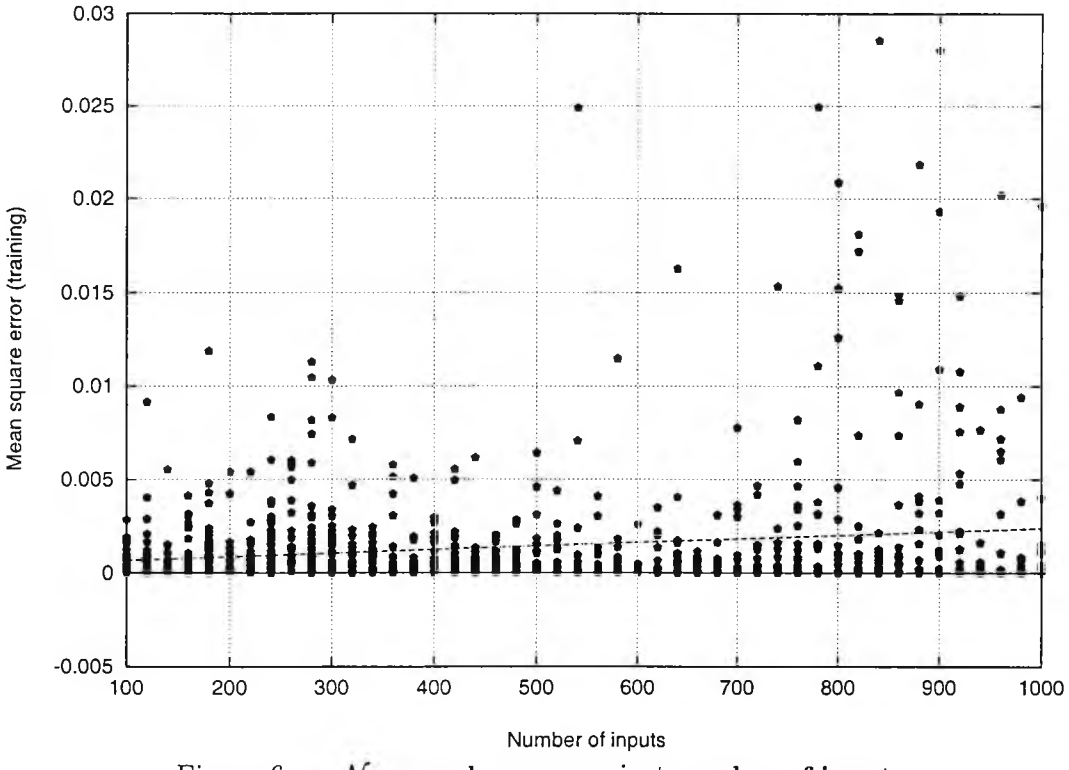

Figure 6.29: $\mathcal{N}_1$, **sample error** against **number of inputs**

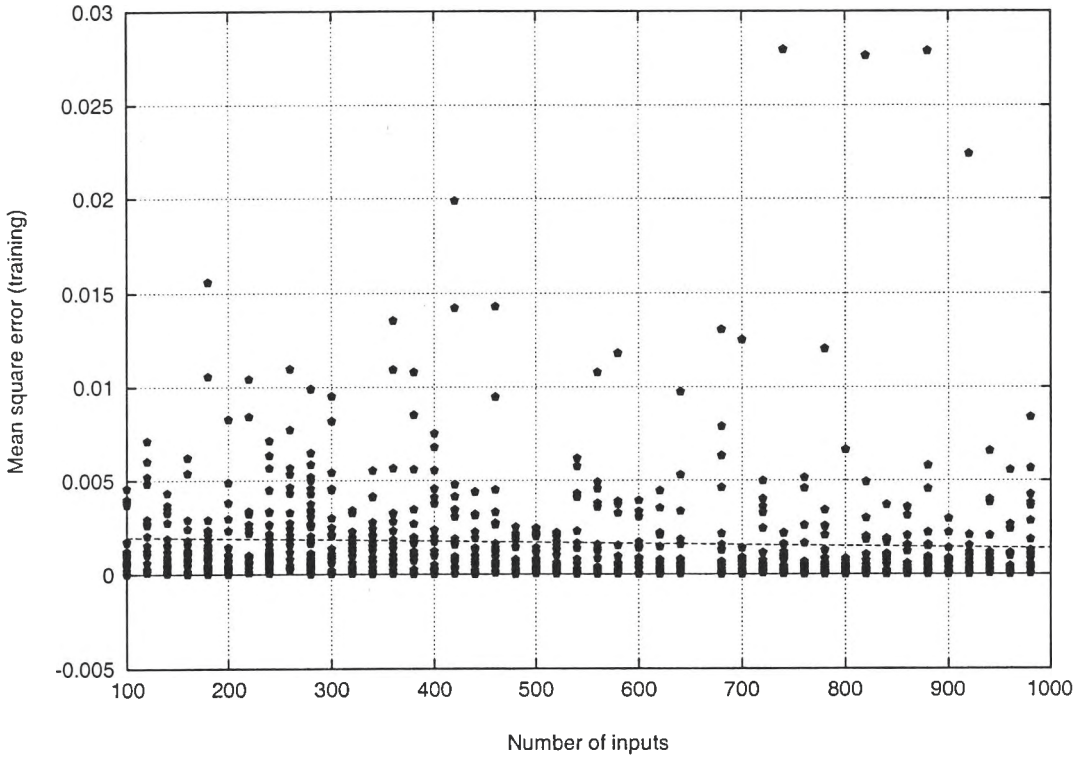

Figure 6.30: $\mathcal{N}_2$, **sample error against number of inputs**

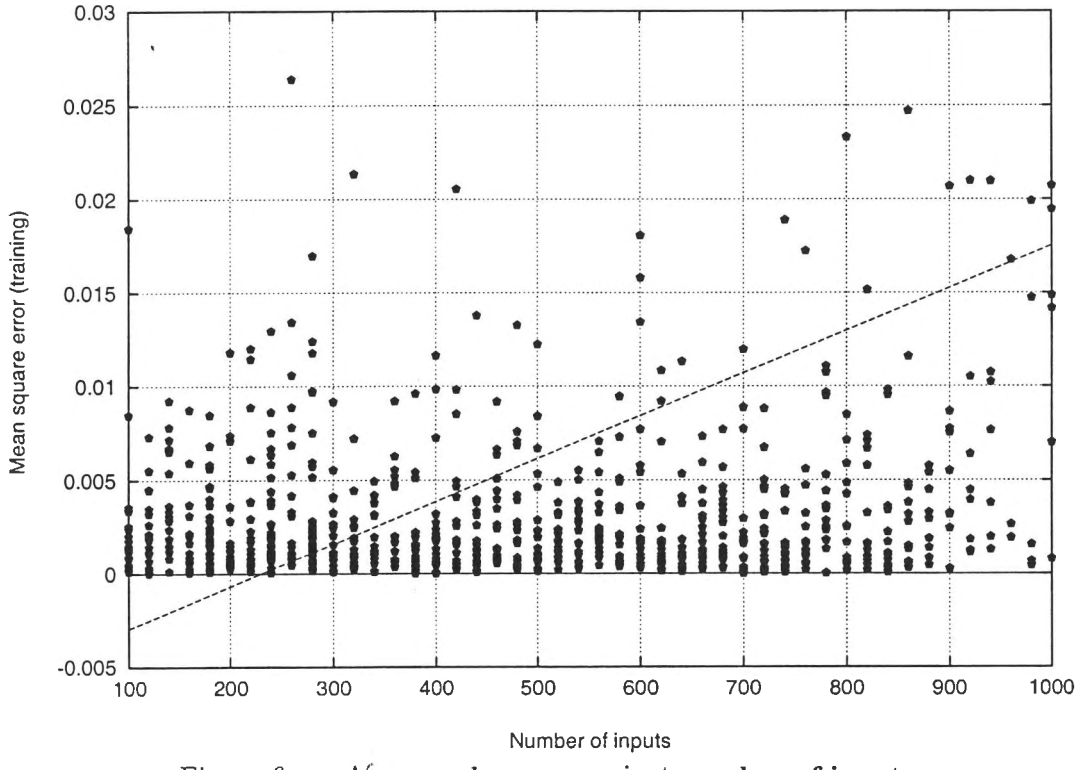

Figure 6.31: $\mathcal{N}_3$, **sample error against number of inputs**

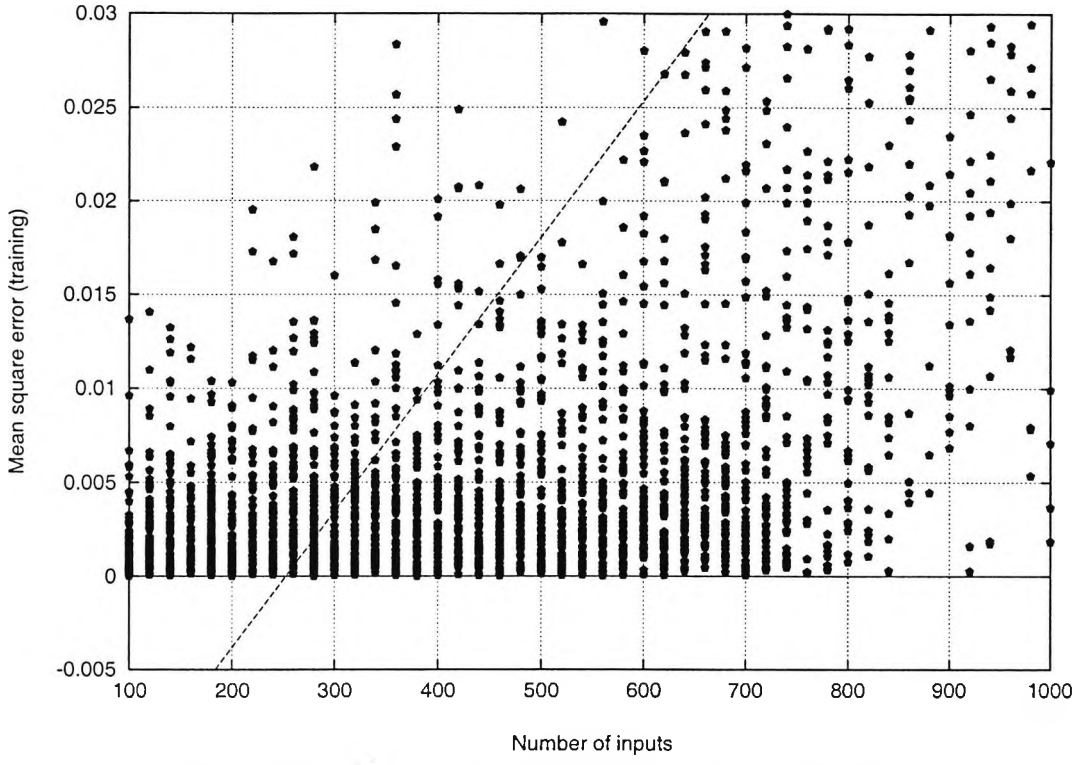

Figure 6.32: $\mathcal{N}_4$, **sample error** against **number of inputs**

6.3.6.5 VARIDIM: approximation error results

The entities invariably yield a lower **approximation error** than that of the single FFNN. In particular, $\mathcal{C}_1$ yields – on average – an error of 60×10^{-03}, $\mathcal{C}_{1,big}$ and $\mathcal{C}_2$ follow with marginally *higher* errors of 62×10^{-03} and 67×10^{-03}. The *highest* **error** is due to $\mathcal{C}_3$, with 104×10^{-03}.

	$\mathcal{C}_1$		$\mathcal{C}_{1,big}$		$\mathcal{C}_2$		$\mathcal{C}_3$	
	C_1 entity		C_1 (66% more weights)		C_2 entity		C_3 entity	
$\times 10^{-03}$	MEAN	STD.DEV	MEAN	STD.DEV	MEAN	STD.DEV	MEAN	STD.DEV
lowest	46.803	1.738	49.427	2.193	55.735	4.171	87.216	3.10
average	60.036	6.780	61.410	7.509	66.779	8.831	104.12	11.49
highest	70.139	14.757	74.769	14.270	83.654	16.147	130.71	19.62

Table 6.8: VARIDIM, **approximation error** statistics for the entities

In contrast, the **approximation error** of the single FFNN is *one and a half to two times higher* than that of the entities. In particular, $\mathcal{N}_3$ (note that this network has 55 % more weights than $\mathcal{C}_1$) yields – on average – the lowest error among the single FFNN with 96×10^{-03}, $\mathcal{N}_2$, $\mathcal{N}_3$ and $\mathcal{N}_4$ follow with errors of 121×10^{-03}, 125×10^{-03} and 127×10^{-03}, respectively.

	$\mathcal{N}_1$ 35% less weights		$\mathcal{N}_2$ standard		$\mathcal{N}_3$ 55% more weights		$\mathcal{N}_4$ 135% more weights	
$\times 10^{-03}$	MEAN	STD.DEV	MEAN	STD.DEV	MEAN	STD.DEV	MEAN	STD.DEV
lowest	89.147	8.972	87.986	9.066	39.704	19.487	90.219	8.99
average	126.58	21.929	120.82	17.185	95.932	37.118	124.52	35.54
highest	157.49	43.722	144.23	35.327	209.83	117.39	169.87	161.16

Table 6.9: VARIDIM, **approximation error** statistics for single FFNN

The statistical significance tests comparing the *mean* **approximation errors** of the participating networks (the *t-test*) show that below 500 input dimensions (see table 6.10) only $\mathcal{N}_3$'s **approximation error** is comparable to that of the entities: the *null hypothesis*, H_0, that the means of $\mathcal{C}_1$'s, $\mathcal{C}_{1,big}$'s and $\mathcal{C}_2$'s **approximation errors** do not differ significantly from that of $\mathcal{N}_3$, is true by 43 %, 57 % and 43 %, respectively (see the $\mathcal{N}_3$-column of table 6.10). $\mathcal{C}_3$'s **approximation error** is comparable to that of $\mathcal{N}_1$ and $\mathcal{N}_2$, higher than that of $\mathcal{N}_3$ but, lower than $\mathcal{N}_4$'s (see the $\mathcal{C}_3$-row of table 6.10).

%, %, %	$\mathcal{N}_1$	$\mathcal{N}_2$	$\mathcal{N}_3$	$\mathcal{N}_4$
$\mathcal{C}_1$	0, 0, 100	0, 0, 100	43, 19, 38	0, 0, 100
$\mathcal{C}_{1,big}$	0, 0, 100	0, 0, 100	57, 24, 19	0, 0, 100
$\mathcal{C}_2$	0, 0, 100	0, 0, 100	43, 24, 33	0, 0, 100
$\mathcal{C}_3$	33, 29, 38	33, 24, 43	5, 95, 0	19, 43, 38

Table 6.10: VARIDIM, statistical significance (*t-test*) of the **approximation error** results for 100 to 500 input dimensions

%, %, %	$\mathcal{N}_1$	$\mathcal{N}_2$	$\mathcal{N}_3$	$\mathcal{N}_4$
$\mathcal{C}_1$	0, 0, 100	0, 0, 100	0, 0, 100	0, 0, 100
$\mathcal{C}_{1,big}$	0, 0, 100	0, 0, 100	0, 0, 100	0, 0, 100
$\mathcal{C}_2$	0, 0, 100	0, 0, 100	4, 0, 96	0, 0, 100
$\mathcal{C}_3$	4, 0, 96	4, 0, 96	48, 12, 40	8, 0, 92

Table 6.11: VARIDIM, statistical significance (*t-test*) of the **approximation error** results for more than 500 input dimensions

When the number of input dimensions exceeds 500, the generalisation ability of the entities is clearly superior. In particular, the hypothesis H_2 that the mean of an entity's **approximation error** is lower than that of a single FFNN is true by 100 % (see the top 3 rows of table 6.11). Even $\mathcal{C}_3$ is now doing much better than $\mathcal{N}_1$, $\mathcal{N}_2$ and $\mathcal{N}_4$. Its performance is comparable to that of $\mathcal{N}_3$, even if $\mathcal{N}_3$ has twice as many weights.

The **approximation error** of all entity models is clearly *independent of, if not decreasing with, the number of input dimensions.* This can be seen from the graphs of the *mean* **approximation error** for $\mathcal{C}_1$, $\mathcal{C}_{1,big}$, $\mathcal{C}_2$ and $\mathcal{C}_3$ contained in figures 6.33, 6.34, 6.35 and 6.36, respectively. This conclusion is further supported by examination of the scatter plots and the lines fitted on them, depicted in figures 6.41, 6.42, 6.43 and 6.44, for $\mathcal{C}_1$, $\mathcal{C}_{1,big}$, $\mathcal{C}_2$ and $\mathcal{C}_3$ respectively.

%, %, %	$\mathcal{N}_1$	$\mathcal{N}_2$	$\mathcal{N}_3$	$\mathcal{N}_4$
$\mathcal{C}_1$	19, 0, 81	33, 0, 67	0, 0, 100	10, 0, 90
$\mathcal{C}_{1,big}$	52, 0, 48	57, 0, 43	0, 0, 100	29, 0, 71
$\mathcal{C}_2$	38, 5, 57	47, 5, 48	0, 0, 100	38, 5, 57
$\mathcal{C}_3$	76, 10, 14	80, 10, 10	5, 0, 95	52, 5, 43

Table 6.12: VARIDIM, statistical significance (*F-test*) of the **approximation error** results for 100 to 500 input dimensions

%, %, %	$\mathcal{N}_1$	$\mathcal{N}_2$	$\mathcal{N}_3$	$\mathcal{N}_4$
$\mathcal{C}_1$	0, 0, 100	0, 0, 100	0, 0, 100	0, 0, 100
$\mathcal{C}_{1,big}$	0, 0, 100	0, 0, 100	0, 0, 100	0, 0, 100
$\mathcal{C}_2$	0, 0, 100	0, 0, 100	0, 0, 100	0, 0, 100
$\mathcal{C}_3$	0, 0, 100	26, 0, 74	0, 0, 100	0, 0, 100

Table 6.13: VARIDIM, statistical significance (*F-test*) of the **approximation error** results for more than 500 input dimensions

In particular, the **approximation error** of $\mathcal{C}_1$ is *almost constant.* This is indicated by the slope of the least mean squares line fitted on the scattered points; it is almost zero (-3.2×10^{-06}). The same can be said for $\mathcal{C}_{1,big}$ (the slope is -5.7×10^{-06}) and $\mathcal{C}_2$ (the slope is -5.6×10^{-06}). $\mathcal{C}_3$ follows a rather strange "wavy" pattern which contributes to its large *standard deviation* value. Notice also that all slopes, no matter how small, are **negative**, thus showing a descending trend.

The single FFNN's generalisation ability consistently shows a *strong dependence on the number of input dimensions: it decreases (e.g. the* **approximation error** *increases) as the number of input dimensions increases.* This is evident from the scatter plots of $\mathcal{N}_1$, $\mathcal{N}_2$, $\mathcal{N}_3$ and $\mathcal{N}_4$, in figures 6.45, 6.46, 6.47 and 6.48, respectively. The least mean squares lines fitted on these scattered points have a large **positive slope** which indicates an increasing trend. In particular, the standard FFNN, $\mathcal{N}_2$, has the smallest slope with 42×10^{-06}, whereas $\mathcal{N}_3$ has a slope of 165×10^{-06}. The corresponding slope values for $\mathcal{N}_1$ and $\mathcal{N}_4$ are somewhere in between these two extremes with 65×10^{-06}

and 71×10^{-06}, respectively.

The statistical significance tests comparing the *variance* of **approximation error** of the participating networks (the *F-test*) reveal that for a number of input dimensions higher than 500, the *variance* of all the entity networks is significantly lower than that of any single FFNN. For lower dimensions, only $\mathcal{N}_1$'s and $\mathcal{N}_2$'s *variances* are comparable to that of the entities.

It is important to note that for the whole range of input dimensions, the variance of $\mathcal{N}_3$ is higher than that of any entity 100 % of the times (see the $\mathcal{N}_3$-column of tables 6.12 and 6.13). *Thus, the benefits gained by the low approximation error of $\mathcal{N}_3$ are nullified by its large inconsistency. This inconsistency gets larger as the number of input dimensions increases.*

Some more conclusions can be drawn by visual inspection of the scatter plots:

Firstly, there is a striking difference between the compactness of the plots for the entities and the single FFNN. For example compare the scatter plots of $\mathcal{C}_1$ and $\mathcal{C}_{1,big}$ (figures 6.41 and 6.42) to the plots of $\mathcal{N}_2$ and $\mathcal{N}_3$ (figures 6.46 and 6.47).

Secondly, whereas the entities' **approximation error** remains *constant* or, in some cases, it even *decreases* with the increasing number of input dimensions, the single FFNN's **approximation error** *increases.*

Thirdly, the number of points which significantly deviate from the mean (outliers) for the case of the majority of the entity models is insignificant (e.g. $\mathcal{C}_1$ and $\mathcal{C}_{1,big}$) or just very small ($\mathcal{C}_2$). Furthermore, these outliers are *reduced* as the number of input dimensions **increases**. The opposite is observed for the single FFNN: when the number of input dimensions is small, the outliers are not so many but they become more and more as the number of inputs increases. Evidence of this can also be found in figures 6.33 to 6.40. In particular, the *standard deviation* plots for the entities (figures 6.33 to 6.36) are, in general, non-increasing whereas the corresponding curves for the single FFNN (contained in the other four figures, 6.37 to 6.40) show a clearly increasing trend.

The following figures show the *minimum*, *maximum*, *mean* and *standard deviation* of the **approximation error** of each of the evaluated networks after a 1,000 iterations as a function of the number of their **inputs**.

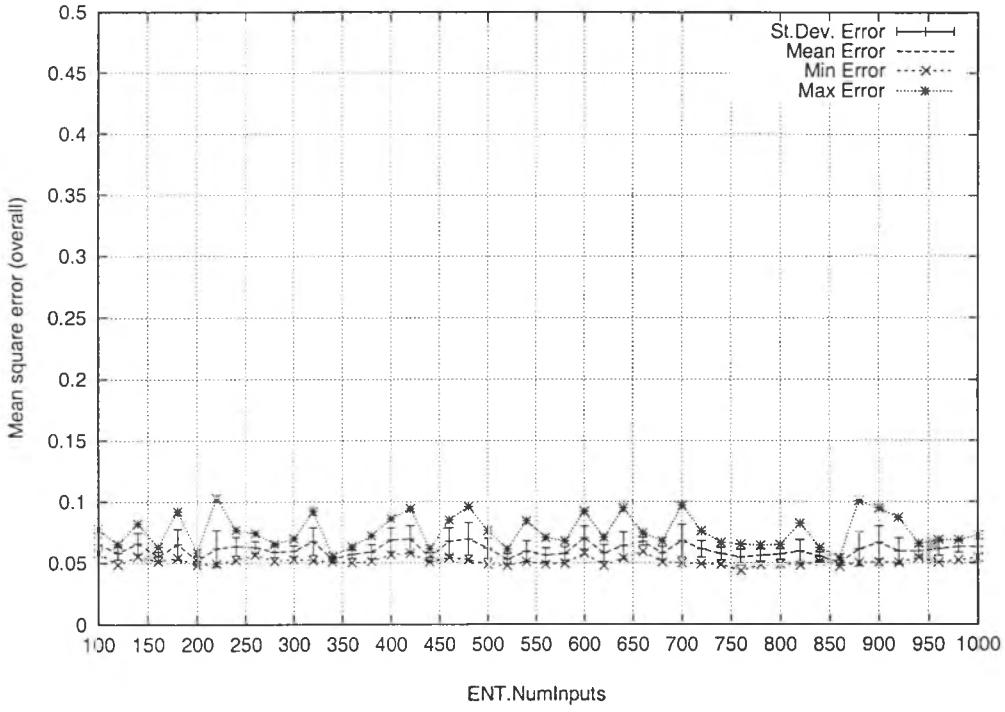

Figure 6.33: C_1, **approximation error** against **number of inputs**

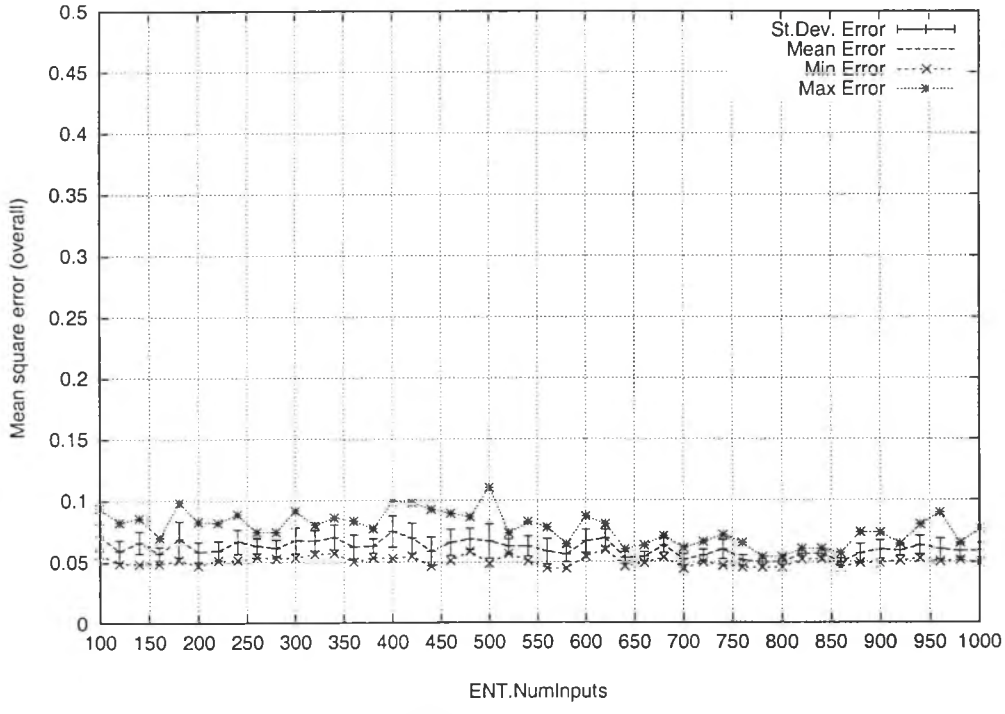

Figure 6.34: $C_{1,big}$, **approximation error** against **number of inputs**

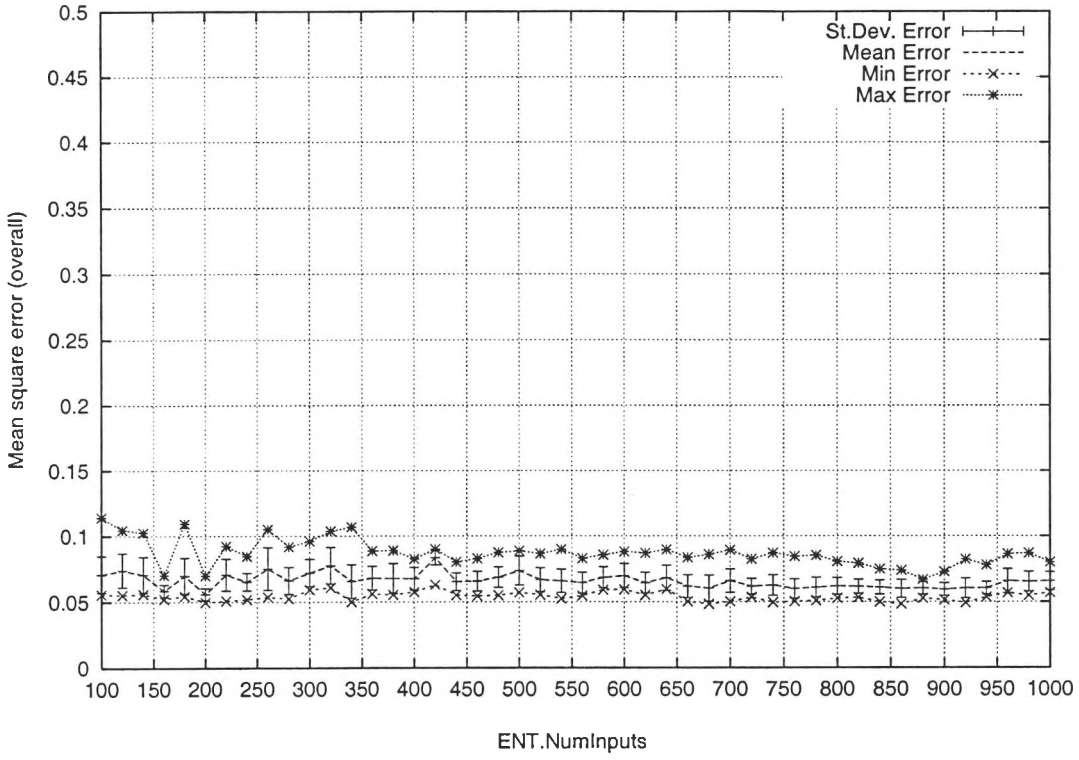

Figure 6.35: C_2, **approximation error** against **number of inputs**

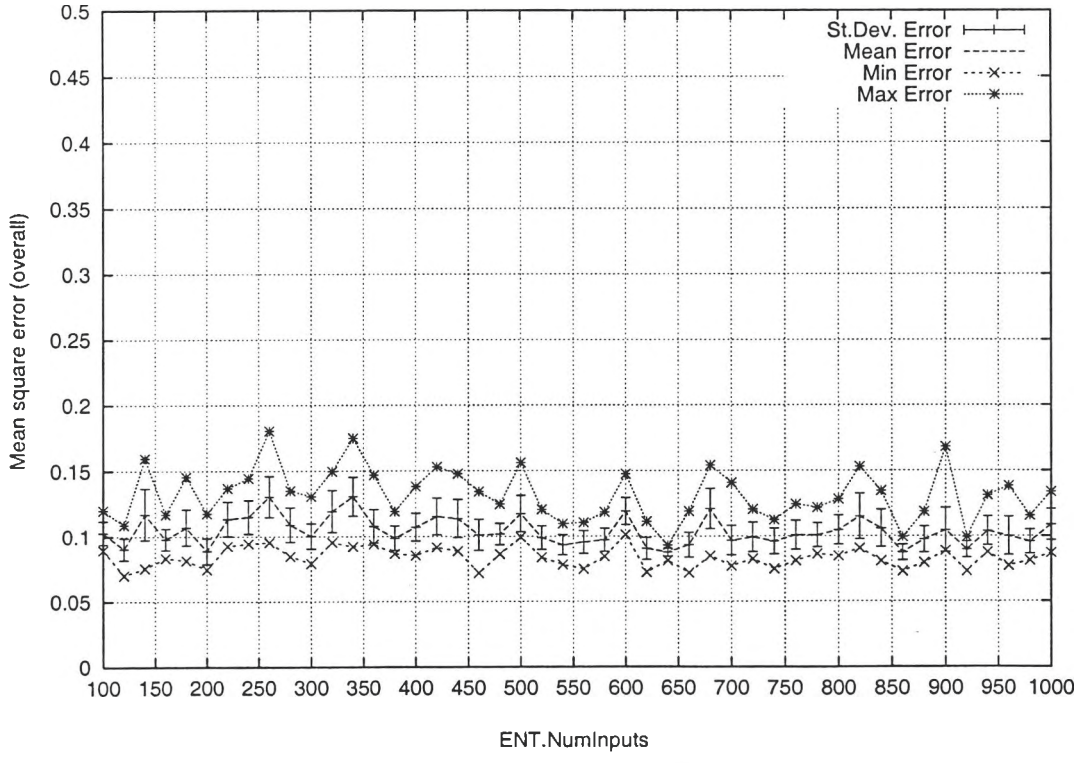

Figure 6.36: C_3, **approximation error** against **number of inputs**

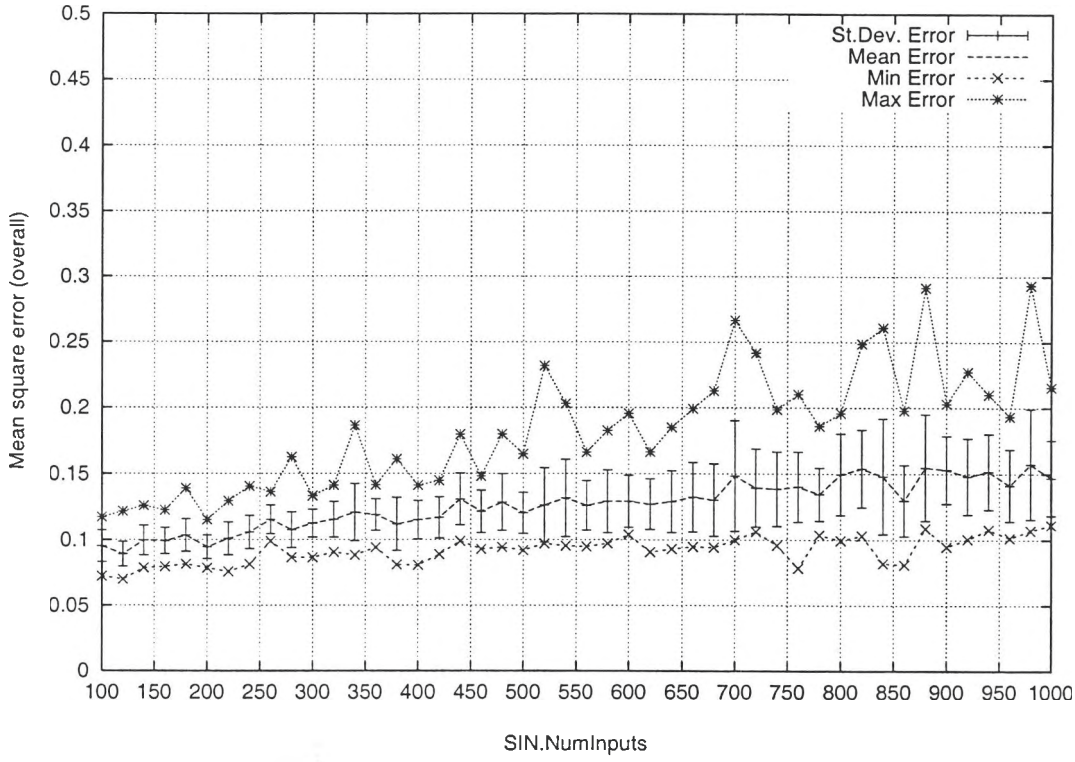

Figure 6.37: $\mathcal{N}_1$, **approximation error** against **number of inputs**

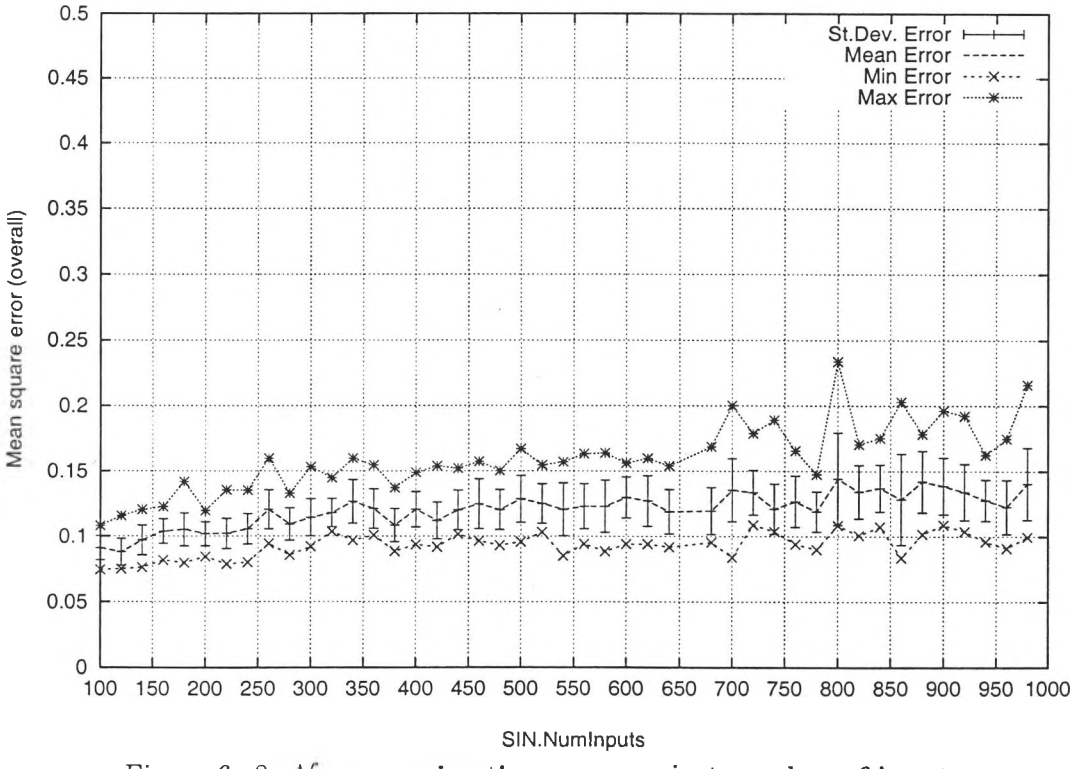

Figure 6.38: $\mathcal{N}_2$, **approximation error** against **number of inputs**

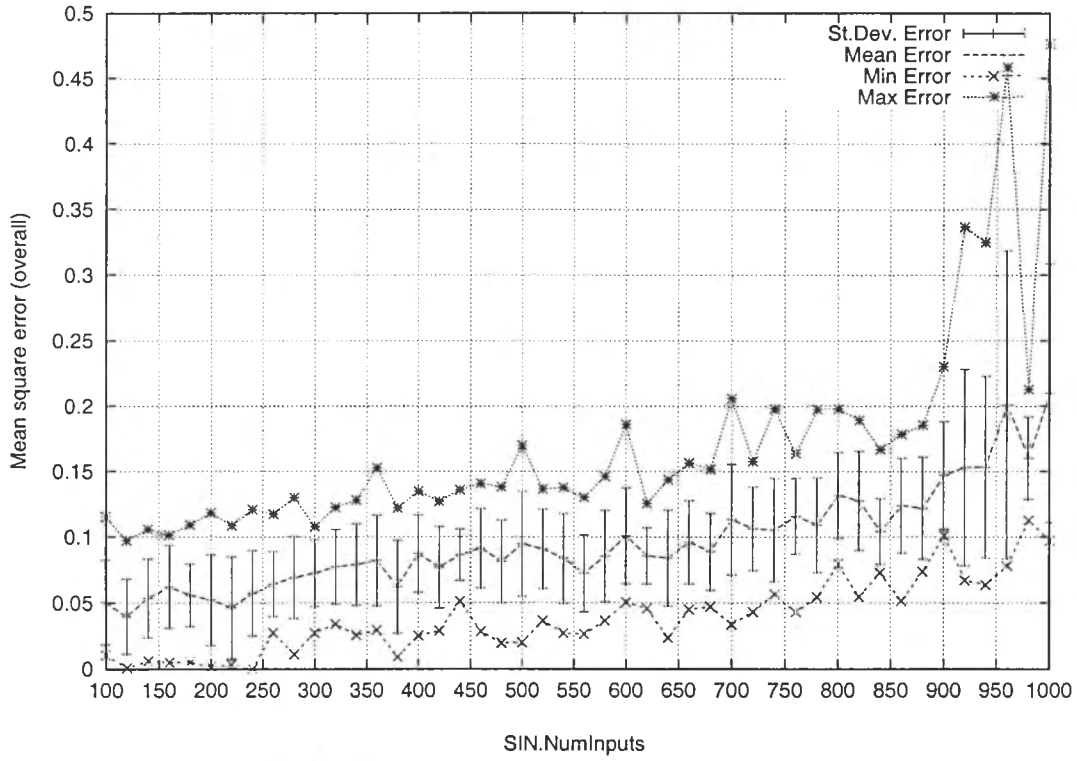

Figure 6.39: $\mathcal{N}_3$, **approximation error** against **number of inputs**

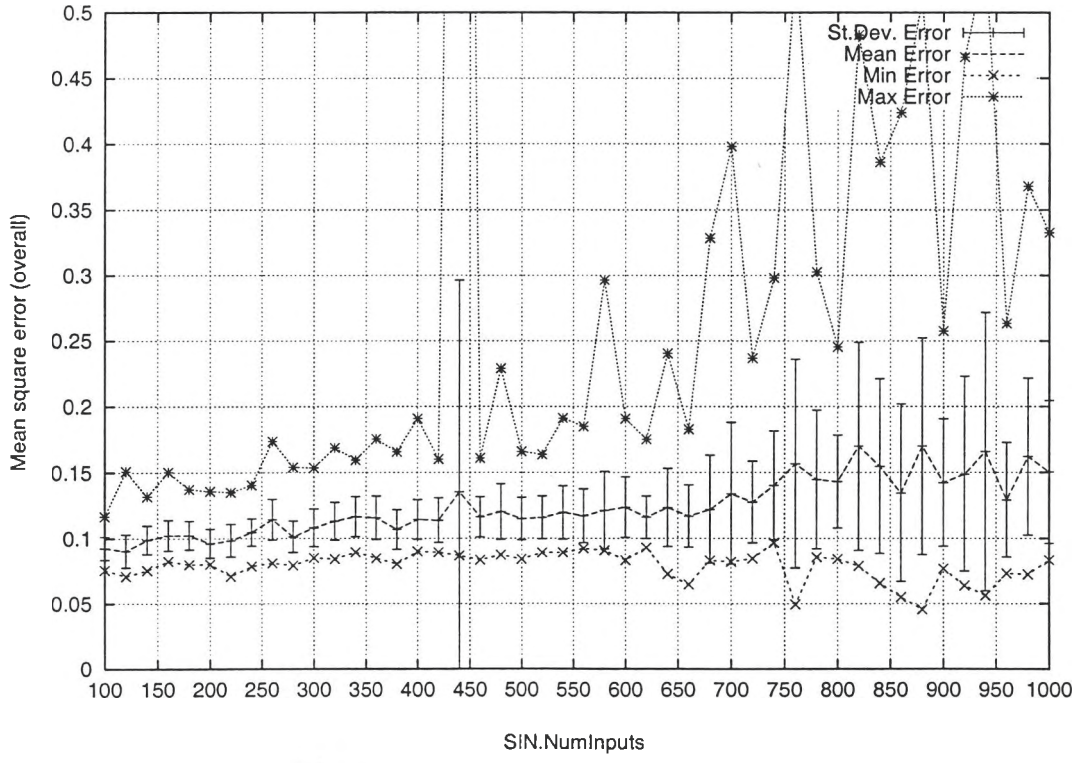

Figure 6.40: $\mathcal{N}_4$, **approximation error** against **number of inputs**

The following figures contain scatter plots of the *approximation error*, and least mean squares lines fitted on them, for all evaluated networks, as the number of their **inputs** increases.

