

**Putting Artificial Intelligence into Wearable Human-
Machine Interfaces – Towards a Generic, Self-Improving
Controller**

Marcel Admiraal

Imperial College London

Department of Mechanical Engineering

Ph.D. Dissertation

Declaration of Originality

I declare that:

The content of this thesis is original and of my own work. All assistance received and sources of materials (data, theoretical analysis, figures, and text) have been acknowledged; and the work has not been submitted to any other institution for any degree or diploma.

Declaration of Copyright

The copyright of this thesis remains with the author and is made available under the Creative Commons Attribution and Share-alike licence version 4. This work may be copied, distributed, displayed and derivative works created if the author is attributed and derivative works made available under an identical or less restrictive license. The full Creative Commons Attribution and Share-alike licence is available at <https://creativecommons.org/licenses/by-sa/4.0/>.

The copyright of the software created also remains with the author and is made available under the GNU General Public Licence version 2, and specifically not any later version. The libraries and programmes are free software, they can be redistributed and modified under the terms of the licence. They are shared in the hope that they will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose. The full GNU General Public License version 2 is available at <https://www.gnu.org/licenses/old-licenses/gpl-2.0.html>.

Acknowledgements

I would like to thank my supervisor Ravi Vaidyanathan and fellow research group members for all their contributions and my friends and family for their continual support and encouragement.

Abstract

The standard approach to creating a machine learning based controller is to provide users with a number of gestures that they need to make; record multiple instances of each gesture using specific sensors; extract the relevant sensor data and pass it through a supervised learning algorithm until the algorithm can successfully identify the gestures; map each gesture to a control signal that performs a desired outcome. This approach is both inflexible and time consuming.

The primary contribution of this research was to investigate a new approach to putting artificial intelligence into wearable human-machine interfaces by creating a Generic, Self-Improving Controller. It was shown to learn two user-defined static gestures with an accuracy of 100% in less than 10 samples per gesture; three in less than 20 samples per gesture; and four in less than 35 samples per gesture. Pre-defined dynamic gestures were more difficult to learn. It learnt two with an accuracy of 90% in less than 6,000 samples per gesture; and four with an accuracy of 70% after 50,000 samples per gesture.

The research has resulted in a number of additional contributions:

- The creation of a source-independent hardware data capture, processing, fusion and storage tool for standardising the capture and storage of historical copies of data captured from multiple different sensors.
- An improved Attitude and Heading Reference System (AHRS) algorithm for calculating orientation quaternions that is five orders of magnitude more precise.
- The reformulation of the regularised TD learning algorithm; the reformulation of the TD learning algorithm applied the artificial neural network back-propagation algorithm; and the combination

of the reformulations into a new, regularised TD learning algorithm applied to the artificial neural network back-propagation algorithm.

- The creation of a Generic, Self-Improving Predictor that can use different learning algorithms and a Flexible Artificial Neural Network.

Table of Contents

Abstract.....	2
Table of Figures.....	10
Table of Tables.....	17
Table of Acronyms.....	18
Chapter 1	
Introduction.....	20
1.1 The Need for a Generic, Self-Improving Controller.....	20
1.2 Research Goals.....	22
1.2.1 Research Objectives.....	22
1.2.2 Research Scope.....	23
1.2.3 Research Constraints.....	24
1.3 Original Contributions.....	25
1.4 Publications.....	26
1.5 Document Structure.....	27
Chapter 2	
Literature Review.....	31
2.1 Introduction.....	31
2.2 Learning Human Hand Gestures.....	33
2.3 Requirements for Real-Time Device Control.....	35
2.4 Capturing Muscle Activation Data.....	37
2.5 Recent Successes and Ongoing Challenges with Using Machine Learning and Artificial Neural Networks.....	42
2.6 Using Machine Learning to Play Games.....	46
2.7 Using Machine Learning in Device Control.....	51
2.8 Sensor Data Fusion.....	54
2.9 Summary.....	58
Chapter 3	
Source-Independent Hardware Data Capture, Processing, Fusion and Storage.....	60

3.1	Introduction.....	60
3.2	IMU Type Independent Data Capture Software Interface.....	62
3.3	IMU Device Interfaces.....	67
3.3.1	x-IMU.....	68
3.3.2	nuIMU.....	71
3.3.3	nuIMU Recorded Data.....	74
3.4	IMU Specific Software.....	76
3.4.1	x-IMU GUI.....	76
3.4.2	nuIMU GUI.....	86
3.4.3	nuIMU Firmware Updates.....	91
3.5	IMU Type Independent Control of a Virtual Robot.....	93
3.6	Real-Time Digital Signal Processing.....	97
3.6.1	Simple Moving Average Filter.....	99
3.6.2	Median Filter.....	99
3.6.3	Short-Term Energy Filter.....	100
3.6.4	Simple Low-Pass Filter.....	101
3.6.5	Simple High-Pass Filter.....	101
3.6.6	Butterworth Difference Filters.....	102
3.6.7	Envelope Filters.....	103
3.6.8	Trigger Filters.....	104
3.6.9	Multi-Filter.....	105
3.6.10	Testing the Digital Signal Filters.....	107
3.7	Fusing Multiple Sensors' Data.....	107
3.8	Summary.....	110

Chapter 4

An Improved AHRS Orientation Quaternion Algorithm.....	112
4.1 Introduction.....	112
4.2 Overview of Quaternions, Orientation and Correction.....	113
4.3 Evaluation of the Madgwick AHRS Algorithm.....	121
4.4 Improved AHRS Orientation Quaternion Algorithm Formulation.....	124
4.5 Algorithm Simulation Environment.....	132
4.6 Algorithm Comparison.....	136
4.7 Summary.....	152