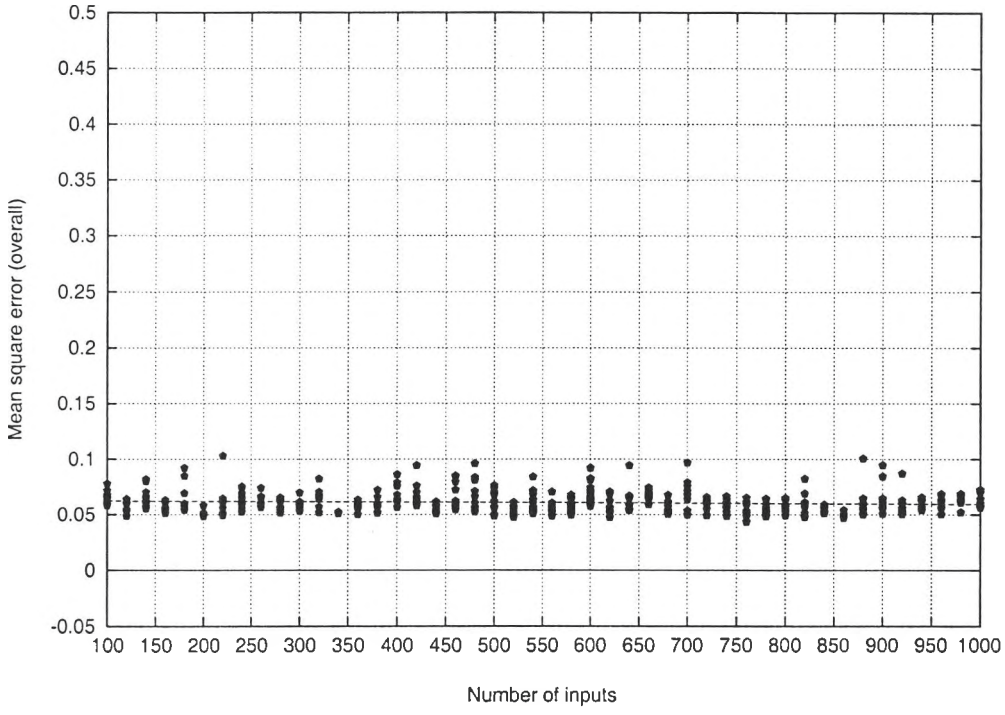

Figure 6.41: $\mathcal{C}_1$, **approximation error** against **number of inputs**

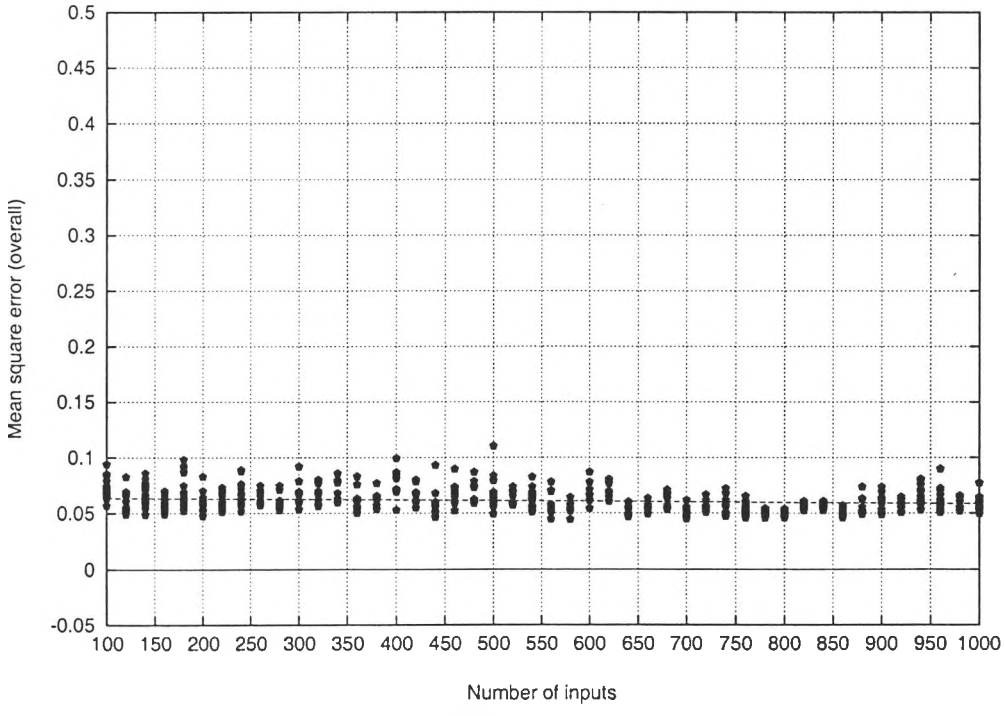

Figure 6.42: $\mathcal{C}_{1,big}$, **approximation error** against **number of inputs**

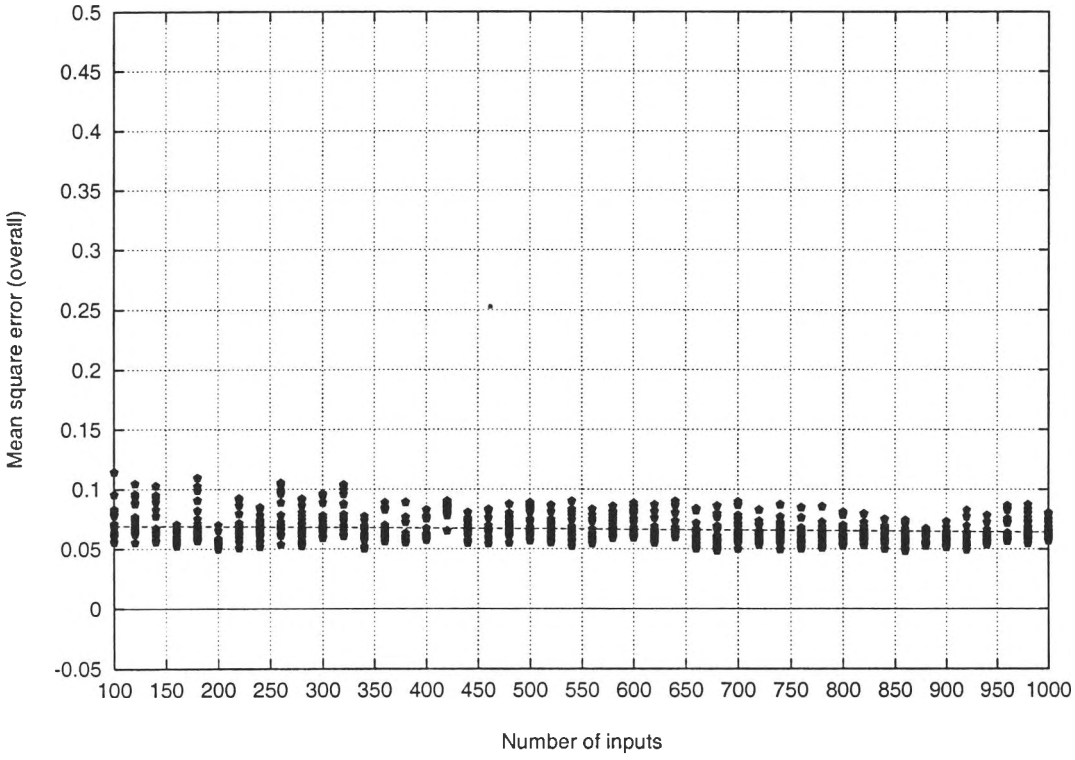

Figure 6.43: $\mathcal{C}_2$, **approximation error** against **number of inputs**

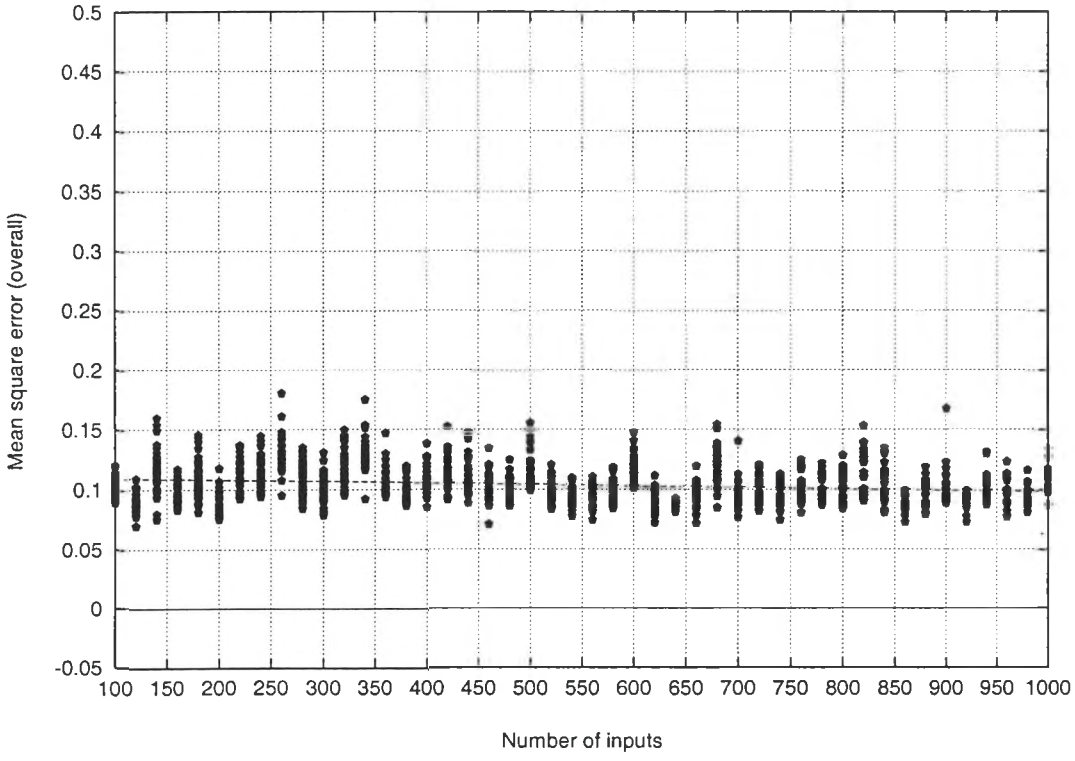

Figure 6.44: $\mathcal{C}_3$, **approximation error** against **number of inputs**

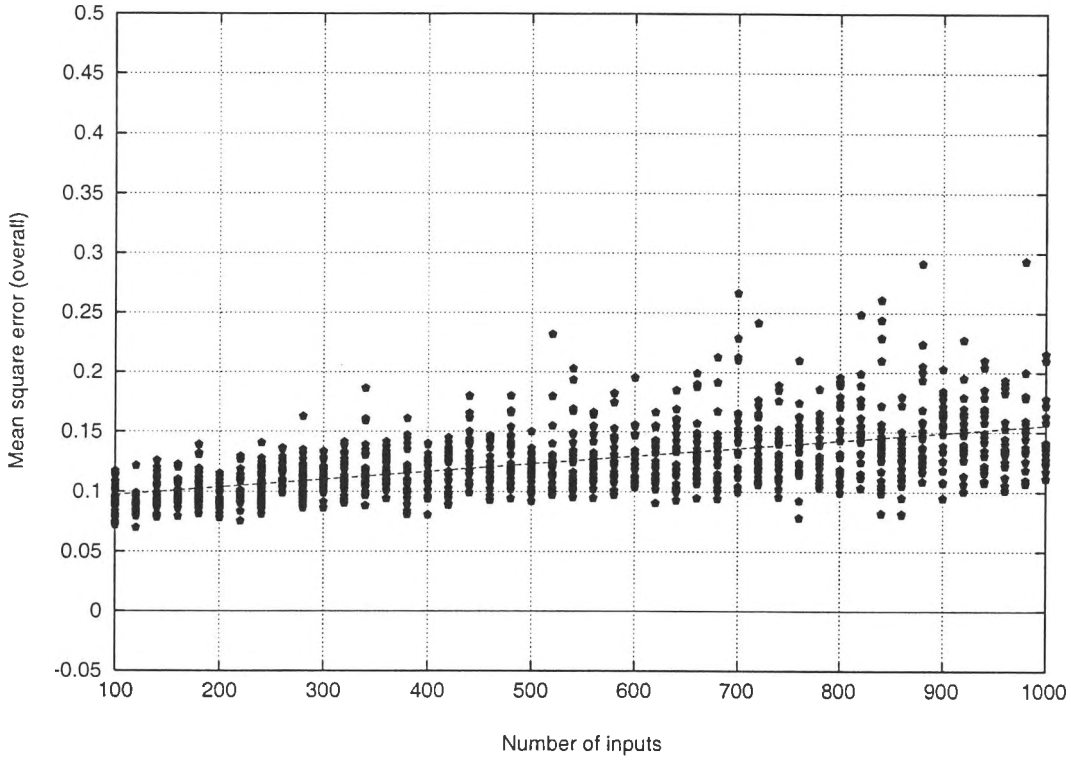

Figure 6.45: $\mathcal{N}_1$, **approximation error** against **number of inputs**

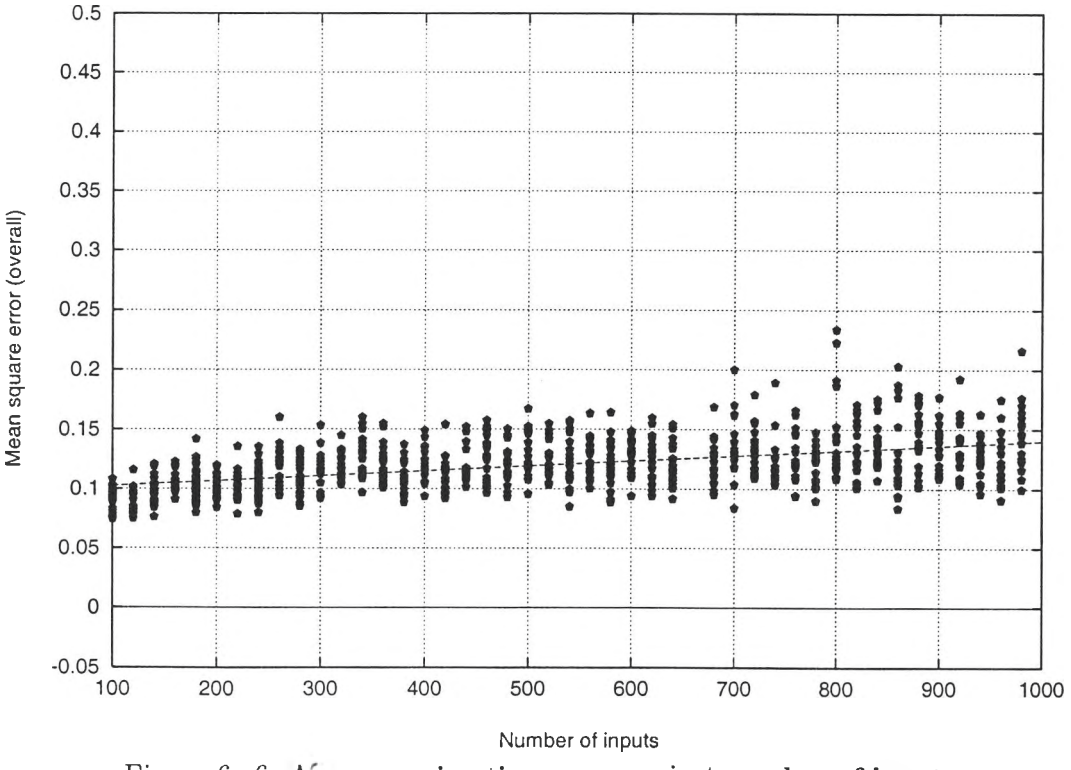

Figure 6.46: $\mathcal{N}_2$, **approximation error** against **number of inputs**

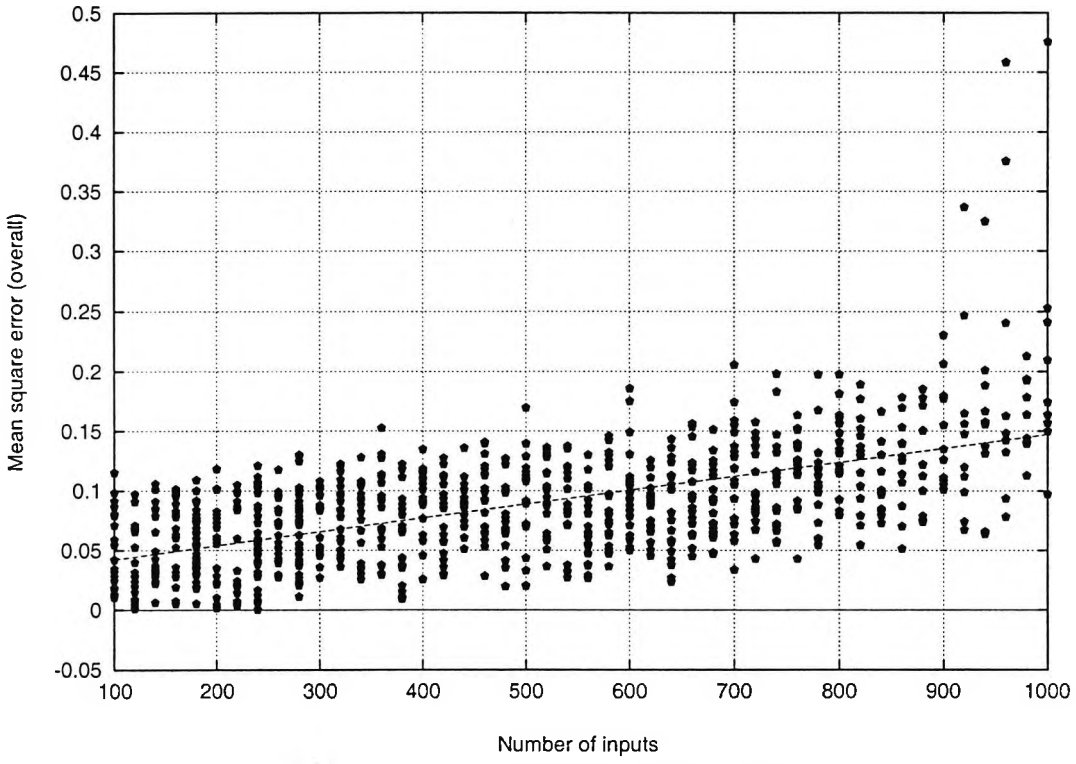

Figure 6.47: $\mathcal{N}_3$, **approximation error** against **number of inputs**

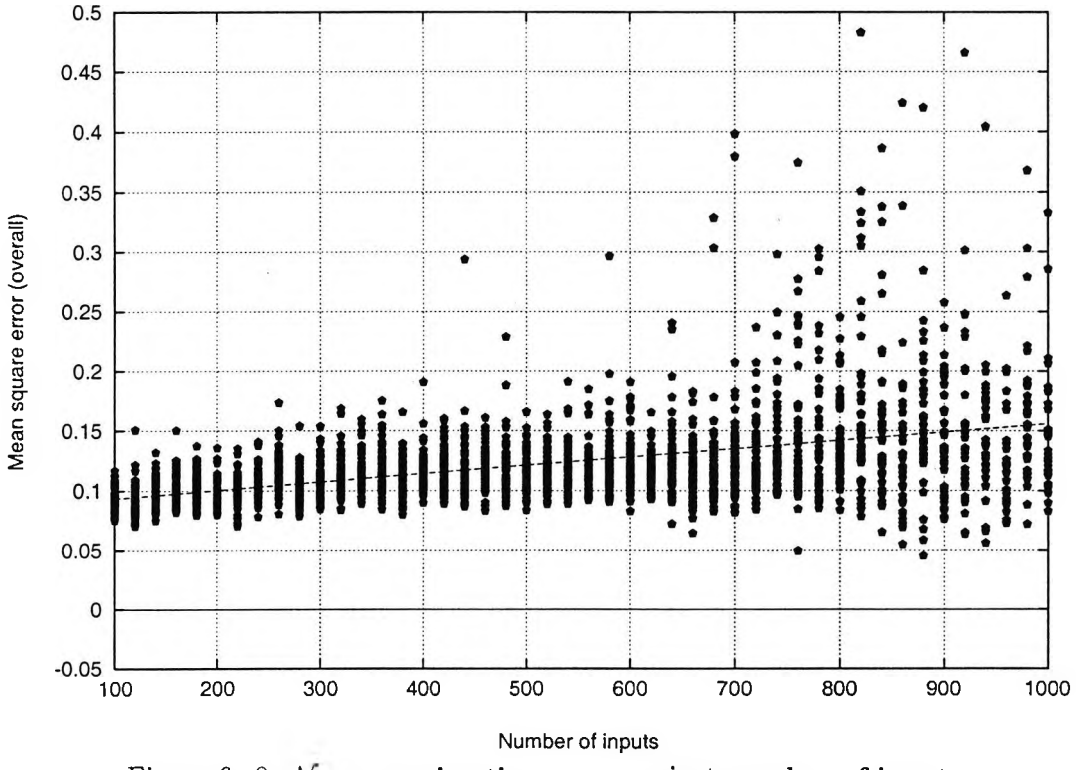

Figure 6.48: $\mathcal{N}_4$, **approximation error** against **number of inputs**

6.3.7 Generalisation Ability: the ConsDim test

6.3.7.1 ConsDim: Methodology

In this test we are interested in the performance (**training time**, **sample error** and **approximation error**) of the networks when, for a given number of input dimensions, the number of vectors in their training set increases. The networks' configuration and number of weights remains constant throughout this test.

The procedure for ConsDim is outlined below:

- The networks to be tested are described in the following table:

NAME	DESCRIPTION
$\mathcal{C}_1$	a C_1 **entity** model with 12,625 weights
$\mathcal{C}_{1,big}$	a C_1 **entity** model with 20,010 weights
$\mathcal{C}_2$	a C_2 **entity** model with 12,340 weights
$\mathcal{C}_3$	a C_3 **entity** model with 12,340 weights
$\mathcal{N}_1$	a **single** FFNN with 12,525 weights (arch: $500 \times 25 \times 1$)
$\mathcal{N}_2$	a **single** FFNN with 20,040 weights (arch: $500 \times 40 \times 1$)
$\mathcal{N}_3$	a **single** FFNN with 25,050 weights (arch: $500 \times 50 \times 1$)
$\mathcal{N}_4$	a **single** FFNN with 30,060 weights (arch: $500 \times 60 \times 1$)

Table 6.14: ConsDim: description of the evaluated networks

- each FFNN unit of the entity models had 12 to 35 inputs, one hidden layer of 10 to 20 units, and a single output,
- the number of training vectors was varied from 20 to 220 (step 10) and the input dimensions were fixed at 500,
- for each different training data set the entities and the single FFNN were trained for 1,000 iterations,
- the test data consisted of 2,000 vectors. None of these vectors was used for training. The final **sample** and **approximation errors** were calculated using the *mean squared error measure* (see equation 3.6 on page 26),
- for a given number of training vectors, the training / test procedure was repeated for 50 times with the same data and network architecture but with different starting weights. The **training time** and **sample** and **approximation errors** were recorded for each of these training attempts and then the *maximum*, *minimum*, *mean* and *standard deviation* were calculated as follows:

```
procedure ConsDim:
  produce neural network/entity
  calculate total number of weights, numW
  for number of training vectors i := 20 to 220 step 10 do
    produce training data set
    produce test data set
    initialise sum and sumsum to zero
    reset minimum error_i and maximum error_i
    for training attempts j := 1 to 50 do
      initialise weights to random
      train neural network/entity for 1,000 iterations
      test neural network/entity
      calculate error_ij using equation for mean squared error
      store error_ij
      update minimum error_i := minimum of (error_ij and minimum error_i)
      update maximum error_i := maximum of (error_ij and maximum error_i)
      sum := sum + error_ij
      sumsum := sumsum + error_ij^2
    end
    mean error_i := sum / 50
    standard deviation of error_i := sqrt( sumsum / 50 - (mean error_i^2) )
    store minimum error_i and maximum error_i
    store mean error_i and standard deviation of error_i
  end
end ConsDim.
```

- the quantities *minimum error*$_i$, *maximum error*$_i$, *mean error*$_i$ and *standard deviation of error*$_i$ were plotted against the total number of training vectors, i. See, for example, figure 6.57 which refers to the **sample error** of the C_1 entity network.

- the values of *error*$_{ij}$ were plotted (as scattered points) against the number of training vectors, i. See, for example, figure 6.65 which refers to the **sample error** results of the C_1 entity network.

- the *mean*, *lowest* and *highest* values of *average error*$_i$ and *standard deviation of error*$_i$ over the whole range of training vectors are shown in various tables throughout the rest of this chapter. See, for example, table 6.15 which refers to the **sample error** results of all the entity networks.

6.3.7.2 ConsDim: training time results

The **training time** of all networks, both entities and single FFNN, is directly proportional to the number of training examples. This is confirmed by figures 6.49, 6.50, 6.51, 6.52, 6.53, 6.54, 6.55 and 6.56, which depict the relationship of training time of the networks $\mathcal{C}_1$, $\mathcal{C}_{1,big}$, $\mathcal{C}_2$, $\mathcal{C}_3$, $\mathcal{N}_1$, $\mathcal{N}_2$, $\mathcal{N}_3$ and $\mathcal{N}_4$ and the number of training vectors.

Again, although the experiments were performed on computers of equal CPU power, the effects of network traffic are noticeable.

The following figures depict the relationship between **training time** and the number of **training vectors**. In particular, four quantities are plotted: *minimum*, *maximum*, *mean* and *standard deviation* of **training time**, for 1,000 iterations.

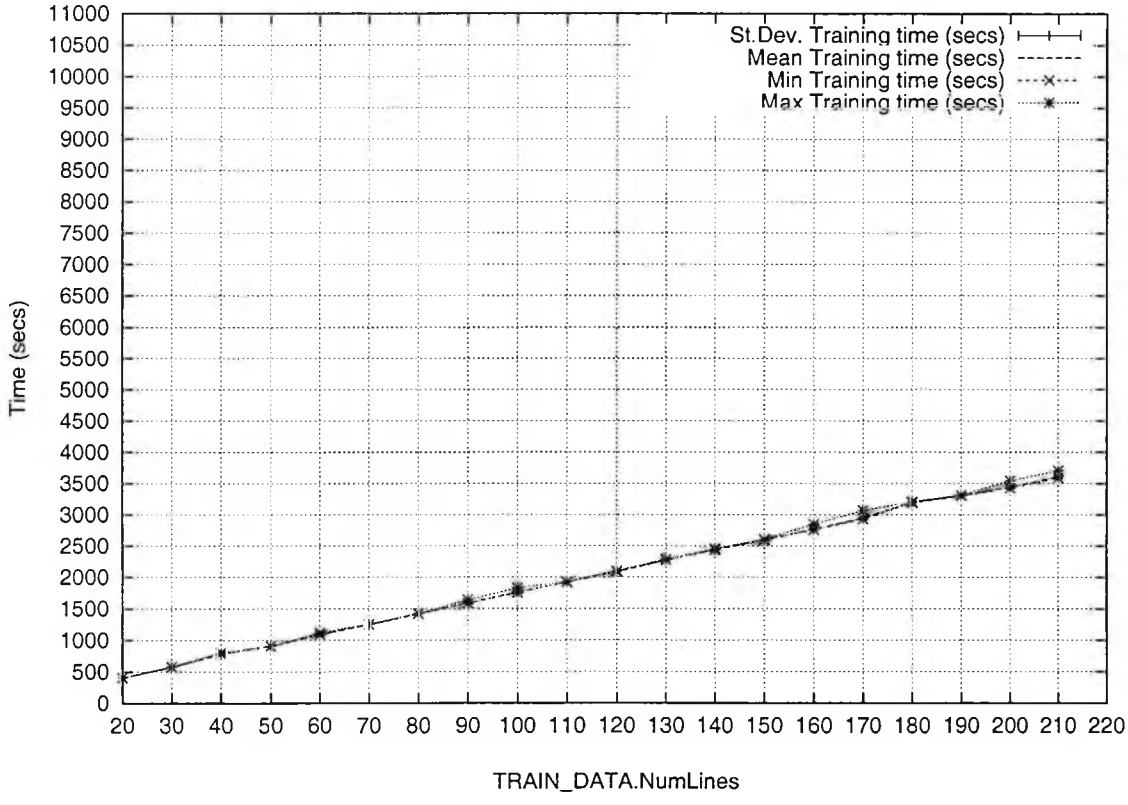

Figure 6.49: $\mathcal{C}_1$, **training time** against number of **training vectors**

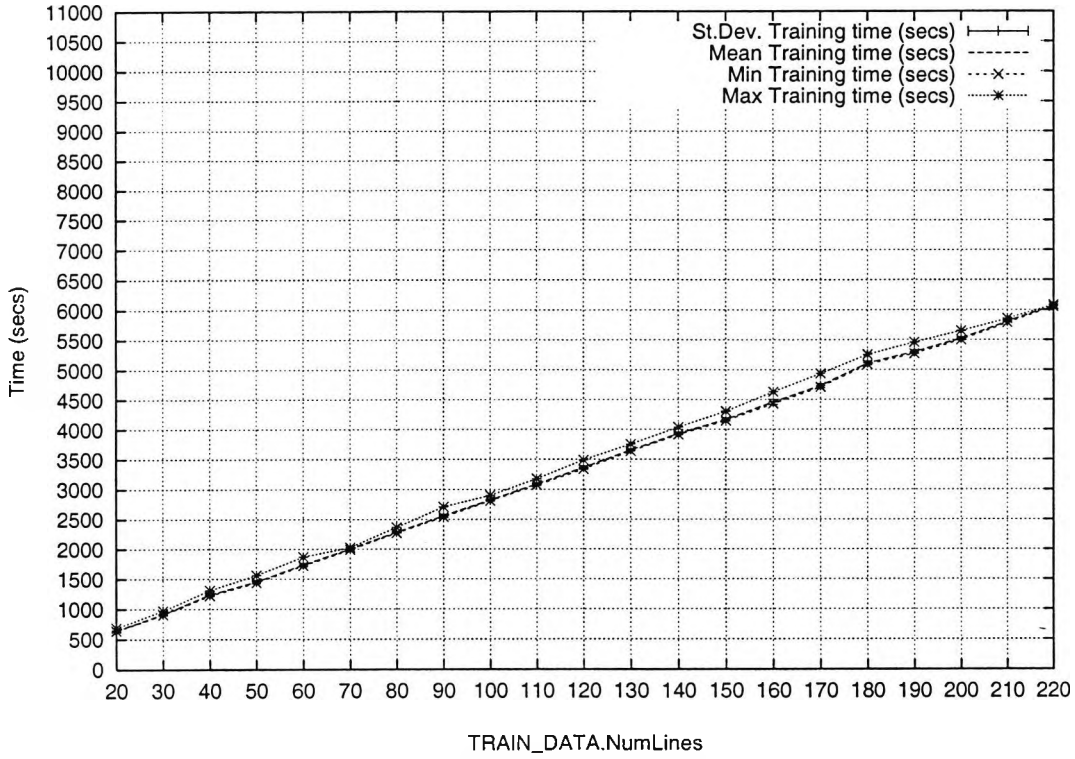

Figure 6.50: $C_{1,big}$, **training time** against number of **training vectors**

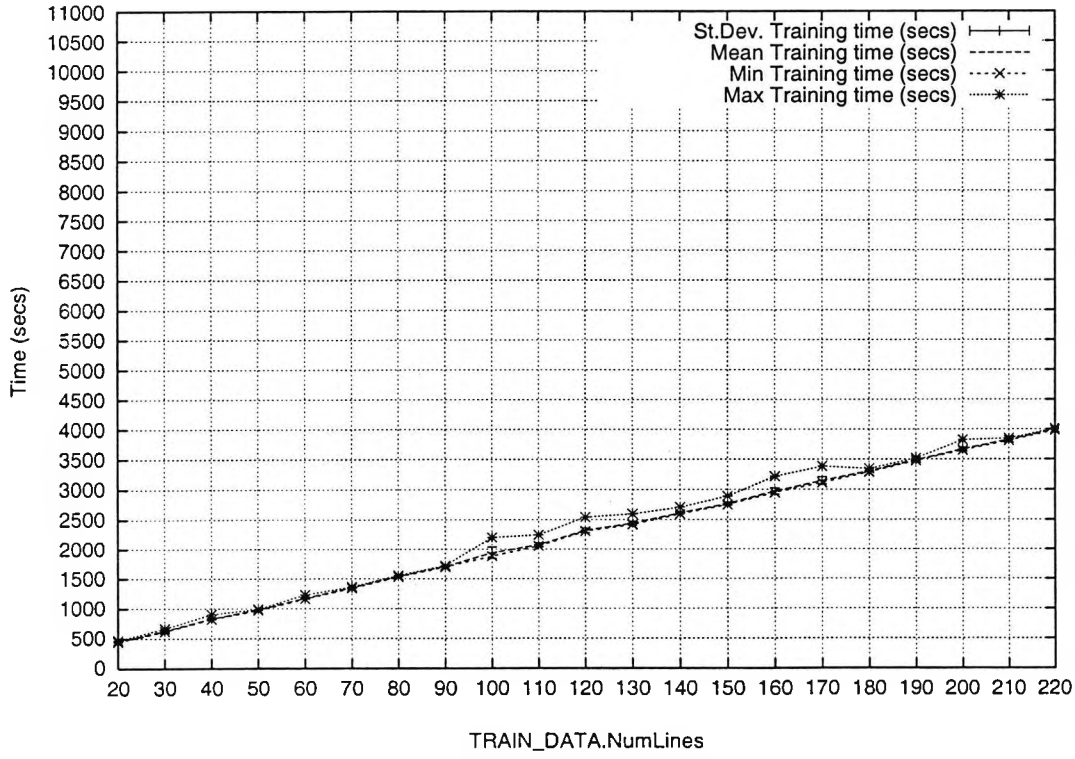

Figure 6.51: C_2, **training time** against number of **training vectors**

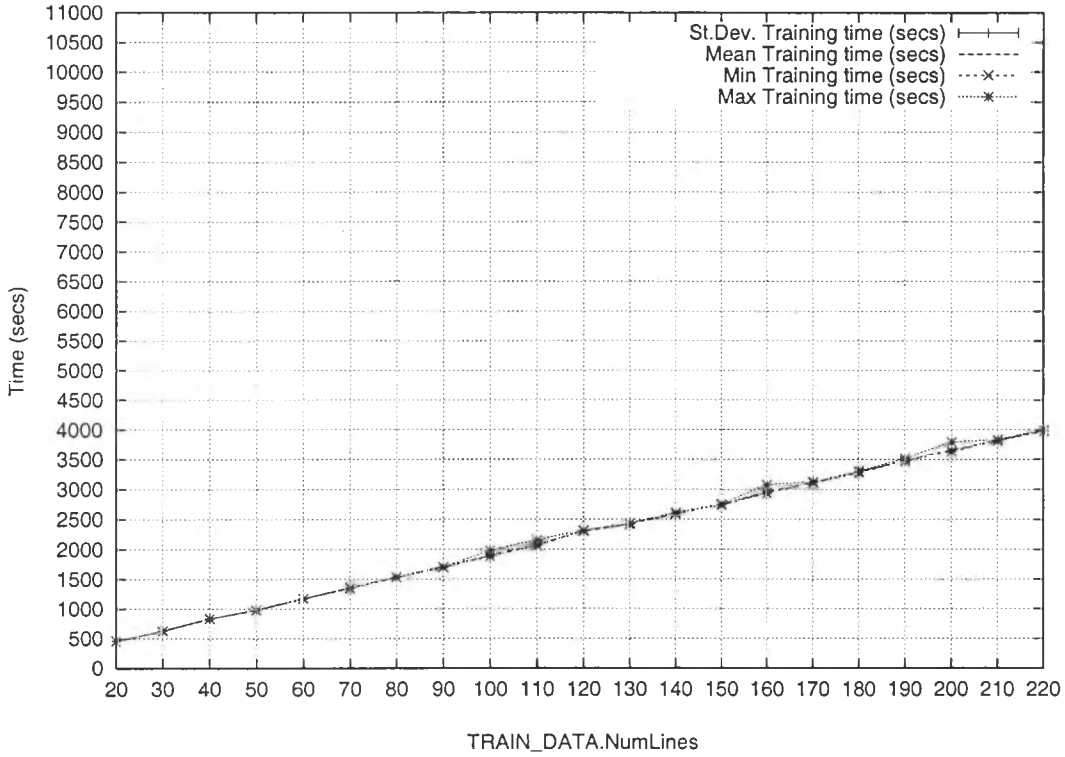

Figure 6.52: $\mathcal{C}_3$, **training time** against number of **training vectors**

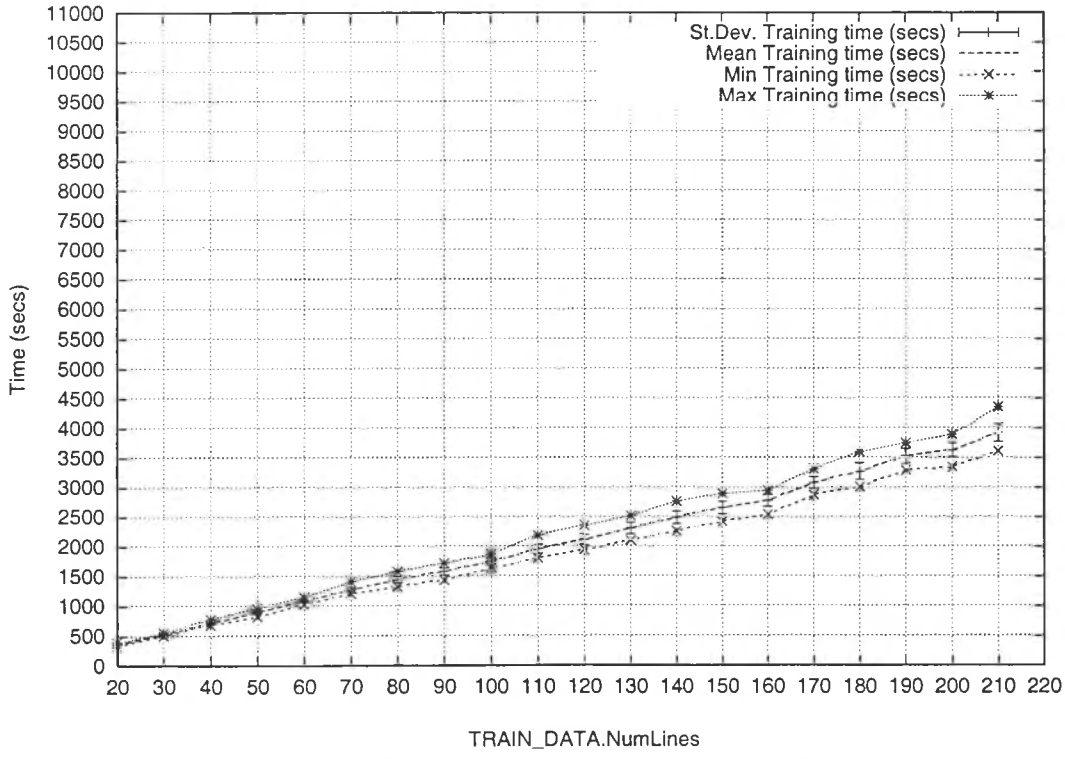

Figure 6.53: $\mathcal{N}_1$, **training time** against number of **training vectors**

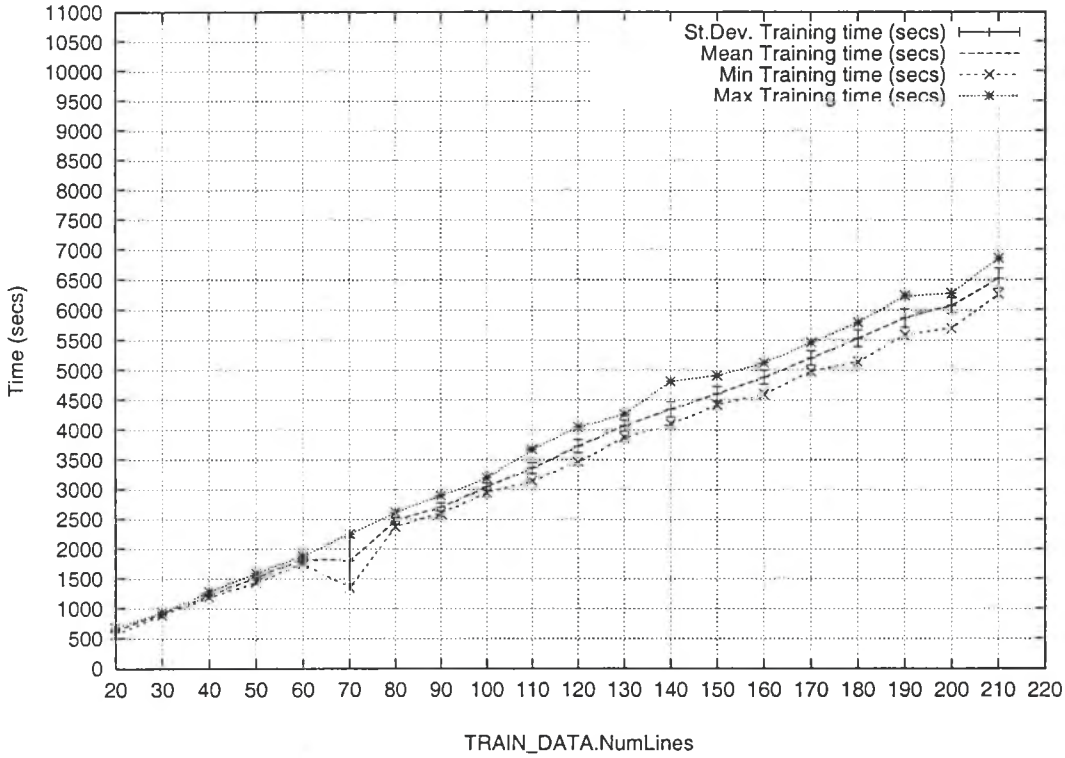

Figure 6.54: $\mathcal{N}_2$, **training time** against number of **training vectors**

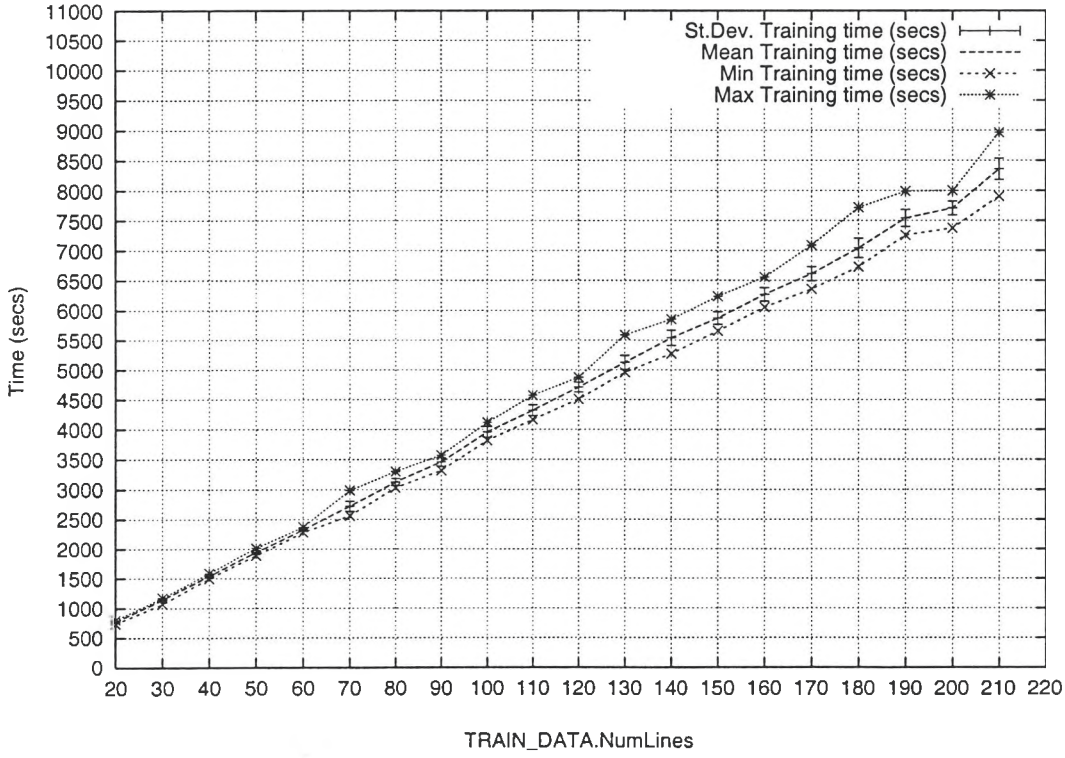

Figure 6.55: $\mathcal{N}_3$, **training time** against number of **training vectors**

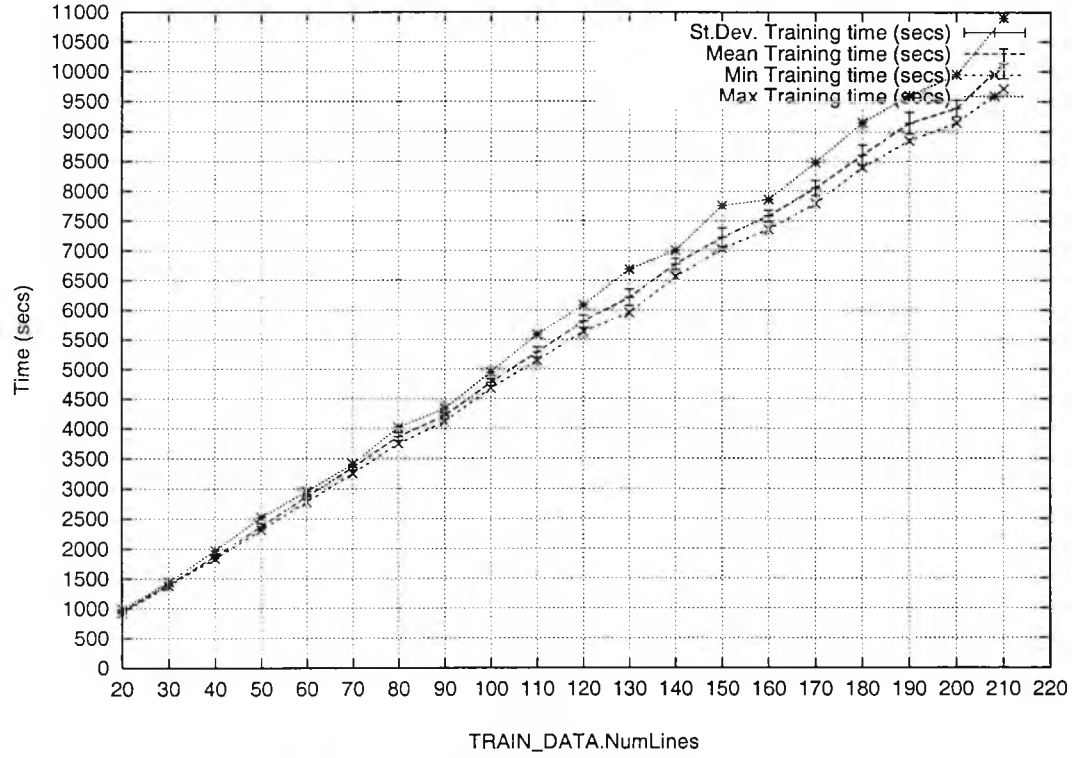

Figure 6.56: $\mathcal{N}_4$, **training time** against number of **training vectors**

6.3.7.3 CONSDIM: sample error results

Figures 6.57, 6.58, 6.59 and 6.60 indicate that all entity models reach a very low **sample error** which, however, increases slightly with the increasing number of training vectors. $\mathcal{C}_{1,big}$ achieves the *lowest* error among the entities (see table 6.15) with 0.8×10^{-03}, on average. Also, the slope of the least mean squares line fitted on the scatter points representing its **sample error** values (see figure 6.66) is, again, the lowest with 7×10^{-06}. Its training is *consistent* and there are very few outliers beyond the 5×10^{-03} error level.

The same can be said for $\mathcal{C}_1$, although its sample error is, on average, slightly higher with 1×10^{-03}. The corresponding least mean squares line has a slope of 11×10^{-06} (figure 6.65).

$\mathcal{C}_3$'s **sample error** increases with the greatest rate among the entities, with a slope of 21×10^{-06} (figure 6.68). Additionally, its **sample error** is, on average, 2.2×10^{-03}, e.g. double than that of $\mathcal{C}_1$. Its variation is, also, twice as much.

The *variation* of **sample error**, among the entity networks, *increases* as the number of training examples increases, but, even so, it is still very low compared to that of the single FFNN.

The single FFNN, on the other hand, yield a *much higher* **sample error** than that of the entity networks. $\mathcal{N}_1$ has the lowest **sample error**, among the single FFNN, with 2.9×10^{-03} (see table 6.16). $\mathcal{N}_2$, $\mathcal{N}_3$ and $\mathcal{N}_4$ follow with 4.2×10^{-03}, 8.6×10^{-03} and 15.6×10^{-03}, respectively.

	$\mathcal{C}_1$		$\mathcal{C}_{1,big}$		$\mathcal{C}_2$		$\mathcal{C}_3$	
	$\mathcal{C}_1$ (12,625)		$\mathcal{C}_1$ (20,010 weights)		$\mathcal{C}_2$ (12,340 weights)		$\mathcal{C}_3$ (12,340 weights)	
$\times 10^{-03}$	MEAN	STD.DEV	MEAN	STD.DEV	MEAN	STD.DEV	MEAN	STD.DEV
lowest	0.144	0.124	0.094	0.050	0.054	0.033	0.04	0.033
average	0.956	0.755	0.784	0.668	1.523	1.317	2.163	1.443
highest	3.018	2.728	1.722	2.016	3.693	3.615	4.837	3.267

Table 6.15: CONSDIM, **sample error** statistics for the entities

	$\mathcal{N}_1$, STANDARD (12,525 weights)		$\mathcal{N}_2$ (20,040 weights)		$\mathcal{N}_3$ (25,050 weights)		$\mathcal{N}_4$ (30,060 weights)	
$\times 10^{-03}$	MEAN	STD.DEV	MEAN	STD.DEV	MEAN	STD.DEV	MEAN	STD.DEV
lowest	0.005	0.024	0.120	0.332	0.687	0.813	2.845	2.920
average	2.917	3.812	4.208	4.332	8.581	13.170	15.608	23.590
highest	6.134	8.609	7.902	8.403	24.896	74.533	25.888	44.783

Table 6.16: CONSDIM, **sample error** statistics for single FFNN

Visual inspection of the scatter plots (figures 6.65, 6.66, 6.67, 6.68, 6.69, 6.70 and 6.71 for the networks $\mathcal{C}_1$, $\mathcal{C}_{1,big}$, $\mathcal{C}_2$, $\mathcal{C}_3$, $\mathcal{N}_1$, $\mathcal{N}_2$, $\mathcal{N}_3$ and $\mathcal{N}_4$, respectively) reveals that the **sample error** of the single FFNN is too high after 100 training vectors, whereas the **sample error** of the entities is much lower. Adding more weights to the network does not remedy the situation – on the contrary, the outliers of $\mathcal{N}_3$ and, in particular, of $\mathcal{N}_4$ increase dramatically. Notice also how small is the difference between the *mean* and *standard deviation* of the **sample errors** of $\mathcal{C}_1$ and $\mathcal{C}_{1,big}$ ($\mathcal{C}_{1,big}$ having almost twice as many weights as $\mathcal{C}_1$, and both networks belonging to the same entity class) networks.

The statistical significance tests comparing the *means* of the **sample error** of the entities and the single FFNN (the *t-test*) indicate that for less than 120 training vectors, the H_2 hypothesis (e.g. that the mean of the entities' error is significantly lower than that of the single FFNN) is true by 70 % to 90 % (with the exception of $\mathcal{C}_3$ when compared to $\mathcal{N}_1$ – their means do not seem to differ significantly, see table 6.17).

When the number of training vectors exceeds 120, the H_2 hypothesis is true by almost 100 % (see table 6.18).

%, %, %	$\mathcal{N}_1$	$\mathcal{N}_2$	$\mathcal{N}_3$	$\mathcal{N}_4$
$\mathcal{C}_1$	18, 18, 64	18, 0, 82	18, 0, 82	27, 0, 73
$\mathcal{C}_{1,big}$	18, 18, 64	18, 0, 82	9, 0, 91	27, 0, 73
$\mathcal{C}_2$	18, 27, 55	9, 0, 91	9, 0, 91	27, 0, 73
$\mathcal{C}_3$	64, 9, 27	36, 0, 64	27, 0, 73	27, 0, 73

Table 6.17: CONSDIM, statistical significance (*t-test*) of the **sample error** results for 10 to 120 training vectors

%, %, %	$\mathcal{N}_1$	$\mathcal{N}_2$	$\mathcal{N}_3$	$\mathcal{N}_4$
$\mathcal{C}_1$	0, 0, 100	0, 0, 100	0, 0, 100	0, 0, 100
$\mathcal{C}_{1,big}$	0, 0, 100	0, 0, 100	0, 0, 100	0, 0, 100
$\mathcal{C}_2$	22, 0, 78	0, 0, 100	0, 0, 100	0, 0, 100
$\mathcal{C}_3$	44, 0, 56	11, 0, 89	0, 0, 100	0, 0, 100

Table 6.18: CONSDIM, statistical significance (*t-test*) of the **sample error** results for more than 120 training vectors

%, %, %	$\mathcal{N}_1$	$\mathcal{N}_2$	$\mathcal{N}_3$	$\mathcal{N}_4$
$\mathcal{C}_1$	9, 18, 73	0, 0, 100	9, 0, 91	0, 0, 100
$\mathcal{C}_{1,big}$	18, 9, 73	0, 0, 100	0, 0, 100	0, 0, 100
$\mathcal{C}_2$	9, 9, 82	9, 0, 91	0, 0, 100	0, 0, 100
$\mathcal{C}_3$	36, 0, 64	0, 0, 100	0, 0, 100	0, 0, 100

Table 6.19: CONSDIM, statistical significance (*F-test*) of the **sample error** results for 10 to 120 training vectors

%, %, %	$\mathcal{N}_1$	$\mathcal{N}_2$	$\mathcal{N}_3$	$\mathcal{N}_4$
$\mathcal{C}_1$	0, 0, 100	0, 0, 100	0, 0, 100	0, 0, 100
$\mathcal{C}_{1,big}$	0, 0, 100	0, 0, 100	0, 0, 100	0, 0, 100
$\mathcal{C}_2$	11, 0, 89	0, 0, 100	0, 0, 100	0, 0, 100
$\mathcal{C}_3$	0, 0, 100	0, 0, 100	0, 0, 100	0, 0, 100

Table 6.20: CONSDIM, statistical significance (*F-test*) of the **sample error** results for more than 120 training vectors

As far as the *variation* of the **sample error** is concerned, the relevant statistical significance tests (the *F-test*, see tables 6.19 and 6.20) indicate that for *any* number of training vectors (both below and over 120) the H_2 hypothesis is always true. It approaches 100 % for more than 120 training vectors. *This proves that the entities' training is more consistent than that of single* FFNN *and that the consistency of the latter deteriorates with the increasing number of training examples.*

The following figures contain plots of the *minimum*, *maximum*, *mean* and *standard deviation* of **sample error** against the number of **training vectors**.

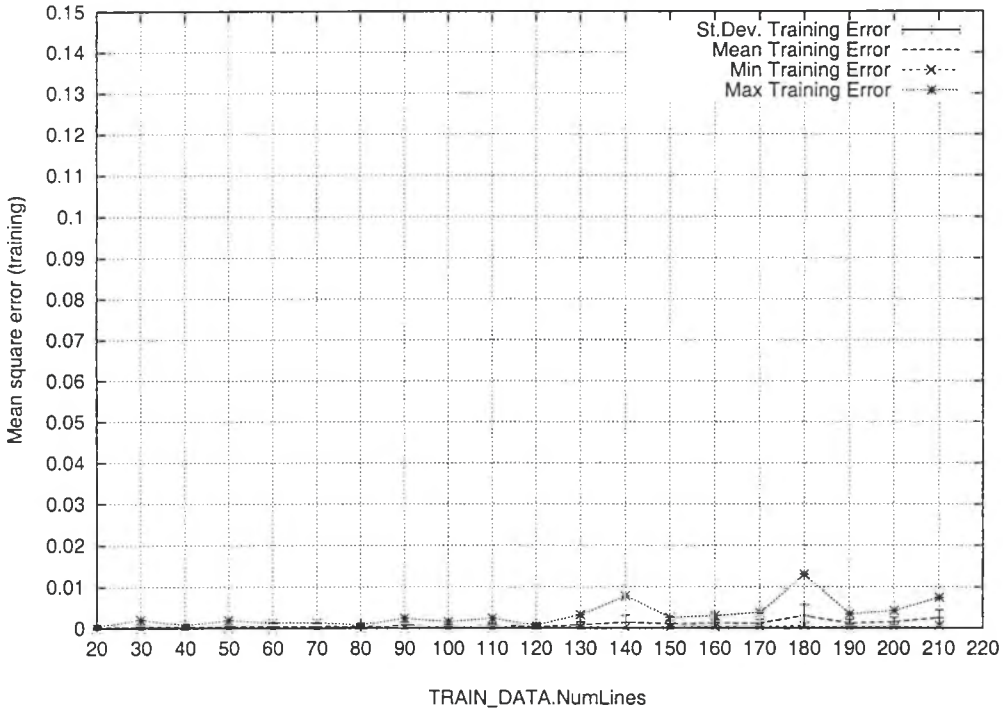

Figure 6.57: $\mathcal{C}_1$, **sample error** against number of **training vectors**

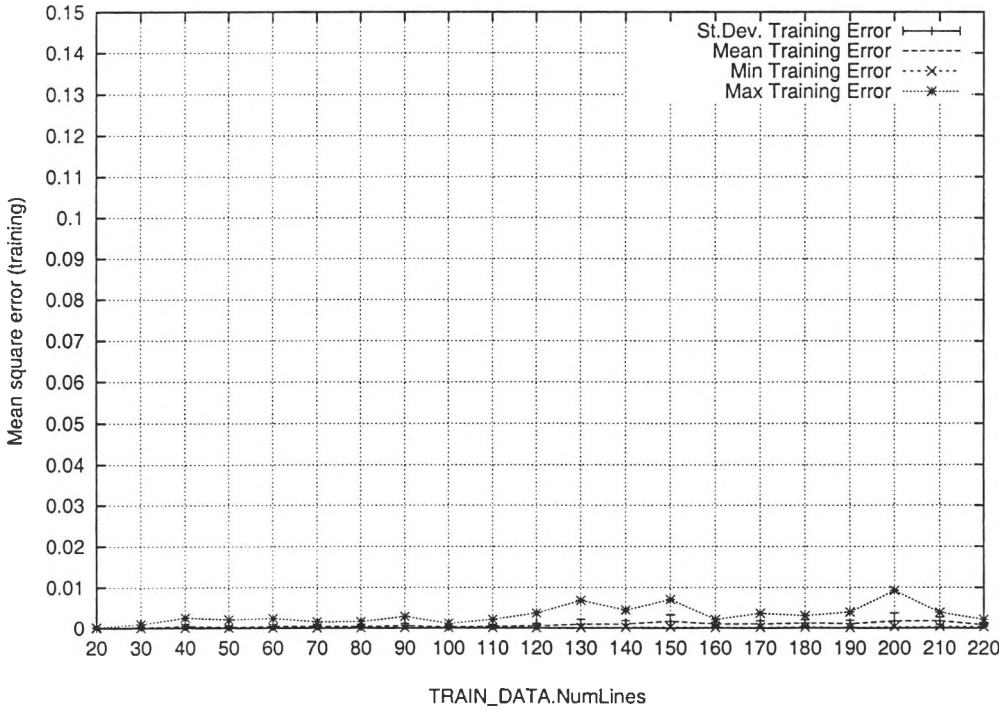

Figure 6.58: $\mathcal{C}_{1,big}$, **sample error** against number of **training vectors**

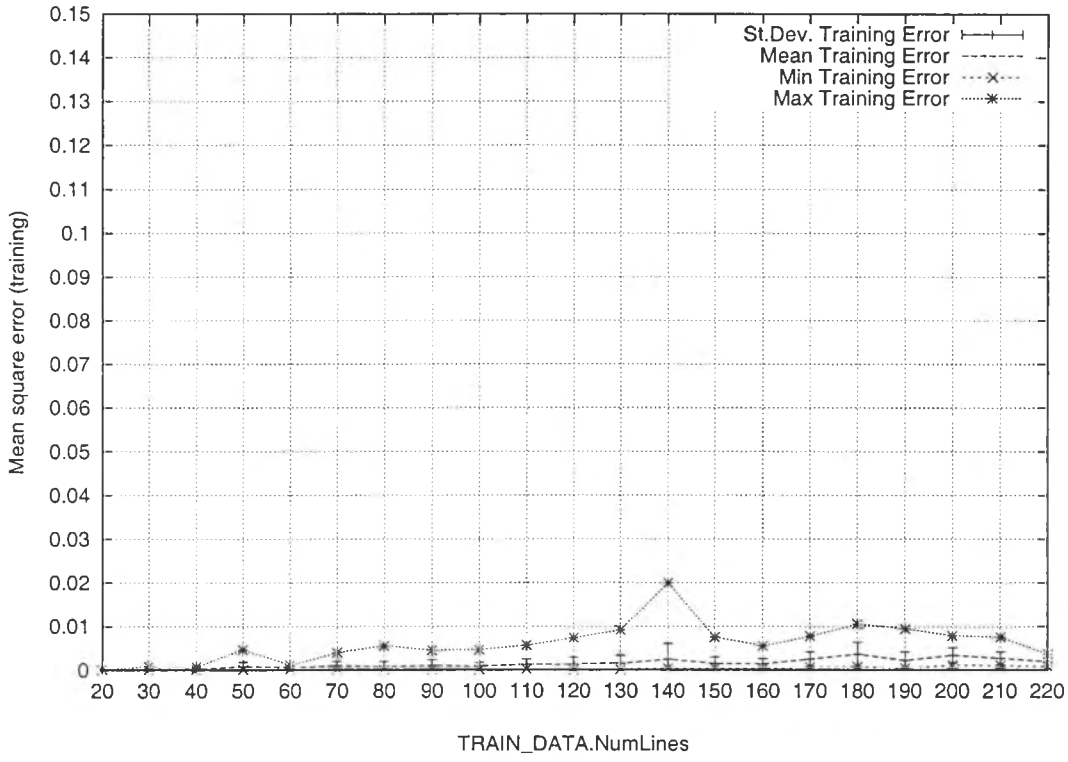

Figure 6.59: $\mathcal{C}_2$, **sample error** against number of **training vectors**

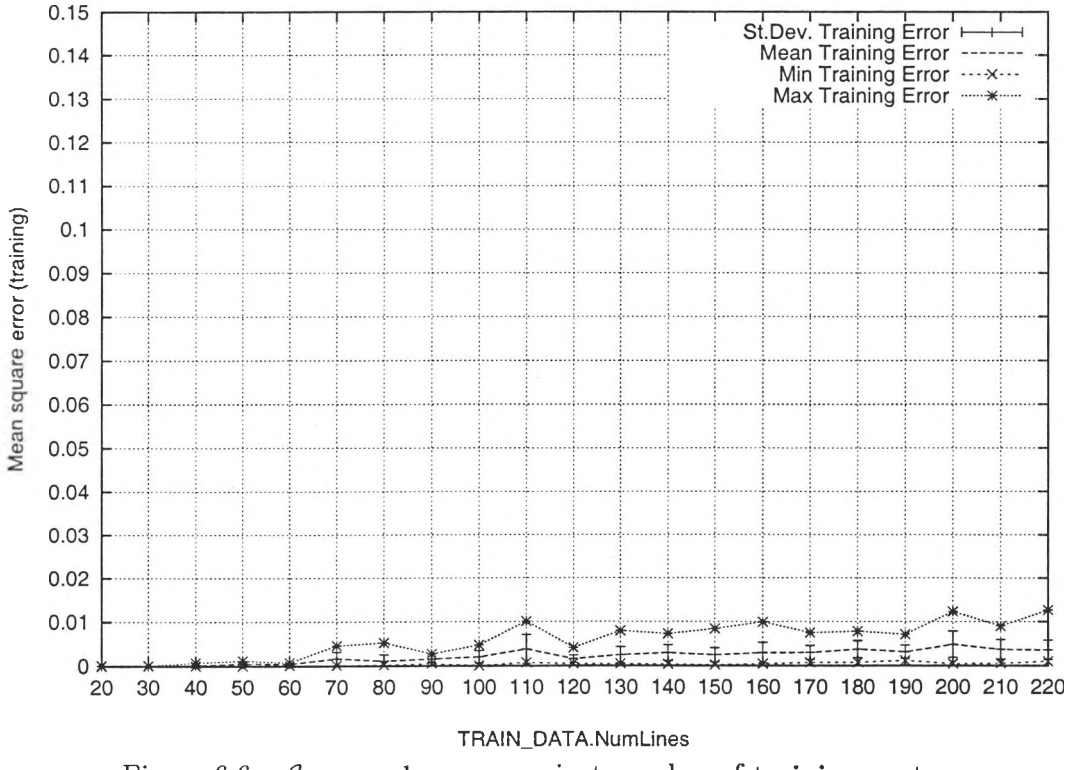

Figure 6.60: $\mathcal{C}_3$, **sample error** against number of **training vectors**

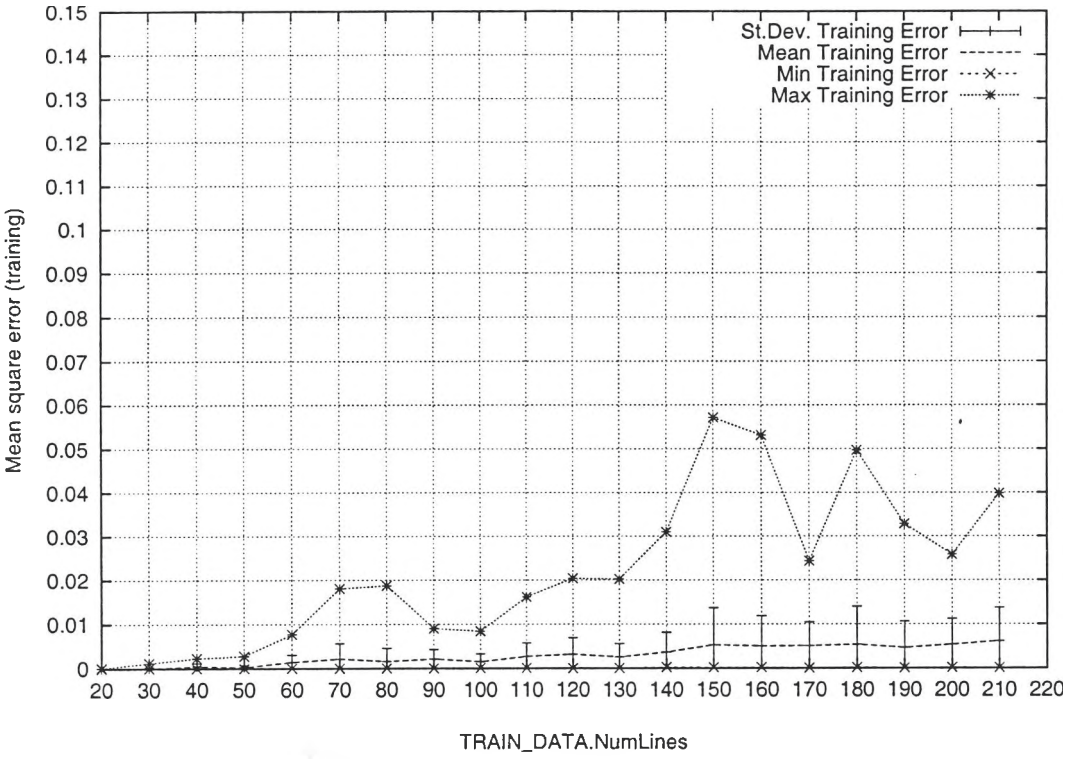

Figure 6.61: $\mathcal{N}_1$, **sample error** against number of **training vectors**

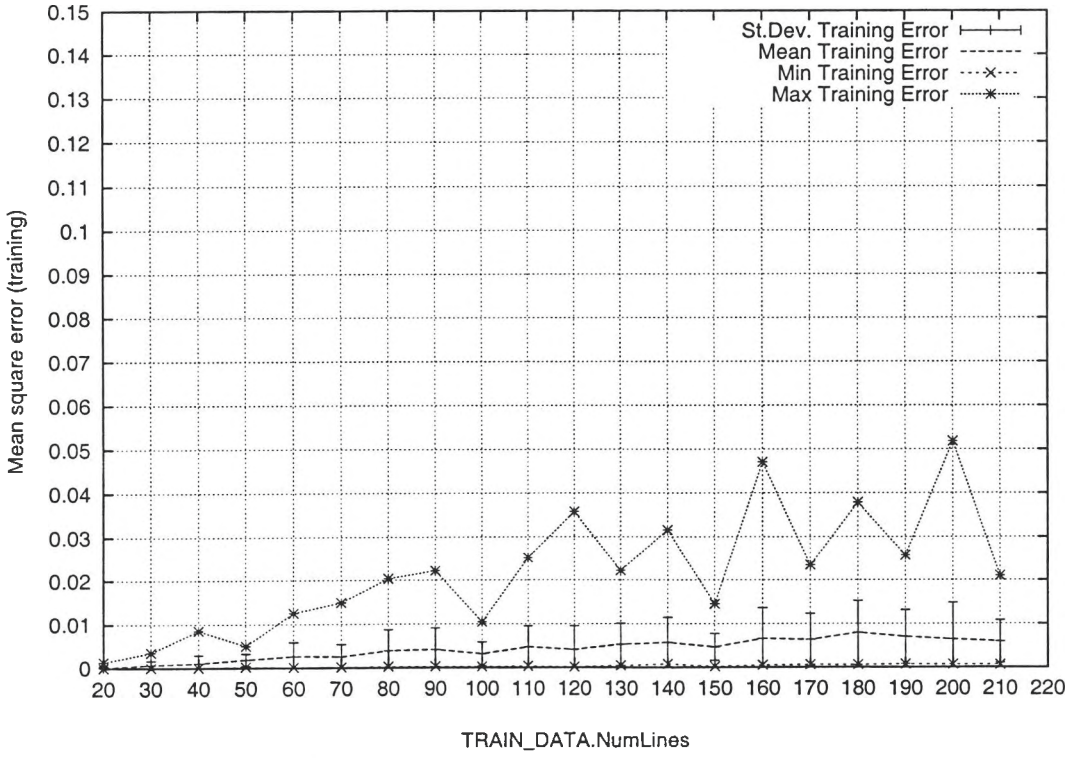

Figure 6.62: $\mathcal{N}_2$, **sample error** against number of **training vectors**

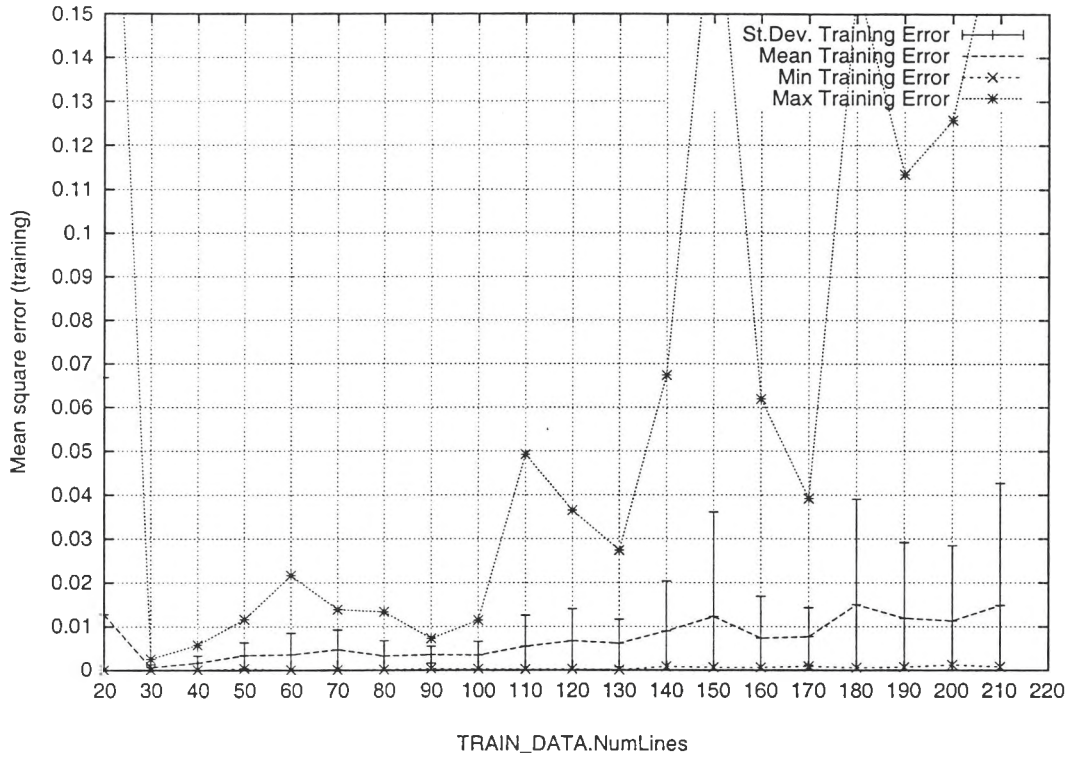

Figure 6.63: $\mathcal{N}_3$, **sample error** against number of **training vectors**

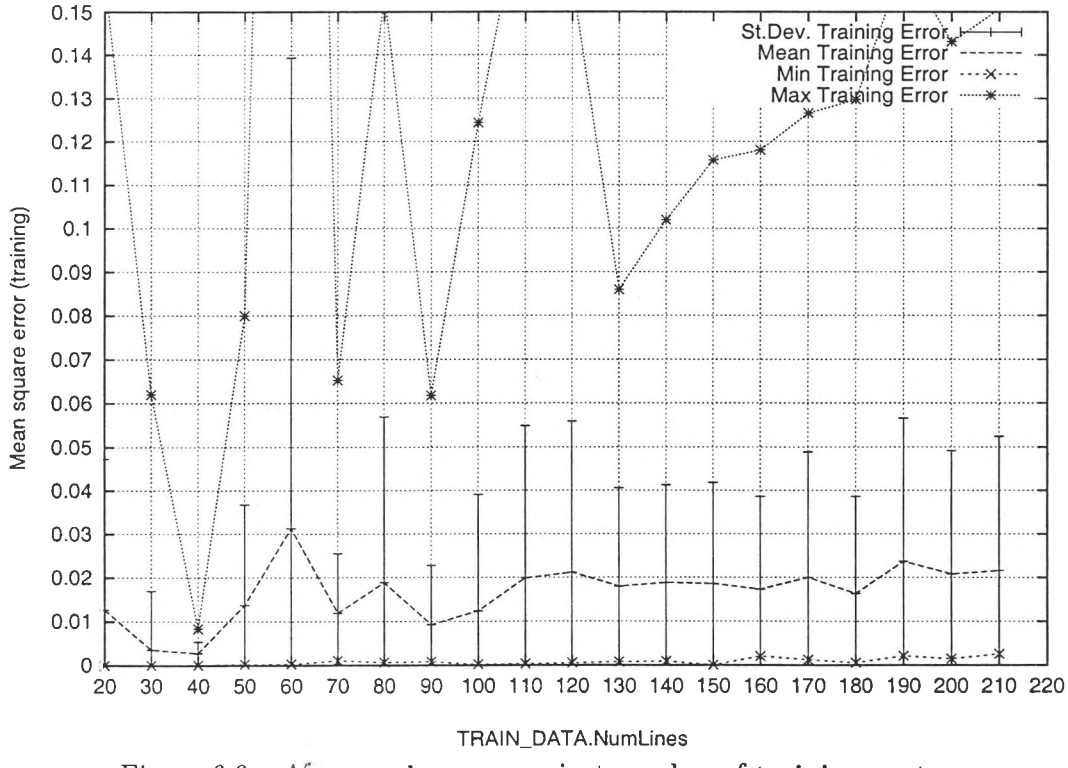

Figure 6.64: $\mathcal{N}_4$, **sample error** against number of **training vectors**

The following figures contain scatter plots of the **sample error** for all evaluated networks as the number of **training vectors** increases.

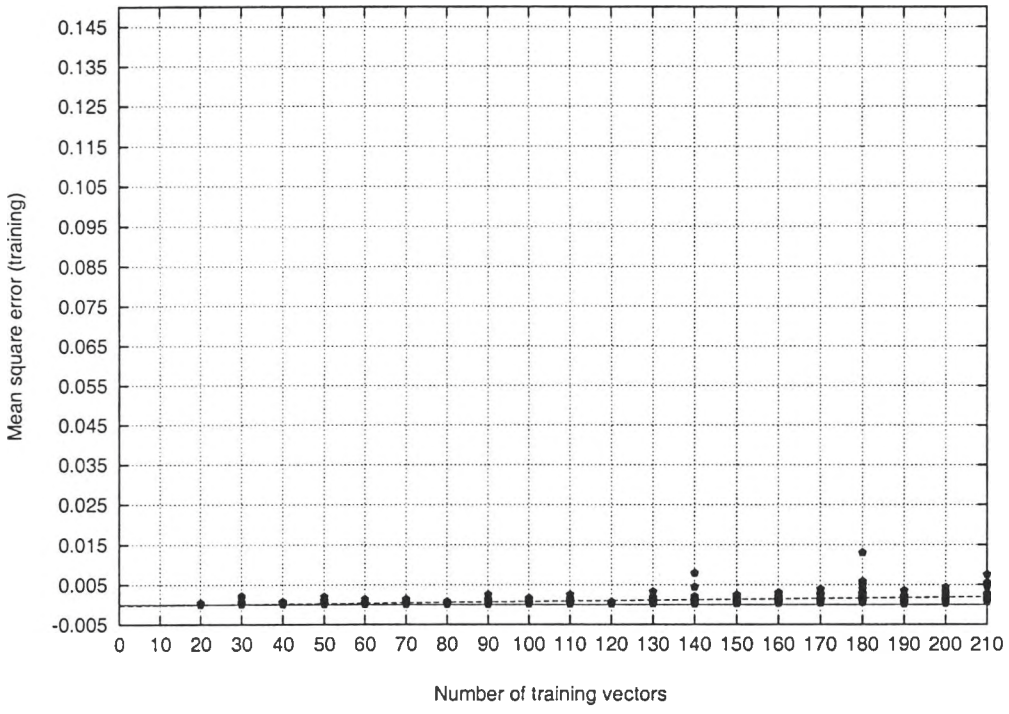

Figure 6.65: C_1, **sample error** against number of **training vectors**

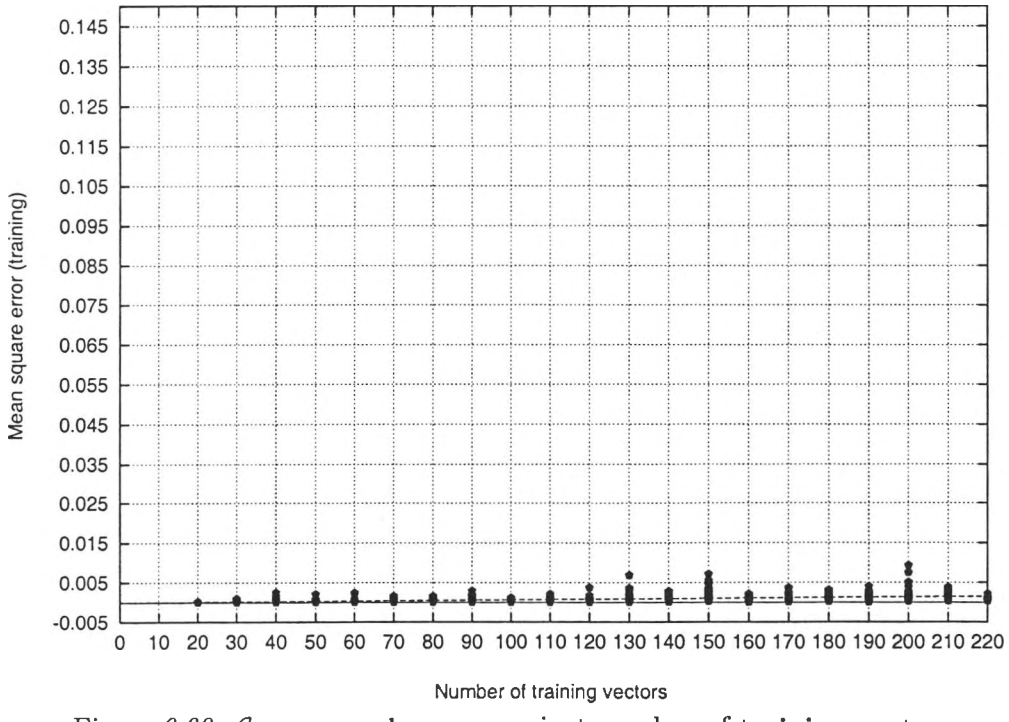

Figure 6.66: $C_{1,big}$, **sample error** against number of **training vectors**

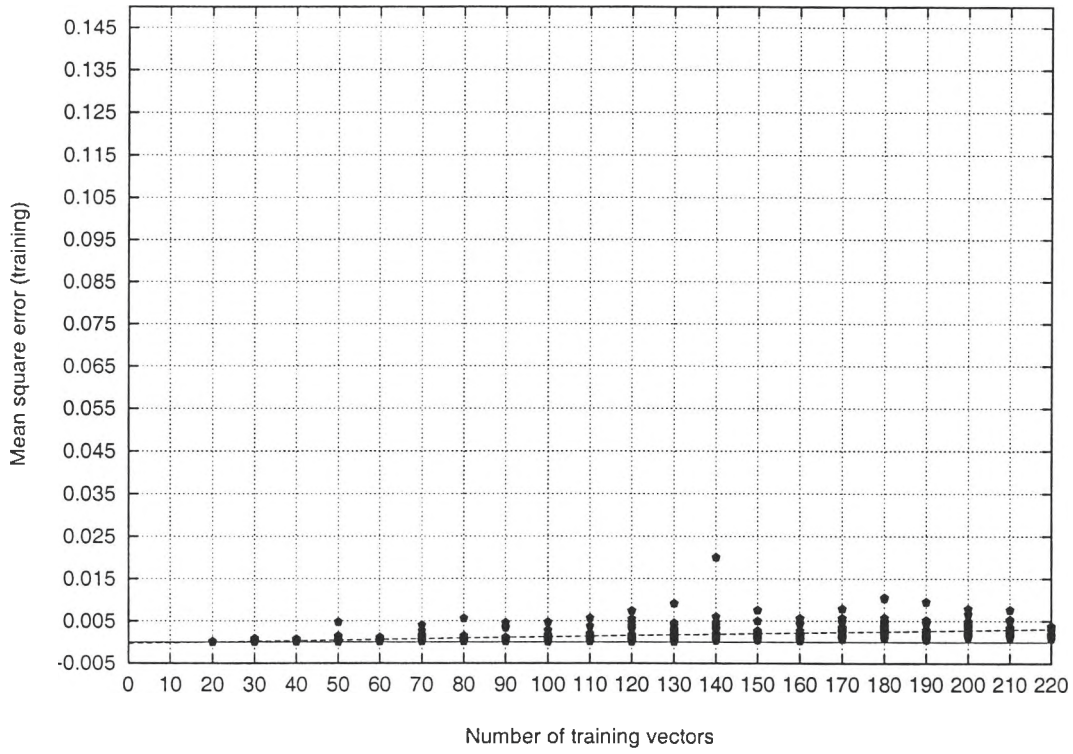

Figure 6.67: C_2, **sample error** against number of **training vectors**

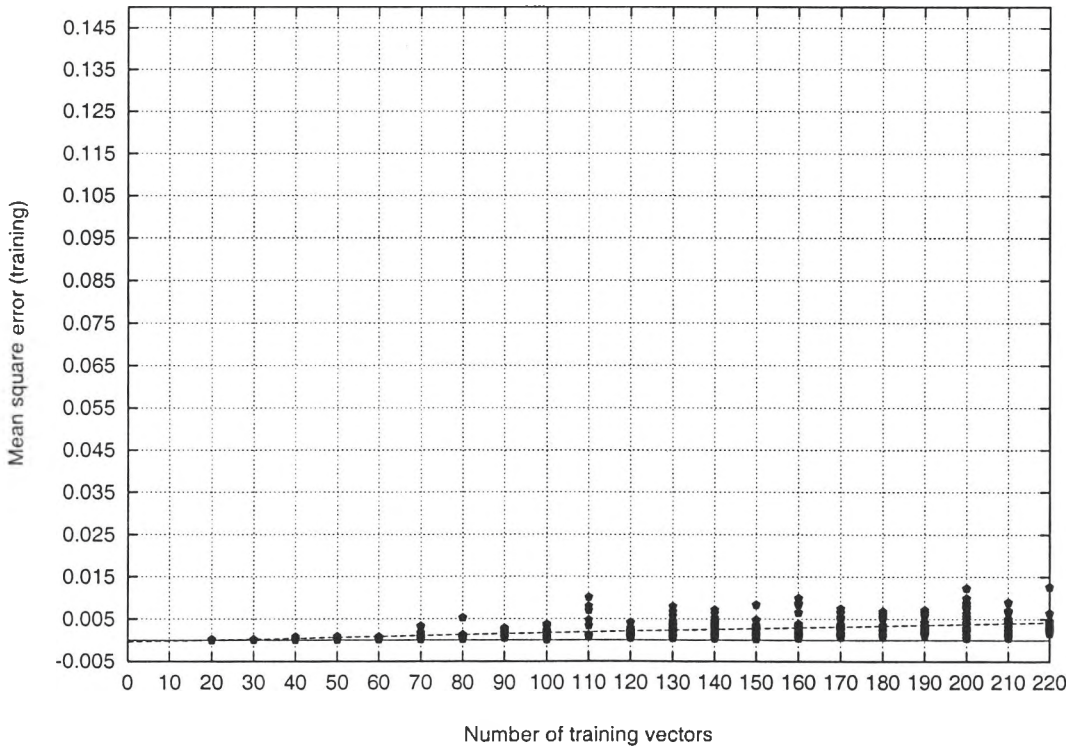

Figure 6.68: C_3, **sample error** against number of **training vectors**

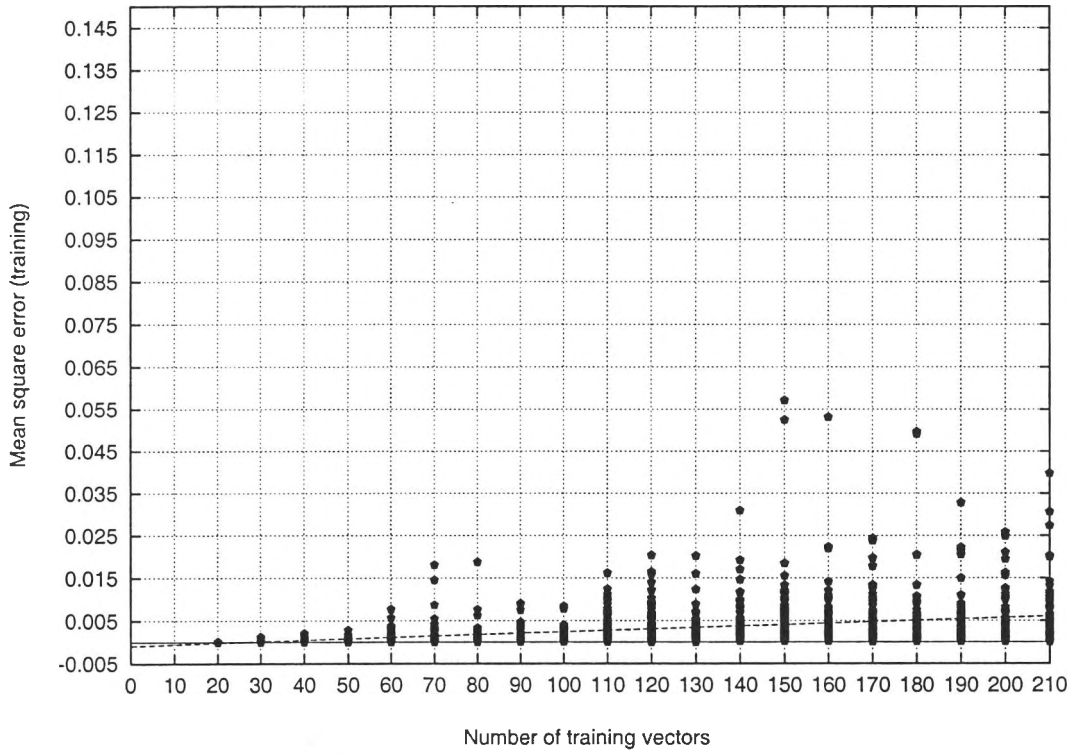

Figure 6.69: $\mathcal{N}_1$, **sample error** against number of **training vectors**