Chapter 5

Formulation of New Machine Learning Algorithms and Policies.....	154
5.1 Introduction.....	154
5.2 Overview of TD Learning.....	156
5.3 Regularised TD Learning Algorithm.....	158
5.4 TD Learning Applied to the Artificial Neural Network Back-Propagation Algorithm.....	161
5.5 Regularised TD Learning Applied to the Artificial Neural Network Back-Propagation Algorithm.....	164
5.6 Prioritised Reinforcement Learning Policy.....	164
5.7 Blended Control Policy.....	168
5.8 Summary.....	170

Chapter 6

The Generic, Self-Improving Predictor.....	172
6.1 Introduction.....	172
6.2 Flexible Artificial Neural Network.....	173
6.3 Temporal Learning Interface.....	178
6.4 MCMC Learner.....	179
6.5 MC Network Learner.....	180
6.6 TD Network Learner.....	182
6.7 Summary.....	182

Chapter 7

The Generic, Self-Improving Controller.....	185
7.1 Introduction.....	185
7.2 Creating the Generic, Self-Improving Controller.....	187
7.3 Managing the Generic, Self-Improving Controller.....	188
7.3.1 Managing the IMU Data Collector.....	189
7.3.2 Managing the Generic, Self-Improving Predictor.....	190
7.3.3 Managing the Blended Mouse Control Policy.....	194
7.4 Training the Generic, Self-Improving Controller.....	195
7.5 Cross-Platform Mouse Pointer Override System.....	204
7.6 Blended Mouse Control Policy.....	206
7.7 Summary.....	207

Chapter 8

Testing the Generic, Self-Improving Predictor.....	209
8.1 Introduction.....	209
8.2 Testing the Artificial Neural Network.....	210
8.3 Testing the Iterative Learners.....	219
8.4 Creating a Generic Game Interface to Test the Generic, Self-Improving Predictor.....	223
8.5 Testing the Generic, Self-Improving Predictor by Playing Tic-Tac-Toe.....	227
8.5.1 Learning Tic-Tac-Toe Using the MCMC Learner.....	229
8.5.2 Learning Tic-Tac-Toe Using the Flexible Artificial Neural Network Based Learners	231
8.5.3 Learning Tic-Tac-Toe Using the TD Network Learner.....	240
8.6 Testing the Generic, Self-Improving Predictor by Playing Backgammon.....	242
8.7 Summary.....	247

Chapter 9

Testing the Generic, Self-Improving Controller.....	249
9.1 Introduction.....	249
9.2 Predicting User-Defined Static Gestures.....	251
9.2.1 Experimental Set Up.....	251
9.2.2 Experimental Results.....	253
9.2.3 Discussion of the Results.....	264
9.3 Predicting Pre-Defined Dynamic Gestures.....	265
9.3.1 Experimental and Simulation Set Up.....	266
9.3.2 Simulation Results.....	268
9.3.3 Discussion of the Results.....	289
9.4 Summary.....	291

Chapter 10

Conclusion and Future Work.....	294
10.1 Thesis Summary.....	294
10.2 Achievements.....	297
10.3 Future Work.....	298
10.4 Final Thoughts.....	301
References.....	304

Appendix A Software Created.....	318
Appendix A.1 Software Selection Decisions.....	318
Appendix A.2 Library and Application Dependencies.....	321
Appendix A.3 MAUtils Library.....	322
Appendix A.3.1 String Utilities.....	322
Appendix A.3.2 Maths Utilities.....	322
Appendix A.3.3 Communication Utilities.....	322
Appendix A.3.4 Collection Utilities.....	323
Appendix A.3.5 Optimisation Utilities.....	323
Appendix A.3.6 Conjugate Gradient Descent Algorithm.....	323
Appendix A.3.7 Logging Utilities.....	323
Appendix A.3.8 Other Utilities.....	324
Appendix A.3.9 Algorithms.....	324
Appendix A.3.10 Compile-Time Series Calculations.....	325
Appendix A.4 MawxUtils Library.....	325
Appendix A.4.1 Control Utilities.....	325
Appendix A.4.2 Display Utilities.....	325
Appendix A.4.3 Factory Utilities.....	326
Appendix A.5 MAANN Library.....	326
Appendix A.5.1 Network Training.....	326
Appendix A.6 MADSP Library.....	327
Appendix A.6.1 Filters.....	327
Appendix A.7 MAIMU Library.....	328
Appendix A.8 MAGame Library.....	328
Appendix A.8.1 Reinforcement Learning Policies.....	328
Appendix A.9 MAIMUData Library.....	329
Appendix A.9.1 IMU Data Collection.....	329
Appendix A.10 Tic-Tac-Toe Application.....	329
Appendix A.11 Backgammon Application.....	329
Appendix A.12 nuIMU Data Viewer Application.....	329
Appendix A.13 x-IMU GUI Application.....	330
Appendix A.14 nuIMU GUI Application.....	330
Appendix A.15 IMU Body Movement Application.....	330

Appendix A.16 Action Predictor GUI Application.....	330
Appendix B Overview of Machine Learning and Artificial Neural Networks.....	331
Appendix B.1 Introduction.....	331
Appendix B.2 Linear Regression and Classification.....	333
Appendix B.3 Non-Linear Regression and Classification.....	335
Appendix B.4 Regularisation.....	336
Appendix B.5 Artificial Neural Networks.....	337
Appendix B.6 Minimisation.....	345
Appendix B.7 TD Learning.....	349
Appendix B.8 TD Learning in Artificial Neural Networks.....	354
Appendix B.9 Adaptive Control.....	355
Appendix B.10 Reinforcement Learning.....	357
Appendix B.11 Hyper-Parameters.....	362

Table of Figures

Figure 1.1: High-level design of the Generic, Self-Improving Controller.....	27
Figure 3.1: Data capture software interface overview for simultaneously capturing data from multiple, different IMUs and presenting only the required data to multiple applications in a standardised form.....	63
Figure 3.2: Cross-platform Serial Panel used to manage serial ports.....	65
Figure 3.3: MMG external sensor used to capture muscle activity.....	68
Figure 3.4: x-io Technologies x-IMU.....	69
Figure 3.5: nuIMU.....	71
Figure 3.6: Cross-platform x-IMU GUI's serial port connection panel.....	77
Figure 3.7: Cross-platform x-IMU GUI's register display and modification panel.....	78
Figure 3.8: Cross-platform x-IMU GUI's date and time display and modification panel.....	82
Figure 3.9: Cross-platform x-IMU GUI's command panel.....	83
Figure 3.10: Cross-platform x-IMU GUI's sensor data panel.....	84
Figure 3.11: Cross-platform x-IMU GUI's Gyroscope Graph.....	85
Figure 3.12: Orientation quaternion visualisation.....	86
Figure 3.13: Cross-platform nuIMU GUI.....	87
Figure 3.14: Three-dimensional vector data visualisation.....	88
Figure 3.15: Cross-platform nuIMU GUI's online selection of the nuIMU Data Interface Type panel.....	89
Figure 3.16: nuIMU GUI's register configuration panel.....	90
Figure 3.17: IMU controlled Virtual Robot.....	94
Figure 3.18: Specifying the number, placement and starting orientations of the Virtual Robot's IMUs.....	95
Figure 3.19: Tracking body motion using IMUs.....	96
Figure 3.20: Multi-Filter configuration tool.....	105

Figure 3.21: Testing and comparing real-time, real-time recorded and full-speed recorded filters.	106
Figure 3.22: Interface for configuring the IMU Data Collector.....	109
Figure 4.1: Simultaneously comparing the orientation quaternion calculated by multiple algorithms using various combinations of sensors.....	133
Figure 4.2: Gyroscope, accelerometer and magnetometer device simulation environment.....	134
Figure 4.3: AHRS orientation quaternion algorithm controls.....	136
Figure 4.4: Comparison between errors in the Madgwick AHRS algorithm and the Improved AHRS algorithm after 1,000 iterations under different rotational speeds around the x-axis...	138
Figure 4.5: Comparison between errors in the Madgwick AHRS algorithm and the Improved AHRS algorithm after 1,000 iterations under different rotational speeds around the y-axis...	138
Figure 4.6: Comparison between errors in the Madgwick AHRS algorithm and the Improved AHRS algorithm after 1,000 iterations under different rotational speeds around the z-axis...	139
Figure 4.7: Errors in the Improved AHRS algorithm after 1,000 iterations under different rotational speeds around the x-axis at five-orders of magnitude greater resolution.....	139
Figure 4.8: Errors in the Improved AHRS algorithm after 1,000 iterations under different rotational speeds around the y-axis at five-orders of magnitude greater resolution.....	140
Figure 4.9: Errors in the Improved AHRS algorithm after 1,000 iterations under different rotational speeds around the z-axis at five-orders of magnitude greater resolution.....	140
Figure 4.10: Impact of the step size of the gradient descent algorithm (α) on the precision of the Improved AHRS algorithm under different rotational speeds around the x-axis.....	142
Figure 4.11: Impact of the step size of the gradient descent algorithm (α) on the precision of the Improved AHRS algorithm under different rotational speeds around the y-axis.....	142
Figure 4.12: Impact of the step size of the gradient descent algorithm (α) on the precision of the Improved AHRS algorithm under different rotational speeds around the z-axis.....	143
Figure 4.13: Impact of varying α on the precision of the fixed-point, shifter-integer version of the Improved AHRS algorithm under different rotational speeds around the x-axis.....	144
Figure 4.14: Impact of varying α on the precision of the fixed-point, shifter-integer version of the Improved AHRS algorithm under different rotational speeds around the y-axis.....	145

Figure 4.15: Impact of varying α on the precision of the fixed-point, shifter-integer version of the Improved AHRS algorithm under different rotational speeds around the z-axis.....	145
Figure 4.16: Impact of the stated precision of the gyroscope on the Improved AHRS algorithm with different gradient descent step-sizes (α).....	147
Figure 4.17: Impact of the stated precision of the accelerometer on the Improved AHRS algorithm with different gradient descent step-sizes (α).....	148
Figure 4.18: Impact of the stated precision of the magnetometer on the Improved AHRS algorithm with different gradient descent step-sizes (α).....	148
Figure 4.19: Impact of the measured precision of the gyroscope on the Improved AHRS algorithm with different gradient descent step-sizes (α).....	150
Figure 4.20: Impact of the measured precision of the accelerometer on the Improved AHRS algorithm with different gradient descent step-sizes (α).....	151
Figure 4.21: Impact of the measured precision of the magnetometer on the Improved AHRS algorithm with different gradient descent step-sizes (α).....	151
Figure 7.1: Action Predictor GUI used to manage and train the Generic, Self-Improving Controller	188
Figure 7.2: Action Predictor GUI's Inputs menu.....	189
Figure 7.3: Action Predictor GUI's Learning menu.....	190
Figure 7.4: Action Predictor GUI's New Learner type selection panel.....	191
Figure 7.5: Warning presented when attempting to modify the TD algorithm's prediction discount rate when not using a TD learning algorithm.....	193
Figure 7.6: Action Predictor GUI's Outputs menu.....	194
Figure 7.7: Action Predictor GUI's Instruction Editor.....	196
Figure 7.8: Action Predictor GUI's File menu.....	197
Figure 7.9: Action Predictor GUI's toolbar showing it updating to reflect the current state.....	200
Figure 8.1: Impact of varying the number of hidden neurons (n) when batch training an artificial neural network to classify ten 20x20 grey-scale hand-written numerals from 5000 samples.....	211