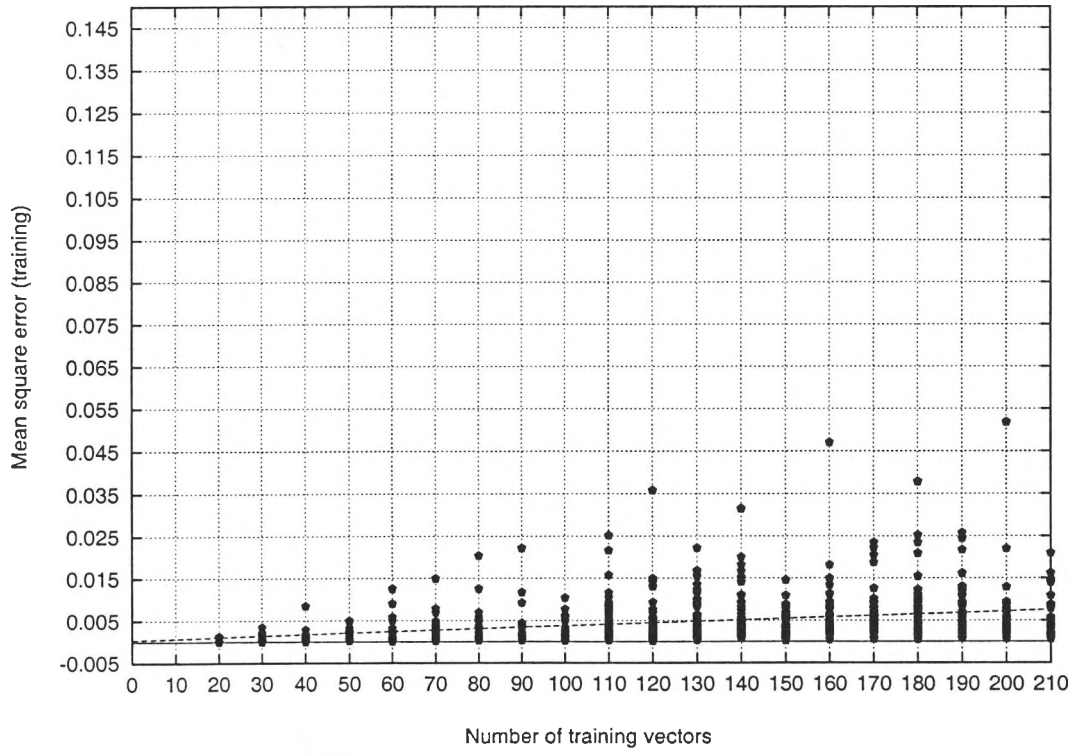

Figure 6.70: $\mathcal{N}_2$, **sample error** against number of **training vectors**

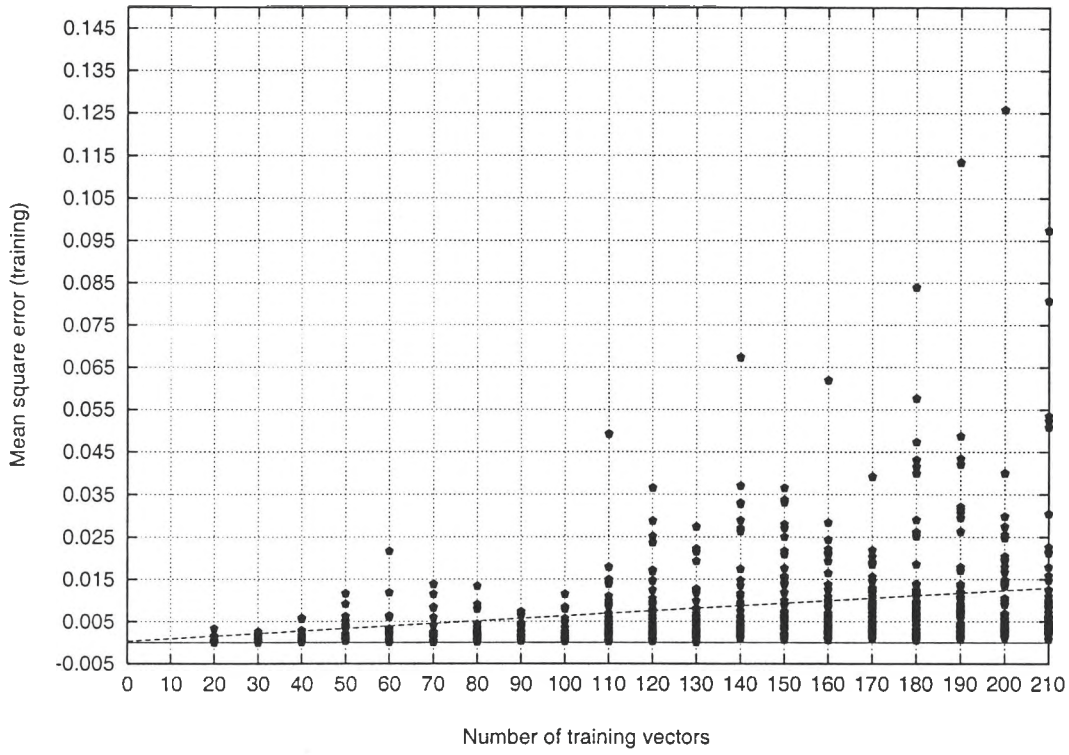

Figure 6.71: $\mathcal{N}_3$, **sample error** against number of **training vectors**

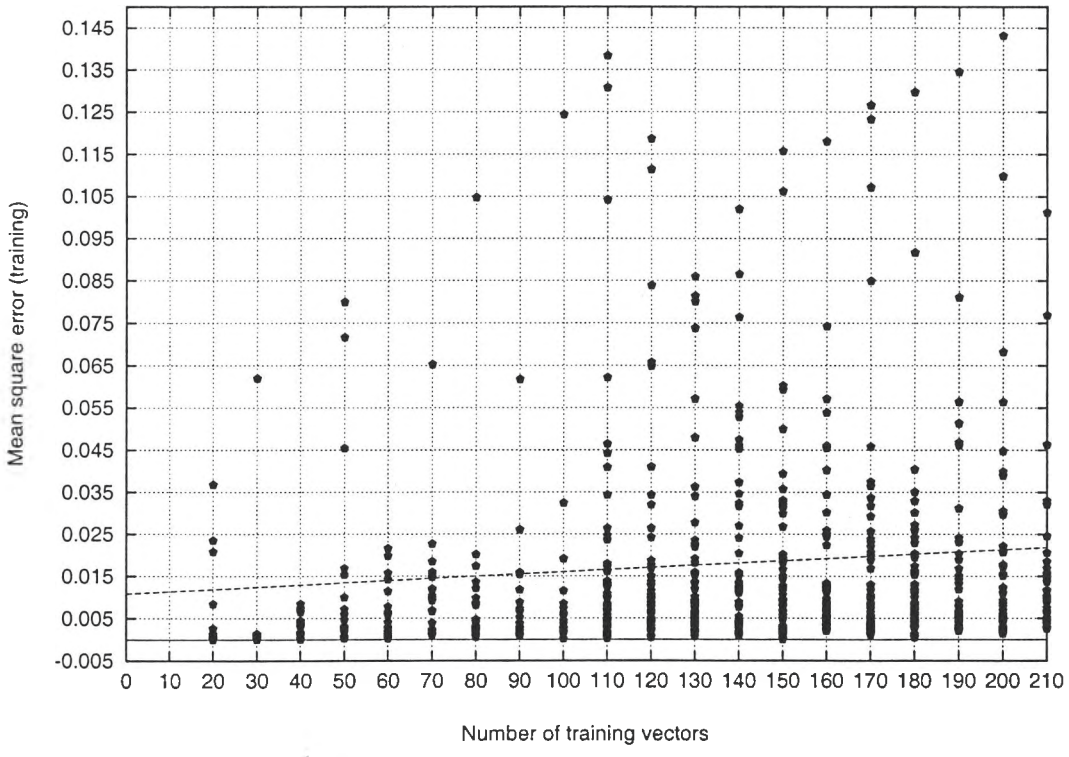

Figure 6.72: $\mathcal{N}_4$, **sample error** against number of **training vectors**

6.3.7.4 CONSDIM: approximation error results

The **approximation error** results for the entity networks are all quite satisfactory with, perhaps, the exception of $\mathcal{C}_3$ which yields a *one and a half times higher* error than that of $\mathcal{C}_1$ and $\mathcal{C}_{1,big}$. In particular, the best generalisation ability is exhibited by $\mathcal{C}_{1,big}$ with an **approximation error** of 64×10^{-03}, on average (see table 6.21). $\mathcal{C}_1$ and $\mathcal{C}_2$ follow with the slightly higher errors of 70×10^{-03} and 77×10^{-03}, respectively. $\mathcal{C}_3$ gives an error of 104×10^{-03} – a relatively high value compared to that of the other entity networks but, still, lower than that of any of the single FFNN.

	$\mathcal{C}_1$		$\mathcal{C}_{1,big}$		$\mathcal{C}_2$		$\mathcal{C}_3$	
	$\mathcal{C}_1$ entity		$\mathcal{C}_1$ (66% more weights)		$\mathcal{C}_2$ entity		$\mathcal{C}_3$ entity	
$\times 10^{-03}$	MEAN	STD.DEV	MEAN	STD.DEV	MEAN	STD.DEV	MEAN	STD.DEV
lowest	57.635	3.503	54.579	5.846	61.900	5.914	93.244	5.841
average	69.687	10.368	63.825	9.083	76.579	15.310	104.20	8.823
highest	106.18	27.649	79.008	15.257	91.036	26.143	134.54	18.34

Table 6.21: CONSDIM, **approximation error** statistics for the entities

	$\mathcal{N}_1$ STANDARD (12,525 weights)		$\mathcal{N}_2$ (20,040 weights)		$\mathcal{N}_3$ (25,050 weights)		$\mathcal{N}_4$ (30,060 weights)	
$\times 10^{-03}$	MEAN	STD.DEV	MEAN	STD.DEV	MEAN	STD.DEV	MEAN	STD.DEV
lowest	116.14	14.027	110.87	2.882	107.29	12.002	113.09	20.48
average	123.97	19.056	117.48	13.933	118.53	19.645	129.28	33.35
highest	142.72	24.310	125.55	23.049	125.30	34.933	139.26	78.06

Table 6.22: CONSDIM, **approximation error** statistics for single FFNN

The **approximation error** plots of the entities (see, for example, the *minimum*, *maximum*, *mean* and *standard deviation* plots of error in figures 6.73, 6.74, 6.75 and 6.76 or the scatter plots in figures 6.81, 6.82, 6.83 and 6.84 for $\mathcal{C}_1$, $\mathcal{C}_{1,big}$, $\mathcal{C}_2$ and $\mathcal{C}_3$, respectively) indicate that, as the number of training examples increases, the **error** value follows a *parabolic locus* with its minimum (e.g. the number of training vectors required for optimum training with 1,000 iterations) at about 100.

One may also observe that $\mathcal{C}_1$, $\mathcal{C}_{1,big}$ and $\mathcal{C}_3$ networks have a much more consistent generalisation behaviour than $\mathcal{C}_2$. This is indicated by the fact that these networks' scatter plots show a *significantly smaller number of outliers*. Additionally, the *standard deviation* of $\mathcal{C}_2$ is 15×10^{-03} compared to the values of 10×10^{-03}, 9×10^{-03} and 8.8×10^{-03} for $\mathcal{C}_1$, $\mathcal{C}_{1,big}$ and $\mathcal{C}_3$, respectively. The results of the statistical significance *F-test* (see tables 6.25 and 6.26) also confirm this conclusion.

From a first glance, the **approximation error** of all single FFNN networks is, on average, *much higher* than that of the entities. The standard single FFNN network $\mathcal{N}_1$, which has the same number of weights as $\mathcal{C}_1$, yields an **approximation error** of 124×10^{-03} (see table 6.22) – this is twice as much as that of $\mathcal{C}_1$.

%, %, %	$\mathcal{N}_1$	$\mathcal{N}_2$	$\mathcal{N}_3$	$\mathcal{N}_4$
$\mathcal{C}_1$	9, 0, 91	0, 0, 100	9, 0, 91	9, 0, 91
$\mathcal{C}_{1,big}$	0, 0, 100	0, 0, 100	0, 0, 100	0, 0, 100
$\mathcal{C}_2$	0, 0, 100	0, 0, 100	0, 0, 100	0, 0, 100
$\mathcal{C}_3$	18, 0, 82	36, 9, 55	27, 18, 55	27, 0, 73

Table 6.23: CONSDIM, statistical significance (*t-test*) of the **approximation error** results for 10 to 120 training vectors

%, %, %	$\mathcal{N}_1$	$\mathcal{N}_2$	$\mathcal{N}_3$	$\mathcal{N}_4$
$\mathcal{C}_1$	0, 0, 100	0, 0, 100	0, 0, 100	0, 0, 100
$\mathcal{C}_{1,big}$	0, 0, 100	0, 0, 100	0, 0, 100	0, 0, 100
$\mathcal{C}_2$	0, 0, 100	0, 0, 100	0, 0, 100	0, 0, 100
$\mathcal{C}_3$	0, 0, 100	0, 0, 100	0, 0, 100	0, 0, 100

Table 6.24: CONSDIM, statistical significance (*t-test*) of the **approximation error** results for more than 120 training vectors

The generalisation ability of the single FFNN remains largely the same as the number of their weights increases: $\mathcal{N}_2$ and $\mathcal{N}_3$ have an error of 118×10^{-03} and $\mathcal{N}_4$ has an error of 129×10^{-03}. However, the *variation* of error – indicated by the presence of a large number of outliers in the scatter plots in figures 6.69, 6.86, 6.87 and 6.88 (for the single FFNN $\mathcal{N}_1$, $\mathcal{N}_2$, $\mathcal{N}_3$ and $\mathcal{N}_4$, respectively), and by the high values of *standard deviation* – is such that makes the single FFNN candidates, with the exception of $\mathcal{N}_2$, very *inconsistent*.

Another indication of this huge variation in the **approximation error** of the single FFNN is the large difference between the *minimum* and *maximum* error values corresponding to the same number of **training vectors**. This can be seen in the plots of figures 6.77, 6.78, 6.79 and 6.80 for $\mathcal{N}_1$, $\mathcal{N}_2$, $\mathcal{N}_3$ and $\mathcal{N}_4$, respectively.

The results of the statistical significance test comparing the differences between the *means* of the **approximation error** of the entities and single FFNN (the *t-test*) indicate that for a number of training vectors below 120 (see table 6.23), the H_2 hypothesis is clearly true (by 100 %) for the entity networks $\mathcal{C}_1$, $\mathcal{C}_{1,big}$ and $\mathcal{C}_2$. This percentage is lower for $\mathcal{C}_3$.

When the number of training vectors exceeds 120 (see table 6.24), the superiority of the entities generalisation ability over that of single FFNN is absolute. The acceptance of the H_2 hypothesis is by 100 %.

The statistical significance tests comparing the variances of the different networks (the *F-test*) shows that for less than 120 training vectors, the *variation* of the entities **approximation error** is lower than or, at most comparable to that of the single FFNN (see table 6.25).

%, %, %	$\mathcal{N}_1$	$\mathcal{N}_2$	$\mathcal{N}_3$	$\mathcal{N}_4$
$\mathcal{C}_1$	27, 0, 73	27, 18, 55	45, 0, 55	9, 0, 91
$\mathcal{C}_{1,big}$	0, 0, 100	27, 9, 64	27, 0, 73	0, 0, 100
$\mathcal{C}_2$	27, 0, 73	64, 18, 18	36, 0, 64	9, 0, 91
$\mathcal{C}_3$	18, 0, 82	27, 9, 64	18, 0, 82	0, 0, 100

Table 6.25: CONSDIM, statistical significance (*F-test*) of the **approximation error** results for 10 to 120 training vectors

%, %, %	$\mathcal{N}_1$	$\mathcal{N}_2$	$\mathcal{N}_3$	$\mathcal{N}_4$
$\mathcal{C}_1$	11, 0, 89	33, 0, 67	0, 0, 100	0, 0, 100
$\mathcal{C}_{1,big}$	0, 0, 100	33, 0, 67	0, 0, 100	0, 0, 100
$\mathcal{C}_2$	44, 11, 45	56, 33, 11	44, 0, 56	44, 0, 56
$\mathcal{C}_3$	0, 0, 100	22, 0, 78	0, 0, 100	0, 0, 100

Table 6.26: CONSDIM, statistical significance (*F-test*) of the **approximation error** results for more than 120 training vectors

For a number of training vectors greater than 120, the acceptance of the H_2 hypothesis is overwhelming, at least when comparing $\mathcal{C}_1$, $\mathcal{C}_{1,big}$ and $\mathcal{C}_3$ with all the other single FFNN (see table 6.26). For $\mathcal{C}_2$, this variation is either equal to that of the singe FFNN or lower.

The following figures contain plots of the *minimum*, *maximum*, *mean* and *standard deviation* of **approximation error** against the number of **training vectors.**

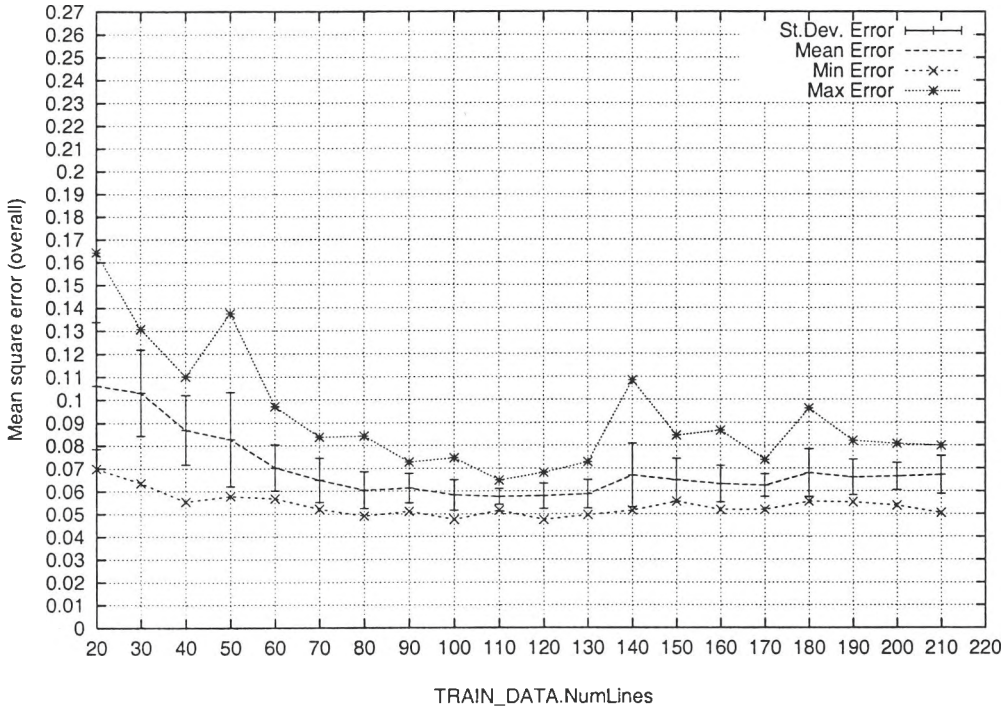

Figure 6.73: C_1, **approximation error** against number of **training vectors**

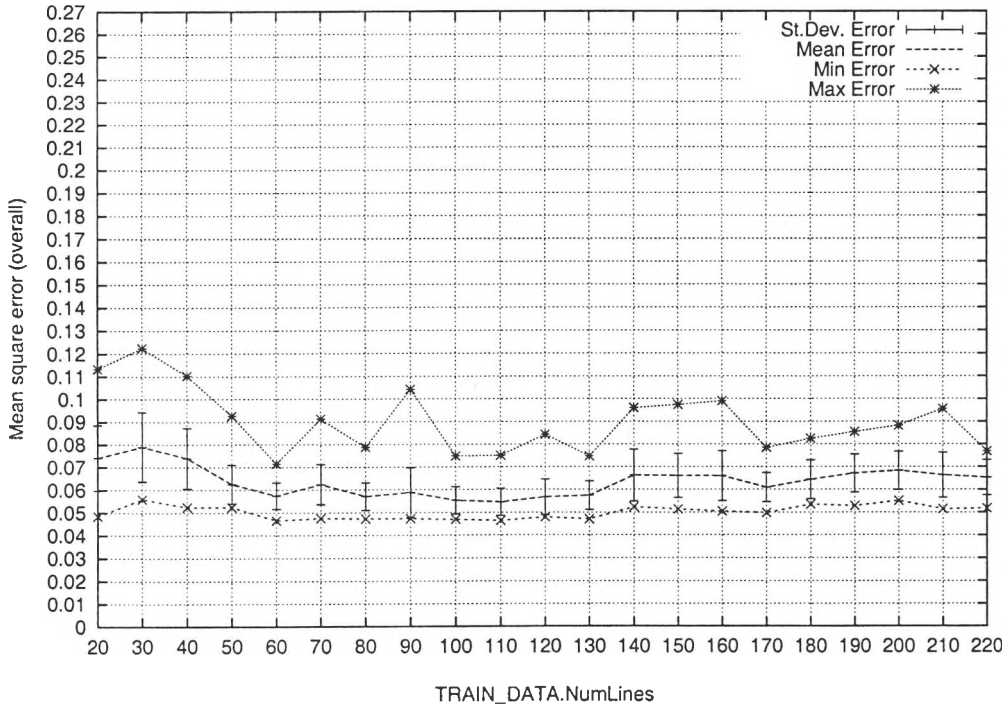

Figure 6.74: $C_{1,big}$, **approximation error** against number of **training vectors**

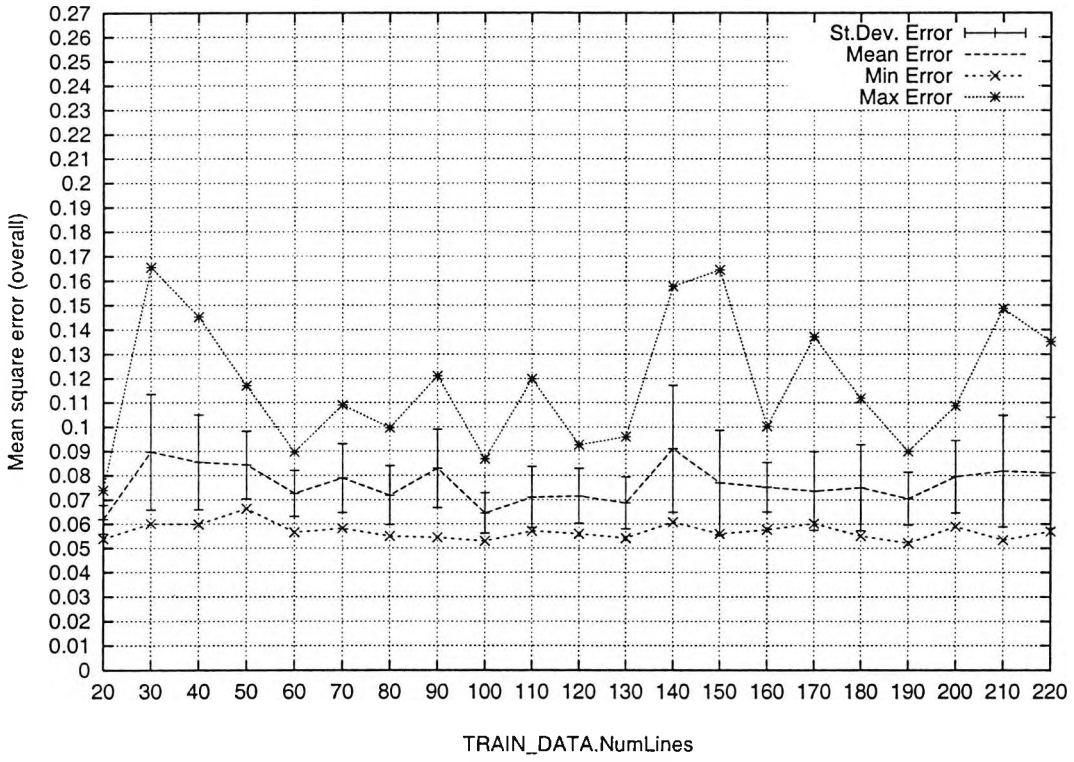

Figure 6.75: $\mathcal{C}_2$, **approximation error** against number of **training vectors**

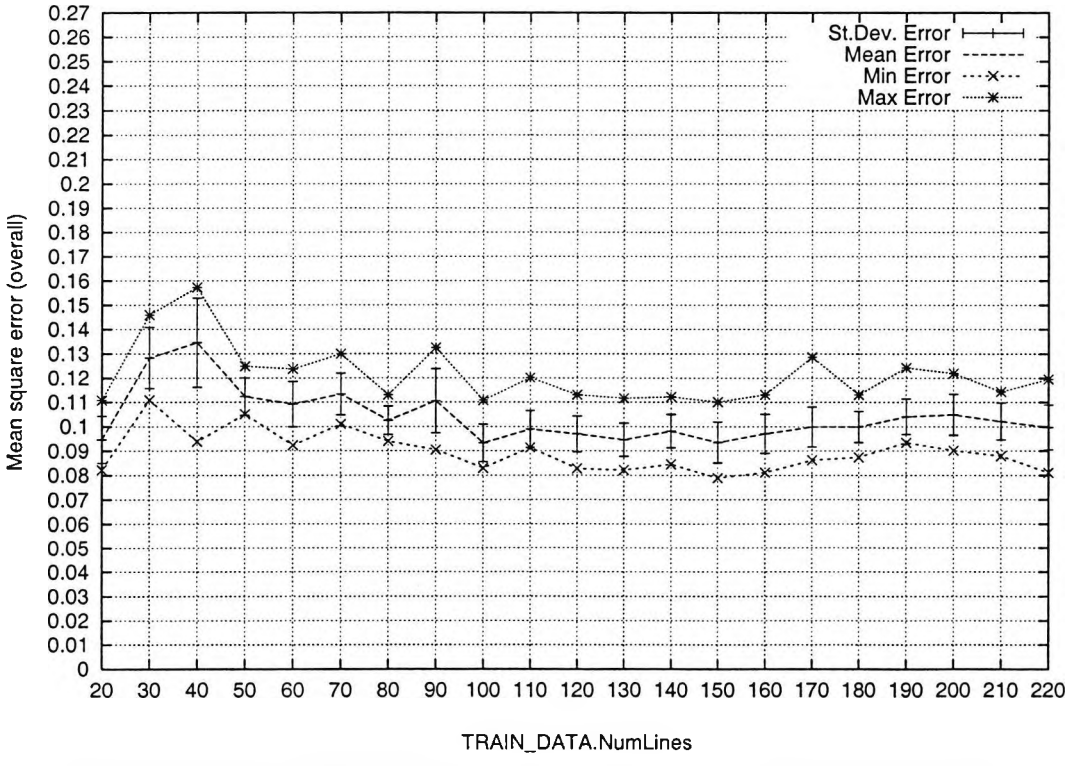

Figure 6.76: $\mathcal{C}_3$, **approximation error** against number of **training vectors**

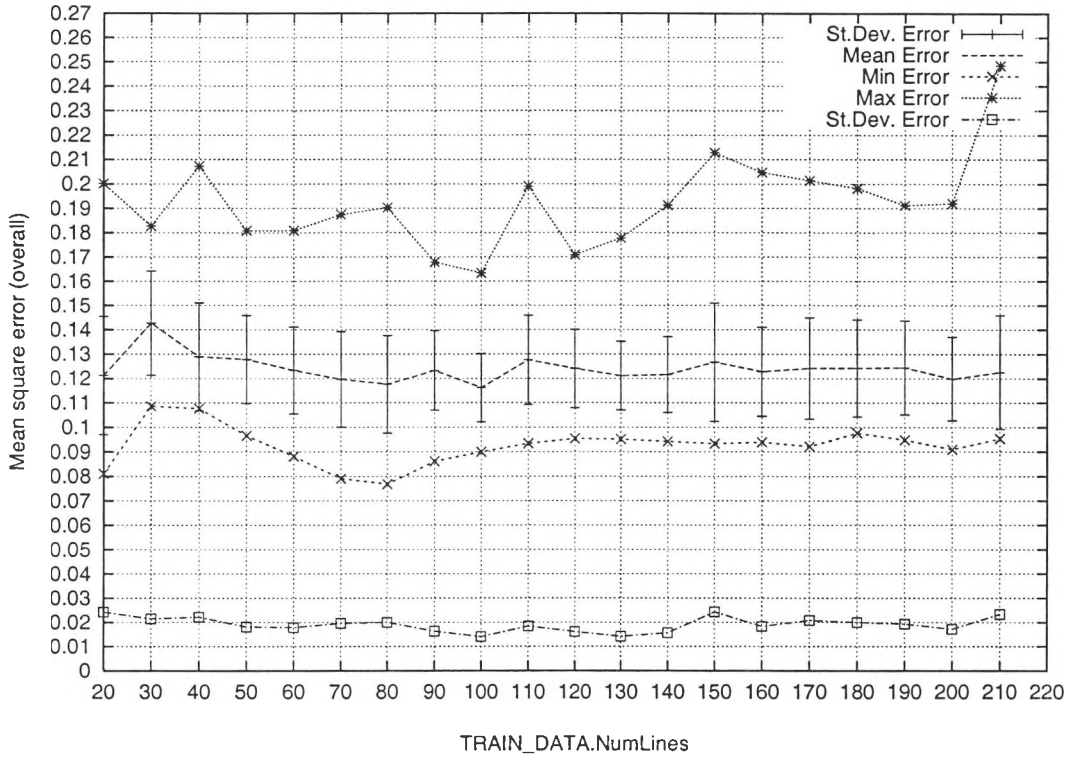

Figure 6.77: $\mathcal{N}_1$, **approximation error** against number of **training vectors**

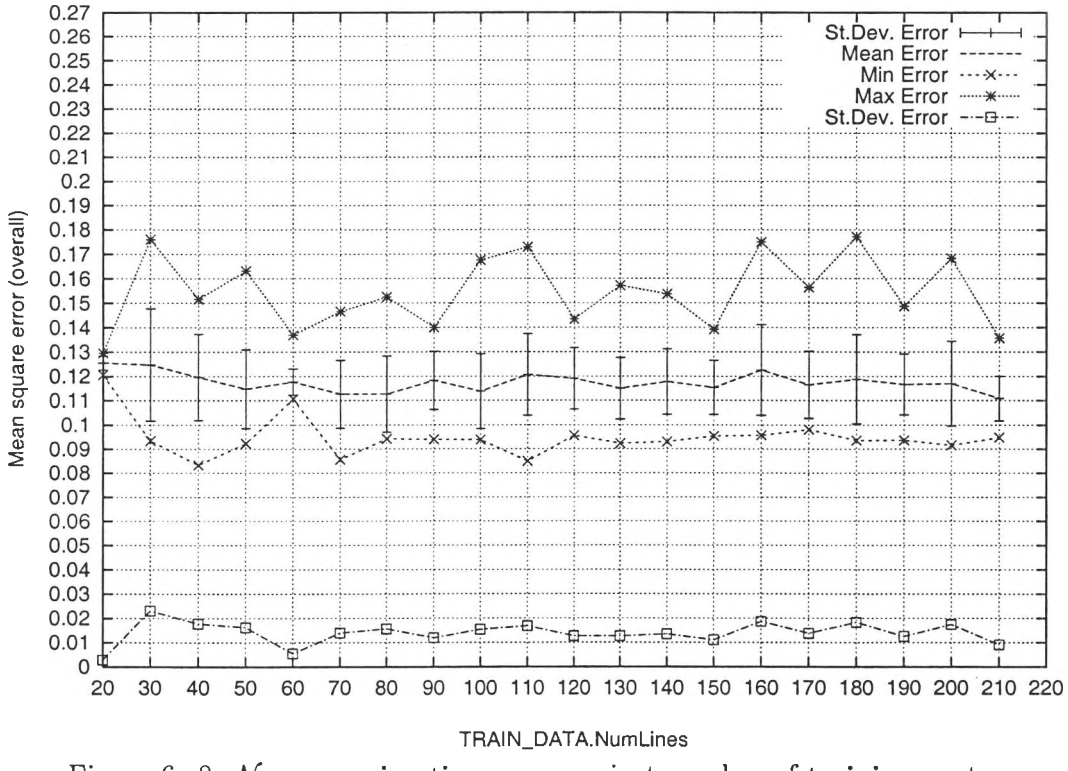

Figure 6.78: $\mathcal{N}_2$, **approximation error** against number of **training vectors**

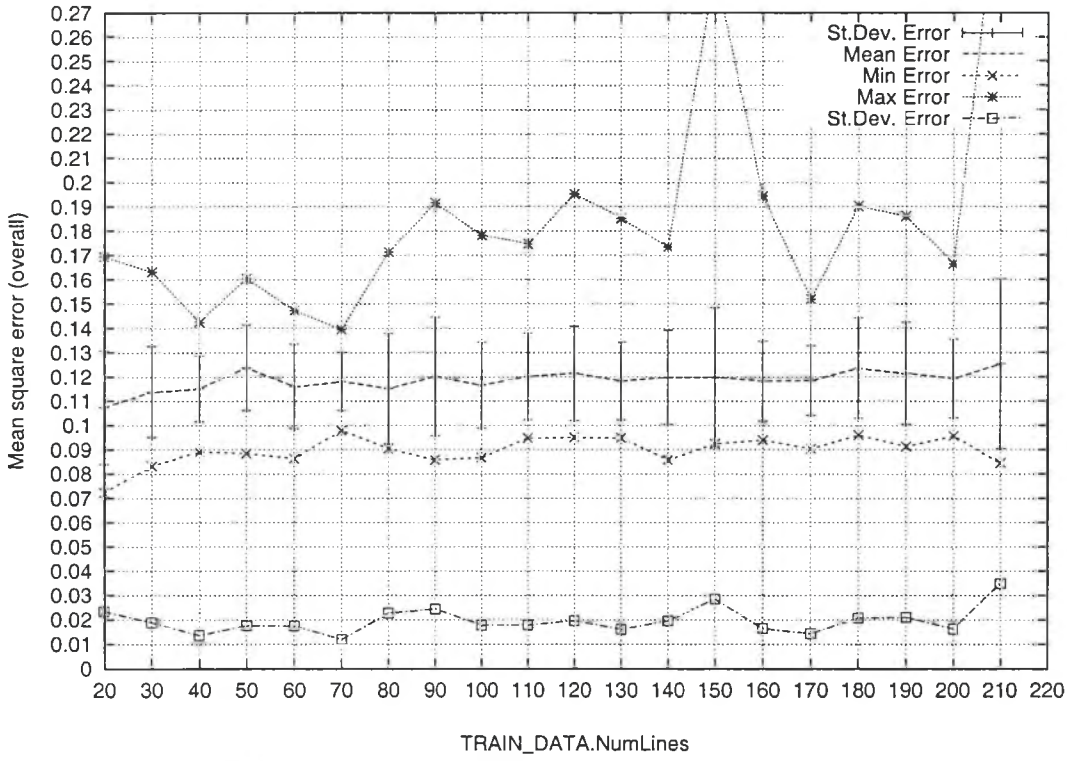

Figure 6.79: $\mathcal{N}_3$, **approximation error** against number of **training vectors**

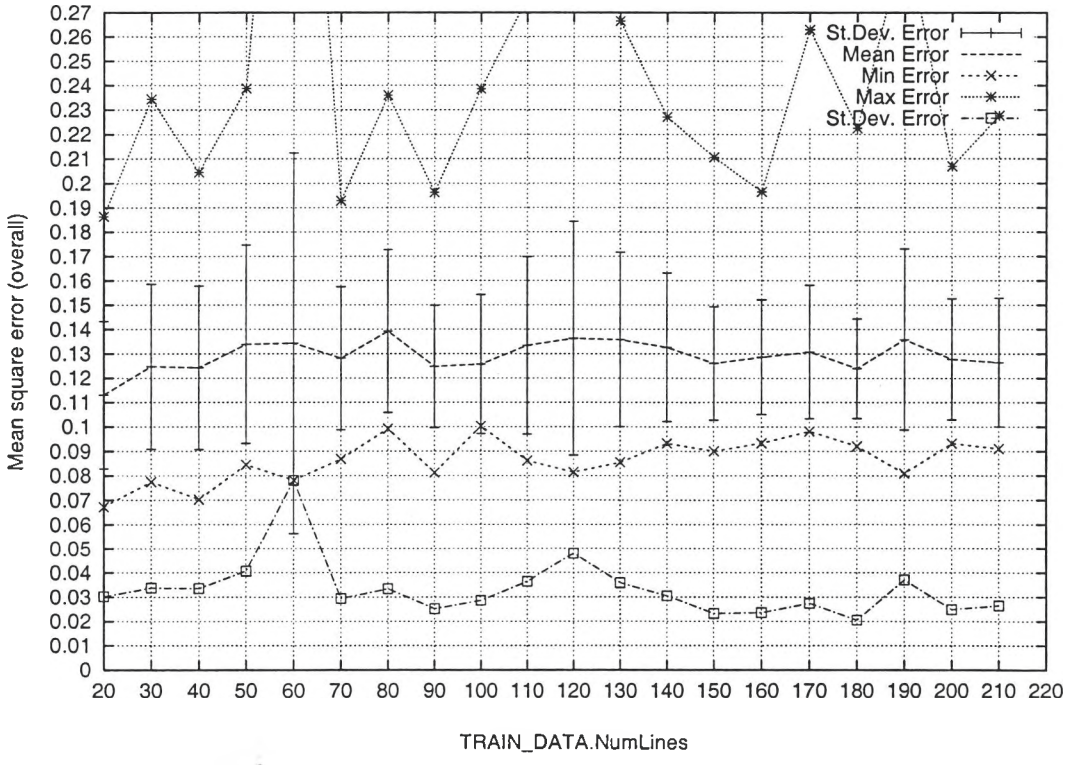

Figure 6.80: $\mathcal{N}_4$, **approximation error** against number of **training vectors**

The following figures contain scatter plots of the *approximation error* of all evaluated networks as the number of **training vectors** increases.

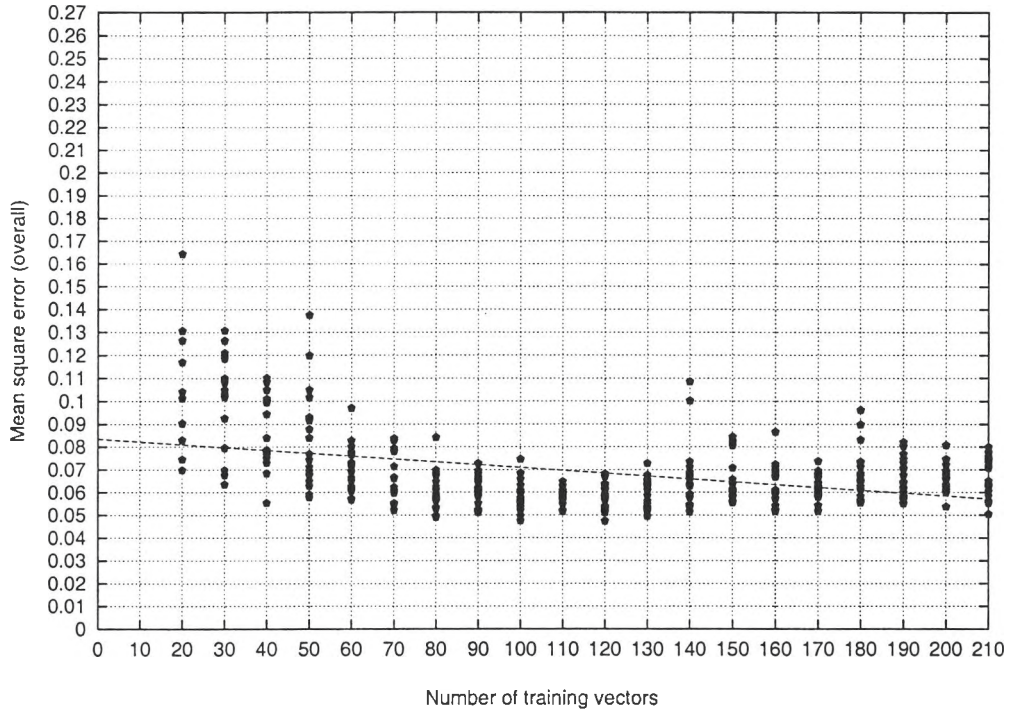

Figure 6.81: C_1, **approximation error** against number of **training vectors**

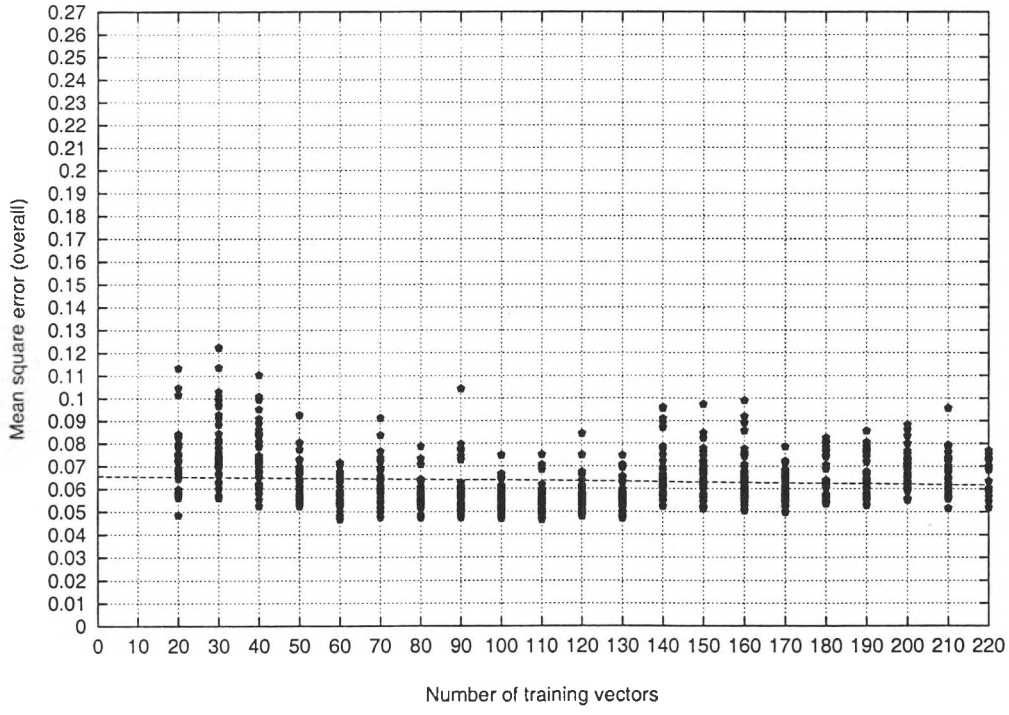

Figure 6.82: $C_{1,big}$, **approximation error** against number of **training vectors**

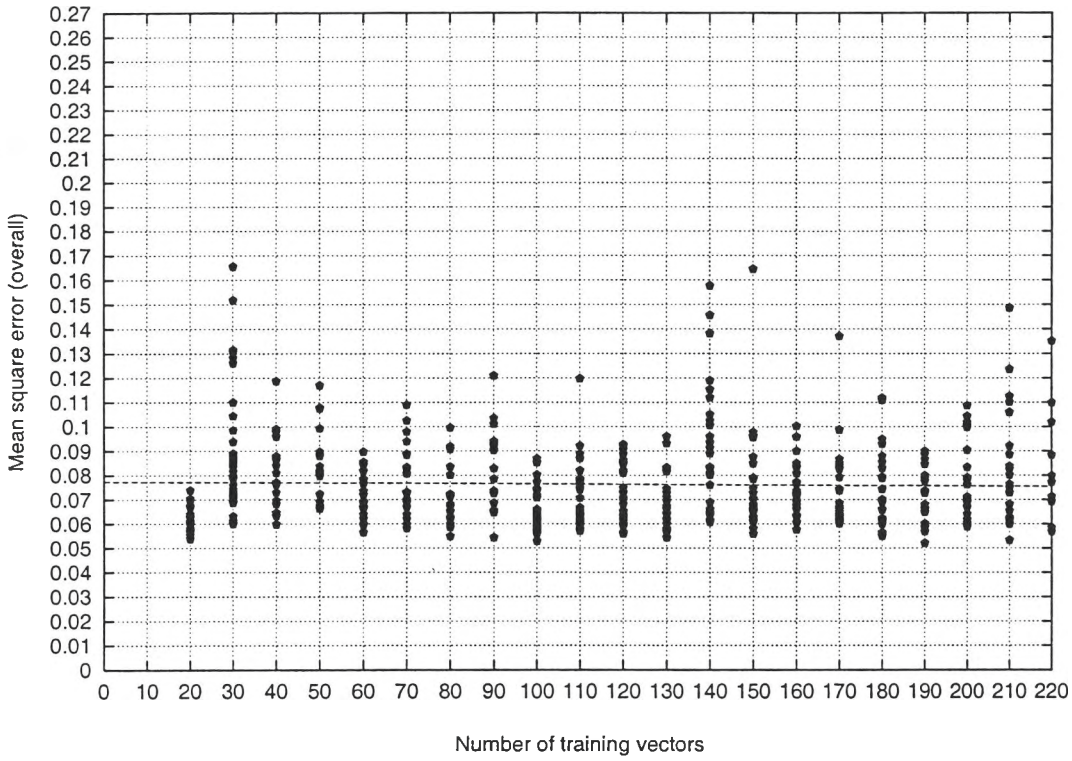

Figure 6.83: $\mathcal{C}_2$, **approximation error** against number of **training vectors**