Figure 8.2: Impact of varying the regularisation factor (γ) when batch training an artificial neural network with 20 hidden neurons to classify ten 20x20 grey-scale hand-written numerals from 5000 samples.....	212
Figure 8.3: Impact of varying the learning rate (α) when using an artificial neural network with 20 hidden neurons to learn to classify ten 20x20 grey-scale hand-written numerals one image at a time.....	214
Figure 8.4: Impact of varying the number of hidden neurons (n) when using an artificial neural network with a learning rate of 0.001 to learn to classify ten 20x20 grey-scale hand-written numerals one image at a time.....	215
Figure 8.5: Impact of varying the regularisation factor (γ) when using an artificial neural network with 20 hidden neurons and a learning rate of 0.001 to learn to classify ten 20x20 grey-scale hand-written numerals one image at a time.....	216
Figure 8.6: Impact of varying the learning rate (α) when using an artificial neural network with 20 hidden neurons and a regularisation factor of 0.002 to learn to classify ten 20x20 grey-scale hand-written numerals one image at a time.....	218
Figure 8.7: Comparing the number of iterations required to learn logic gates averaged over ten simulations using (a) the MCMC Learner and (b) Flexible Artificial Neural Network based learners with one hidden layer with four neurons, and a learning rate of 0.1....	220
Figure 8.8: Impact of varying the learning rate (α) on the number of iterations required to learn (a) AND, (b) OR, (c) NAND and (d) XOR logic gates using each of the Iterative Learners	221
Figure 8.9: Comparing the learning speeds of different learners solving the five position one dimensional random walk problem with different learning rates (α).....	222
Figure 8.10: Watching the Generic, Self-Improving Predictor learning to play Tic-Tac-Toe.....	228
Figure 8.11: Impact of varying the learning rate (α) when using the MCMC Learner to learn to play Tic-Tac-Toe.....	230
Figure 8.12: Initial results from learning to play Tic-Tac-Toe using the MC Network Learner (a) and the TD Network Learner (b).....	232
Figure 8.13: Results from using a binary game state representation to learn to play Tic-Tac-Toe using the MC Network Learner (a) and the TD Network Learner (b).....	233

Figure 8.14: Impact of varying the regularisation factor (γ) when learning to play Tic-Tac-Toe using an artificial neural network learner with a learning rate (α) of 0.0005, 18 hidden neurons and a binary representation of the game state.....	234
Figure 8.15: Impact of varying the regularisation factor (γ) when learning to play Tic-Tac-Toe using the MC Network Learner with 15 hidden neurons, a binary representation of the game state, and a learning rate (α) of (a) 0.0005, (b) 0.0032, (c) 0.025, (d) 0.05.....	235
Figure 8.16: Impact of varying the learning rate (α) when using an MC Network Learner to learn to play Tic-Tac-Toe with (a) 5, (b) 10, (c) 20 and (d) 25 hidden neurons and a binary representation of the game state.....	237
Figure 8.17: Impact of varying the temporal discount rate (β) when using an MC Network Learner and a binary representation of the game to learn to play Tic-Tac-Toe with a learning rate of (a) 0.05, (b) 0.025, (c) 0.0125 and (d) 0.0064.....	238
Figure 8.18: Learning to play Tic-Tac-Toe with the MC Network Learner, 15 hidden neurons, a temporal discount rate (β) of 0.7 and various learning rates (α).....	240
Figure 8.19: Impact of varying the weight preference (λ) using a TD Network Learner with 15 hidden neurons and a learning rate (α) of (a) 0.05, (b) 0.025, (c) 0.0125 and (d) 0.0064	241
Figure 8.20: Watching the Generic, Self-Improving Predictor learning to play Backgammon.....	244
Figure 8.21: Results from different Generic, Self-Improving Predictors learning to play Backgammon by playing 100,000 games.....	246
Figure 9.1: Strap used to attach an IMU and multiple MMGs to the user's forearm.....	250
Figure 9.2: Comparing the number of samples per gesture vs. predictive accuracy required to predict increasing numbers of user-defined static gestures.....	263
Figure 9.3: Initial results from learning to predict two pre-defined dynamic gestures.....	269
Figure 9.4: Learning to predict two pre-defined gestures using historical MMG data with (a) 200 ms of historical data and (b) 100 ms of historical data and an extended artificial neural network hidden layer.....	270
Figure 9.5: Learning two pre-defined dynamic gestures using 100 ms of historical data and 9,000 simulated samples per gesture.....	272

Figure 9.6: Learning two pre-defined dynamic gestures using 200 ms of historical data and 9,000 simulated samples per gesture.....	273
Figure 9.7: Learning two pre-defined dynamic gestures using 500 ms of historical data and 9,000 simulated samples per gesture.....	274
Figure 9.8: Learning two pre-defined dynamic gestures using 500 ms of historical data and 9,000 simulated samples per gesture with a learning rate of (a) 0.001, (b) 0.0001, (c) 1×10^{-5} , and (d) 1×10^{-6}	275
Figure 9.9: Learning two pre-defined dynamic gestures using data captured from an IMU and 8 MMGs at 1000 Hz and 9,000 simulated samples per gesture plotting the predictions after (a) every 10 th tranche and (b) every tranche.....	277
Figure 9.10: Consistently learning two pre-defined dynamic gestures using data captured from an IMU and 8 MMGs at 1000 Hz.....	277
Figure 9.11: Learning two pre-defined dynamic gestures using data captured from an IMU and 8 MMGs at 1000 Hz using (a) 200 ms of historic data without extending the length of the hidden layer, (b) 100 ms, (c) 200 ms and (d) 500 ms of historic data and extending the hidden layer to twice the length of the input state vector.....	278
Figure 9.12: Learning two pre-defined dynamic gestures using data captured from an IMU and 8 MMGs at 1000 Hz and 90,000 simulated samples per gesture.....	279
Figure 9.13: Learning two pre-defined dynamic gestures using data captured from an IMU and 8 MMGs at 1000 Hz and 90,000 simulated samples per gesture using a learning rate of (a) 1×10^{-2} , (b) 1×10^{-3} , (c) 1×10^{-4} and (d) 1×10^{-5}	280
Figure 9.14: Learning three pre-defined dynamic gestures using data captured from an IMU and 8 MMGs at 1000 Hz and 90,000 simulated samples per gesture.....	282
Figure 9.15: Learning four pre-defined dynamic gestures using data captured from an IMU and 8 MMGs at 1000 Hz and 90,000 simulated samples per gesture.....	283
Figure 9.16: Learning five pre-defined dynamic gestures using data captured from an IMU and 8 MMGs at 1000 Hz and 90,000 simulated samples per gesture.....	284
Figure 9.17: Learning six pre-defined dynamic gestures using data captured from an IMU and 8 MMGs at 1000 Hz and 90,000 simulated samples per gesture.....	285

Figure 9.18: Comparing the number of samples per gesture vs. predictive accuracy required to predict increasing numbers of pre-defined dynamic gestures with a learning rate of 0.001.....	286
Figure 9.19: Learning (a) three, (b) four, (c) five and (d) six pre-defined dynamic gestures using data captured from an IMU and 8 MMGs at 1000 Hz and 90,000 simulated samples per gesture and a learning rate of 0.0001.....	287
Figure 9.20: Comparing the number of samples per gesture vs. predictive accuracy required to predict increasing numbers of pre-defined dynamic gestures with a learning rate of 0.0001.....	288
Figure A.1: Software library and application dependencies.....	321

Table of Tables

Table 3.1: Comparison between 16 bit (2 byte) precision available and the specified precision of the LSM9DS0 accelerometer, gyroscope and magnetometer components.....	73
Table 9.1: Results from learning to predict two user-defined static gestures.....	253
Table 9.2: Results from learning to predict three user-defined static gestures.....	256
Table 9.3: Results from learning to predict four user-defined static gestures – Part 1.....	258
Table 9.4: Results from learning to predict four user-defined static gestures – Part 2.....	259
Table 9.5: Results from learning to predict five user-defined static gestures – Part 1.....	260
Table 9.6: Results from learning to predict five user-defined static gestures – Part 2.....	261
Table 9.7: Results from learning to predict five user-defined static gestures – Part 3.....	262
Table 9.8: Per subject dynamic gesture metadata.....	266

Table of Acronyms

ACRL	Actor-Critic Reinforcement Learning
AHRS	Attitude and Heading Reference System
BCI	Brain Computer Interface
CNN	Convolutional Neural Networks
DARPA	Defense Advanced Research Projects Agency
DPS	Degrees per Second
DOF	Degrees of Freedom
EEG	Electroencephalography
EMG	Electromyogram
EPP	Extended Physiological Proprioception
GUI	Graphical User Interface
HMI	Human-Machine Interface
ICA	Independent Components Analysis
IID	Independent and Identically Distributed
IMU	Inertial Measurement Unit: Devices used to simultaneously capture data from multiple hardware sensors.
IO	Input-Output
LSTM	Long Short-Term Memory

MARG	Magnetic, Angular Rate and Gravity: Combined sensor chip that includes a three axis accelerometer, a three axis gyroscope and a three axis magnetometer
MCMC	Markov Chain, Monte-Carlo
MDP	Markov Decision Process
MEMS	Microelectromechanical System
MMG	Mechanomyogram
MNIST	Modified National Institute of Standards and Technology
NAND	Not And
PCA	Principal Component Analysis
PWM	Pulse Width Modulation
RBM	Restricted Boltzmann Machine
RC	Resistance-Capacitance
RMS	Root-Mean-Square
RNN	Recurrent Neural Network
SDK	Software Development Kit
SEM	Standard Error of the Mean
TD	Temporal Difference
UART	Universal Asynchronous Receiver Transmitter
XOR	Exclusive Or

Chapter 1

Introduction

1.1 The Need for a Generic, Self-Improving Controller

The main challenge in human-machine interfaces (HMIs) remains the inability to adequately recognize the actions and intentions of the human. Furthermore, disabilities prevent many people from using the HMIs that we use everyday. For example, it's hard to use a computer mouse without a hand.

Chapter 2 will show that there is a large body of research using various sensors to control various devices, but they all require considerable engineering effort. Furthermore, combining multiple sensors remains difficult. In addition, these efforts are not transferable to other sensors or other

devices. Although some generic controllers have been created recently, these controllers remain limited to playing games and do not interact with the real-world.

The best generic controller we know is the human brain. It consists of millions of individual, noisy sensors. It successfully integrates the signals from these heterogeneous sensors to provide adaptable and reliable control of dextrous limbs. We are able to move in continuously changing conditions, while learning how to control any number of new tools we choose to use.