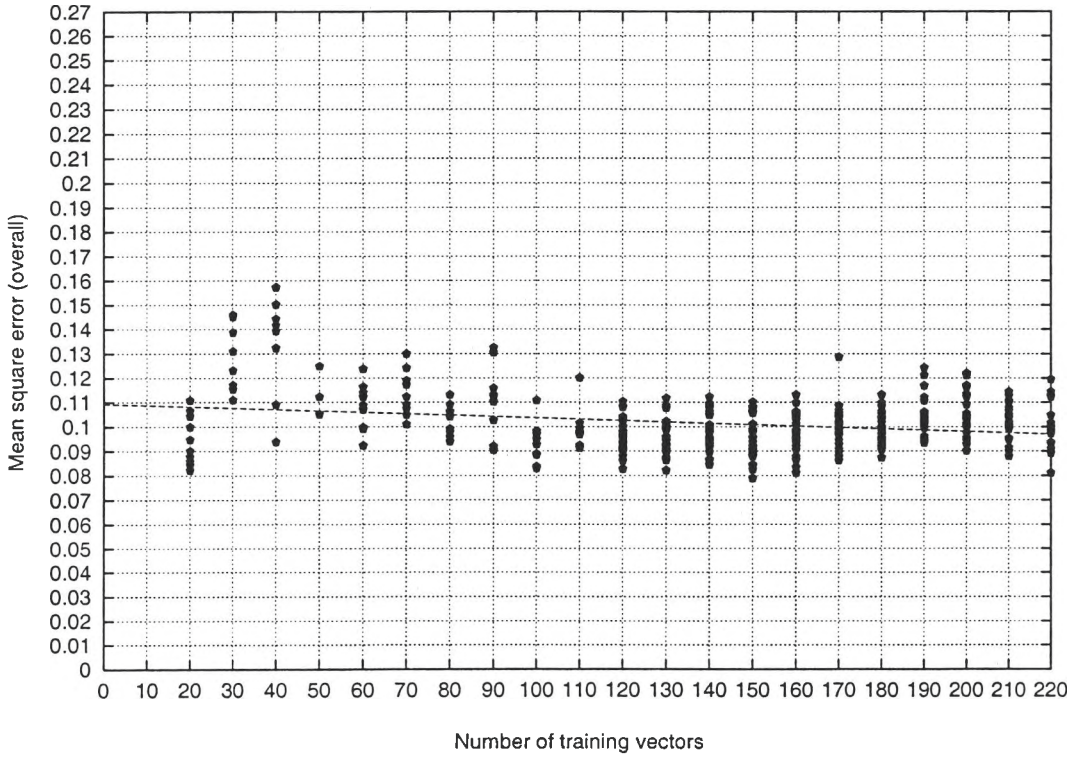

Figure 6.84: $\mathcal{C}_3$, **approximation error** against number of **training vectors**

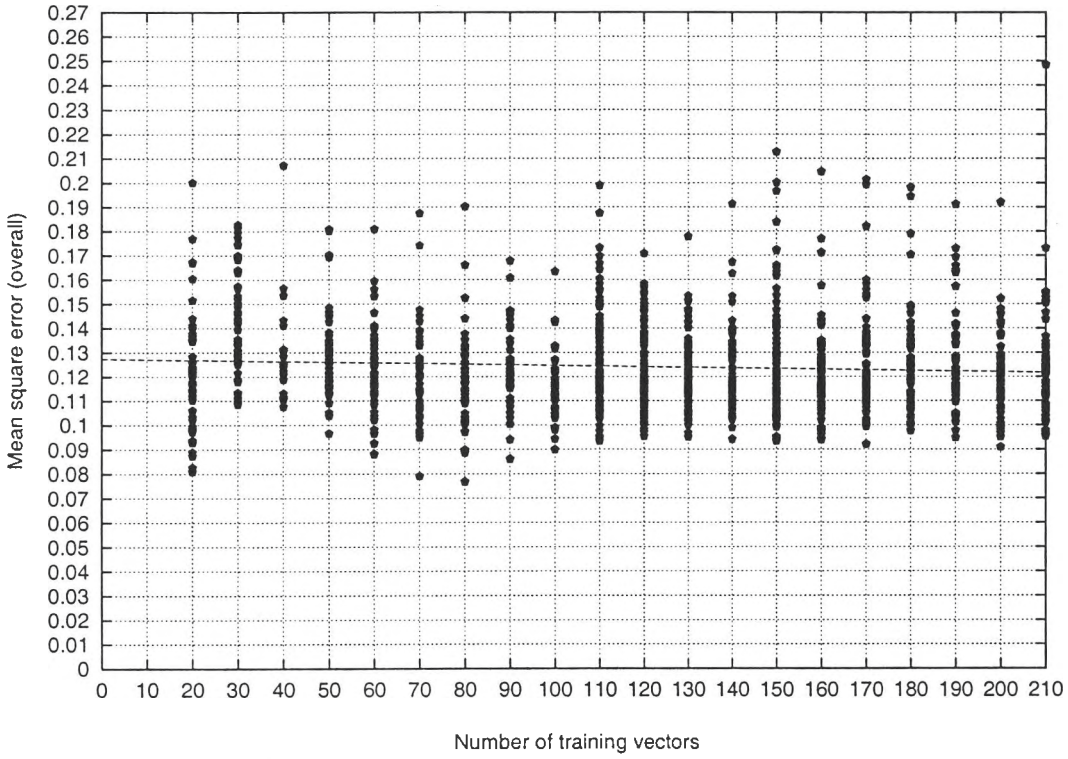

Figure 6.85: $\mathcal{N}_1$, **approximation error** against number of **training vectors**

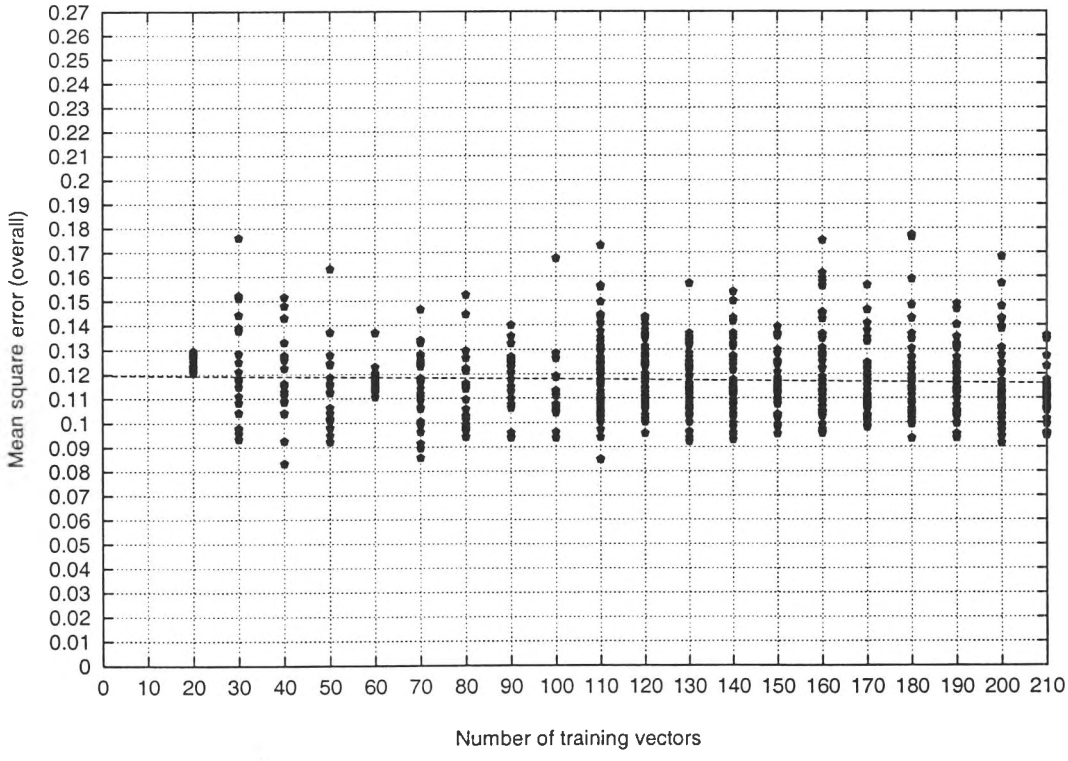

Figure 6.86: $\mathcal{N}_2$, **approximation error** against number of **training vectors**

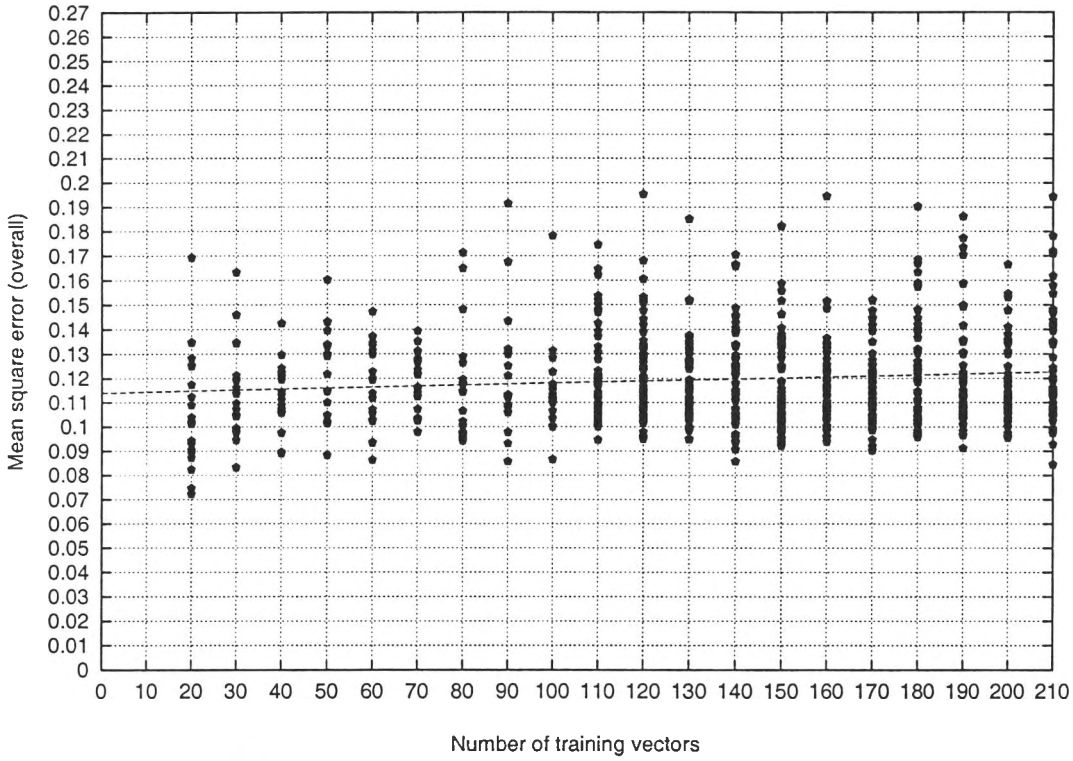

Figure 6.87: $\mathcal{N}_3$, **approximation error** against number of **training vectors**

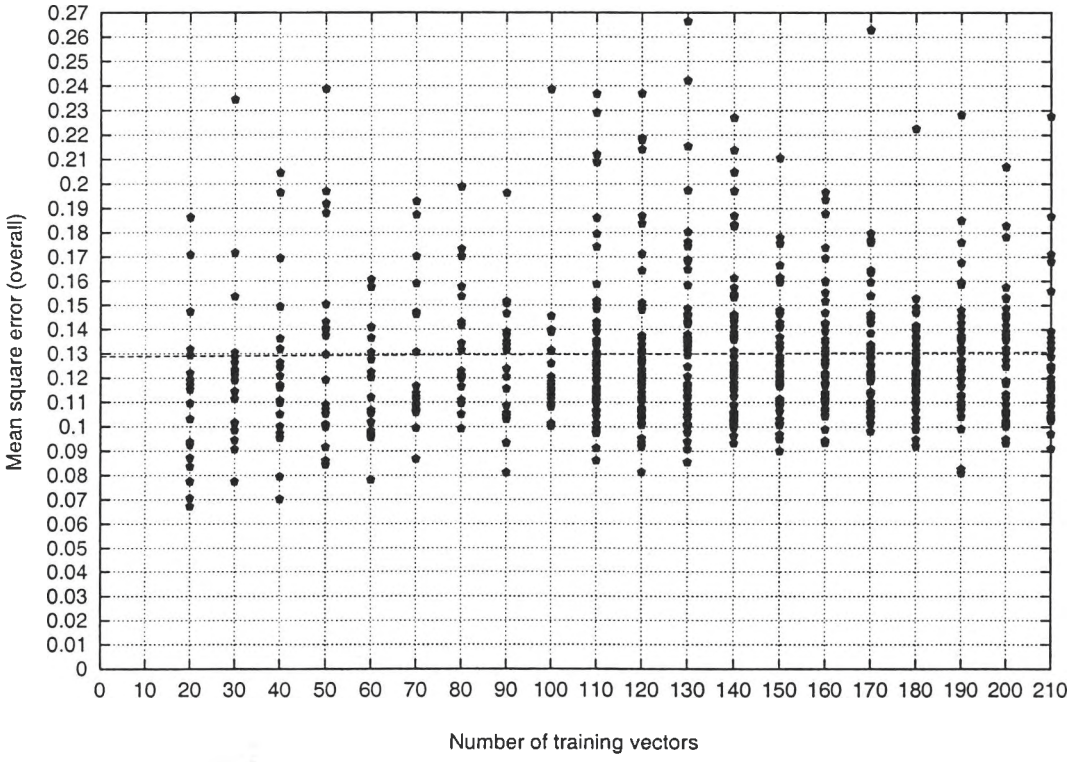

Figure 6.88: $\mathcal{N}_4$, **approximation error** against number of **training vectors**

6.4 Parallelisation of the training process

6.4.1 Introduction

In section 5.8 on page 76 it was argued that one of the advantages of the entities over the single FFNN networks is that the structure of the former favours *coarse-grain* parallelism as opposed to the *fine-grain* parallelism inherent in the architecture of the latter. In this section we will present the results of tests carried out to compare the training times of a C_1 entity model in *sequential* and *parallelised* modes of operation.

The parallelisation scheme used was a very simple one. At first, the network's units are grouped according to their respective layers. Secondly, the units of the first layer are divided evenly among the available processors and their training is done concurrently. When all first layer units are trained, the procedure repeats with the units of the second layer, until all the units of the entity are trained. After each layer is processed, the weights are transfered to a central store where the test procedure will take place sequentially. All the test scripts were written in the ***mp*** language and can be found in appendix G on page 221. Note that parallelising the training procedure of the entities does not require a specific parallel hardware platform. All is needed are some networked workstations talking TCP/IP and running Unix.

There were *four* C_1 entity networks of different sizes, participating in this test. Their architecture is detailed the following table:

NAME	DESCRIPTION
$\mathcal{P}_1$	500 inputs, 32 units, 13,030 weights
$\mathcal{P}_2$	1,000 inputs, 60 units, 26,035 weights
$\mathcal{P}_3$	1,500 inputs, 91 units, 39,120 weights
$\mathcal{P}_4$	2,000 inputs, 120 units, 52,490 weights

Table 6.27: Parallelised training, description of the evaluated networks

Each network was trained for 1,000 iterations on the same data set. In the first instance, training was done *sequentially*, that is, in exactly the same way as with all the networks of VARIDIM and CONSDIM in section 6.3. In the second instance, the training process was split evenly among **four** different workstations of equal CPU power. Each network's training procedure, either in parallel or sequential mode, was repeated for 50 times and the training time was recorded each time. The *minimum*, *maximum*, *mean* and *standard deviation* of the *training time*, calculated over the 50 repeats, are presented in table 6.28.

It should be noted that the purpose of this test is not to measure exactly the difference in training time between sequential and parallelised training schemes but rather to demonstrate the ability of the entities to be parallelised efficiently.

On the one hand, the parallelisation methodology and algorithms employed are not the best or the most efficient. More experimentation and in-depth knowledge of computer networking and the **Unix** programming environment would have probably yielded better parallelisation schemes.

On the other hand, the computer resources at our disposal could not be used exclusively for the purposes of these simulations. For example, a factor that determines training time in such parallelised schemes is the traffic of the computer network and the load in accessing the file system (our central store). These parameters are determined by the number of users and what they are doing at a given instance – something which was largely beyond our control.

Almost none of these restrictions exist when training is done sequentially and on a single user machine.

6.4.2 Results and discussion

		$\mathcal{P}_1$	$\mathcal{P}_2$	$\mathcal{P}_3$	$\mathcal{P}_4$
NETWORK SIZES	inputs	500	1,000	1,500	2,000
	weights	13,030	26,035	39,120	52,490
	FFNN units	32	60	91	120
SEQUENTIAL TRAINING TIME (seconds)	minimum	992	1,977	2,711	3,539
	maximum	1,026	2,030	2,739	3,577
	stand. dev.	9.22	10.11	8.1	8.71
	mean	**1,009**	**1,997**	**2,727**	**3,555**
PARALLELISED TRAINING TIME (seconds)	minimum	577	859	987	1,273
	maximum	645	895	1,003	1,312
	std. dev.	14.88	8.25	9.33	8.78
	mean	**595**	**879**	**993**	**1,296**

Table 6.28: Parallelised and sequential training time results

The results of the **Parallelisation** test are presented in table 6.28. For each network, its size (e.g. number of **inputs**, **weights** and FFNN **units**) and the *minimum*, *maximum*, *mean* and *standard deviation* of the corresponding **training time**, calculated over the 50 repeats, are reported.

These results indicate that there are significant benefits to be gained from parallelising the training procedure of the entities.

In particular, the time required by $\mathcal{P}_1$ networks trained sequentially is, on average, 1,009 seconds. This is 69.6%[13] more than the 595 seconds required by the same networks when trained in parallel.

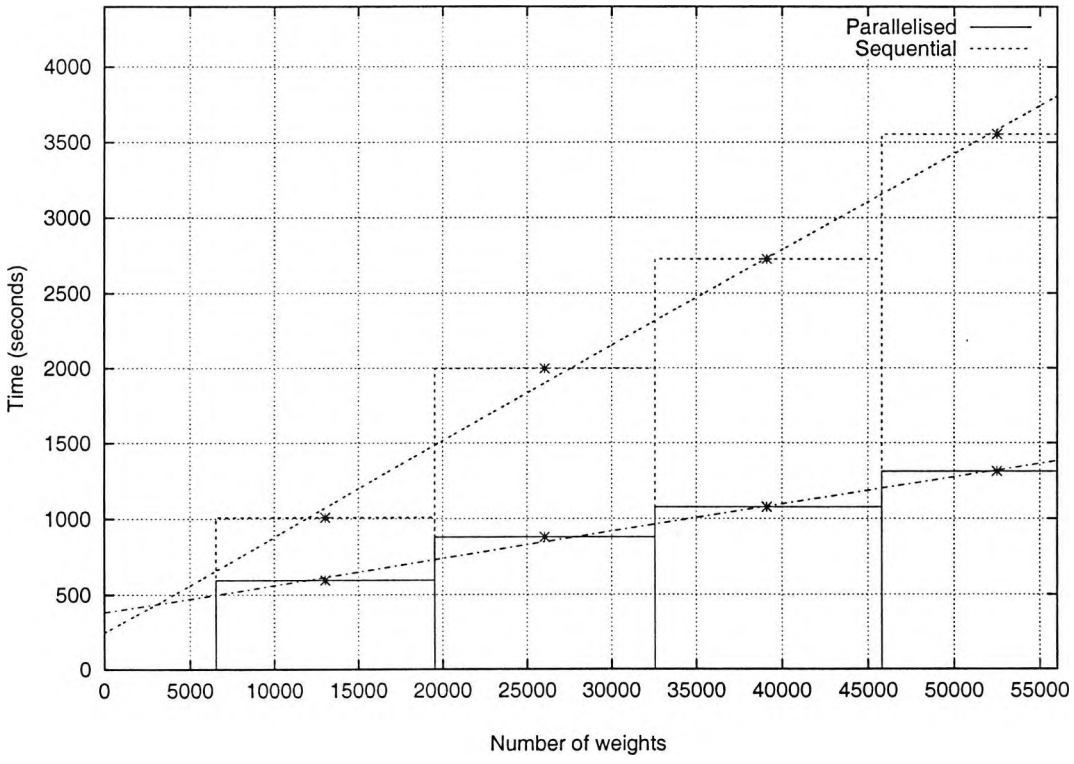

Figure 6.89: Comparison of training times for sequential and parallelised training schemes

When the number of weights is doubled (e.g. $\mathcal{P}_2$), the sequential training requires 1,997 seconds to complete, compared to 879 seconds for the parallelised training scheme of the same network; this makes the sequential training process 127.2% slower than the parallelised equivalent. Similar results hold for $\mathcal{P}_3$ and $\mathcal{P}_4$ networks where the difference between sequential and parallelised training times is 161.8% and 173.4%, respectively.

Figure 6.89 contains bar charts of the **training time** against the **total number of weights** for sequential and parallelised training. Least mean squares lines have also been fitted on each of the two charts with a very high correlation (99%). These lines have slopes of 63.7×10^{-03} and 17×10^{-03} for the sequential and parallelised schemes,

[13] The calculation is as follows: $\frac{1,009-595}{595} \times 100\% = 69.6\%$.

respectively. The equations of the least mean squares lines (t_s for sequential and t_p for parallelised), as a function of the total number of weights (w) are given below:

$$t_s = 0.064\, w + 242.6 \tag{6.3}$$

$$t_p = 0.018\, w + 382.2 \tag{6.4}$$

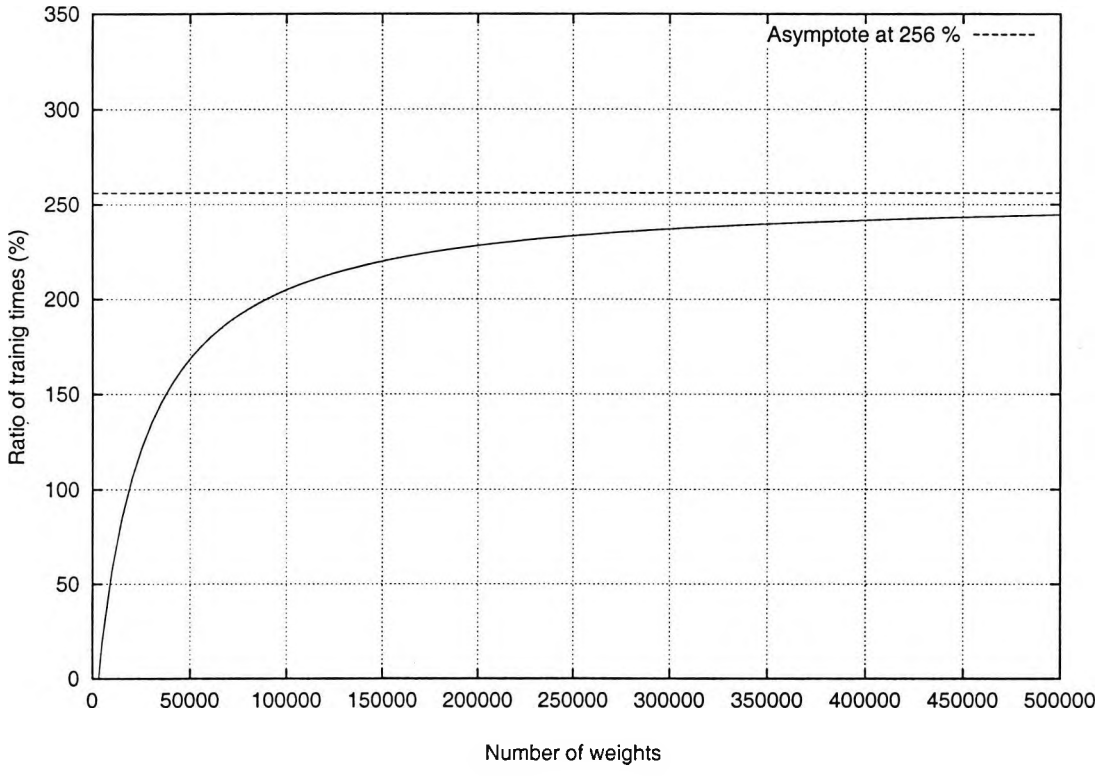

Figure 6.90: Plot of $t_{s/p}(w)$ against the total number of weights, w

The above equations will be used in order to derive an expression for the training time benefits obtained from using parallelised instead of sequential training, for a given number of weights. For this purpose we may use the following percentage:

$$t_{s/p}(w) = \frac{t_s - t_p}{t_p} \times 100\% = \frac{0.064\, w + 242.6 - 0.018\, w - 382.2}{0.018\, w + 382.2} \times 100\%$$

$$= \frac{0.046\, w - 139.6}{0.018\, w + 382.2} \times 100\% \tag{6.5}$$

A plot of $t_{s/p}(w)$ against the number of weights, w is shown in figure 6.90. The horizontal asymptote of $t_{s/p}(w)$, for positive w, is at:

$$T_{s/p} = \lim_{w \to \infty} t_{s/p}(w) = \frac{0.064}{0.046} \times 100\% = 255.56\% \tag{6.6}$$

In conclusion, it can be said that for large w, the time required for sequential training of a C_1 entity is longer by more than 250%, compared to the time taken by a parallelised version of the same procedure distributed over **four** workstations.

6.5 Summary and conclusions

6.5.1 Generalisation Ability

The first test (VARIDIM) under the **Generalisation Ability** category aimed at investigating the effects of increasing the number of input dimensions on the overall performance of the participating networks. Below are the conclusions and supporting evidence in terms of the **sample** and **approximation error** results.

VARIDIM, **sample error**: The conclusions drawn from the **sample error** (e.g. the error on the training set) results of this test are the following:

1. For a small number of input dimensions (less than 500), the training of $\mathcal{C}_1$ and $\mathcal{C}_{1,big}$ entity networks is much more consistent than that of single FFNN. The training of $\mathcal{C}_2$ and $\mathcal{C}_3$ is as consistent as that of single FFNN. When the number of input dimensions exceeds 500, the training of all entities is more consistent than that of single FFNN. This implies that the entities' error surface is smoother and the number of local minima fewer than those in the respective single FFNN.

2. As the number of input dimensions increases, the consistent training of the entity networks *improves*, thus they appear to be virtually unaffected by the *curse of dimensionality* and other problems which usually plague single FFNN with a lot of inputs.

These conclusions are supported by the following evidence:

1. The statistical significance tests comparing the variances of the entity and single FFNN networks (the *F-test*) indicate that for a number of input dimensions less than 500, the hypothesis that the *variance* of the entities is lower than that of single FFNN, e.g. H_2 is always true for $\mathcal{C}_1$ and $\mathcal{C}_{1,big}$ (the top two rows of table 6.6).

   As far as training consistency is concerned, the superiority of the entities over the single FFNN is obvious, especially when the number of input dimensions exceeds 500. In this case, the *F-test* results, outlined in table 6.7, show that the H_2 hypothesis is always true and by as much as 100 %.

2. The best entity's *standard deviation* (see table 6.2 on page 99, $\mathcal{C}_1$ entity) is, on average, 0.4×10^{-03}. The best single FFNN's *standard deviation* (see table 6.3 on page 99, $\mathcal{N}_2$ single FFNN) is, on average, 2.5×10^{-03} – a difference of more than 500%.

3. The scatter plots in figures 6.25 on page 106 to 6.28 on page 108, associated with the sample error of the **entities**, show that **outliers** *do exist but their number* **decreases** *as the number of input dimensions increases.*

4. The scatter plots in figures 6.29 on page 108 to 6.32 on page 110, associated with the sample error of the **single** FFNN, show that *the number of outliers increases as the number of input dimensions increases.* Adding more weights to these networks does not remedy the situation – on the contrary it makes it worse.

VARIDIM, **approximation error**: The conclusions that can be drawn after examination of the **approximation error** results of VARIDIM are the following:

1. The entities generalise better than single FFNN.

2. The entities' generalisation ability is not reduced by the increasing number of input dimensions – on the contrary, it increases.

3. The entities' generalisation ability is more consistent than that of single FFNN.

These conclusions are supported by the following evidence:

1. The best entity's **approximation error** (see table 6.8 on page 110, $\mathcal{C}_1$ entity) is, on average, 60×10^{-03}. The best single FFNN **approximation error** (see table 6.9 on page 111, $\mathcal{N}_3$) is, on average, 96×10^{-03} – a difference of 60%. Observe that the single FFNN has 55% more weights than the entity and that its *standard deviation* is 445% more than the entity's (e.g. 37×10^{-03} compared to 6.78×10^{-03})!

2. The statistical significance tests comparing the means of the **approximation error** of the entities and single FFNN (the *t-test*, see table 6.10) indicate that very rarely the hypothesis that a single FFNN's error is lower than that of any entity is accepted. For example, this happens only below 500 input dimensions and when comparing $\mathcal{N}_3$ with all the entities (the $\mathcal{N}_3$ column of table 6.10) or $\mathcal{C}_3$ with all the single FFNN (the $\mathcal{C}_3$ row of table 6.10).

   For a number of input dimensions larger than 500, the H_2 hypothesis that the mean of the **approximation error** is significantly lower than that of the single FFNN is always accepted and often with a percentage of 100 % (except when comparing $\mathcal{C}_3$ with $\mathcal{N}_3$ – they are comparable, see table 6.11).

3. The best entity's *standard deviation* is, on average, 6.78×10^{-03} (this is $\mathcal{C}_1$, the best generaliser) whereas the best single FFNN average *standard deviation* is 17×10^{-03} (this is $\mathcal{N}_2$) – a difference of 150%. *Observe that among the single* FFNN, *the best generaliser (e.g. $\mathcal{N}_3$) is not the most consistent one (e.g. $\mathcal{N}_2$).*

4. The statistical significance tests comparing the variances of the different networks (the *F-test*) show that, when the number of input dimensions is below 500, the variance of the **approximation error** of the entities is either comparable or lower than that of the single FFNN (see table 6.12), but, for a larger number of input dimensions (exceeding 500) the superiority of the entities, as far as training consistency is concerned, is indisputable with acceptance of the H_2 hypothesis by 100 % (see table 6.13).

5. All *least mean squares lines* fitted on the scatter plots of the entities (in figures 6.41 on page 118 to 6.44 on page 119) have negative slopes, indicating a **decreasing trend** of the approximation error with the increasing number of input dimensions. In contrast, the respective plots for the single FFNN (in figures 6.45 on page 120 to 6.48 on page 121) indicate a clearly **increasing trend.**

6. The outliers appearing in the entities' scatter plots are very few and the plots appear very compact. In contrast, the outliers in the single FFNN's scatter plots are much more and increase with the increasing number of dimensions.

The second test (CONSDIM) under the **generalisation Ability** category aimed at investigating the effects of increasing the number of training vectors on the overall performance of the participating networks whose sizes were kept fixed. Below are the conclusions and supporting evidence in terms of the **sample** and **approximation error** results.
CONSDIM, **sample error:** The conclusions drawn from the **sample error** results of this test are the following:

1. The entities' training is very consistent compared to that of single FFNN.

2. The entities' training consistency decreases with the increasing number of training vector but at a much slower pace than that of single FFNN. Therefore, it can be said, that the entities are more flexible because they can be trained equally well for a wide range of training vector numbers without any side effects. The single FFNN, on the other hand, suffers serious side effects when the number of training vectors 100.

These conclusions are supported by the following evidence:

1. The best entity's *standard deviation* of the **sample error** is, on average, 0.668×10^{-03}. The best single FFNN average *standard deviation* is 3.8×10^{-03} – a difference of more than 450%.

2. The statistical significance tests comparing the variances of the different networks (the *F-test*) show that for more than 120 training vectors, the H_2 hypothesis is accepted by 100 % (table 6.20).

   For a number of training vectors less than 120, the acceptance of the H_2 hypothesis is almost as absolute as above with, perhaps, the exception of $\mathcal{N}_1$ (see table 6.19, the $\mathcal{N}_1$ column), where it is only by 70 %.

3. All *least mean squares lines* fitted on the entities' scatter plots (in figures 6.65 on page 135 to 6.68 on page 136) have positive slopes. This indicates an *increasing trend of the* **sample error** *with the increasing number of training vectors.* The same applies for the case of the scatter plots of single FFNN (in figures 6.69 on page 137 to 6.72 on page 138). The difference between the entity networks and single FFNN is quantitative: the slopes of the former range from 7×10^{-06} to 20×10^{-06}, whereas the latter's range from 33×10^{-06} to 52×10^{-06}.

CONSDIM, **approximation error:** The conclusions drawn from the **approximation error** results of this test are the following:

1. The entities are better generalisers than single FFNN for the whole range of the number of training vectors.

2. The entities are more consistent generalisers than the single FFNN.

3. The entities, particularly those of class 1 and 3, have their lowest **approximation error** when the number of training vectors is about 100.

These conclusions are supported by the following evidence:

1. The best entity's **approximation error** is, on average, 63×10^{-03} (see table 6.21 on page 139, $\mathcal{C}_{1,big}$) and does not fall below (again $\mathcal{C}_{1,big}$) 54×10^{-03}. The best single FFNN's **approximation error** is, on average, 117×10^{-03} (see table 6.22 on page 139, $\mathcal{N}_2$) and does not fall below 107×10^{-03} ($\mathcal{N}_3$).

2. The statistical significance tests comparing the means of the **approximation error** (the *t-test*) of the various networks when the number of training vectors is less than 120, show that the H_2 hypothesis is true for all entities with as high a percentage as 100 % (see table 6.23). $\mathcal{C}_3$ is an exception with lower confirmation percentages ranging from 55 % to 82 % (see the $\mathcal{C}_3$ row table 6.23).

   When the number of training vectors is more than 120, the hypothesis that the mean **approximation error** of the entities is lower than that of any single FFNN (e.g. the H_2 hypothesis) is accepted absolutely by 100 % (see table 6.24).

3. The best entity's *standard deviation* of the **approximation error** is 8.8×10^{-03} (table 6.21 on page 139, $\mathcal{C}_3$) while the best single FFNN's is 13.9×10^{-03} (table 6.22 on page 139, $\mathcal{N}_2$).

4. The statistical significance tests comparing the variances of the different networks (the *F-test*) for a number of training vectors less than 120, show that in most cases the H_2 hypothesis is true. In a few other cases, however, the *null hypothesis* – e.g. that the variances of the two tested networks do not differ significantly – seems to be true (see table 6.25).

   When the number of training vectors exceeds 120, the situation improves and the confirmation of the H_2 hypothesis is more substantial. The *null hypothesis* is mainly true when comparing the $\mathcal{C}_2$ entity with all single FFNN (see table 6.26, the $\mathcal{C}_2$ row).

5. Almost all single FFNN's scatter plots show a large number of outliers compared to those appearing tin the scatter plots of the entities.

6.5.2 Parallelisation of training

The test comparing the times required to train a class 1 entity in sequential and parallelised modes has shown that, as expected, parallelised training requires much shorter time than sequential training.

In particular, sequential training can take up to 250% more time than when training is distributed among four networked workstations of the equal CPU power.

CHAPTER VII

CONCLUSION

This chapter concludes the thesis by recapitulating on the requirements which motivated this work and how they were met.

This thesis has addressed one of the fundamental requirements of feed forward neural networks: the ability to expand in size and deal efficiently with high-dimensional data.

The novelty of this work lies in its development of a set of modular neural network architectures, the FFNN entities, which are based on *emergence* and the principles of connectionism while, at the same time, are characterised by the diversity in complexity, level of abstraction and functionality of its constituent elements.

This chapter concludes the thesis by recapitulating on the requirements which motivated this work and how they were met. It presents areas of future work which are either related to, but outside the direct scope of, this thesis or are possible directions for extending this research further.

7.1 Recapitulation

Central in our research is the development of a framework for replacing the traditionally monolithic FFNN with an entity of small and flexible units. Unlike other attempts to *modularise* neural networks, our methodology is not based on connecting neural networks together under the central control of a higher authority which selects the best network, neither is our primary aim to achieve better generalisation by *task decomposition* or *bootstrapping* (see for example section 5.3 on page 52).

7.1.1 Motivations

The motivation to design and implement the modular architecture of the entities stems

from the inability of single FFNN to scale up and, consequently, deal efficiently and effectively with high-dimensional data without resorting to dimensionality reduction techniques. Furthermore, although existing feed forward neural network models, be they modular or monolithic, are relatively successful in addressing issues of generalisation, specialisation and confidence of prediction, the problems associated with high-dimensional data and scaling remain basically unanswered. The thesis that the brain is not only characterised by a massively connected network of neurons but also by the existence of different computational systems operating at different levels of abstraction and specialising at different functions, [Freeman, 1991], is by itself a right justification to replace the single FFNN with the entities.

At a theoretical level, the approximation capabilities of the proposed entity classes C_1, C_2 and C_3 are equivalent to those of single FFNN as they, too, can approximate arbitrarily well any real, continuous function. At a practical level, a comparison of the generalisation ability and training consistency of the two models favours the entities. This becomes clearer as the dimensionality of the training data increases.

7.1.2 FFNN entities: the model

The entities are characterised by:

- vast but controlled connectivity[1] and at various levels, of relatively simple elements whose type is not restricted only to neurons,
- a number of localised processes each assigned only a small part of the input data parameters,
- training procedures which are local at the level of single FFNN and, at the same time, global, at the level of the whole entity.

What is more, the entities promote a framework of constructing complex connectionist structures by interconnection of computing elements which differ in complexity but are similar as far as structure is concerned. For example, neurons $\rightarrow$ single FFNN $\rightarrow$ entities of single FFNN $\rightarrow \cdots \rightarrow$ entities of entities of ...

A possible taxonomy of various entity designs based on two criteria: *topology* and *complexity of their training targets*, has yielded the following three classes:

[1] *Vast connectivity* characterises the neurons which make up the various single FFNN of the entity. On the other hand, the connectivity between these single FFNN or other, less complex entities that make up an entity can be *controlled* at the design stage.

- C_1 entities are constructed based on partial connectivity of which the only requirement is that of feed forward signal propagation. Weights of variable strength can be incorporated along the connections of the different elements of the entity. The adjustment of these weights is effected by the use of a generalised version of back-propagation, when all the localised training procedures – which also use back-propagation – are completed.

- C_2 entities are based on a cascaded architecture. This results in a more regular structure which allows for better control as far as size and task allocation are concerned.

- The topology of C_3 entities is the same as that of C_2. The difference is that the training target of the units composing C_3 entities is a measure of the discrepancy between actual and desired outputs of the previous unit.

In effect, the difference between classes C_1 and C_2 is one between *unordered* and *ordered*, *unstructured* and *structured* topology. Whereas, the difference between classes C_2 & C_2 and C_3 is one of more refined training targets. It is interesting to note that the idea of refined training targets could not have been applied to C_1 without imposing serious constraints on its architecture. On the one hand, one risks losing the generality offered by the C_1's unstructured topology and, probably, ending up with the same cascaded architecture of C_2 and C_3, anyway. On the other hand, each unit of the C_1 class may have more than two inputs. This calls for a multi-dimensional error metric with all the complications this implies. Thus, the design and implementation of a fourth entity class was not attempted because it was considered to be time consuming and would not add significantly to the novelty of this work.

7.1.3 Utility of the entities

The adoption of the entities as a means for the analysis of high-dimensional data not only solves the important problems of scaling and deals effectively with the *curse of dimensionality* but also yields a number of significant advantages over single FFNN.

Firstly, a *coarse-grain* parallelisation model is promoted which allows for better resource allocation, more balanced mapping of processes to processors and, more importantly, a huge reduction in the communication needs between the various processes. This feature is not only important for the *efficient parallelisation* of the training process of the entities, but also it is decisive for a feasible *hardware implementation*. In this respect, the engineering problems associated with transferring an entity to silicon are

eased considerably compared to those of single FFNN[2].

Secondly, the size of an entity is determined by two factors: the size of each FFNN unit and the total number of these units. Thus, the construction of an entity is more flexible than that of a single FFNN. Consequently, the size of each FFNN unit is no longer governed by the dimensionality of the training data. FFNN can, therefore, be kept conveniently small so as to avoid the usual pathologies associated with large FFNN, namely *premature neuron saturation*, *complex error surface* and *numerous local minima*.

Thirdly, the entities are a connectionist system where information is not only distributed and stored *implicitly* in the weights of each FFNN unit and in the weights *between* the FFNN units, but also exists on a higher level, that of the FFNN (and all the other higher level units composing the entity) which can identify, more readily, meaningful information. Thus, the entities may be studied with an arbitrary level of abstraction.

7.1.4 Theoretical results

The approximation capabilities of the three proposed entity classes are, theoretically, equivalent to those of single FFNN as it was proved that they are *universal function approximators* and thus, can approximate arbitrarily well any real, continuous function. The proof is contained in section 5.6 on page 70, and is based on the *Stone-Weierstrass theorem*. This same theorem was used by [Hornik, 1991] and others in order to prove that the class of feed forward neural networks of a single hidden layer containing an *arbitrary* number of hidden units holds the property of universal function approximation.

In addition to this theoretical guarantee of the capabilities of the entities, a further investigation of their performance at a practical level and comparison with single FFNN was deemed necessary. These empirical results are summarised in the following section.

7.1.5 Experimental results

Two sets of experiments were carried out in order to assess the performance of the entities in practice. In particular, VARIDIM compared the *training time*, *sample error*

[2] An analogy from the field of micro-electronics is the following: currently the number of transistors that can be accommodated to a single silicon chip for the purposes of building CPU is restricted by technology and the laws of physics. One way to overcome this limitation is parallel computing where a large number of CPU of average power are connected together. The efficiency of this parallel computing device is a trade-off between communication needs and power of the constituent CPU.

and *approximation error* of a set of single FFNN and entities as the number of training data dimensions increased while the number of training vectors was kept constant. CONSDIM, on the other hand, compared the same quantities but on a set of single FFNN and entities whose input dimensions and number of weights were kept fixed while the number of training data vectors varied.

The results of these experiments systematically favour the entities which not only exhibit a very *consistent training* with very few deviations from the mean but also, their *generalisation ability* is much better compared to that of equivalent single FFNN – sometimes as much as two or three-fold.

Moreover, the generalisation ability and training consistency of the entities improves or, at worst, remains unchanged when the number of dimensions of the input data increased. By contrast, according to our experiments, the performance of single FFNN clearly deteriorates when the number of input dimensions exceeds 400 or 500. A detailed analysis of these empirical results is contained in section 6.5.1 on page 154.

Chapters 4 and 5 contain ample proof that the training time of both the entities and single FFNN is directly proportional to the number of weights they contain[3]. Thus, an entity and a single FFNN with the same number of weights will take the same time to train. However, we can not overlook the fact that training time is *polynomial* to the *number of input dimensions.* In this respect, the analysis given in section 5.7 on page 72 shows, that provided that all single FFNN are constructed using the same parameters (e.g. the ratio of the single layer units to the number of inputs) an entity will be trained faster than a single FFNN by a factor which approaches asymptotically the quantity $\beta\gamma$ as the number of input data dimensions increases. β is the ratio of the number of inputs per single FFNN unit in the entity (assumed constant) to the number of data dimensions and γ is the ratio of the sum of the number of inputs of all available FFNN units to the number of data dimensions.