Noisy sensor measurements can be overcome by either increasing the fidelity of the instrumentation, or by increasing the number of sensors. Using statistical optimisation, measurements can be improved by utilising different types of sensor with different qualities. The same way the human body combines vision, which is accurate but slow, with our reflexes, which are inaccurate but fast, to maintain balance during high velocity and high acceleration movements. Existing sensors are not the limitation. What is missing is an easy way to combine different types of sensors.

Current machine learning techniques have eliminated the need for accurate models to successfully map enormous amounts of input data onto desired outcomes. These mappings can be learnt and then used to programme feed-forward controllers. The standard approach to creating a machine learning based controller is:

- Provide users with a number of gestures that they need to make.
- Record multiple instances of each gesture using specific sensors.
- Extract the relevant sensor data and pass it through a supervised learning algorithm until the algorithm can successfully identify the gestures.
- Map each gesture to a control signal that performs a desired outcome.

This approach has been used by many researchers, but it is time consuming and inflexible. Feed-forward controllers are also inadequate for developing self-improving control systems. Even for fixed tasks, the distribution of input patterns might and does change over time.

It would be useful if a controller could be created that would continually adapt to changing conditions. A controller that was intuitive to use, because the user decided what gestures to use. A controller that was generic enough to use different input sensors and control different devices depending on need. This research introduces a new approach and new tools for creating a generic, adaptable and intuitive controller: a Generic, Self-Improving Controller.

1.2 Research Goals

The focus of the research group is in creating low-cost wearable robotics for the disabled; especially for disadvantaged people and people in under-developed countries. The aim of this research was to investigate a new approach to putting artificial intelligence into wearable human-machine interfaces. This was to be achieved by a creating a Generic, Self-Improving Controller. It would be generic, because first, it could use multiple, different sensors as input, and second, its output could be used to control any device. It would be self-improving, because it can be continuously and iteratively trained to use new data and feedback to improve the control signals generated while being used.

1.2.1 Research Objectives

To achieve the research aim, the specific research objectives were:

- Combine the signals from multiple different hardware sensors to create a standard input signal that is independent of the actual sensors used. The specific hardware sensors to be used will be

inertial measurement unit (IMU)¹ that are used to measure movement and orientation; and multiple mechanomyographic (MMG) sensors that are used to measure muscle activation. The IMU and MMG sensors will be attached to users' forearms. The IMUs will measure forearm movement and orientation. The MMGs will measure forearm muscle activation to capture hand and finger movement.

- Develop a self-improving predictor by using artificial intelligence to continually improve the conversion of a standard input signal into the desired output signal. The output signal should also be standardised, so it can be used to control various devices.
- Use standard machine learning testing methods to test the functionality of the self-improving predictor and its components. Use games to create a controlled, simulated environment with automatic feedback.
- Combine the hardware-independent data capture solution with the self-improving predictor to create a self-improving controller.
- Test the self-improving controller by using user-defined gestures to control a mouse pointer. To make the control intuitive, allow users to define gestures they want to use.
- Determine the limits of the solutions created.

1.2.2 Research Scope

The scope of this research includes:

- Writing all the required software. The software development includes updating the firmware of the hardware when required and available. It also includes creating the simulation environments

¹ During the course of this research, the term IMU (Inertial Measurement Unit) is used to describe the devices used to simultaneously capture data from multiple hardware sensors. The term MARG (Magnetic, Angular Rate and Gravity) is combined sensor chip. It includes a three axis accelerometer (measuring changes in linear inertia), a three axis gyroscope (measuring the rotational inertia), and a three axis magnetometer. Each IMU includes a MARG chip.

required to test the software developed and generate the simulated data for the simulation experiments. A list of the software created is provided in Appendix A.

- Developing and improving any algorithms required to achieve the objectives.

The scope of this research excludes:

- Developing and creating the sensors. Other members of the research team are actively developing and creating the new IMUs and MMGs that were used in this research.

1.2.3 Research Constraints

This research had the following constraints:

- **Budget:** The budget available for equipment and materials was £2,000 per year. This is aligned with the research group's focus on creating low-cost wearable robotics.
- **Equipment availability:** The new IMUs and MMGs used in this research were developed in parallel by other members of the research group. They only became available mid-way through this research. Furthermore, only a limited number were produced, and they were shared with other members of the research group.
- **Equipment prioritisation:** There was a robotic arm and a robotic hand available for testing. However, this equipment was prioritised to other research projects. The availability to this research was insufficient to both learn how to use the equipment and apply it to meaningful experiments.

1.3 Original Contributions

This research details the design, development and initial testing of the Generic, Self-Improving Controller created. Its development and testing resulted in the making of a number of additional original contributions including:

- The creation of a source-independent hardware data capture, processing, fusion and storage tool for standardising the capture and storage of historical copies of data captured from multiple different sensors. It included the creation of a generic IMU interface to combine sensor input from multiple types of IMU, which eliminates the dependence on hard-coded APIs, and enables the quick integration of future technologies. It also included the reformulation of multiple, digital signal filters that provide the same results whether used in real-time, simulated-time or offline.
- An improved Attitude and Heading Reference System (AHRS) algorithm for calculating orientation quaternions. The improved algorithm results in a five order of magnitude improvement in the precision of the calculated orientation quaternions.
- The reformulation of the regularised TD learning algorithm; the reformulation of the TD learning algorithm applied the artificial neural network back-propagation algorithm; and the combination of the reformulations into a new, regularised TD learning algorithm applied to the artificial neural network back-propagation algorithm.
- The creation of a Generic, Self-Improving Predictor that can use different learning algorithms. The input and output interfaces of the Generic, Self-Improving Predictor are simple vectors. Therefore, not only could it be used in the Generic, Self-Improving Controller, but it can also be used in any iterative learning environment. It includes the creation of a Flexible Artificial Neural Network. The sizes of the layers in the Flexible Artificial Neural Network can be changed while it is being used without restarting the application and without losing previous learning.

Furthermore, the Flexible Artificial Neural Network can use any learning algorithm; including the new, regularised TD learning algorithm applied to the artificial neural network back-propagation algorithm.

1.4 Publications

This research has resulted in the following publications:

- Admiraal, M., Wilson, S. & Vaidyanathan, R.: Improved formulation of the IMU and MARG orientation gradient descent algorithm for motion tracking in human-machine interfaces. In: Multisensor Fusion and Integration for Intelligent Systems (MFI), 2017 IEEE International Conference on. 2017 IEEE. pp. 403–410.
- Angeles, P., Mace, M., Admiraal, M., Burdet, E., et al.: A Wearable Automated System to Quantify Parkinsonian Symptoms Enabling Closed Loop Deep Brain Stimulation. In: Conference Towards Autonomous Robotic Systems. 2016 Springer. pp. 8–19.

In addition, the following have been submitted for publication:

- Admiraal, M., & Vaidyanathan, R.: A Generic Learning Interface for Using Multiple Sensors to Continually Improve the Fidelity of Control of Multiple Devices. To: Swarm Human Blended Intelligence Conference.
- Fadhil, A., Kanneganti, R., Gupta, L., Admiraal, M. & Vaidyanathan, R.: Fusion of Enhanced and Synthetic Vision System Images for Runway and Horizon Detection. To: Information Fusion Journal. Elsevier.

1.5 Document Structure

The primary engineering goal of this project was to create the Generic, Self-Improving Controller. This document details how biomechatronics and computer science have been combined to design, develop and test the Generic, Self-Improving Controller. A controller that can be used with multiple, arbitrary sensors to control multiple, arbitrary devices. A controller that would learn using feedback to iteratively improve the desired outcomes. As illustrated in Figure 1.1, the Generic, Self-Improving Controller is comprised of three components: First, the filtering, recording and combining of multiple sensors' signals to create a standardised input signal that consists of either binary or normalised analogue data: the **Input**. Second, the regular processing of a temporal sequence of input values into predictions: the **Self-Improving Predictor**. The predictions are continuously updated based on feedback. The predictions will be used as probabilities for various desired outcomes. Third, the conversion of the predicted probabilities or desired outcomes into the required control signals: the **Control Policy**.

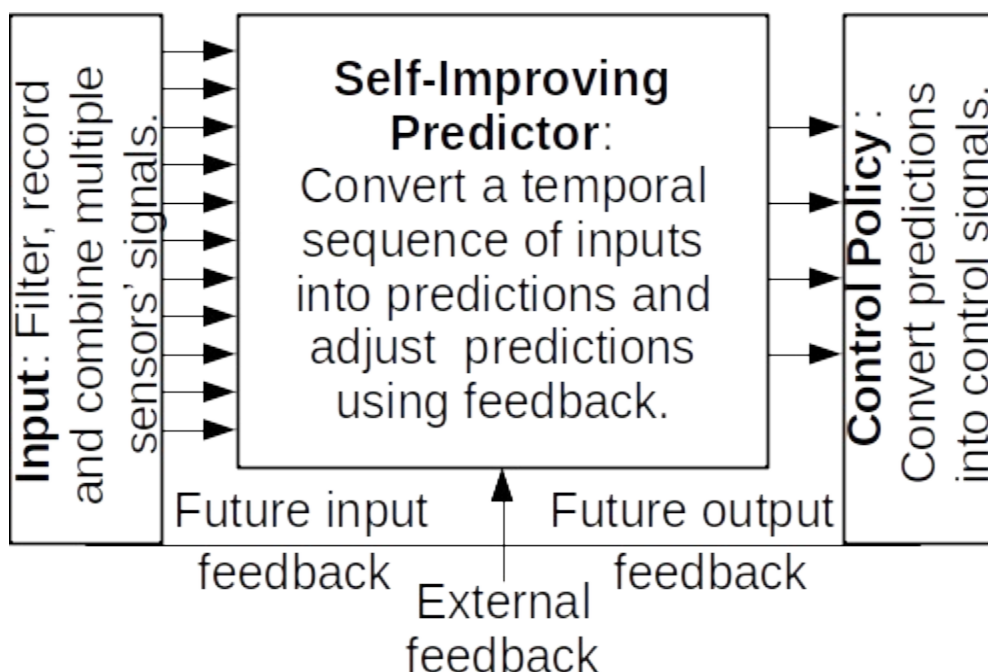


Figure 1.1: High-level design of the Generic, Self-Improving Controller.

Chapter 2 reviews the existing literature with regards to putting artificial intelligence into wearable human-machine interfaces (HMIs), and creating generic, self-improving controllers. The literature review includes the existing problems encountered with learning hand gestures; the requirements for real-time device control; the methods for capturing muscle activation; the recent successes and ongoing challenges faced when using machine learning; the use of machine learning in games; the use of machine learning in device control; and techniques used to fuse sensors.

Chapter 3 describes the creation of the **Input**: the left-hand side of Figure . It explains how data from multiple sensors are captured, filtered, recorded and combined into a format that is independent of the type or number of sensors used. It details how signals from multiple, different hardware devices are combined to create the source-independent input signal required by the Self-Improving Predictor. It also includes an overview of the specific hardware used; the creation of a generic device interface to enable simultaneous and hardware-independent data capture; and the creation of multiple, real-time digital filters.