In addition, there is a wide margin for improvement to the training time of the entities due to the *coarse-grain* parallelism model they promote. In particular, a comparison of training times between two identical C_1 entities, one trained sequentially and the other one distributed over three different processors, reveals that the sequential training time is longer than the parallelised one by more than 250%.

[3] For a fixed number of training iterations.

7.2 Future work

Directions for future research include, but are not restricted to, the following:

1. Improvement of the parallelisation methodology in terms of *load balancing* and *communication overheads* by investigating alternative ways of partitioning the training task and distributing it to the available processors. Also, investigation of the relationship of various entity architectural parameters[4] such as the *number*, *size* and *number of inputs* of the single FFNN composing the entities, to the efficiency of the parallelisation scheme.

2. Investigate how the entities model might effect rule extraction. The entities may be studied with an arbitrary level of abstraction because they consist of units of arbitrary type and complexity. Moreover, they promote a model where information may be identified not only among the weights of the various units but also, more readily, among the entity units themselves.

3. Extend the entities to support multiple outputs.

4. Create new entity classes and experiment with them.

5. Assess, at a theoretical level, the generalisation bounds of the entities. A possible direction would be to establish bounds of the *Vapnik-Chervonenkis dimension* (see appendix C on page 173) for the entities.

6. Extend the benchmarks by including more data sets and comparing the performance of the entities with more learning machines – for example Support Vector Machines.

[4] See also 5.7 on page 72 for a reference to some of these parameters.

Appendices

APPENDIX A

DERIVATION OF THE BACK-PROPAGATION ALGORITHM

A.1 Introduction

The derivation of the back-propagation and weight update formulae is quite standard and included in many neural network textbooks. For example in [Freeman, 1991].

Recall the notation introduced in section 3.2 on page 20 and that according to definition 3.5 and equation 3.6 on page 26, for a given input vector $\underline{\mathbf{X}}$, the corresponding FFNN output response $\underline{\mathbf{Y}}$, and a target output response $\underline{\mathbf{T}}$, the discrepancy at the output is:

$$E = \frac{1}{2}\sum_{k=1}^{u(L)}(T_k - Y_k)^2$$

Recall also that for a given FFNN, a weight modification rule based on gradient descent is the following:

$$\underline{\mathbf{W}}(t+1) = \underline{\mathbf{W}}(t) - \beta\,\nabla E$$

where $\underline{\mathbf{W}}(t)$ denotes the set of its free parameters at time t and $\beta \in \mathbb{R}^+$.

In the next two sections we will calculate ∇E, a set of partial derivatives of E with respect to each of the elements in $\underline{\mathbf{W}}$. A distinction will be made for derivatives with respect to weights of the output layer and weights of the hidden layers.

A.2 Derivatives with respect to the output layer weights

We begin with the calculation of the partial derivative of E with respect to w_{kj}^L, that is the weight connecting the k^{th} unit of the output layer, L, to the j^{th} unit of the layer $L-1$:

$$\begin{aligned}\frac{\partial E}{\partial w_{kj}^L} &= -\frac{1}{2}\,2\,(T_k - Y_k)\cdot\frac{\partial Y_k}{\partial w_{kj}^L}\\ &= -(T_k - Y_k)\cdot\frac{\partial Y_k}{\partial w_{kj}^L}\end{aligned}$$

Recall that $Y_k = y_k^L = \sigma_k^L(A_k^L(\underline{\mathbf{y}}^{L-1}))$ where, $A_k^L(\cdot)$ and σ_k^L are the affine transform and activation function associated with the k^{th} unit of the L^{th} layer. Then, by chain rule[1],

$$\begin{aligned}\frac{\partial Y_k}{\partial w_{kj}^L} &= \frac{\partial \sigma_k^L(A_k^L(\underline{\mathbf{y}}^{L-1}))}{\partial A_k^L(\underline{\mathbf{y}}^{L-1})} \cdot \frac{\partial A_k^L(\underline{\mathbf{y}}^{L-1})}{\partial w_{kj}^L} \\ &= \sigma_k^{L'}(A_k^L(\underline{\mathbf{y}}^{L-1})) \cdot \frac{\partial A_k^L(\underline{\mathbf{y}}^{L-1})}{\partial w_{kj}^L}\end{aligned}$$

Recall that:

$$A_k^L = \sum_{j=1}^{u(L-1)} w_{kj}^L \, y_j^{L-1} + b_k^L$$

therefore[2],

$$\frac{\partial A_k^L}{\partial w_{kj}^L} = y_j^{L-1} \quad \text{and also} \quad \frac{\partial A_k^L}{\partial b_k^L} = 1$$

The required derivative is then:

$$\frac{\partial E}{\partial w_{kj}^L} = -\sigma_k^{L'}(A_k^L(\underline{\mathbf{y}}^{L-1})) \cdot (T_k - Y_k) \cdot y_j^{L-1} \tag{A-1}$$

and the derivative with respect to the bias term of the output unit is:

$$\frac{\partial E}{\partial b_k^L} = -\sigma_k^{L'}(A_k^L(\underline{\mathbf{y}}^{L-1})) \cdot (T_k - Y_k) \tag{A-2}$$

A.3 Derivatives with respect to the hidden layer weights

Let us now see what applies to units of the hidden layers. We will proceed in the same way as above except that we need to find the partial derivatives of the mean squared error with respect to the weights of the last hidden layer $(L-1)$, that is $\frac{\partial E}{\partial w_{ji}^{L-1}}$:

$$\frac{\partial E}{\partial w_{ji}^{L-1}} = -\sum_{k=1}^{u(L)} (T_k - Y_k) \cdot \frac{\partial Y_k}{\partial w_{ji}^{L-1}} \tag{A-3}$$

Recall that $Y_k = y_k^L = \sigma_k^L(A_k^L(\underline{\mathbf{y}}^{L-1}))$. Therefore, by chain rule,

[1] Note that with σ' we mean $\frac{d\sigma}{dx}$.

[2] w_{kj}^L appears only once in the above sum and therefore the partial derivatives of all the other terms with respect to w_{kj}^L are zero and, thus, the sum is eliminated.

$$\begin{aligned}\frac{\partial Y_k}{\partial w_{ji}^{L-1}} &= \frac{\partial \sigma_k^L(A_k^L(\underline{\mathbf{y}}^{L-1}))}{\partial A_k^L(\underline{\mathbf{y}}^{L-1})} \cdot \frac{\partial A_k^L(\underline{\mathbf{y}}^{L-1})}{\partial w_{ji}^{L-1}} \\ &= \sigma_k^{L\prime}(A_k^L(\underline{\mathbf{y}}^{L-1})) \cdot \frac{\partial A_k^L(\underline{\mathbf{y}}^{L-1})}{\partial w_{ji}^{L-1}} \end{aligned} \tag{A-4}$$

Recall that:

$$A_k^L(\underline{\mathbf{y}}^{L-1}) = \sum_{j=1}^{u(L-1)} w_{kj}^L \, y_j^{L-1} + b_k^L$$

then by chain rule,

$$\begin{aligned}\frac{\partial A_k^L(\underline{\mathbf{y}}^{L-1})}{\partial w_{ji}^{L-1}} &= \frac{\partial A_k^L(\underline{\mathbf{y}}^{L-1})}{\partial y_j^{L-1}} \cdot \frac{\partial y_j^{L-1}}{\partial w_{ji}^{L-1}} \\ &= w_{kj}^L \cdot \frac{\partial y_j^{L-1}}{\partial w_{ji}^{L-1}} \end{aligned} \tag{A-5}$$

Similarly, recall that the output of the j^{th} unit of the $L-1$ layer is given by:

$$y_j^{L-1} = \sigma_j^{L-1}\Big(\sum_{i=1}^{u(L-2)} w_{ji}^{L-1} y_i^{L-2} + b_j^{L-1}\Big)$$

then,

$$\frac{\partial y_j^{L-1}}{\partial w_{ji}^{L-1}} = \sigma_j^{L-1\prime}(y_i^{L-2}) \cdot y_i^{L-2} \tag{A-6}$$

and the derivative of y_j^{L-1} with respect to the bias term, b_j^{L-1}, is:

$$\frac{\partial y_j^{L-1}}{\partial b_j^{L-1}} = \sigma_j^{L-1\prime}(y_i^{L-2}) \tag{A-7}$$

Substituting equations A-6 into A-5 and A-5 into A-4, we get

$$\frac{\partial Y_k}{\partial w_{ji}^{L-1}} = \sigma_j^{L-1\prime}(y_i^{L-2}) \cdot \sigma_k^{L\prime}(y_j^{L-1}) \cdot w_{kj}^L \cdot y_i^{L-2}$$

Substituting the above equation into A-3 yields the final expressions for the error derivatives:

$$\frac{\partial E}{\partial w_{ji}^{L-1}} = -y_i^{L-2} \cdot \sigma_j^{L-1\prime}(y_i^{L-2}) \cdot \sum_{k=1}^{u(L)} (T_k - Y_k) \cdot \sigma_k^{L\prime}(y_j^{L-1}) \cdot w_{kj}^L$$

$$\frac{\partial E}{\partial b_j^{L-1}} = -\sigma_j^{L-1\prime}(y_i^{L-2}) \cdot \sum_{k=1}^{u(L)} (T_k - Y_k) \cdot \sigma_k^{L\prime}(y_j^{L-1}) \cdot w_{kj}^L$$

A.4 Final back-propagation equations

Let us first introduce the term δ associated with the j^{th} unit of the l^{th} layer as follows:

$$\delta_j^l = \begin{cases} (T_j - Y_j) \cdot \sigma_j^{l\prime}(A_j^l(\underline{\mathbf{y}}^{l-1})) & \text{if } l \text{ is the output layer, } L, \\ \sigma_j^{l\prime}(A_j^l(\underline{\mathbf{y}}^{l-1})) \cdot \sum_{k=1}^{u(l+1)} \delta_k^{l+1} w_{kj}^{l+1} & \text{if } l \text{ is hidden layer, } 1 < l < L. \end{cases} \quad \text{(A-8)}$$

This simplifies the partial derivative expressions to:

$$\frac{\partial E}{\partial w_{ji}^l} = -\delta_j^l y_i^{l-1} \quad \text{and} \quad \frac{\partial E}{\partial b_j^l} = -\delta_j^l$$

Finally, the update expression for the weight connecting the j^{th} unit of the l^{th} layer with the i^{th} unit of the previous layer is given by:

$$w_{ji}^l(t+1) = w_{ji}^l(t) + \beta \delta_j^l y_i^{l-1} \quad \text{(A-9)}$$

the update expression for the bias term[3] of the j^{th} unit of the l^{th} layer is given by:

$$b_j^l(t+1) = b_j^l(t) + \beta \delta_j^l \quad \text{(A-10)}$$

and δ_j^l is given by equation A-8.

[3] In general, the bias term can be treated as a weight with a unit input. This is why the derivative expressions with respect to the bias are the same as those with respect to the weights except for the term y_i^{l-1} which, in the case of the bias, is always 1.

APPENDIX B

THE XOR PROBLEM AND THE PERCEPTRON

The *exclusive*-OR or XOR problem is one of the most classical cases indicating the inability of the perceptron to classify *linearly inseparable* patterns.

The training patterns are taken from the truth table of the XOR boolean function shown in table B.1(a). Figure B.1(a) indicates that it is impossible to separate the two classes (0 and 1) by any surface or line. This implies that a perceptron will never be successful in the XOR task.

A	B	A ⊕ B
0	0	0
0	1	1
1	0	1
1	1	0

(a)

A	B	C	A ⊕ B
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

(b)

Table B.1: Two-variable and two-variable-plus-dummy XOR truth tables

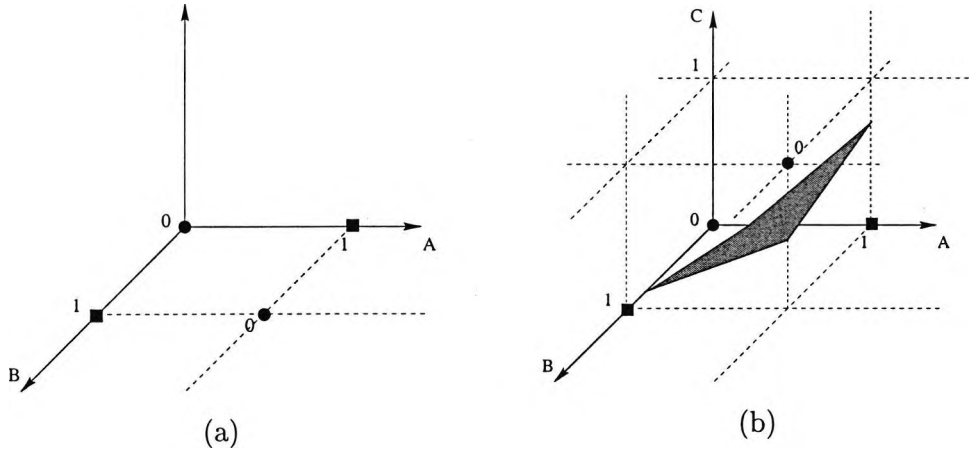

Figure B.1: A geometric representation of a two-variable and two-variable-plus-dummy XOR truth table. The gray surface indicates one possible decision surface implemented by a perceptron

However, if a dummy variable (C) is strategically added to the XOR truth table, as

shown in table B.1(b), the input patterns (A, B and C) become *linearly separable* and there exists a surface to separate them into the two classes (the gray plane in figure B.1(b)). This result is also predicted by equation 4.1 on page 34 which expresses the number of linearly separable patterns in terms of the number of input dimensions and number of training examples.

Table B.1(a) represents the case where there are 4, two-dimensional training examples. Thus, the number of linearly separable patterns[1] is:

$$\sum_{i=0}^{2} \binom{4-1}{i} = \binom{3}{0} + \binom{3}{1} + \binom{3}{2} = 7$$

(where $\binom{\alpha}{\beta}$ denotes the number of combinations of taking β items from a pool containing a total of α items and is equal to $\frac{\alpha!}{\beta!(\alpha-\beta)!}$)

We can see that out of 8 possible dichotomies of 4, two-dimensional vectors only 7 of them are linearly separable. There is a pair which is not linearly separable and this is contained in the XOR truth table.

On the other hand, table B.1(b) represents the case where there are 4, **three**-dimensional training examples (with the addition of the dummy variable, C). The number of linearly separable patterns now becomes:

$$\sum_{i=0}^{3} \binom{4-1}{i} = \binom{3}{0} + \binom{3}{1} + \binom{3}{2} + \binom{3}{3} = 8$$

Therefore, in this case, all the possible dichotomies are linearly separable.

Note that the above formula requires that all points be in *general positions*. This requires that there is no subset of 3 (the number of dimensions) or fewer points which are *linearly dependent*. Thus, the results above will not be valid when the C column of table B.1(b) is filled with only 1's or only 0's (that would have made the points linearly dependent).

Whether this trick will aid the perceptron to eventually overcome its linearity handicap is beyond doubt, but one may justifiably ask how practical and systematic it is.

[1] The total number of dichotomies, linearly separable or not, is $2^{(4-1)} = 2^3 = 8$. This should not be confused with the total number of *labellings* which is $2^N = 2^4 = 16$.

APPENDIX C

THEORETICAL FRAMEWORKS FOR LEARNING

C.1 Introduction

The *generalisation ability* of learning machines is a concept which is very difficult to be described and quantified in a theoretical framework. This section is devoted to describing the idea of *Probably Approximately Correct* (PAC) learning – a widely acceptable theoretical model of learning due to Valiant, [Valiant, 1994] – and the attempt by Vapnik and Chervonenkis to quantify the generalisation ability of (binary) classifiers.

C.2 Formulation of the learning problem

Let us now proceed in formally defining the setup for learning a sequence of *labelled examples* and formulating the learning problem.

A **learning machine**, $\mathcal{L}$, of which the family of **neural networks** with binary output is just a small subset, is capable of implementing a family of functions, called its hypothesis space:

$$\mathcal{G}^m = \{g : \mathbb{R}^m \mapsto \{0, 1\}\}$$

by changing the connectivity of its internal components and adjusting the value of its adaptive parameters.

The information presented to $\mathcal{L}$ during training consists only of a set of N labelled examples in the form of $(\underline{\mathbf{x}}, y)$ pairs. These N pairs for which the input vectors, $\underline{\mathbf{x}}_i$, have been generated by some unknown probability distribution $P(\underline{\mathbf{x}})$ on an example space $\mathcal{X}$, and the corresponding labels, y_i, are given by an, also unknown, function $h(\underline{\mathbf{x}})$, are called a **training sample** of length N:

$$\underline{\mathbf{T}} = ((\underline{\mathbf{x}}_1, y_1), (\underline{\mathbf{x}}_2, y_2), \ldots, (\underline{\mathbf{x}}_N, y_N))$$

The set $\underline{\mathbf{T}}$ is only a subset of the *concept space* $\mathcal{T}$ which consists of all possible input vectors and their respective labels as assigned by h.

Learning is the process by which $\mathcal{L}$ selects a *hypothesis*, g, from its *hypothesis space*, $\mathcal{G}^m$. It is hoped that the selected hypothesis, g, will assign each input vector, $\underline{\mathbf{x}}_i$, to

its correct label, y_i (as determined by $h(\underline{\mathbf{x}}_i)$). The selection of a particular g may be done according to many different criteria but it only depends on the training sample $\underline{\mathbf{T}}$ (since nothing else is known).

Let us define the *sample* error of the selected hypothesis g on the training set, $\underline{\mathbf{T}}$, to be the number of *disagreements* of the hypothesis with the true label of each input vector in $\underline{\mathbf{T}}$ normalised over the number of samples, N:

$$\hat{E}_{\underline{\mathbf{T}}}(g) = \frac{1}{N}|\{g(\underline{\mathbf{x}}_i) \neq y_i\}_{i=1}^{N}| \tag{C-1}$$

The *sample* error, however, is of limited importance as there are no guarantees that it is related to the performance of $\mathcal{L}$ in classifying **unknown** input vectors generated with the same probability, $P(\underline{\mathbf{x}})$, as the ones in the training sample. Persons with experience in training neural networks can confirm this fact (over-fitting is a classical example).

The need for a more *pragmatic* measure of the performance of $\mathcal{L}$ leads to the definition of another error measure defined over the *whole concept space* $\mathcal{T}$ as:

$$E(g) = P(\{(\underline{\mathbf{x}}, y) \in \mathcal{T} : g(\underline{\mathbf{x}}) \neq y\}) \tag{C-2}$$

Equations C-1 and C-2 defined above are similar, in principle, to the classical split in probability theory between the notions of observed *frequency* of occurrence of an event and the *probability* of that event occurring. Like frequency and probability, the observed and pragmatic error, above, will only be identical when the sample length, N, becomes (impractically) large (according to the law of "*large numbers*").

Finally, one more error function is introduced. This is termed as *approximation error* and refers to the *pragmatic* error (as defined in equation C-2) of the **best** hypothesis in the hypothesis space $\mathcal{G}^m$:

$$E_{apr}(\mathcal{G}^m) = \min_{g \in \mathcal{G}^m} E(g) \tag{C-3}$$

$E_{apr}(\mathcal{G}^m)$ has – at least directly – nothing to do with the ability of the learning machine to *find* a good hypothesis based on the training sample. Instead, it tells us about whether $\mathcal{L}$ (and consequently $\mathcal{G}^m$) *contains* a potentially good hypothesis and how good this is.

C.3 Probably Approximately Correct Learning

PAC learning talks about the probability that the discrepancy between the *approxima-*

tion and *pragmatic* error indicators for a learning machine, $\mathcal{L}$, is below a given accuracy level.

Formally, the PAC framework is defined as following:

DEFINITION C.1 *Given a learning machine $\mathcal{L}$ associated with a hypothesis space, $\mathcal{G}^m$, and a concept space $\mathcal{T}$ from which a training sample, $\underline{\mathbf{T}}$, of length N is selected, and for arbitrary $\epsilon, \delta \in (0,1)$, we say that $\mathcal{G}^m$ is* **learnable** *and that N is a* **sufficient sample size** *for $(\epsilon-\delta)$-learning if the probability that the discrepancy between the approximation and pragmatic error indicators is less than ϵ, is at most δ:*

$$P(|E_{apr}(\mathcal{G}^m) - E(g)| < \epsilon) < \delta \tag{C-4}$$

C.4 The Vapnik-Chervonenkis dimension

The VC dimension gives *worst-case* bounds on the generalisation ability of a learning machine. Thus, the predicted results are extremely pessimistic and are rarely confirmed in practice.

Never-the-less, the perspective of Vapnik and Chervonenkis gives a theoretical direction to the study of learning and allows us an insight into the generalisation abilities and limitations of neural networks. It also unifies previous efforts to quantify these limitations, for example the ideas behind such notions as the *order of a predicate*, [Minsky and Papert, 1969], and *linear separability* (see section 4.2.2 on page 33).

Firstly, we need a few definitions:

DEFINITION C.2 *Let the number of possible (binary) classifications of N input vectors (drawn from the example space $\mathcal{X}$) $\underline{\mathbf{X}} = (\underline{\mathbf{x}}_1, \underline{\mathbf{x}}_2, \ldots, \underline{\mathbf{x}}_N)$ by a learning machine $\mathcal{L}$ implementing the hypothesis space $\mathcal{G}^m$ be indicated by $\Pi_{\mathcal{G}^m}(\underline{\mathbf{X}})$. For a binary classifier, $\Pi_{\mathcal{G}^m}(\underline{\mathbf{X}}) \leq 2^N$.*

DEFINITION C.3 *We say that a sample $\underline{\mathbf{X}}$ of length N is* **shattered** *by the hypothesis space $\mathcal{G}^m$ or that $\mathcal{G}^m$* **shatters** *$\underline{\mathbf{X}}$ if $\Pi_{\mathcal{G}^m}(\underline{\mathbf{X}}) = 2^N$.*

DEFINITION C.4 *The* **growth function** *associated with a sample of length N and a hypothesis space $\mathcal{G}^m$ is the maximum possible number of classifications by $\mathcal{G}^m$ and is defined as:*

$$\Delta(N) = \max_{\underline{\mathbf{X}} \in \mathcal{X}} \Pi_{\mathcal{G}^m}(\underline{\mathbf{X}}) \tag{C-5}$$

The growth function is a special property of a learning machine associated with a hypothesis space $\mathcal{G}^m$ defined on an example space $\mathcal{X}$. However, it has been shown by [Cover, 1965] and also by [Vapnik and Chervonenkis, 1971] that the general form of this function either grows as 2^N or is *bounded above* by the relation:

$$\Delta(N) \leq N^{d_{VC}(\mathcal{G}^m)} + 1 \tag{C-6}$$

In particular, the growth function for small N is equal to 2^N. However, at a critical sample length called the VC dimension, $N = d_{VC}(\mathcal{G}^m)$, of the particular learning machine, the growth function grows only *polynomially* and is bounded by equation C-6 above.

Formally the VC dimension is defined as following:

DEFINITION C.5 *The VC dimension of a hypothesis space $\mathcal{G}^m$ is the maximum length of a sample, $\underline{\mathbf{X}}$ shattered by $\mathcal{G}^m$ and is given by:*

$$d_{VC}(\mathcal{G}^m) = \max \{N : \Delta(N) = 2^N\} \tag{C-7}$$

where we take the maximum to be infinite if the set is unbounded.

The main contribution of Vapnik and Chervonenkis to the study of learning is the following theorem:

THEOREM C.1 *For any hypothesis space $\mathcal{G}^m$, the condition that $\mathcal{G}^m$ has finite VC dimension is both necessary and sufficient for potential learnability. Thus potentially learnable hypothesis spaces are those of finite VC dimension.*

C.5 Some generalisation bounds

A first result is a theorem from [Vapnik and Chervonenkis, 1971] which gives an *upper bound* on the probability of the discrepancy between the *sample* and *pragmatic* error indicators (see equations C-1 and C-2). Given $\epsilon \in (0, 1)$ and a sample $\underline{\mathbf{T}}$, the following holds:

$$P(\max_{g \in \mathcal{G}^m} |\hat{E}_{\underline{\mathbf{T}}}(g) - E(g)| > \epsilon) \leq 4\Delta(2N)e^{-\frac{\epsilon^2 N}{8}} \tag{C-8}$$

The above equation gives an upper bound on the difference between our estimated error (e.g. the sample error indicator, $\hat{E}_{\underline{\mathbf{T}}}(g)$) and the **true** generalisation performance indicated by $E(g)$. In order to make this difference as small as possible, for a given

accuracy ϵ, we need to make the right hand side of the equation small by increasing N **if and only if** we are sure that the growth function is in the polynomial growth region (e.g. it grows at most as $N^{d_{VC}(\mathcal{G}^m)} + 1$ and not as 2^N. If $\Delta(2N)$ grows as 2^N, increasing N will increase the right hand side of the expression (since the base of the natural logarithms, e, is less than 2).

In [Baum and Haussler, 1989] multi-layer, feed-forward neural networks with **threshold** activation functions are considered. An upper bound of the VC dimension of such a network comprising of $|\underline{\mathbf{W}}|$ weights and M neurons, is given by:

$$d_{VC} \leq 2|W|\log_2(eM) \tag{C-9}$$

where e is the base of natural logarithms.

Furthermore, they showed that if a sample size given by:

$$N \leq \frac{|W|}{\epsilon}\log_2(\frac{M}{\epsilon}) \tag{C-10}$$

can be learned by the network with an accuracy of $1 - \epsilon/2$ (where $0 < \epsilon \leq 1/8$), then there is a high probability that the network will classify unknown inputs with an accuracy of at least $1 - \epsilon$.

Finally, for a large two-layer network, they derived the approximate rule of thumb that the minimum sample length to guarantee correct classification of $1 - \epsilon$ unknown inputs must be $N \approx |W|/\epsilon$. For example, if an accuracy of 90% is required, then the number of training examples must be approximately equal to ten times the number of weights!

C.6 Support Vector Machines

C.6.1 Introduction

The Support Vector (SV) algorithm is a non-linear generalisation of the *Generalised Portrait* algorithm developed in the Soviet Union in the sixties, [Vapnik and Lerner, 1963] and [Vapnik and Chervonenkis, 1964]. As expected, the SV algorithm is grounded in the framework of *statistical learning theory* which has been developed by Vapnik, Chervonenkis and others over the last thirty years, [Vapnik and Chervonenkis, 1974], [Vapnik, 1979], [Vapnik, 1995], etc.

The present form of the SV algorithm was developed at AT&T Bell Labs by Vapnik and colleagues, [V.N. Vapnik and Smola, 1997], and is commonly referred to by the term Support Vector Machines (SVM).

C.6.2 SVM basics

SVM is a new way of training polynomial, neural network and Radial Basis function learning machines for either classification or regression tasks[1]. The novelty of SVM lies in the fact that training is not based on minimising the *sample* or *empirical error* (see equation C-1), like so many neural network training algorithms do and are usually faced with one of the many guises of the same problem, namely the *bias-variance trade-off* or *over-fitting*, [Geman and Bienenstock, 1992]. Instead, SVM attempts to minimise the upper bound of the *g*eneralisation error.

Another difference between SVM and more conventional optimisation methods is that SVM choose the most suitable function space for the task out of a **pool** of function spaces. Most optimisation methods attempt to minimise the sample error for some **fixed** function space.

SVM use a different induction principle to minimise the upper bound of the *g*eneralisation error with the aid of Structural Risk Minimisation (SRM). SRM's goal is to choose among various learning machines with different learning capacities the one which yields a good trade-off between low empirical risk and small capacity. In order to achieve this, one sets up a hierarchy of function spaces and chooses the space with the smallest complexity that can attain the desired sample error.

SVM and SRM are based on the existence of the family of bounds governing the relation between capacity of a learning machine and its performance – the result of Vapnik's previous work on statistical learning theory and the VC dimension, [Vapnik, 1979].

In practice, training the SVM consists of minimising a cost function with a number of constraints. Problems such as training the SVM fall into the category of *standard constrained Quadratic Programming* (QP) and appear to be simple and straightforward. However, it is well known that finite numerical precision can cause QP solvers to give non-optimum solutions, [Burges, 1998]. Also, the complexity of the QP solver is highly dependent on the training data, its size and its dimensions. Unfortunately, there is no known method to define the data complexity analytically.

[1] E.g. with binary/discrete or continuous outputs.

APPENDIX D

FFNN, THE STONE-WEIERSTRASS THEOREM AND THE UNIVERSAL FUNCTION APPROXIMATION PROPERTY

D.1 Introduction

The *Stone-Weierstrass theorem* has been used by many theoreticians in the field of neural networks to prove that the family of FFNN can approximate arbitrarily well any real continuous function over a compact set (see for example [Hornik, 1991] and [Hornik et al., 1992]).

In this section, we present a proof along the same lines. The definitions of Chapter 3 regarding the family of sigmoids (definition 3.1 on page 22) and the family of affine functions (definition 3.2 on page 22) will be used. For the sake of clarity, the definition of the family of all FFNN functions (see definition 3.2 on page 23) will be re-written as a single output, a single hidden layer of q units, a single linear output with zero bias, neural network:

$$\mathcal{G}^n = \{g_n : \mathbb{R}^n \mapsto \mathbb{R} | g_n(\underline{\mathbf{x}}) = \sum_{i=1}^{q} \beta_i \sigma(A_i(\underline{\mathbf{x}})), \quad \beta_i \in \mathbb{R}, \underline{\mathbf{x}} \in \mathbb{R}^n, A_i \in \mathcal{A}^n, \sigma \in \mathcal{S}\} \tag{D-1}$$

D.2 Metric spaces

In the following definition the notions of a **metric** and a **metric space** are introduced (see [Rudin, 1964] and [Hornik et al., 1992]):

DEFINITION D.6 *A metric on a set* $\underline{\mathbf{X}}$ *is a function* $\rho : \underline{\mathbf{X}} \times \underline{\mathbf{X}} \mapsto \mathbb{R}$ *which satisfies the following three conditions*:

1. **Positivity**: $\rho(x, y) > 0$ *unless* $x = y$ *in which case* $\rho(x, y) = 0$
2. **Symmetry**: $\rho(x, y) = \rho(y, x)$
3. **Triangle inequality**: $\rho(x, y) \leq \rho(x, z) + \rho(z, y)$

where $x, y, z \in \underline{\mathbf{X}}$. *A set* $\underline{\mathbf{X}}$, *equipped with a metric,* $\rho(\cdot, \cdot)$ *is called a metric space, denoted by* $(\underline{\mathbf{X}}, \rho)$.

Here, we are introducing the terms ρ-**dense** and **uniform convergence**

DEFINITION D.7 *A subset* $\underline{\mathbf{M}}$ *of a metric space* $(\underline{\mathbf{X}}, \rho)$ *is* ρ-**dense** *in a subset* $\underline{\mathbf{T}}$ *if for every* $\varepsilon > 0$ *and for every* $\tau \in \underline{\mathbf{T}}$, *there is a* $\mu \in \underline{\mathbf{M}}$ *such that* $\rho(\mu, \tau) < \varepsilon$.

EXAMPLE D.1 *Let* $\mathcal{F}^n$ *be the set of bounded functions from* $\mathbb{R}^n \mapsto \mathbb{R}$ *and* $f, g \in \mathcal{F}^n$. *Then,*

$$\rho(f, g) \equiv sup_{x \in \mathbb{R}^n} |f(x) - g(x)|$$

is a metric on $\mathcal{F}^n$.

The closeness of a **class** or **family of functions** to another class is described by the concept of **denseness**:

DEFINITION D.8 $\underline{\mathbf{S}} \subseteq \mathcal{F}^n$ *is* **uniformly dense on compacta** *if for all compact sets* $\underline{\mathbf{K}}$, $\underline{\mathbf{S}}$ *is* ρ_K*-dense in* $\mathcal{F}^n$ *where* $\rho_K(f, g) = sup_{x \in \underline{\mathbf{K}}} |f(x) - g(x)|$.

D.3 The Stone-Weierstrass theorem

Below are some definitions which will be used in the *Stone-Weierstrass theorem* following.

DEFINITION D.9 *A family* $\mathcal{F}$ *of real functions defined on a set* $\underline{\mathbf{E}}$ *is an* algebra, *if* $\mathcal{F}$ *is closed under:*

1. **addition**: $x + y \in \mathcal{F}$
2. **multiplication**: $x \cdot y \in \mathcal{F}$
3. **scalar multiplication**: $\alpha \cdot x \in \mathcal{F}$

for $x, y \in \mathcal{F}$ *and* α *scalar.*

DEFINITION D.10 *A family of functions* $\mathcal{F}$ *separates points on a set* $\underline{\mathbf{E}}$ *if for every* $x_1, x_2 \in \underline{\mathbf{E}}$ *and* $x_1 \neq x_2$, *there exists a function* $f \in \mathcal{F}$ *such that* $f(x_1) \neq f(x_2)$.

The above statement implies that there is at least one function in $\mathcal{F}$ that "knows" the difference between any two points of $\underline{\mathbf{E}}$ and, thus, "treats" (maps) them differently.

DEFINITION D.11 *A family of functions* $\mathcal{F}$ *vanishes at no point of* $\underline{\mathbf{E}}$ *if for each* $x \in \underline{\mathbf{E}}$ *there exists a function* $f \in \mathcal{F}$ *such that* $f(x) \neq 0$.

The *Stone-Weierstrass theorem* may be used in determining whether a family of functions over a (compact) set $\underline{\mathbf{K}}$ can approximate arbitrarily well any continuous function. It is, therefore, the key tool in proving that a certain neural network architecture (implementing a certain family of transfer functions) is a *universal function approximator.*

THEOREM D.2 *Let $\mathcal{F}$ be an algebra of real continuous functions on a compact set $\underline{\mathbf{K}}$. If $\mathcal{F}$ separates points on $\underline{\mathbf{K}}$ and if $\mathcal{F}$ vanishes at no point on $\underline{\mathbf{K}}$, then the uniform closure $\underline{\mathbf{B}}$ of $\mathcal{F}$ consists of all real continuous functions on $\underline{\mathbf{K}}$ (i.e. $\mathcal{F}$ is ρ_K-dense in the space of real continuous functions on $\underline{\mathbf{K}}$).*

D.4 FFNN are Universal Function Approximators

THEOREM D.3 *Any feed-forward neural network implementing the family of functions $\mathcal{G}^n$ (see equation D-1) and whose inputs belong to a compact set $\underline{\mathbf{K}} \subset \mathbb{R}^n$, is uniformly dense on compacta in the set of all continuous functions in $\mathbb{R}^n$ (mapping $\mathbb{R}^n$ to $\mathbb{R}$).*

Proof

- *$\mathcal{G}^n$ is an algebra of functions because it satisfies the three conditions set in definition D.9, namely:*

  1. **addition**: *The sum $g_n(\underline{\mathbf{x}}) + h_n(\underline{\mathbf{x}})$ for $g_n, h_n \in \mathcal{G}^n$, $\underline{\mathbf{x}} \in \underline{\mathbf{K}}$ and $|\underline{\mathbf{K}}| = n$, belongs to $\mathcal{G}^n$ since:*

$$\sum_{i=1}^{q} \beta_i \sigma(A_i(\underline{\mathbf{x}})) + \sum_{j=1}^{q'} \beta'_j \sigma(A'_j(\underline{\mathbf{x}})) = \sum_{k=1}^{q+q'} \beta''_k \sigma(A''_k(\underline{\mathbf{x}}))$$

$$\text{where } \beta''_k = \begin{cases} \beta_k, & \text{for } 1 \leq k \leq q \\ \beta_{q+q'-k}, & \text{for } q < k \leq q+q' \end{cases}$$

$$\text{and } A''_k(\underline{\mathbf{x}}) = \begin{cases} A_k(\underline{\mathbf{x}}), & \text{for } 1 \leq k \leq q \\ A'_{q+q'-k}(\underline{\mathbf{x}}), & \text{for } q < k \leq q+q' \end{cases}$$

  2. **multiplication**: *The product $g_n(\underline{\mathbf{x}}) \cdot h_n(\underline{\mathbf{x}})$ for $g_n, h_n \in \mathcal{G}^n$, and $\underline{\mathbf{x}} \in \underline{\mathbf{K}}$, belongs to $\mathcal{G}^n$ since:*

$$\sum_{i=1}^{q} \beta_i \sigma(A_i(\underline{\mathbf{x}})) \cdot \sum_{j=1}^{q'} \beta'_j \sigma(A'_j(\underline{\mathbf{x}})) = \sum_{i,j}^{q \times q'} \beta_i \beta'_j \sigma(A_i(\underline{\mathbf{x}})) \sigma(A'_j(\underline{\mathbf{x}}))$$

The condition will be satisfied if the product of the two sigmoids is still the same sigmoid. [Hornik et al., 1992] *solve this problem by using the "cosine squasher",* [Gallant and White, 1992]:

$$\sigma_{cos}(x) = \begin{cases} 0 & for \ -\infty < x \leq -\frac{\pi}{2} \\ \frac{cos(x+\frac{3\pi}{2})+1}{2} & for \ -\frac{\pi}{2} < x \leq \frac{\pi}{2} \\ 1 & for \ \frac{\pi}{2} < x < \infty \end{cases} \tag{D-2}$$

3. **scalar multiplication**: *The product* $\alpha g_n(\underline{\mathbf{x}})$, *for* $g_n \in \mathcal{G}^n$, *and* α *scalar, belongs to* $\mathcal{G}^n$ *since*:

$$\alpha \sum_{i=1}^{q} \beta_i \sigma(A_i(\underline{\mathbf{x}})) = \sum_{i=1}^{q} \beta_i' \sigma(A_i(\underline{\mathbf{x}}))$$

- $\mathcal{G}^n$ *separates points on* $\underline{\mathbf{K}}$, *e.g. if* $\underline{\mathbf{x}}_1, \underline{\mathbf{x}}_2 \in \underline{\mathbf{K}}$ *and* $\underline{\mathbf{x}}_1 \neq \underline{\mathbf{x}}_2$ *then there exists at least one* $g_n \in \mathcal{G}^n$ *such that* $g_n(\underline{\mathbf{x}}_1) \neq g_n(\underline{\mathbf{x}}_2)$. *This is true when* $\sigma(A(\underline{\mathbf{x}}_1)) \neq \sigma(A(\underline{\mathbf{x}}_2))$ *e.g. when* $A(\underline{\mathbf{x}}_1) \neq A(\underline{\mathbf{x}}_2)$ *(due to monotonicity of* σ*). To ensure* $A(\underline{\mathbf{x}}_1) \neq A(\underline{\mathbf{x}}_2)$, *choose* $A \in \mathcal{A}^n$ *so that for* $\lambda_1, \lambda_2 \in \mathbb{R}$, $\lambda_1 \neq \lambda_2$, *we have* $A(\underline{\mathbf{x}}_1) = \lambda_1$ *and* $A(\underline{\mathbf{x}}_2) = \lambda_2$. *This will ensure that* $\mathcal{G}^n$ *separates points on* $\underline{\mathbf{K}}$.

- $\mathcal{G}^n$ *vanishes at no point of* $\underline{\mathbf{K}}$, *e.g. for each* $\underline{\mathbf{x}} \in \underline{\mathbf{K}}$ *there exists at least one* $g_n \in \mathcal{G}^n$ *such that* $g_n(\underline{\mathbf{x}}) \neq 0$. *To see this, choose* $\lambda \in \mathbb{R}$ *such that* $\sigma(\lambda) \neq 0$. *Then construct* $A \in \mathcal{A}^n$ *such that* $A(\underline{\mathbf{x}}) = \lambda$. *This is possible even if* $\underline{\mathbf{x}} = \emptyset$ *because of the presence of the constant term* b *in the affine functions equation. This ensures that* $\mathcal{G}^n$ *vanishes at no point of* $\underline{\mathbf{K}}$.

APPENDIX E

TESTING FOR STATISTICAL SIGNIFICANCE

E.1 Introduction

In the comparison and interpretation of various experimental results, it must be remembered that these are mere samples drawn from a much bigger population, by the experimental procedure. The degree in which the observations reflect the actual population's properties (e.g. *mean* and *variance*) have to be estimated and the conclusions be corrected approximately. This procedure is particularly important when we have only a small number of samples available – e.g. when the experiments were repeated for 40, 50 or less times.

In such cases, a test for the *statistical significance* of the experimental results (the samples) is required in order to see how safe it is to generalise upon the whole population. A framework for testing for *statistical significance* of experimental results is provided by the *Small Sampling Theory*, [Spiegel, 1971] and [Brookes and Dick, 1963]. This framework is as follows:

1. **Construct the null hypothesis** related to the objectives of the experiments. For example, one of the objectives of the experiments described in chapter 6 was to compare the generalisation ability of entity networks and single FFNN. In this case, the *null hypothesis* can be "*for any number of inputs, the mean approximation error of the entity does not differ significantly from that of the equivalent single* FFNN". Another objective of the experiments was to compare the training inconsistency – e.g. the variation in the sample error – of the networks. An appropriate *null hypothesis* can be "*for any number of inputs, the variance of the approximation error of the entity does not differ significantly from that of the equivalent single* FFNN".

2. **Construct the alternative hypothesis** which will be adopted in the case of the *null hypothesis* being rejected. There are situations where it is enough to have an *alternative hypothesis* which simply states that there are significant differences between the means or variances of the two populations. For example, "*for any number of inputs, the mean approximation error of the entity differs significantly*

from that of the equivalent single FFNN". In this case we will use a *two-tailed test.* On the other hand, if it is desirable to identify the *direction* of the difference in the examined quantities, we will have to use a *one-tailed test.* In this case the alternative hypothesis can be: "*for any number of inputs, the mean approximation error of the entity is lower than that of the equivalent single* FFNN".

3. **Choose the level of significance**. The *level of significance* is defined as the probability of making a *Type I* error: that is, reject the *null hypothesis* when it should have been accepted. It usually is safe to assume a *significance level* of 5 % or less, [Brookes and Dick, 1963].

4. **Choose a statistical significance test**. There exist a large number of such tests (for example the *t-test*, the *F-test*, the χ^2-*test*, etc.) each appropriate for a certain type of experimental results, number of samples, objectives and selected hypotheses.

5. **Estimate the statistic**. This is calculated using the sample data's *mean*, *standard deviation*, *number of samples* and the appropriate formulae specific to the type of each test.

6. **Accept or reject the null hypothesis** depending on whether the *statistic* calculated earlier is lower or higher than the respective entry (i.e. *critical value*) in the *statistical table* containing the distribution associated with the chosen significance test and the *degrees of freedom.* The latter is a function of the number of observations.

E.2 Testing the difference between two populations' means: the *t-test*

Two samples of n_1 and n_2 observations with means m_1 and m_2 and standard deviations s_1 and s_2 are drawn independently from two populations with means μ_1 and μ_2 and standard deviations σ_1 and σ_2. In general, the populations' properties are unknown while the samples' properties are known.

In order to test the difference between the population *means*, proceed as follows:

1. Calculate the *t-statistic* which is the ratio of the difference of the samples' means over a normalised expression of their standard deviations. The exact formula is given below:

$$t = \frac{m_1 - m_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}} \qquad \text{(E-1)}$$

2. Calculate the *degrees of freedom*, ν, of the observations using the following formula:

$$\nu = \begin{cases} n_1 + n_2 - 2 & \text{if } n_1 = n_2 \\ \frac{(s_1^2/n_1 + s_2^2/n_2)^2}{\frac{(s_1^2/n_1)^2}{n_1-1} + \frac{(s_2^2/n_2)^2}{n_2-1}} & \text{if } n_1 \neq n_2 \end{cases} \quad \text{(E-2)}$$

3. Consult a *t-distribution* table (available in most statistics textbooks, see for example [Spiegel, 1971, p.344]) and find the *critical value*, $t_{a,\nu}$, corresponding to the observations' *degrees of freedom*, ν, and the chosen *level of significance*, a.

4. Depending whether a *one-tailed* or a *two-tailed* test is required, the *null hypothesis* is rejected at the *chosen significance level* and, thus, the alternative hypothesis is accepted, if:

   - *one-tailed test*: $t > t_{a,\nu}$
   - *two-tailed test*: $|t| > t_{a/2,\nu}$

E.3 Testing the ratio of two populations' variances: the *F-test*

Two samples of n_1 and n_2 observations with means m_1 and m_2 and standard deviations s_1 and s_2 are drawn from two populations with means μ_1 and μ_2 and standard deviations σ_1 and σ_2.

In order to find out whether the difference between the *variances* of the two samples is significant, proceed as follows:

1. Calculate the *F-statistic* which is the ratio of the variances of the two samples. The exact formula is given below:

$$f = \frac{s_1^2}{s_2^2} \quad \text{(E-3)}$$

2. The *degrees of freedom* associated with the statistic mentioned above are calculated as follows:

$$\text{for the numerator,} \quad \nu_1 = n_1 - 1$$
$$\text{for the denominator,} \quad \nu_2 = n_2 - 1$$

3. Consult a *F-distribution* table (available in most statistics textbooks, see for example [Hinton, 1995, p.308]) and find the *critical value*, f_{a,ν_1,ν_2}, corresponding to the *degrees of freedom* ν_1 and ν_2 and the chosen *level of significance*, a.

4. Depending whether a *one-tailed* or a *two-tailed* test is required, the *null hypothesis* is accepted with the *chosen significance level*, if:

   - *one-tailed test*: $f < f_{a,\nu_1,\nu_2}$
   - *two-tailed test*: $f < f_{a/2,\nu_1,\nu_2}$

Following is an example of using the *t-test* and *F-test* to assess the statistical significance of the results obtained in chapter 6:

EXAMPLE E.2 *A single* FFNN *and a* C_1 *entity with the same number of inputs (*700*) were trained with the same data for* 40 *and* 50 *times respectively. The approximation error of each network was recorded each time. In the end, the mean approximation error (over the* 40 *training attempts) of the single* FFNN *was* $m_1 = 0.1185$ *and its standard deviation* $s_1 = 0.042$. *For the entity, the mean was* $m_2 = 0.0968$ *and the standard deviation was* $s_2 = 0.0115$. *In view of the fact that the sample means and standard deviations differ does not necessarily imply that the population means and standard deviations will differ significantly, establish the following*:

1. **Can we conclude that the single** FFNN **is a worst generaliser (e.g. higher approximation error) than the entity at a 5 % significance level?**

2. **Can we conclude that the single** FFNN**'s variation of the approximation error is greater than that of the entity at a 5 % significance level?**

Solution: *Let* μ_1, μ_2, σ_1 *and* σ_2 *denote population mean and standard deviation of the approximation errors for the single* FFNN *and the entity respectively.*

1. *We have to decide between the hypotheses*:

   - *null hypothesis,* $H_0 : \mu_1 = \mu_2$, *the means do not differ significantly – the two networks have the same generalisation ability,*
   - *alternatively,* $H_1 : \mu_1 > \mu_2$, *the single* FFNN *has a higher mean approximation error than the entity and, consequently, is a worse generaliser.*

   We will use the one-tailed t-test because our hypotheses are concerned with the difference and the direction of the difference between the two populations' means. The t-statistic is calculated as:

$$t = \frac{0.1185 - 0.0968}{\sqrt{\frac{0.042^2}{40} + \frac{0.0115^2}{50}}} = 3.17$$

On the basis of a one-tailed test at a 5 % level of significance and with

$$\nu = \frac{(0.042^2/40 + 0.0115^2/50)^2}{\frac{(0.042^2/40)^2}{39} + \frac{(0.0115^2/50)^2}{49}} \approx 43$$

degrees of freedom, the t-distribution table entry indicates a critical value[1] *of* $t_{5\%,40} = 1.684$. *Because* $t > t_{5\%,40}$ *(e.g.* $3.17 > 2.684$*), we have to reject the null hypothesis and accept* H_1 *– the entity is a better generaliser than the single* FFNN.

2. *We have to decide between the hypotheses:*

   - *null hypothesis,* $H_0 : \sigma_1^2 = \sigma_2^2$*, the variances do not differ significantly,*
   - *alternatively,* $H_1 : \sigma_1^2 > \sigma_2^2$*, the single* FFNN*'s approximation error has a higher variance than the entity's.*

   Because we want to check hypotheses regarding the variances of the populations we will use the F-test (again, the one-tailed variation because we are interested in the direction of this difference). The F-statistic is calculated as:

$$f = \frac{0.042^2}{0.0115^2} = 13.34$$

On the basis of a one-tailed test at a 5 % level of significance and with $\nu_1 = 40 - 1 = 39$ *and* $\nu_2 = 50 - 1 = 49$ *degrees of freedom for the single* FFNN *and entity observations respectively, the F-distribution table entry indicates a critical value of* $f_{5\%,39,49} \rightarrow f_{5\%,30,40} = 1.174$. *Because* $f > f_{5\%,30,40}$ *(e.g.* $13.34 > 1.174$*), we have to reject the null hypothesis and accept* H_1 *– the entity's approximation error has a lower variance.*

[1] Note that common *t-distribution* tables might not contain the critical value for the calculated degrees of freedom. For example, the table might contain entries for 40 and 50 degrees of freedom, but not for the required 43. In this case we may use linear interpolation between the two available entries in order to obtain the critical value for the 43 degrees of freedom. Alternatively, use the next lowest value available (e.g. 40).

APPENDIX F

THE *np* SCRIPT LANGUAGE AND INTERPRETER

F.1 Overview of the *np* interpreter

F.1.1 Introduction

np is an interpreter of the *np* script language, a set of simple commands which allow a user to create, train and test single feed-forward neural networks (FFNN), as well as entities of FFNN.

```
TRAIN_DATA = ProduceVectoredDataSet {
                NumInputs = 100;
                NumOutputs = 1;
                NumLines = 60;
                Y = Levy6;
}
ENTITY = CreateEntity {
                NumInputs = 100;
                EntityType = FFNN;
                EntityClass = 1;
                MinNumInputs = 10;
                MaxNumInputs = 20;
                ConfFile = EntityConfig;
                Seed = 1974;
}
TrainEntity {
                Obj = ENTITY;
                InpFileObj = TRAIN_DATA;
                Iters = 1000;
}
SINGLE = CreateSingle {
                Arch = 100 140 20 1;
                SingleType = FFNN;
                Weights = single;
}
TrainSingle {
                Obj = SINGLE;
                Iters = 2000;
                InpFileObj = TRAIN_DATA;
}
$
```

A sample *np* script used to train a single FFNN and an entity with 100 inputs and a single output.

np scripts are composed of the following five elements:

1. **Instruction Identifier**: This is simply the name of an instruction. An instruction (e.g *CreateSingle*) will:

(a) initiate an action according to the **parameters** specified (e.g. *TrainEntity*), or,

(b) create an **object** which will hold some or all of the **parameters** specified (e.g *CreateSingle*), or,

(c) do both 1(a) and 1(b) (e.g. *ProduceVectoredDataSet*).

2. **Object Identifier**: When an instruction does either 1(b) or 1(c), an **object** is created and referenced by an identifier (e.g TRAIN_DATA). There are three classes of objects:

   (a) **File Object**: An object which has been created by an *OpenFileObject*, or equivalent[1], command. A **File object** represents either a local file or data which will be/has been received from the network. See appendix F.7 for more details.

   (b) **Single Object**: This object represents a single unit (be it a FFNN or an ADALINE etc.) and holds data relevant to it. For example, it holds the architecture of the network.

   (c) **Entity Object**: This object represents an entity of single units (again FFNN or ADALINE etc.).

3. **Parameter Identifier**: Certain **instructions** require certain parameters to be set. The parameter identifiers refer to these. For example, the parameter `Arch` refers to the architecture of the unit to be created with the *CreateSingle* instruction).

4. **Parameter Values**: The value that a parameter takes. It is the text between the symbols '=' and ';'. For example **100 140 20 1** in the architecture parameter.

5. **Separators** (#{} =;$): These are symbols which separate different instructions, objects, parameters and values.

   $ is the **program terminator** symbol. Whatever there is after that symbol is ignored.

   # is the **comment** symbol. Whatever there is between this and the end-of-line symbols is ignored.

   {} are the **start** and **end-of-block** symbols. A block always (even if empty) follows an instruction and within it all the relevant parameters should be specified.

[1] For example the *MergeObjects* instruction returns a **file object**.

= is the **assignment** symbol.

; denotes the end of the **parameter value.**

, is used as a separator of a list of objects.

.. is used to indicate a range in a list. For example 1..3 is expanded to 1, 2, 3.

F.1.2 Parallel execution

A useful feature of *np* is the parallel execution of lengthy training processes. It is only applicable to those ENTITY architectures for which decomposition of the training process into independent sub-tasks is possible. So far, such a decomposition has meaning only with the first and second FFNN entity classes.

np will proceed to parallel training if the `Hosts` field of the *TrainEntity* contains a list of remote machines for which unix's **rsh** and **rcp** (remote shell and remote copy) are enabled for that particular user. The local host will divide the process of training into as many sub-tasks as the number of hosts in the `Hosts` field, distribute it to the specified remote hosts and suspend the interpetation of the script. When **all** hosts have completed their tasks, the results are sent back to the local host which resumes the interpretation of the script.

F.1.3 Running *np*

Usage:

np [**-log** *logfile*] [**-(no)bell**] [**-syntax**] [**-template**] [**-h**] [**-silent**] [**-instruction** *ins*] *file*

Invoking *np* is very easy. Given an *np* script file, let us call it *myscript.np*, do:

% ***np*** *myscript.np* ¬

np will first parse the file and then process each instruction. Messages about the current instruction being processed are sent to the **stdout**. You can send most of them to a log file by specifying the '**-log** *mylogfile*' option **before** the input script name, i.e.:

% ***np*** -log *mylogfile myscript.np* ¬

By default, a bell will sound whenever an error occurs. You can avoid this by specifying the '**-nobell**' option.

The command line option '**-syntax**', (always before the input file), tells *np* that it need not process the input file but rather to check whether it is syntactically correct.

Another option is '**-template**'. This option tells ***np*** to treat the input file as a *template* file. A template file is an ***np*** file for which some of its parameters are set to the statement **ASK**(*question*). For each of these statements and unless it has been encountered before, ***np*** will ask you the *question* and will substitute the whole **ASK** statement with your answer. The final ***np*** file with all the **ASK** statements substituted is printed on the **stdout**.

Finally, the options '**-help**', '**-h**' and '**-usage** will cause ***np*** (irrespective of any other specified options) to print a short usage message and exit. The option '**-instruction**' followed by the name of an instruction will print a short description about it and all the parameters which it requires. Use the keyword '*all*' in order to print **all** the instructions in the ***np*** system.

F.2 *np* instructions: general object interaction

F.2.1 ExtractColumnsFromObject

A number of file objects, in which data is arranged in columns, can be used, in conjunction with this command, in order to extract some of their columns and form a new file object.

An example situation is the following: We have two training files composed of 5 columns each (3 input columns and 2 output columns). We need to construct a third data file which will train a network with 6 inputs and 2 outputs, say, X_1, X_2, X_3, X_4, X_5, X_6 and Y_2, Y_3. The instruction *ExtractColumnsFromObject* takes a sequence of:

FileObjectName[*Lists*]

separated by colons and appends them side by side to the output file. For example, in our situation, we will give the following *Columns* specification:

Columns = Obj_1[1 TO 3]:Obj_2[1 TO 3]:Obj_1[4]:Obj_2[5]

IDENTIFIER = *ExtractColumnsFromObjects* {

`DefaultFileObj` = { The **file object** from some or all of the *Column* items in the `Columns` description may be omitted if this parameter is defined. }
Optional

`OutFileName` = { The name of the local file or channel where the result should be sent to. If omitted, the file name will be constructed as **temp_Identifier**. See appendix F.7 for channel categories. }
Optional

`Columns`
Required
= { A colon separated collection of *Columns*. A *Column* (see definition below) consists of *Lists* (see definition below and in appendix F.8) enclosed in **square brackets** and, optionally, preceded by the identifier of an existing file object. If no file object is mentioned for a given *Column* then the `DefaultFileObj` (see below) will be used. A *Column* is formally defined as:

Column = [*Lists*] | FileObject[*Lists*]
Columns = Column | Columns:Column

See appendix F.8 for more information about *Lists*. }

}

F.2.2 MergeObjects

Merging can be done either as a simple concatenation (sequential) of the file objects or in a side-by-side (parallel) fashion.

IDENTIFIER = *MergeObjects* {

`InpFileObj`
Required
= { This must be a comma separated list of at least two **file objects**. The order of merging is the same as the order of the **file objects** in this field. }

`MergeMethod`
Optional
= { It could either be **Parallel** where the contents of the file objects are all put side-by-side, or **Sequential** where the operation is a simple concatenation of the specified file objects. }

`OutFileName`
Optional
= { The name of the local file or channel where the result should be sent to. If omitted, the file name will be constructed as **temp_Identifier**. See appendix F.7 for channel categories. }

}

The following example merges 3 files in parallel and mails the result to the user:

```
TRAIN_DATA = MergeObjects {
    InpFileObj = FILE_1, FILE_2, FILE_3;
    MergeMethod = Parallel;
    OutFileName = MAIL<someone@somewhere.ac.uk, The result of merging the 3 files>;
}
```

F.2.3 ColumnsArithmetic

This command performs **row-wise** and **column-wise** operations of the input file objects. For example, there are three file objects (**A**, **B** and **C**) associated with files whose

data is formatted in columns. One would like to evaluate the following operation for each row of the files:

$$sqrt(abs(\mathbf{A}_i[3] \text{ - } log((\mathbf{B}_i[2]+\mathbf{C}_i[5])/\mathbf{A}_i[1])))$$

where $\mathbf{A}_i$[j], $\mathbf{B}_i$[j] and $\mathbf{C}_i$[j] are the values of the i^{th} row, j^{th} column of the three files, i runs from the first to the last row (or to the shortest row of the three) and j has the value specified between square brackets.

Then, one might like to calculate the *average* value of the resultant column. All these can be done with the *ColumnsArithmetic* instruction. Here is its syntax,

IDENTIFIER = *ColumnsArithmetic* {

`RowExpr` (Required) = This is an arithmetic expression where the following symbols can be used and have their usual meaning: +-/*(), as well as some user-defined (look for the '**# ADD HERE**' string in **NeuralLib.pl**) functions like *abs* and all *awk* built-in mathematical functions (*sqrt*, *log*, etc.). The operands are defined formally as:

Column = [*Lists*] | FileObject[*Lists*]

It is the same as the `Column` description in the *ExtractColumns-FromObjects* instruction with the exception that each of the *Lists* items must specify a SINGLE column.

`DefaultFileObj` (Optional) = The **file object** from some or all of the *Column* items in the `Expression` description may be omitted if this parameter is defined.

`ColExpr` (Optional) = A comma separated list of operations can be specified here so that they be applied to the resultant column. These operations include user-defined functions such as *average* and *sum*.

`OutFileName` (Optional) = The name of the local file or channel where the result should be sent to. If omitted, the file name will be constructed as **temp_Identifier**. See appendix F.7 for channel categories.

}

The following will produce the desired results for the problem introduced at the beginning of this section:

```
TRAIN_DATA = ColumnsArithmetic {
    Row = sqrt(abs(A[3] - log((B[2]+C[5])/A[1])));
    Col = average;
}
```

The *ColumnsArithmetic* instruction uses *awk*. However, functions like *abs* or *average* are not known to *awk*. *abs* is defined as a function at the beginning of the *awk* program, every time it is called. *average* is a bit more complicated because it operates on all the row elements. See the perl library **NeuralLib.pl** at the subroutine *ColumnsArithmetic*. There is a section where one can define functions and functions in *awk* are fairly simple. However, operations like *average* are a bit more difficult[2]. Consult you local *awk* man page for more information and a list of the built-in functions.

Finally, note that *awk* has a limitation of the input variables it can use. The maximum number of columns *awk* can handle is set at compile time. In any case, if you have any problems it is better to get *gawk*[3] from GNU's archives and compile it to suit your needs.

F.2.4 Deletion of objects

An object can be deleted at two levels. The first level is the computer memory. Deleting an object from the memory still leaves files in the hard-storage device of the computer. *Unlink* refers exactly to this second level of deletion and will include **all** the files associated with this object (apart from `InpFileObjs`).

DeleteObjects{

`Obj` (Required) = { A comma separated list of objects (all three types of objects are allowed). }

`Unlink` (Optional) = { It can be **Yes** or **No**. **No** is the default. **Yes** will delete all the files associated with **all** the objects specified. No `InpFileObjs` will be deleted however. }

}

F.3 *mp* instructions: produce and/or format data sets

There are three commands relevant to producing and/or formatting data sets. One is for vectored data of any sort, created by a function (artificial) or read from a file and then formatted to so many inputs and outputs. The other two are for formatting time series data read from a file and constructing a data set made of samples from various images.

[2] Actually the *average* operation was copied from the man page.

[3] *mp* uses *gawk* and not *awk*.

F.3.1 Vectored data sets

The command *ProduceAndFormatVectoredDataSet* will either read data from an ASCII file and then format it, given the length of the required input and output vectors, or create data using a function.

IDENTIFIER = *ProduceAndFormatVectoredDataSet* {

`NumInputs`
Required
= {The number of inputs, the size of the input vector. }

`NumOutputs`
Required
= {The number of outputs, the size of the output vector. }

`NumLines`
Required
= {The number of input and output vector pairs. Usually every pair is on a single line, hence `NumLines`.}

`NumOutputClasses`
Optional
= {In the case when the output should be **discrete** and restricted to take values from a finite set, the `NumOutputClasses` specifies the size of this set. If discrete output is not required then either omit it or set it to **0**, which is the default. Use in conjunction with the `FirstClassAt` and `LastClassAt` parameters (see below).}

`FirstClassAt`
Optional,
`NumOutputClasses` > 0
= {When the output is expected to be discrete and the `NumOutputClasses` parameters was set to a positive integer, this parameter must be set to the **real number** indicating the value of the first class. See below for an example.}

`LastClassAt`
Optional,
`NumOutputClasses` > 0
= {This is the value of the last class. An example follows: we need to classify 5 images as follows: 0.0 maps to the first image, 0.5 maps to the second image, $\cdots$, 2.5 maps to the fifth image. To achieve, this one should set the `FirstClassAt` to 0.0 and the `LastClassAt` to 2.5, while the `NumOutputClasses` must be 5.}

`OutFileName`
Optional
= {The name of the local file or channel where the result should be sent to. If omitted, the file name will be constructed as **temp_Identifier**. See appendix F.7 for channel categories.}

`QuantiseInputs`
Optional,
`NumOutputClasses` > 0
= {It specifies whether the inputs as well as the outputs should be quantised (discrete). The default is **No**, which means that the inputs should be represented as continuous variables. A **Yes** will quantise both inputs and outputs.}

`InpFileObj`

Required,

if `Y` is not present

= { The file object, created by an *OpenFileObject* or equivalent command, associated with data the user wishes to read and format it accordingly. If this parameter is specified, then the `Y` parameter, which declares a function to produce data artificially, should be omitted. }

`Y`

Required,

if `InpFileObj` is not present

= { This parameter must specify the name of the function which will be used to produce data artificially. So far, there are only two functions available: **Levy6** and **Random6**. Do not use the `InpFileObj` parameter if this one is used as they are mutually exclusive. These functions are not built-in the ***np*** perl libraries. Instead, they have been created independently using C (any other language can be used). The only restriction is in the format of the command-line parameters and the output.Inspect the files **Levy6.c** and **Random6.c** if you want to develop more functions. }

`Seed`

Optional

= { By setting this parameter to a positive integer (the seed to a random number generator), it is guaranteed that as long as you use the same number for the seed and the same random number generator, you will obtain the same set of data, given that the rest of the specified parameters remain constant. This is useful in situations where you want to produce the same data set for training different networks at different times or systems, where storage or transfer is difficult or in the case when the original data set was deleted. If on the other hand you want your data set to be randomly chosen then either set this parameter to **Any** or do not specify it at all. }

}

F.3.2 Data sets based on time series

The command *FormatTimeSeriesDataSet* will read time series data from an ASCII file and format it by sampling at the given input and output time points. A time series simply consists of a measurement of a time-varying quantity at fixed time intervals. FFNN may be used in exploring the correlations between various past time points and the future. A multi-dimensional representation of the time series must be created by constructing input and output vectors with the time series values at these past time points. For example, if one wishes to check whether the next time series point can be predicted by using the past five time point values, then one constructs the following training file:

Input					Output
T-5	T-4	T-3	T-2	T-1	T

where T runs through all the time series points. This process is called *state-space reconstruction*, [Weigend, 1993], and this is what this command is supposed to do.

It is also possible to use many different time series files. For example one might want to use the variation in the prices of petrol, iron and wheat (in files **petrol.txt**, **iron.txt**, **wheat.txt**) in the prediction of some stock-market index (in file **dow_jones.txt**). In this case, declare all four file objects in the `InpFileObj` field with the object representing the prediction data being last. If you want to include previous values of the Dow Jones time series, include it twice. See the example at the end of this section for more details.

IDENTIFIER = *FormatTimeSeriesDataSet* {

`InpTimePoints` = { It specifies the time points which will compose the input vector. This is a *Lists* item and therefore one can use the shortcuts provided. See appendix F.8 for the *Lists* specification and the relevant format. However, one is supposed not to use the FIRST, LAST and ALL keywords, as they make no sense in this context. }

Required

`OutTimePoints` = { It specifies the time points which will compose the output vector. Same as above. }

Required

`NumLines` = { The number of input and output vector pairs **for each output class**. If you set this field to 20, say, and if you had 5 output classes then the resultant file would have had 100 vectors. }

Required

`InpFileObj`
Required
= { A list of comma separated file objects, created by an *OpenFileObject* or equivalent command, associated with the time series data file(s). The file(s) must be ASCII and the values (reals or integers) must be separated by white space. }

`NumOutputClasses`
Optional
= { In the case when the output should be **discrete** and restricted to take values from a finite set, the `NumOutputClasses` specifies the size of this set. }

`FirstClassAt`
Optional,
`NumOutputClasses > 0`
= { When the output is expected to be discrete and the `NumOutputClasses` parameters was set to a positive integer, this parameter must be set to the **real number** indicating the value of the first class. }

`LastClassAt`
Optional,
`NumOutputClasses > 0`
= {This is the value of the last class. }

`QuantiseInputs`
Optional,
`NumOutputClasses > 0`
= { It specifies whether the inputs as well as the outputs should be quantised. The default is **No**, which means that the inputs should be represented as continuous variables. A **Yes** will quantise both inputs and outputs. }

`OutFileName`
Optional
= { The name of the local file or channel where the result should be sent to. If omitted, which usually this is the case, then the file name will be constructed as **temp_Identifier**. See appendix F.7 for channel categories. }

`Seed`
Optional
= { By setting this parameter to a positive integer (the seed to a random number generator), it is guaranteed that as long as you use the same number for the seed, you will obtain the same set of data, given that the rest of the specified parameters remain constant. This is useful in situations where you want to produce the same data set for training different networks at different times or systems, where storage or transfer is difficult or in the case when the original data set was deleted. If on the other hand you want your data set to be randomly chosen then either set this parameter to **Any** or do not specify it at all. }

}

In the following example we create the data set for training a neural network to predict the value of the Dow Jones index using previous Dow Jones values (predict T+40 using T to T+20 and T+30 to T+39):

```
DOW_JONES = OpenFileObject {
    Filename = dow_jones.txt;
}
TRAIN_DATA = FormatTimeSeriesData {
    InpTimePoints = 1..20, 30..39;
    OutTimePoints = 40;
    NumLines = 100;
    InpFileObj = DOW_JONES;
}
```

While the following example uses four additional time series in the input vector:

```
PETROL = OpenFileObject {
   Filename = petrol.txt;
}
IRON = OpenFileObject {
   Filename = iron.txt;
}
WHEAT = OpenFileObject {
    Filename = wheat.txt;
}
DOW_JONES = OpenFileObject {
    Filename = dow_jones.txt;
}
TRAIN_DATA = FormatTimeSeriesData {
    InpTimePoints = 1..20, 30..39;
    OutTimePoints = 40;
    NumLines = 100;
    InpFileObj = PETROL, IRON, WHEAT, DOW_JONES;
}
```

F.3.3 Data sets for image classification

The command *FormatImagesDataSet* will read a binary (1 byte = 1 pixel value, no header information) file which has been constructed by concatenation of all the images[4] that the user needs to classify. For example, if one has some images that form 5 categories (classes). Suppose there are 5 directories called CLASS_1, CLASS_2, ..., CLASS_5, and each of the directories contains 3 images of the same class. Note that each directory must have an equal number of images and all images must be of the same dimensions,, say W×H. In order to construct the master data file, one can do (assume *csh* or derivatives):

```
% foreach i (CLASS_*) ¬
? cd $i ¬
? foreach j (*) ¬
? echo "Class: $i, Image: $j" ¬
? cat $j >> ../MASTER_IMAGES ¬
? end ¬
? cd .. ¬
? end ¬
```

[4] The individual images must be of the same dimensions.

Remember that the **MASTER_IMAGES** file contains 15 images of 5 classes and the dimensions of **each class** are W×3×H.

IDENTIFIER = *FormatImagesDataSet* {

`Width` = {The width of each of the images (W in the above example) in the input file.}
Required

`Height` = {The height of each of the images (3×H) in the input file.}
Required

`WindowWidth` = {It is common in image classification to operate on subsets of the images rather than the whole. In this way, better generalisation is achieved, while adequately small data sets have to be processed. This parameter refers to the width of the sampling window which will provide the subsets.}
Required

`WindowHeight` = {The sampled data will, therefore, consist of: $WindowWidth \times WindowHeight$ elements. The height of the sampling window.}
Required

`NumLines` = {The number of input and output vector pairs taken from **each** individual image.}
Required

`FirstClassAt` = {This parameter must be set to the **real number** indicating the value of the first class. See below for an example.}
Required

`LastClassAt` = {This is the value of the last class. For example, in order to classify the **5** images of our example as follows: **0.0** maps to the **first image**, **0.5** maps to the **second image** ··· and **2.5** maps to the **fifth image**, one may set the `FirstClassAt` to **0.0** and the `LastClassAt` to **2.5**.}
Required

`InpFileObj` = {The file object, created by an *OpenFileObject* or equivalent command, associated with a binary file which holds all the images which will be used in the classification (this file is called **MASTER_IMAGES** in our example).}
Required

`ScalingFactor` = {The pixel values are integers in the range 0 to 255. If you want to transform them to some other range then specify a real number here which all the pixel values will be **divided** by.}
Optional

`Padding`
Optional
= { If you want to create a data set which is partly composed of the contents of your images data set and the rest being set to an arbitrary pixel value (see `PaddingValue`) then set this parameter to a positive integer representing the width of the frame around the sampling window. It is useful if you want to experiment with testing a network with data which has fewer pixels than its inputs. }

`PaddingValue`
Optional,
valid if `Padding` > 0
= { The pixel values composing the surrounding frame of the data set. See `Padding`. }

`OutFileName`
Optional
= { The name of the local file or channel where the result should be sent to. If omitted, which usually this is the case, then the file name will be constructed as **temp_Identifier**. This file is ASCII. See appendix F.7 for channel categories. }

`Seed`
Optional
= { By setting this parameter to a number you will know that every time you use the same number, you will obtain the same set of data. If on the other hand you want something different every time you run the command then do not specify this variable at all. }

}

F.4 *mp* instructions: single FFNN

The process for training or testing a neural network starts with creating it first. The creation process will associate an IDENTIFIER with a FFNN or ADALINE of the specified architecture and other properties. So, in effect, the IDENTIFIER will carry all the static properties of that FFNN / ADALINE.

Once the network has been created, training and testing can take place. Here too, one has to supply some parameters. However, these parameters are not static, but, rather, they are dynamic and are forgotten when the process is finished, while the network remains to be used again. The results of training are saved to the weights file of the network.

F.4.1 Creation

It will return an IDENTIFIER which will be associated with the parameters specified. Using this IDENTIFIER, the program will be able to retrieve all the user preferences for the particular network. When this network is no longer of any use, destroy it using the *DeleteObjects* command.

IDENTIFIER = *CreateSingle* {

`Arch`
Required
= { The architecture of the net represented as positive integers separated by space. For example, **3 13 12 1** results to a network of 3 inputs, 1 output, and 2 hidden layers with 13 and 12 units each. }

`Sigmoid`
Optional
= { Should the output neurons of the network be sigmoided (i.e. $\mathbb{R} \to [0,1]$)? **Yes** or **No**. **No** is the default. }

`SingleType`
Required
= { Specify the type of the single network. At present, this can be a FFNN or an ADALINE. }

`Weights`
Optional
= { This is the name of the file holding the final weights for this network. If this parameter is not specified a file name will be constructed as **weights_Identifier**, where IDENTIFIER refers to the instruction's identifier. }

`NumOutputClasses`
Optional
= { In the case when the output should be **discrete** and restricted to take values from a finite set, the `NumOutputClasses` specifies the size of this set. }

`FirstClassAt`
Optional, `NumOutputClasses` > 0
= { When the output is expected to be discrete and the `NumOutputClasses` parameters was set to a positive integer, this parameter must be set to the **real number** indicating the value of the first class. }

`LastClassAt`
Optional, `NumOutputClasses` > 0
= {This is the value of the last class. }

`Derivatives`
Optional
= { If the derivative of the output of this unit w.r.t. its input is required, then specify the (base) file name to save it, with this parameter. }

}

F.4.2 Training

The *TrainSingle* command will train the single unit (FFNN or ADALINE) associated with the value of the `Obj` parameter supplied. Training needs a data file (defined by a file object) which must contain a number (`NumLines`) of **input** and **output vectors**, also known as **exemplars**. This file object can be created by the produce and/or format data sets mentioned in previous sections.

The process of training consists of feeding the input vector to the unit, obtain an output vector, compare the obtained output with the expected output, as defined in the training data file mentioned above, and calculate a discrepancy vector. The discrepancy or error vector has to be minimised by re-adjusting the strengths of the internal connections, known as the **weights**, of the network. This process is repeated several times (`Iters`).

The C program *NNengine* will be used for the training process. It can be interrupted at any point by a **Ctrl-C** or sending an **SIGINT** signal to the process (kill -INT).

The training process is often compared to the process of walking down a mountain to a valley. Our aim is to get to a lower altitude as quickly as possible. Therefore at the current position we sample the terrain around us for the steepest slope (gradient) and we jump to a new position in its direction. The question is, how big our jump should be. If we jump in small steps, then we can detect changes in the steepest descent direction faster, but we will reach the valley much later due to the increased number of jumps. On the other hand, we can do really big jumps and risk loosing our direction. We also risk, when we are deep in the valley enough, to jump on the opposite bank... The parameter that controls the magnitude of the change in the weight's vector is called the **rate of learning**, β. Another factor that controls training is the **momentum**, λ. This parameter indicates how much the new weight vector will be composed of the current one. This parameter sometimes leads to unstable behaviour with the error oscillating. Unless somebody is watching over the training process, set this parameter to zero or to some very small (less than 0.05, say) number.

TrainSingle {

`Obj` (Required) = { An single unit object previously created by the *CreateSingle* command. }

`Iters` (Required) = { The number of training iterations. }

`InpFileObj`
Required
= { This is the file object, created by *OpenFileObject* or equivalent, associated with the training data file. It is an ASCII file of floats or integers and consists of lines of pairs of input and output vectors. }

`WeightsUpdate`
Optional
= { This parameter controls the way the weight vector is updated. There are two options: **Exemplar** which updates the weights every time a new exemplar is presented to the network, or **Epoch** which updates the weights after **all** the exemplars have been presented to the network. The default value is **Exemplar**. }

`Lamda`
Optional
= { The *momentum* term. Another parameter which controls the training process. Use a very small number. If omitted, a default value will be used. }

`XDisplay`
Optional
= { Specifies an X-windows display name (e.g. air-gialla.sarc.city.ac.uk:0.0) where a real-time plot of the training error and rate of training error should be sent for monitoring. }

`Seed`
Optional
= { The seed to feed the random number generator. }

`Beta`
Optional
= { The *rate of learning*. Usually set it between 0.05 to 1.2. Better, still, set it to a high number at the beginning and then use the **kill -USR1 pid** or **kill -USR2 pid** (**pid** is the NNengine process id) to increase or decrease beta at run-time. There is a simple front end for this process called **NNChange.tcl** for those who have the **tcl/tk** package. if this parameter is not given, the default value will be used. }

`TrainingType`
Optional, `NumOutputClasses > 0` (from `Obj`)
= { The output of the neural network is, by nature, continuous. If discrete output was specified (`Obj` parameters) then quantisation takes place. In this case the error can be calculated either as the discrepancy between **expected output** and **actual continuous output** or **expected output** and **actual discrete output**. The `TrainingType` parameter indicates which of the two methods of error calculation should be followed: **Continuous** or **Discrete**. The default is **Continuous**. }

}

F.4.3 Testing

Once the network has been created and trained, and the weight vector is saved in a file, one may use the *TestSingle* command in order to feed some inputs in the network and obtain an output.

TestSingle {

`Obj` (Required) = { A single unit object previously created by a *CreateSingle* command }

`InpFileObj` (Required) = { This is the file object, created by *OpenFileObject* or equivalent, associated with the testing data file. It is an ASCII file of floats or integers and consists of lines of input vectors (only). }

`ShowInputs` (Optional) = { The output of this command may consist of either both the input and obtained output vectors (**Yes**) or only the output vectors (**No**, the default). }

`OutFileName` (Optional) = { The name of the file or channel where the output of the network should be sent to. If omitted a local file will be created with the name **final_output_Identifier**. See appendix F.7 for channel categories. }

}

F.5 *mp* instructions: Entities

Creating, training and testing the entities is similar to the case of the single units. Given the required entity class, the program will create a file which will contain instructions to create, train and test the entity. Note that an entity can be composed of not only FFNN but also of ADALINE or any other single unit might be implemented.

There is a quicker way to describe a FFNN or an entity of FFNN. This is called a **configuration script** (see appendix F.9 for more details). The description of an entity of a given specification is generated by some C language programs (**ProduceClass?Script.c**) as a configuration script. However, unless the user needs to create some more entity classes, or modify the existing ones, these files as well as the configuration script may be completely ignored.

F.5.1 Creation

The creation of an entity is more or less the same as the creation of a single unit. Some of the parameters, however, differ.

IDENTIFIER = *CreateEntity* {

`NumInputs`
Required
= This is the number of elements that the input data vector contains. It must be greater than 10 because, otherwise, it is not very practical to bother with an entity.

`Sigmoid`
Optional
= Should the output neurons of the network be sigmoided (i.e. $\mathbb{R} \to [0,1]$)? **Yes** or **No**. **No** is the default. This parameter is relevant to the single units composing the entity.

`EntityClass`
Required
= Currently, there are three classes that one may chose from: **1**, **2** and **3**.

`EntityType`
Required
= Specify the type of single units that make up this entity. Presently there are two choices FFNN or ADALINE.

`MinNumInputs`
Required, if the `ConfFile` is to be created.
= The number of inputs to every single unit composing the entity is selected at random. However, the user may chose an upper and lower bound to the number of inputs. This is the lower bound.

`MaxNumInputs`
Required
= This specifies the upper bound to the number of inputs each single unit should have.

`BPWeights`
Optional
= If an entity with adjustable connections between its various units is required then specify the (base) file name to hold this weights with this parameter.

`Derivatives`
Optional
= If the derivatives of the output of each single unit w.r.t. its input are required, then specify the (base) file name to hold them, with this parameter.

`Seed`
Optional
= By setting this parameter to a number you will know that every time you use the same number, you will obtain the same configuration for the entity. The same number of inputs, etc...

`Weights`
Optional
= { This is the **base** name for all the weights files that will be used in association with all the single units. The weight file for the single unit whose IDENTIFIER is, say, SFI, is **weights_SFI**. If this parameter is not specified then the default value given will be **temp_Identifier**, where IDENTIFIER refers to the Entity identifier. }

`NumOutputClasses`
Optional
= { In the case when the output should be **discrete** and restricted to take values from a finite set, the `NumOutputClasses` specifies the size of this set. This parameter is relevant to the single units composing the entity. }

`C1, C2`
Optional
= { These two parameters are relevant only to the **Class 1** entity. This class is a collection of randomly interconnected units (i.e. there is no predefined structure). `C1` specifies the number of first layer units (i.e. those that accept inputs from the data input vector and not from another unit). `C2` specifies the total number of layers in the entity interconnection scheme. By 'layers' we mean those units (single units) which belong to the same probability group of input assignment. }

`ConfFile`
Optional
= { As mentioned above, some external executables will create a configuration script which will be the basis for constructing the final ***np*** script for the entity. This parameter specifies the name of this file. The *configuration script* will have the extension '.con', the ***np*** script file will have the extension '.create'. If omitted, the default base name **temp_Identifier** will be used. If this parameter is defined but the `Min/MaxNumInputs` are undefined, the program will understand that there is already an existing configuration script which should be read, instead of creating a new one. }

`FirstClassAt`
Optional,
`NumOutputClasses > 0`
= { When the output is expected to be discrete and the `NumOutputClasses` parameter was set to a positive integer, this parameter must be set to the **real number** indicating the value of the first class. This parameter is relevant to the single units composing the entity. }

`LastClassAt`
Optional,
`NumOutputClasses > 0`
= {This is the value of the last class. }

}

F.5.2 Training

This process is similar to that of training a single unit. At first, the *configuration script* file is read and the *np* script[5] to train the entity is constructed. Then it is called and the training procedure commences.

TrainEntity {

`Obj` (Required) = { An Entity object previously created by the *CreateEntity* command. }

`InpFileObj` (Required) = { This is the file object, created by *OpenFileObject* or equivalent, associated with the training data file. It is an ASCII file of floats or integers and consists of lines of pairs of input and output vectors. One input file is used for the entire entity. The user needs not to worry about partitioning it. The program will take care of that. So, if you are training a 1000-input entity your input file must contain 1000 columns for the input vector. }

`WeightsUpdate` (Optional) = { This parameter controls the way the weight vector is updated. There are two options: **Exemplar** which updates the weights every time a new exemplar is presented to the network, or **Epoch** which updates the weights after **all** the exemplars have been presented to the network. The default value is **Exemplar**. This parameter is relevant to all the single units. }

`XDisplay` (Optional) = { Specifies an X-windows display name (e.g. air-gialla.sarc.city.ac.uk:0.0) where a real-time plot of the training error and rate of training error should be sent for monitoring. }

`TrainingType` (Optional) = { The output of the neural network is usually continuous. If discrete output is required then quantisation must take place. In this case the error can be calculated either as the discrepancy between **expected output** and **actual continuous output** or **expected output** and **actual discrete output**. The `TrainingType` parameter indicates which of the two methods of error calculation should be followed: **Continuous** or **Discrete**. The default is **Continuous**. This parameter is relevant to all the single units. }

[5] The `ConfFile` of the creation process with the '.train' extension.

`Hosts` (Optional) = { A comma separated list of hosts (internet address format, numerical or other). If this field is defined, the process of training an entity is broken into sub-tasks which are sent to each specified host and , thus, parallelising the training process. The user must have access to all of the hosts (using unix's **rsh** and **rcp**, note that some systems have such features disabled for security reasons) and each host must have a copy of all the necessary files to run ***np*** . The user needs not to be concerned with copying the data files to all hosts. Because of how **rsh** works, ***np*** requires a list of paths for each host. See the *SetPath* instruction. An example script which parallelises the training process can be found in **EXAMPLES/Parallel.np**. Parallelisation of training is only possible with entity architectures which allow for breaking the process into sub-tasks (Classes 1 and 2). }

`Iters` (Optional) = { The **minimum** number of training iterations for each single unit. If omitted a default of 1000 iterations will be used. This parameter is relevant to all the single units. }

`Beta` (Optional) = { The *rate of learning*. Everything mentioned for the single unit case applies here too. This parameter is relevant to all the single units. }

`Lamda` (Optional) = { The *momentum* term. This parameter is relevant to all the single units. }

}

F.5.3 Entities with connections of adjustable strength

The connections between the various units composing an entity may be variable. In this case, after each single unit is trained in the usual way (using the *TrainEntity* instruction), the weights of the entity connections can be optimised using gradient descent. This instruction does exactly this. With or without adjustable strength connections, the entities can be tested using the *TestEntity* instruction.

TrainEntity {

`Obj` (Required) = { An Entity object previously created by the *CreateEntity* command. }

`InpFileObj` (Required) = { This is the file object, created by *OpenFileObject* or equivalent, associated with the training data file. }

`WeightsUpdate` = { This parameter controls the way the weight vector is updated. There are two options: **Exemplar** which updates the weights every time a new exemplar is presented to the network, or **Epoch** which updates the weights after **all** the exemplars have been presented to the network. The default value is **Exemplar**. This parameter is relevant to all the single units. }
Optional

`XDisplay` = { Specifies an X-windows display name (e.g. airgialla.sarc.city.ac.uk:0.0) where a real-time plot of the training error and rate of training error should be sent for monitoring. }
Optional

`TrainingType` = { The output of the neural network is usually continuous. If discrete output is required then quantisation must take place. In this case the error can be calculated either as the discrepancy between **expected output** and **actual continuous output** or **expected output** and **actual discrete output**. The `TrainingType` parameter indicates which of the two methods of error calculation should be followed: **Continuous** or **Discrete**. The default is **Continuous**. This parameter is relevant to all the single units. }
Optional

`Iters` = { The **minimum** number of training iterations for each single unit. If omitted a default of 1000 iterations will be used. This parameter is relevant to all the single units. }
Optional

`Beta` = { The *rate of learning*. Everything mentioned for the single unit case applies here too. This parameter is relevant to all the single units. }
Optional

`Lamda` = { The *momentum* term. This parameter is relevant to all the single units. }
Optional

`WeightsRange` = { Four real numbers separated by space or comma to denote the range (min,max) of the **starting** weights and biases. The actual value of each weight and bias will be determined by the random number generator and the specified seed. }
Optional

`Seed` = {The seed to feed the random number generator. }
Optional

}

F.5.4 Testing

Again, this process is similar to that of training a single unit. At first, the *configuration* script file is read and the *nn* script for training (the `ConfFile` of the creation process with the '.test' extension) the entity is constructed. Then it is called and the forward pass (testing) begins.

TestEntity {

`Obj` (Required) = { An Entity object previously created by the *CreateEntity* command and trained with the *TrainEntity* and, optionally, the *BackpropagateEntity* instructions. }

`InpFileObj` (Required) = { This is the file object, created by *OpenFileObject* or equivalent, associated with the testing data file. It is an ASCII file of floats or integers and consists of lines of input vectors (only). Again, this file will be used for the entire entity, no need to partition it for the needs of the single units. }

`OutFileName` (Optional) = { The name of the file or channel where the output of the network should be sent to. If omitted a local file will be created with the name **final_output_Identifier**. See appendix F.7 for channel categories. }

`ShowInputs` (Optional) = { The output of this command may consist of either both the input and obtained output vectors (**Yes**) or only the output vectors (**No**, the default). }

}

F.6 Various other *nn* instructions

F.6.1 Unlink a file

It will unlink (delete) local files given their filename (plus path).

Unlink {

`Filename` (Required) = {A comma separated list of file names (not file objects). }

}

F.6.2 Include an *np* script file

It will read the *np* script file specified and will execute all the commands found until the end of file marker. It will then resume the execution of the initial file.

IncludeFile {

`Filename` Required = { The name of a local file or a channel whose contents are valid *np* script language instructions. }

}

F.6.3 Execute a system command

The command *System* executes a *perl* system command:

System {

`Com...` Required = { At least one command is required. More commands can be defined by using a parameter identifier starting with **Com**. Make sure that the command is valid and that can be found in the path. Use full path name if unsure. }

}

F.6.4 Debugging *np* scripts

The *DumpCurrentObjects* instruction will send a list of all the objects currently in memory along with their respective parameters and their values to **stderr** or to the named `OutFileName`.

DumpCurrentObjects {

`OutFileName` Optional = { A local file or channel that the information is to be sent to. }

}

F.6.5 SendInformation

The *SendInformation* instruction will write some information regarding the execution of the current *np* script (like executable name, input script name, host, user name, date and time started, current date and time) plus any message the user defines, to **stderr** or to the named `OutFileName` (a channel).

SendInformation {

`OutFileName` = {A local file or channel that the information is to be sent to. }

Optional

`Message` = {Some text to be added to the information sent. }

Optional

}

F.6.6 SetPath

The *SetPath* instruction will let ***mp*** know the path in which to search for executables for a given host. When ***mp*** attempts to execute a command to a remote host using **rsh** (for example when parallelising the training process of an entity, see the `TrainEntity` instruction), it needs to know where to search for the required executables.

SetPath {

`Address` = {Space separated paths. }

Optional

}

For example:

```
SetPath {
        host1.city.ac.uk = /usr/bin /vol/gnu/bin /homes/fred/bin_solaris;
        host2.city.ac.uk = /usr/bin /vol/gnu/bin /homes/fred/bin_linux;
}
```

F.7 Files and Channels

A `Filename` parameter can be either a **local file** name, obeying the unix system's path conventions, or a **channel** name referring to a remote destination accessible by the network.

There are various categories and sub-categories for the channel. These are:

1. **File handles**:

   - **STDIN**: The standard input, use **HANDLE<STDIN>**,
   - **STDOUT**: The standard output, use **HANDLE<STDOUT>**,
   - **STDERR**: The standard error, use **HANDLE<STDERR>**,

- **User defined**: A handle opened by the perl function *open*, use **HANDLE<A_HANDLE>**

2. **Host to host communication protocols:**
   - **mail**: Send data using **sendmail**, use **MAIL**<*host@e-mail address, subject*>,
   - **rcp**: Remote copy, use **RCP**<*host, remote path/filename*>,
   - **ftp**: File transfer protocol, use **FTP**<*host, login, passwd, filename*>.[6].
3. **Interprocess communication:**
   - **sockets**: Unix sockets, use **SOCKET**<*host, port number*>[7].

F.8 The *Lists* specification

A *Lists* item is defined as:

Integer	= [0-9]+ | FIRST | LAST
Expansion	= Integer..Integer | ALL
Integers	= Integer | Integer Integers | Integers Expansion
List	= Integers | Integers EXCEPT Integers
Lists	= List | List, Lists

Note that the keywords FIRST, LAST and ALL must be defined if you want to use them. For example it is allowed to use them in the *ExtractColumnsFromObject* command because there is a way to find out the total number of columns in the file objects. However, it is not allowed to use them to define time points (e.g. `InpTimePoints`). Special concession is made for FIRST where we assumed that if not defined then it takes the value of 1.

Here are some examples:

1..5	= 1 2 3 4 5
1..5 10	= 1 2 3 4 5 10
FIRST..5 LAST	= 1 2 3 4 5 25 (only when FIRST and LAST are defined)
ALL EXCEPT 5..LAST, 10	= 1 2 3 4 10 (only when ALL is defined)

[6] Not yet implemented.

[7] Not yet implemented. There are enough protocols already.

F.9 The *Configuration Script* format

Some definitions:

- O_i denotes the actual output obtained by a single FFNN. This is a matrix as it covers all the outputs of the FFNN in width and all the exemplars (`NumLines`) in height,
- X is the identifier of the file object associated with the input matrix. $X[Lists]$ denotes a matrix composed of vectors from X as specified by *Lists*,
- Y and $Y[Lists]$, as above,
- E_i denotes the result of vector arithmetic,
- ID denotes the identifier of a single FFNN,
- IV is one of X, O_i and E_i. An empty IV defaults to X,
- OV is one of Y and E_i,
- V is one of IV and OV,
- D is one of O_i or E_i,
- α denotes the number of input and β the number of output vectors to the particular FFNN.
- γ denotes the layer number and δ the total number of units in this layer.

There are three kind of statements that are allowed to exist in a *configuration script*:

1. $O_i = (IV_1[Lists] : IV_2[Lists] : \cdots)$, $(OV_1[Lists] : OV_2[Lists] : \cdots)$, ID, (α, β), (γ, δ);

   This is a declaration of a neural network identified by 'ID', its output is 'O_i'. It has α inputs given by all the $IV_i[Lists]$ and has β outputs given by all the $OV_i[Lists]$. γ and δ are used in the case when decomposition of the training process for parallel execution is required. In this case, units that belong to the same layer (i.e. same γ) may be trained independently (which means that none of these units sends its output to or receives its input from another unit of the same layer). δ informs ***mp*** of the total number of units in this layer so that the task is divided evenly among the remote hosts. Remember that the configuration script is created in conjunction with the *CreateEntity*, *TrainEntity* and *TestEntity*. Therefore the input and output vectors refer to the `InpFileObj`.

2. E_i = V_k[*Integer*] - V_l[*Integer*];
   This creates a vector E_i as the result of the difference between two other vectors V_k[*Integer*] and V_l[*Integer*]. Note that this operation involves vectors and not matrices, hence a single *Integer*.

3. DEL(D_1, D_2, $\cdots$, D_i);
   Will delete (unlink) all the objects identified by D_1, D_2, $\cdots$, D_i.

Consider the following example configuration script:

O_1 =([1,2,3,4]),(Y[1]),N_1,(4,1);
E_1 =O_1[1]-Y[1];
O_2 =(X[5,8]:O_1[1]),(E_1[1]),N_2,(3,1);
E_2 =O_2[1]-E_1[1];
DEL(O_1, E_1, O_2);

The 1^{st} line calls for a neural network identified by N_1 which has 4 inputs and 1 output. Its input vector is constructed from the 1^{st}, 2^{nd}, 3^{rd} and 4^{th} element of the input vector of the training file for the whole entity (`InpFileObj`). The output consists of a single element, the first (and probably the only one) element of the output vector of the training file. The output goes to a file object identified by O_1.

The 2^{nd} line will calculate the discrepancy vector between the first element of the O_1 file object and the first element of Y, the expected output.

The 3^{rd} line calls, again, for a neural network identified by N_2. It has 3 inputs and 1 output. Its inputs come from two sources. The first two inputs are the 5^{th} and 8^{th} vectors of the training file (`InpFileObj`), input vectors part. The 3^{rd} input comes from the 1^{st} vector of the file identified by O_1 (i.e. the output of N_1). Its output comes from the discrepancy vector calculated in line 2 (E_1, the [1] index is somewhat redundant).

The 4^{th} line, again, calculates the discrepancy vector between what the output of N_2 should have been (E_1[1]) and what actually is (O_2[1]).

Finally, the 5^{th} line will delete all the output files of the two networks (O_1 and O_2) as well as the discrepancy vector (E_1).

F.10 Alphabetical listing of all *mp* instructions

IDENTIFIER = *BackpropagateEntity* {
    Obj = ···; (*Required*)
    InpFileObj = ···; (*Required*)
    Iters = ···; (*Optional*)
    WeightsRange = ···; (*Optional*)
    Seed = ···; (*Optional*)
    Beta = ···; (*Required*)
    Iters = ···; (*Required*)
    Lamda = ···; (*Required*)
    TrainingType = ···; (*Optional*)
    XDisplay = ···; (*Optional*)
} IDENTIFIER = *ColumnsArithmetic* {
    RowExpr = ···; (*Required*)
    DefaultFileObj = ···; (*Optional*)
    ColExpr = ···; (*Optional*)
    OutFileName = ···; (*Optional*)
} IDENTIFIER = *CreateEntity* {
    NumInputs = ···; (*Required*)
    EntityClass = ···; (*Required if the* ConfFile *is to be created.*)
    EntityType = ···; (*Required*)
    MinNumInputs = ···; (*Required if the* ConfFile *is to be created.*)
    MaxNumInputs = ···; (*Required*)
    Seed = ···; (*Optional*)
    C1, C2 = ···; (*Optional*)
    ConfFile = ···; (*Optional*)
    Sigmoid = ···; (*Optional*)
    Weights = ···; (*Optional*)
    BPWeights = ···; (*Optional*)
    Derivatives = ···; (*Optional*)
    NumOutputClasses = ···; (*Optional*)
    FirstClassAt = ···; (*Optional,* NumOutputClasses > 0)
    LastClassAt = ···; (*Optional,* NumOutputClasses > 0)
} IDENTIFIER = *CreateSingle* {
    Arch = ···; (*Required*)
    SingleType = ···; (*Required*)
    Sigmoid = ···; (*Optional*)
    Weights = ···; (*Optional*)
    NumOutputClasses = ···; (*Optional*)
    FirstClassAt = ···; (*Optional,* NumOutputClasses > 0)
    LastClassAt = ···; (Optional, NumOutputClasses > 0)
} *DeleteObjects* {
    Obj = ···; (*At least one is required,* **comma** *separated*)
    Unlink = ···; (*Optional*)
}*DumpCurrentObjects* { } IDENTIFIER = *ExtractColumnsFromObjects* {
    Columns = ···; (*Required*)
    DefaultFileObj = ···; (*Optional*)
    OutFileName = ···; (*Optional*)
} IDENTIFIER = *FormatImagesDataSet* {
    Width = ···; (*Required*)
    Height = ···; (*Required*)
    WindowWidth = ···; (*Required*)
    WindowHeight = ···; (*Required*)
    NumLines = ···; (*Required*)
    FirstClassAt = ···; (*Optional*)

```
    LastClassAt = ···; (Optional)
    Padding = ···; (Optional)
    PaddingValue = ···; (Optional)
    ScalingFactor = ···; (Optional)
    InpFileObj = ···; (Required)
    OutFileName = ···; (Optional)
    Seed = ···; (Optional)
} IDENTIFIER = FormatTimeSeriesDataSet {
    InpTimePoints = ···; (Required)
    OutTimePoints = ···; (Required)
    NumLines = ···; (Required)
    InpFileObj = ···; (At least one is required, comma separated)
    NumOutputClasses = ···; (Optional)
    FirstClassAt = ···; (Optional)
    LastClassAt = ···; (Optional)
    QuantiseInputs = ···; (Optional)
    OutFileName = ···; (Optional)
    Seed = ···; (Optional)
    } IncludeFile {
    Filename = ···; (Required)
} IDENTIFIER = MergeObjects {
    InpFileObj = ···; (At least one is required, comma separated)
    MergeMethod = ···; (Optional)
    OutFileName = ···; (Optional)
} IDENTIFIER = ProduceAndFormatVectoredDataSet {
    NumInputs = ···; (Required)
    NumOutputs = ···; (Required)
    NumLines = ···; (Required)
    NumOutputClasses = ···; (Optional)
    FirstClassAt = ···; (Optional, NumOutputClasses > 0)
    LastClassAt = ···; (Optional, NumOutputClasses > 0)
    QuantiseInputs = ···; (Optional, NumOutputClasses > 0)
    InpFileObj = ···; (Required if Y is not present)
    Y = ···; (Required if InpFileObj is not present)
    OutFileName = ···; (Optional)
    Seed = ···; (Optional)
} SendInformation {
    OutFileName = ···; (Required)
    Message = ···; (Optional)
} SetPath {
    Hostname = Path; (At least one is required, space separated)
    Hostname = Path; (At least one is required)
    ...
} System {
    Com... = ···; (At least one is required)
} TestEntity {
    Obj = ···; (Required)
    InpFileObj = ···; (Required)
    OutFileName = ···; (Optional)
    ShowInputs = ···; (Optional)
} TestSingle {
    Obj = ···; (Required)
    InpFileObj = ···; (Required)
    OutFileName = ···; (Optional)
    ShowInputs = ···; (Optional)
} TrainEntity {
    Obj = ···; (Required)
    InpFileObj = ···; (Required)
    Iters = ···; (Optional)
```

```
    Beta = ···; (Optional)
    Lamda = ···; (Optional)
    WeightsUpdate = ···; (Optional)
    TrainingType = ···; (Optional)
    Hosts = ···; (Optional)
    XDisplay = ···; (Optional)
} TrainSingle {
    Obj = ···; (Required)
    Seed = ···; (Required)
    Iters = ···; (Required)
    InpFileObj = ···; (Required)
    Beta = ···; (Optional)
    Lamda = ···; (Optional)
    WeightsUpdate = ···; (Optional)
    TrainingType = ···; (Optional)
    XDisplay = ···; (Optional)
} Unlink {
    Filename = ···; (At least one is required, comma separated)
}
```

APPENDIX G

EXAMPLE *np* SCRIPTS

Below is an example of an ***np*** program. It demonstrates how easy training and testing an FFNN entity can be,

```
# Create a training data set of 60 lines using the Levy function
# with 500 input variables.
TRAINING_DATA = ProduceFormatVectoredDataSet {
    NumInputs = 500;
    NumOutputs = 1;
    NumLines = 60;
    Y = Levy6;
    Seed = 1974;
}
# Create a Class 1 entity of  FFNN with 500 inputs
ENTITY = CreateEntity {
    SingleType = FFNN;
    NumInputs = 500;
    EntityClass = 1;
    MinNumInputs = 12;
    MaxNumInputs = 35;
    Seed = 1975;
}
# Train the entity for 1000 iterations with specified rate of learning
# Furthermore, parallelise the training process and distribute it among the four
# remote hosts specified.
TrainEntity {
    Obj = ENTITY;
    InpFileObj = TRAINING_DATA;
    Iters = 1000;
    Beta = 0.095;
    Lamda = 0.0;
    Hosts = altair, zeta, spica, vega;
}
# produce the test set again with the Levy function
# but different seed and a lot more lines (2000)
```

```
TEST_DATA = ProduceAndFormatVectoredDataSet {
    Y = Levy6;
    NumInputs = 500;
    NumOutputs = 1;
    NumLines = 2000;
    Seed = 1971;
    Y = Levy6;
}
# ... and test the entity
# the output is contained in a file called final
TestEntity {
    Obj = ENTITY;
    InpFileObj = TEST_DATA;
    OutFileName = final;
}
```

The first instruction of the above program, *ProduceFormatVectoredDataSet*, will create the training data file which is then referred to by the name of TRAINING_DATA. The number of input dimensions, the number of vectors as well as which data-generating function should be used, are all specified by the parameters included in the body – the text between the two curly brackets following the instruction. Each definition (e.g. `NumInputs = 500;`) in the body of the instruction must end in a semi-colon.

A lot of operations within the *mp* system require a random number generator. For example, the inputs to the Levy function are produced randomly (see also section 6.3.5 on page 87). However, we would also like to be able to reproduce exactly such initial conditions because of issues of repeatability. In this respect, the deterministic nature of the computer's random number generator is an advantage because these "random" decisions can be repeated by supplying the same seed, using the `Seed` keyword.

The instruction *CreateEntity* will create a FFNN entity (a C_1 entity in this case, as specified by the keyword `EntityType`) with a total of 500 inputs (as specified by the keyword `NumInputs`). The *mp* interpreter is instructed to construct each individual FFNN unit in the entity with a number of inputs between 12 and 35 (the keywords `MinNumInputs` and `MaxNumInputs`). The total number of FFNN as well as the interconnection map are determined randomly. The `Seed` keyword may, again, be used to regenerate exactly a previous configuration. If omitted, the random number generator will be seeded with the current time – a usual tactic for creating unique random number sequences.

Following, is the instruction (*TrainEntity*) to train the created entity object, referred to by the identifier ENTITY, with the data set referred to by the identifier TRAINING_DATA.

The Hosts keyword contains a list of **domain names** or **IP** addresses of computers in the network to which the training procedure should be distributed (e.g. parallelised training). Currently, these hosts must be running **Unix**. However, ***mp*** can easily be extended so that it runs on a wider range of operating systems by rewriting certain parts of it in the **Java** language.

Once training is completed, the test data set will be created, using the instruction *ProduceFormatVectoredDataSet*. The seed is different than before because we need the test set to be different from the training set. Also, the number of vectors specified is much larger than before.

Finally, the command to test the entity (*TestEntity*) is given by defining which entity we require and what test data set it should be tested with. The output of the entity goes to a file called **final**. Obtaining a performance measure can be done with the following program. Note that this program has to be appended at the end of the previous program.

```
# Create a separate data object which holds just the  expected output.
# Remove the  input vector columns
EXPECTED_OUTPUT = ExtractColumnsFromObject {
    Columns = TEST_DATA[LAST];
}
# Now, open the file called final which contains the
# obtained entity output ...
ACTUAL_OUTPUT = OpenFileObject {
    Filename = final;
}
# ... and calculate the error.
ERROR = ColumnsArithmetic {
    # mean square error estimate
  RowExpr = 0.5 * ((ACTUAL_OUTPUT[1] - EXPECTED_OUTPUT[1]) ** 2);
  ColExpr = average;
  OutFileName = error;
}
$
```

The first instruction of the above program will extract the last column from the test data set. This is done by specifying which column should be extracted. The expression "TEST_DATA[LAST]" refers to the last column of the test data file – e.g. the expected output. This column should then be compared to the **actual** output, contained in the *final* file which is opened with the second instruction (*OpenFileObject*).

Finally, the *ColumnsArithmetic* instruction will perform the mathematical expression defined by the keyword RowExpr for each row of the specified data objects. Do not forget that, by now, the objects ACTUAL_OUTPUT and EXPECTED_OUTPUT contain just a single column which can be referred to by fixing the '[1]' after the object name, e.g. EXPECTED_OUTPUT[1]. The RowExpr is nothing else than one half of the square of the discrepancy between **expected** and **obtained** outputs. Finally, the keyword ColExpr specifies that all the elements of the resultant column should be summed up and the average be taken and dumped to the output file, **error**.

G.1 Some more *nn* scripts

This is an example *nn* script which will train a C_1 entity (referred as ENTITY) with training data (referred to as TRAIN_DATA) produced by the Levy function. Training takes place in parallel over four different computers in the network.

```
# Author: A.Hadjiprocopis
SetPath {
   altair.soi.city.ac.uk = /usr/bin /vol/gnu/bin;
   vega.soi.city.ac.uk = /usr/bin /vol/gnu/bin;
   spica.soi.city.ac.uk = /usr/bin /vol/gnu/bin;
   zeta.soi.city.ac.uk = /usr/bin /vol/gnu/bin;
}
TRAIN_DATA = ProduceAndFormatVectoredDataSet {
   Y = Levy6;
   NumInputs = 500;
   NumOutputs = 1;
   NumLines = 70;
   Seed = 1974;
}
ENTITY = CreateEntity {
   EntityType = FFNN;
   NumInputs = 500;
   EntityClass = 1;
   MinNumInputs = 12;
   MaxNumInputs = 35;
   Sigmoid = No;
   ConfFile = Conf_C1;
   Weights = W_C1;
   C1 = 22;
   C2 = 4;
   Seed = 1975;
```

```
}
TrainEntity {
   Obj = ENTITY;
   InpFileObj = TRAIN_DATA;
   Iters = 1000;
   Beta = 0.095;
   Lamda = 0;
   Hosts = altair, zeta, spica, vega;
}
SendInformation {
   OutFileName = Message;
   Message = TITLE: the results of C1 entity testing
   Obj = TRAIN_DATA,ENTITY;
}
$
```

This is an ***np*** script which may be used to test an entity trained with the previous script.

```
# Author: A.Hadjiprocopis
TEST_DATA = ProduceAndFormatVectoredDataSet {
   Y = Levy6;
   NumInputs = 500;
   NumOutputs = 1;
   NumLines = 5;
   Seed = 1976;
}
ENTITY = CreateEntity {
   EntityType = FFNN;
   NumInputs = 500;
   EntityClass = 1;
   MinNumInputs = 12;
   MaxNumInputs = 35;
   Sigmoid = No;
   ConfFile = Conf_C1;
   Weights = W_C1;
   C1 = 22;
   C2 = 4;
   Seed = 1975;
}
TestEntity {
   Obj = ENTITY;
   InpFileObj = TEST_DATA;
```

```
   OutFileName = final;
}
INPUT = ExtractColumnsFromObject {
   Columns = TEST_DATA[1..LAST EXCEPT LAST];
}
EXPECTED_OUTPUT = ExtractColumnsFromObject {
   Columns = TEST_DATA[LAST];
}
ACTUAL_OUTPUT = OpenFileObject {
   Filename = final;
}
ERROR = ColumnsArithmetic {
   # mean square error estimate
   RowExpr = 0.5 * ((ACTUAL_OUTPUT[1] - EXPECTED_OUTPUT[1]) ** 2);
   ColExpr = average;
   OutFileName = error;
}
$
```

The following script will train a single FFNN.

```
TRAIN_DATA = ProduceAndFormatVectoredDataSet {
   Y = Levy6;
   NumInputs = 100;
   NumOutputs = 1;
   NumLines = 70;
   Seed = 1974;
}
SINGLE = CreateSingle {
   SingleType = FFNN;
   Arch = 100 49 1;
   Weights = W_SINGLE;
   Sigmoid = No;
}
TrainSingle {
   Obj  = SINGLE;
   InpFileObj  = TRAIN_DATA;
   Iters = 1000;
   Beta = 0.095;
   Lamda = 0;
}
SendInformation {
   OutFileName = Message;
```

```
   Message = Results of testing a single FFNN;
   Obj = TRAIN_DATA,SINGLE;
}
$
```

Finally, the following *np* script may be used in testing a single FFNN which was trained with the previous script.

```
TEST_DATA = ProduceAndFormatVectoredDataSet {
   Y = Levy6;
   NumInputs = 100;
   NumOutputs = 1;
   NumLines = 2000;
   Seed = 1976;
}
SINGLE = CreateSingle {
   SingleType = FFNN;
   Arch = 100 49 1;
   Weights = W_SINGLE;
   Sigmoid = No;
}
INPUT = ExtractColumnsFromObject {
   Columns = TEST_DATA[1..LAST EXCEPT LAST];
}
TestSingle {
   Obj = SINGLE;
   InpFileObj = INPUT;
   OutFileName = final;
}
EXPECTED_OUTPUT = ExtractColumnsFromObject {
   Columns = TEST_DATA[LAST];
}
ACTUAL_OUTPUT = OpenFileObject {
   Filename = final;
}
ERROR = ColumnsArithmetic {
   RowExpr = 0.5 * ((ACTUAL_OUTPUT[1] - EXPECTED_OUTPUT[1]) ** 2);
   ColExpr = average;
   OutFileName = error;
}
$
```

Bibliography

[Anderson and Rosenfeld, 1988] Anderson, J. A. and Rosenfeld, E., editors (1988). *Neurocomputing: Foundations of Research.* MIT Press, Cambridge.

[Baldi and Hornik, 1989] Baldi, P. and Hornik, K. (1989). Neural networks and principal component analysis: Learning from examples without local minima. *Neural Networks*, 2:53–58.

[Baum and Haussler, 1989] Baum, E. and Haussler, D. (1989). What size net gives valid generalization? *Neural Computation*, 1(1):151–160.

[Bianchini et al., 1997] Bianchini, M., Fanelli, S., Gori, M., and Maggini, M. (1997). Terminal attractor algorithms: A critical analysis. *Neurocomputing*, pages 3–13.

[Bianchini et al., 1998] Bianchini, M., Frasconi, S., Gori, M., and Maggini, M. (1998). Optimal learning in artificial neural networks: A review of theoretical results. *Neural Network Systems Techniques and Applications (C. Leondes, ed.)*, pages 1–51.

[Bishop, 1995] Bishop, C. (1995). *Neural Networks for Pattern Recognition.* Clarendon Press, Oxford.

[Blum and Rivest, 1988] Blum, A. and Rivest, R. (1988). Training a 3-node neural network is np-complete. In *Proceedings of the* 1^{st} *Workshop on Computational Learning Theory.* Morgan-Kaufmann.

[Brookes and Dick, 1963] Brookes, B. and Dick, W. (1963). *Introduction to Statistical Method.* Heinemann Educational Books Ltd, London.

[Burges, 1998] Burges, C. (1998). *A tutorial on Support Vector Machines for Pattern Recognition.* Kluwer Academic Publishers, Boston.

[Burrows and Niranjan, 1993] Burrows, T. L. and Niranjan, M. (1993). The use of feed–forward and recurrent neural networks for system identification. Technical report, Cambridge University Engineering Department.

[Chao et al., 1991] Chao, J., Ratanasuwan, W., and Tsujii, S. (1991). How to find global minima in finite times of search for multilayer perceptrons training. *International Joint Conference on Neural Networks*, pages 1079–1083.

[Šima, 1994] Šima, J. (1994). Loading deep networks is hard. *Neural Computation*, 6:842–850.

[Šima, 1996] Šima, J. (1996). Back-propagation is not efficient. *Neural Networks*, 6.

[Cover, 1965] Cover, T. (1965). Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition. *IEEE Transactions on Electronic Computers*, 14:326–334.

[Craven and Shavlik, 1994] Craven, M. and Shavlik, J. (1994). Using sampling and queries to extract rules from trained neural networks. In *Machine Learning: Proceedings of the Eleventh International Conference*, San Francisco, CA.

[Cybenko, 1989] Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2(4):303–314.

[DasGupta et al., 1995] DasGupta, B., Siegelmann, H., and Sontag, E. (1995). On the complexity of training neural networks with continuous activation. *IEEE Transactions on Neural Networks*, 6.

[Descartes, 1637] Descartes (1637). *Discours de la Méthode.*

[Efron, 1982] Efron, B. (1982). The jacknife, the boostrap and other resampling plans. *Society for Industrial and Applied Mathematics.*

[Flexer, 1996] Flexer, A. (1996). Statistical evaluation of neural network experiments: Minimum requirements and current practice. In Trappl, R., editor, *Cybernetics and Systems 1996, Proceedings of the* 13^{th} *European Meeting on Cybernetics and Systems Research*, pages 1005–1008, Austrian Society for Cybernetic Studies.

[Freeman, 1991] Freeman, J. (1991). *Neural Networks: Theory and Practice.* Addison–Wesley.

[Funahashi, 1989] Funahashi, K. (1989). On the approximate realization of continuous mappings by neural networks. *Neural Networks*, 2:183–192.

[Gallant and White, 1992] Gallant, A. and White, H. (1992). There exists a neural network that does not make avoidable mistakes. In White, H., editor, *Artificial Neural Networks: Approximation and Learning Theory*, pages 5–11. Blackwell, Oxford, UK.

[Garey and Johnson, 1979] Garey, M. and Johnson, D. (1979). *Computers and Intractability: A Guide to the theory of NP-completeness.* W.H. Freeman, San Fransisco.

[Gately, 1996] Gately, E. (1996). *Neural Networks for Financial Forecasting,*. John Wiley, New York.

[Geman and Bienenstock, 1992] Geman, S. and Bienenstock, E. (1992). Neural networks and the bias-variance dilemma. *Neural Computation*, 4:1–58.

[Griewank, 1989] Griewank, A. (1989). On automatic differentiation. In Iri, M. and Tanabe, K., editors, *Mathematical Programming: Recent Developments and Applications*, pages 83–107. Kluwer Academic Publishers.

[Hadjiprocopis and Smith, 1997a] Hadjiprocopis, A. and Smith, P. (1997a). Feed forward neural network entities. In J. Mira, R. M.-D. and Cabestany, J., editors, *Lecture Notes in Computer Science: Biological and Artificial Computation: From Neuroscience to Technology*, pages 349–359. Springer–Verlag.

[Hadjiprocopis and Smith, 1997b] Hadjiprocopis, A. and Smith, P. (1997b). Feed forward neural network entities in the analysis of high-dimensional data. In *Proceeding of the* 3^{rd} *International Conference on Neural Networks and their Applications*, pages 147–154.

[Hadjiprocopis and Smith, 1998] Hadjiprocopis, A. and Smith, P. (1998). Feed forward neural network entities in time series prediction and image classification. In *Proceeding of the International ICSC/IFAC Symposium on Neural Computation / NC98*, pages 1002–1008, Technical University of Vienna.

[Hadjiprocopis et al., 1994] Hadjiprocopis, A., Smith, P., Comley, R., and Lakkos, S. (1994). A neural network scheme for earthquake prediction based on the seismic electric signals. *IEEE Conference on Neural Networks and Signal Processing.*

[Höffgen, 1993] Höffgen, K. (1993). Computational limitations on training sigmoidal neural networks. *Information Processing Letters*, 46.

[Hebb, 1949] Hebb, D. O. (1949). *The Organization of Behavior.* Wiley, New York.

[Hecht-Nielsen, 1990] Hecht-Nielsen, R. (1990). *Neurocomputing.* Addison–Wesley, Menlo Park, CA.

[Hilbert, 1902] Hilbert, D. (1902). Mathematical problems. *Bulletin of the American Mathematical Society*, 8:437–479.

[Hinton, 1995] Hinton, P. (1995). *Statistics Explained.* Routledge, New York, english edition.

[Hornik, 1991] Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4:251–257.

[Hornik et al., 1992] Hornik, K., Stinchcombe, M., and White, H. (1992). Multilayer feedforward networks are universal approximators. In White, H., editor, *Artificial Neural Networks: Approximation and Learning Theory*, pages 12–28. Blackwell, Oxford, UK.

[Jacobs et al., 1991] Jacobs, R. A., Jordan, M. I., Nowlan, S. J., and Hinton, G. (1991). Adaptive mixture of local experts. *Neural Computation*, 3:79–87.

[Jordan and Jacobs, 1992] Jordan, M. I. and Jacobs, R. A. (1992). Hierarchies of adaptive experts. In R. P. Lippmann, J. M. and Touretzky, D. S., editors, *NIPS*, volume 4, pages 985–992. Morgan Kaufmann.

[Judd, 1988] Judd, J. (1988). On the complexity of learning shallow neural networks. *Journal of Complexity*, 4:177–192.

[Judd, 1990] Judd, J. (1990). *Neural Network Design and the Complexity of Learning.* MIT Press, Cambridge, MA.

[Kůrková, 1991] Kůrková, V. (1991). Kolmogorov's theorem is relevant. *Neural Computation*, 3:617–622.

[Kolmogorov, 1957] Kolmogorov, A. N. (1957). On the representation of continuous functions of many variables by superposition of continuous functions of one variable and addition. *Doklady Akademii Nauk SSR*, 114:953–956.

[Kontoravdis et al., 1992] Kontoravdis, D., Stafylopatis, A., and Kollias, S. (1992). Parallel implementation of structured feedforward neural networks for image recognition. *International Journal of Neural Networks*, 2:91–99.

[Landa, 1992] Landa, M. D. (1992). Virtual environments as intuition synthesisers. *Cultural diversity in the global village.*

[Lavine, 1983] Lavine, R. A. (1983). *Neurophysiology: The Fundamentals.* The Collamore Press, Lexington, MA.

[Levy and Montalvo, 1985] Levy, A. V. and Montalvo, A. (1985). The tunnelling algorithm for the global minimization of functions. *SIAM J. Sci. Stat. Comput.*, 6:15–29.

[Lindsey and Lindblad, 1994] Lindsey, C. and Lindblad, T. (1994). Review of hardware neural networks: a user's perspective. In *Proceedings of ELBA94.*

[Lucas, 1997] Lucas, S. (1997). Forward-backward building blocks of evolving neural networks with intrinsic learning behaviours. In J. Mira, R. M.-D. and Cabestany, J., editors, *Lecture Notes in Computer Science: Biological and Artificial Computation: From Neuroscience to Technology*, pages 723–732. Springer–Verlag.

[MacGregor, 1987] MacGregor, R. (1987). *Neural and Brain Modeling.* Academic Press, San Diego, CA.

[Marx, 1887] Marx, K. (1887). *Capital.* www.marx.org, english edition.

[McClelland et al., 1986] McClelland, J., Rumelhart, D., and Hinton, G. (1986). The appeal of parallel distributed processing. In Rumelhart, D. and McClelland, J., editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*, pages 3–44. MIT Press, Cambridge, MA.

[McCulloch and Pitts, 1943] McCulloch, W. S. and Pitts, W. (1943). A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5.

[McKelvey, 1992] McKelvey, T. (1992). Neural networks applied to optimal flight control. Technical report, Linköping University, Sweeden.

[Minsky, 1975] Minsky, M. (1975). A framework for representing knowledge. In Winston, P., editor, *The psychology of computer vision*, pages 211–277, New York. McGraw-Hill.

[Minsky, 1990] Minsky, M. (1990). Logical vs. analogical or symbolic vs. connectionist or neat vs. scruffy. *Artificial Intelligence at MIT, Expanding Frontiers*, 1.

[Minsky and Papert, 1969] Minsky, M. and Papert, S. (1969). *Perceptrons.* MIT Press, Cambridge, MA.

[Minsky and Papert, 1988] Minsky, M. and Papert, S. (1988). *Perceptrons: An Introduction to Computational Geometry.* MIT Press, Cambridge, MA, expanded edition.

[Misra, 1992] Misra, M. (1992). *Implementation of Neural Networks on Parallel Architectures.* Doctoral Dissertation, Electrical Engineering, University of Southern California.

[Murre, 1993] Murre, J. (1993). Transputers and neural networks: An analysis of implementation constraints and performance. *IEEE Transactions on Neural Networks*, 4(2):284–292.

[Newell, 1980] Newell, A. (1980). Physical symbol systems. *Cognitive Science*, 4:135–183.

[Ohno-Machado and Musen, 1996] Ohno-Machado, L. and Musen, M. A. (1996). Modular neural networks for medical prognosis: Quantifying the benefits of combining neural networks for survival prediction. Technical report, Knowledge Systems Laboratory, Medical Computer Science, Stanford University.

[Papert, 1988] Papert, S. (1988). *One AI or Many?* Daedalus.

[Perrone, 1994] Perrone, M. P. (1994). General averaging results for convex optimization. In Mozer, M. C., editor, *Proceedings 1993 Connectionist Models Summer School*, pages 364–371, Hillsdale, NJ. Lawrence Erlbaum.

[Perrone and Cooper, 1993] Perrone, M. P. and Cooper, L. N. (1993). When networks disagree: ensemble methods for hybrid neural networks. *Artificial Neural Networks for Speech and Vision*, pages 126–142.

[Poggio and Girosi, 1989] Poggio, T. and Girosi, F. (1989). A theory of networks for approximation and learning. *MIT AI Memo No. 1140.*

[Rosenblatt, 1962] Rosenblatt, F. (1962). *Principles of Neurodynamics.* Spartan, New York.

[Rudin, 1964] Rudin, W. (1964). *Principles of Mathematical Analysis.* McGraw-Hill, New York.

[Rumelhart, 1975] Rumelhart, D. (1975). Notes on a schema for stories. In Bobrow, D. and Collins, A., editors, *Representation and understanding*, pages 211–236, New York. Academic Press.

[Rumelhart et al., 1986] Rumelhart, D., Hinton, G., and Williams, R. (1986). Learning internal representations by back-propagating errors. In Rumelhart, D. and McClelland, J., editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition.* MIT Press, Cambridge, MA.

[Schank, 1976] Schank, R. (1976). The role of memory in language processing. In Cofer, C., editor, *The structure of human memory*, pages 162–189, San Fransisco. Freeman.

[Spiegel, 1971] Spiegel, M. (1971). *Theory and problems of Statistics.* McGraw-Hill Book Company (UK) Ltd, english edition.

[Torn and Zilinkas, 1987] Torn and Zilinkas (1987). Global optimisation. *Lecture Notes in Computer Science.*

[Valiant, 1994] Valiant, L. (1994). A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142.

[Vapnik, 1979] Vapnik, V. (1979). *Estimation of Dependences Based on Empirical Data [in Russian].* Nauka, USSR.

[Vapnik, 1995] Vapnik, V. (1995). *The Nature of Statistical Learning Theory.* Springer, New York.

[Vapnik and Chervonenkis, 1964] Vapnik, V. and Chervonenkis, A. (1964). A note on one class of perceptrons. *Automation and Remote Control*, 25.

[Vapnik and Chervonenkis, 1971] Vapnik, V. and Chervonenkis, A. (1971). On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications*, 16(2):264–280.

[Vapnik and Chervonenkis, 1974] Vapnik, V. and Chervonenkis, A. (1974). *Theory of Pattern Recognition [in Russian].* Nauka, USSR.

[Vapnik and Lerner, 1963] Vapnik, V. and Lerner, A. (1963). Pattern recognition using generalised portrait method. *Automation and Remote Control*, 24.

[V.N. Vapnik and Smola, 1997] V.N. Vapnik, S. G. and Smola, A. (1997). Support vector method for function approximation, regression, estimation and signal processing. *Advances in Neural Information Processing Systems*, 9:281–287.

[Wang and Hsu, 1991] Wang, S. and Hsu, C. (1991). Terminal attractor learning algorithms for backpropagation neural networks. *International Joint Conference on Neural Networks*, pages 183–189.

[Weigend, 1993] Weigend, A. S. (1993). *Time Series Prediction : Forecasting the Future and Understanding the Past.* Addison–Wesley.

[Werbos, 1974] Werbos, P. J. (1974). *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.* Doctoral Dissertation, Applied Mathematics, Harvard University, Boston, MA.

[Widrow and Hoff, 1960] Widrow, B. and Hoff, M. (1960). Adaptive switching circuits. *IRE WESCON Convention Record*, 4:96–104.

[Wiklicky, 1993] Wiklicky, H. (1993). The neural network loading problem is undecidable. In *Proceedings of Euro-COLT*. Oxford University Press.

[Zell et al., 1993] Zell, A., Mache, N., Vogt, M., and Huettel, M. (1993). Problems of massive parallelism in neural network simulation. In *Proceedings of the IEEE Int. Conf. on Neural Networks*, volume 3, pages 1890–1895, San Francisco, CA.

[Zhigljavsky, 1991] Zhigljavsky (1991). *Theory of Global Random Search*. Kluwer Academic Publishing.

INDEX OF NAMES

Arnold, V.I., 24 (3.3)

Chervonenkis, 173 (C)
Cover, T.M., 32 (4.2)

HAL, 17 (2.4)
Hebb, D., 10 (2.2)
Hilbert, D., 24 (3.3)

Kolmogorov, A., 24 (3.3), 30 (4.1)

Marx, Karl, 16 (2.4)
McCulloch & Pitts, 10 (2.2)
Minsky, M., 11 (2.2), 32 (4.2)
Minsky, M., 16 (2.4)

Newell, 7 (2.1)

Papert, S., 11 (2.2), 32 (4.2)
Papert, S., 16 (2.4)
Parallel Distributed Processing group (Rumelhart et al.), 13 (2.2)
Pavlov, I., 10 (2.2)
Piagiet, 16 (2.3)

Rosenblatt, F., 10 (2.2)

Simon, 7 (2.1)
Sprecher D., 24 (3.3)

Valiant, 173 (C)
Vapnik, 173 (C)
von Neumann, J., 9 (2.2)

Werbos, P., 13 (2.2)

INDEX OF KEYWORDS

abstraction, *see* Connectionism
Adaline, *see* adaptive linear element
adaptive interpolation, *see* Connectionism
adaptive linear element, 20 (3.1)
affine functions, family of, 22 (3.2)
AI, *see* artificial intelligence
approximation error, 174 (C)
artificial data generation, 86 (6.3)
artificial intelligence, 7 (2.1)
associationism, 10 (2.2)

back-propagation of errors
  critique, 38 (4.4)
  description, 27 (3.4)
  discovery, 13 (2.2)
  phases, 27 (3.4)
behaviourism, 10 (2.2)
bias-variance trade-off, 177 (C)
bottom-up, 9 (2.1)

Complexity classes, 35 (4.3)
  class NP, 35 (4.3)
  class P, 35 (4.3)
Connectionism
  a new approach to AI, 14 (2.3)
  abstraction, 16 (2.3)
  adaptive interpolation, 16 (2.3)
  basic properties, 15 (2.3)
  Constructivism, 16 (2.3)
  distributivity, 15 (2.3)
  general-purpose modelling, 16 (2.3)
  generalisation, 16 (2.3)
  graceful degradation, 15 (2.3)
  history of, 9 (2.2)
  Nativism, 16 (2.3)
  robustness, 15 (2.3)
Constructivism, *see* Connectionism
curse of dimensionality, 31 (4.1)

distributivity, *see* Connectionism

emergence, 9 (2.1)
empirical results
  generalisation Ability, 83 (6.3)
  methodology, 81 (6.2)
  parallelisation of the training process, 150 (6.4)
  recapitulation, 154 (6.5)
Entities, *see* Feed Forward Neural Network Entities

Feed Forward Neural Network Entities
  C_1 construction, 59 (5.4)
  C_1 example, 58 (5.4)
  C_1 formalism, 57 (5.4)
  C_1 training, 60 (5.4)
  C_1 with adjustable connections, 60 (5.4)
  C_3 formalism, 66 (5.4)
  C_2 formalism, 65 (5.4)
  benefits of, 76 (5.8)
  framework, 1 (1.1)
  introduction, 55 (5.4)

modified back-propagation, 61 (5.4)
motivation, 51 (5.2)
training time benefits: comparison with single FFNN, 72 (5.7)
universal function approximation, 70 (5.6)
Feed Forward Neural Networks
critique, 29 (4.1)
derivative of transfer function, 62 (5.4)
hardware implementation, 45 (4.5)
introduction, 19 (3.1)
neuron's notation, 21 (3.2)
neuronal operation, 21 (3.2)
neuronal transfer function, 22 (3.2)
notation, 20 (3.2)
NP-hard and NP-complete results, 30 (4.1), 36 (4.3)
number of adjustable parameters, 39 (4.4)
parallel implementation, 43 (4.5)
the non-explicit nature of learning, 45 (4.6)
transfer function, 23 (3.2)
FFNN, *see* Feed forward neural networks
fifth generation project, 17 (2.4)
frames, 7 (2.1)
fuzzy logic, 8 (2.1)

general positions, 33 (4.2), 172 (B)
generalisation, *see* Connectionism
global optimisation, 40 (4.4)
good old-fashioned AI (GOFAI), 9 (2.1)
graceful degradation, *see* Connectionism
gradient descent, 26 (3.4)
growth function, 175 (C)

Hilbert's 13^{th} problem, 24 (3.3)
hill climbing, 30 (4.1)
homunculus, 9 (2.1)
hypothesis space, 173 (C)

linear (in)separability, 12 (2.2), 33 (4.2), 171 (B)
linear classifiers, 29 (4.1), 32 (4.2)
loading problem, 30 (4.1), 36 (4.3)
local minima, 41 (4.4)

mean squared error, 25 (3.4)
Modular Neural Networks
bagging, 53 (5.3)
boosting, 53 (5.3)
committees of networks, 52 (5.3)
mixtures of experts, 53 (5.3)
recursively defined mixture of experts, 54 (5.3)

Nativism, *see* Connectionism
np
BackpropagateEntity, 210 (F)
ColumnsArithmetic, 193 (F)
CreateEntity, 207 (F)
CreateSingle, 203 (F)
DeleteObjects, 195 (F)
ExtractColumnsFromObject, 192 (F)
FormatImagesDataSet, 200 (F)
FormatTimeSeriesDataSet, 198 (F)
MergeObjects, 193 (F)
ProduceAndFormatVectoredDataSet, 196 (F)
TestEntity, 212 (F)
TestSingle, 206 (F)
TrainEntity, 209 (F)
TrainSingle, 204 (F)
alphabetical listing of instructions, 218 (F)

introduction, 67 (5.5)
parallel execution, 191 (F)
reference guide, 189 (F)
running, 191 (F)
some examples, 221 (G)
structure, 68 (5.5)

order of a perceptron, 32 (4.2)
over-fitting, 177 (C)

PAC, *see* Probably Approximately Correct
Pavlov's dogs, 10 (2.2)
Perceptron, 10 (2.2)
dichotomies, 34 (4.2)
generalisation vs memorisation, 35 (4.2)
limitations, 33 (4.2)
physical symbol system hypothesis, 7 (2.1)
pragmatic error, 174 (C)
premature neuron saturation, 31 (4.1), 41 (4.4)
Probably Approximately Correct, 175 (C)

reductionism, 8 (2.1)
representation of functions
approximate representation, 25 (3.3)
exact representation, 24 (3.3)
universal function approximator, 24 (3.3)
robustness, *see* Connectionism
rule-based systems, 8 (2.1)

sample error, 174 (C)
schemata, 7 (2.1)
Scientific Method, 8 (2.1)
scripts, 7 (2.1)
shattering, 175 (C)
sigmoids, family of, 22 (3.2)
Small sampling theory, 183 (E)
Statistical significance tests
F-test, 185 (E)
introduction, 183 (E)
t-test, 184 (E)
Stone-Weierstrass theorem, 179 (D)
Support Vector Machines, 177 (C)
synthesis, 9 (2.1)

this thesis
contribution, 3 (1.2)
motivation, 2 (1.1)
structure, 5 (1.3)
top-down, 9 (2.1)

VC dimension, 175 (C)

XOR problem, 12 (2.2)