During the creation of the Input module, the signals generated by the IMUs used in this research were reviewed. They included a quaternion for describing the orientation of the IMU. This quaternion was calculated using an AHRS orientation quaternion algorithm. It was found that the AHRS orientation quaternion algorithm could be improved. Chapter 4 describes the formulation of an improved AHRS orientation quaternion algorithm. It details the derivation of the improved formulation and describes how to best use magnetometer and accelerometer measurements to correct for the errors inherent in numerically integrating gyroscope measurements.

Chapter 5 describes the development of new machine learning algorithms. It includes the formulation of the new faster, regularised TD learning algorithm for the artificial neural network back-propagation algorithm, which was used in the Self-Improving Predictor. It provides a brief overview of an existing algorithm and the need to reformulate the algorithm when applied to

artificial neural networks. A more complete overview of machine learning, artificial neural networks and the other algorithms used in this research is provided in Appendix B. Chapter 5 also includes the development of the **Control Policy**: the right-hand side of Figure 1.1. A Blended Control Policy was developed that converts the Self-Improving Predictor's changing predictions into continually improving control signals.

Chapter 6 describes the development of the **Self-Improving Predictor**: the centre of Figure 1.1. A Generic, Self-Improving Predictor was developed so that it could be used in other iterative learning environments beyond the Generic, Self-Improving Controller. It includes the creation of a Temporal Learning Interface and three learning algorithms that are applied to it. It also includes the development of the Flexible Artificial Neural Network.

Chapter 7 describes how the above components are combined to create the **Generic, Self-Improving Controller**: the complete Figure 1.1. It includes the creation of the Action Predictor GUI, which is used to manage the Generic, Self-Improving Controller. The Action Predictor GUI also enables users to train the underlying Generic, Self-Improving Predictor using arbitrary, user-defined gestures. Chapter 7 also includes an implementation of the Blended Control Policy for controlling a cross-platform mouse controller: the Blended, Mouse Control Policy. Finally, it includes the linking of the Action Predictor GUI to a new cross-platform mouse controller. The cross-platform mouse controller enables users to use their chosen gestures to control the mouse pointer regardless of the operating system used.

Chapter 8 covers the testing of the different components of the Generic, Self-Improving Predictor. It includes separately testing the Flexible Artificial Neural Network and the Iterative Learners. The Flexible Artificial Neural Network is tested using standard supervised learning to predict hand-written digits. The Iterative Learners are first tested by training them to learn simple logic before

training them to learn hand-written digits. Finally, the Generic, Self-Improving Predictor is tested by training it to learn to play games.

Chapter 9 describes the testing of the Generic, Self-Improving Controller. First, the Generic, Self-Improving Controller is used to predict user-defined static gestures. Then, the Generic, Self-Improving Controller is tested using pre-defined dynamic gestures. It includes a discussion of the test results and the current limitations of the Generic, Self-Improving Controller.

Finally, Chapter 10 provides a summary of the results and achievements. It includes a discussion on how this research can be used and developed further.

Chapter 2

Literature Review

2.1 Introduction

The focus of the research group is on creating low-cost wearable robotics for the disabled; especially for disadvantaged people and people in under-developed countries. A current key research area is in improving the human-machine interfaces in prosthetics; especially artificial hands for amputees. The specific aim of this research was to investigate a new approach to putting artificial intelligence into wearable human-machine interfaces by creating a generic, self-improving controller that can combine multiple sensors and control various devices. Therefore, areas of direct relevance to this research include: understanding existing work done on learning human hand

gestures; the requirements for real-time device control; the previous use of machine learning in device control; and the previous approaches used to fuse data from multiple sensors.

There is a vast amount of literature in each of the relevant areas identified, while simultaneously there is very little literature on the specific subject of using artificial intelligence to fuse multiple sensor data in human-machine interfaces (HMIs). Therefore, literature reviews that were deemed closest to this research were used to structure the reading before the key insights from the relevant papers were collated and summarised.

Three literature reviews were selected. First, Novak and Riener's review of sensor fusion methods in wearable robotics. Novak and Riener's review was selected because, they state that there were no previous detailed reviews of sensor fusion for multi-modal systems. They also highlight that the main challenge in HMIs is the inability to adequately recognize the actions and intentions of the human (Novak & Riener, 2015). Second, Jaimes and Sebe's review of the major approaches and issues to multi-modal human-computer interaction was selected. Although their review is done from a computer vision perspective, they highlight that interactions are no longer limited to explicit control signals. Interactions now include gestures, speech, haptics, eye blinks, eye tracking and gaze detection. Their review explains why multi-modal interfaces are more robust, and reduce errors, and why most interfaces do not account for the social aspect of communication or that humans use emotion to enhance communication (Jaimes & Sebe, 2007). Finally, Asghari Oskoei and Hu's review of pattern and non-pattern recognition in electromyographic (EMG) control was selected, because they highlight the advantage of hands-free control that is based on the user's intention (Asghari Oskoei & Hu, 2007); a key principle behind this research.

Beyond these reviews it was necessary to review the work done on learning human hand gestures. This includes the feedback mechanisms used; both in the control of the device itself, and in the signals sent back to the user. The requirements for real-time device control required a review of the

sensors that have been used to detect users' motion and intentions. Furthermore, since games will be used to create the controlled, simulated environment with automatic feedback used for testing the self-improving predictor, a review of previous research in this field and machine learning in general was required. Of particular interest was Tesauro's success with using temporal difference learning to teach a predictor to play Backgammon at master level (Tesauro, 1995). Finally, using machine learning in device control also required a review of the previous approaches used.

The literature review for this research has been grouped and divided into the following sections. Section 2.2 reviews the work done in learning human gestures. Section 2.3 reviews previous work done to identify the requirements for real-time device control. Section 2.4 reviews the different ways that muscle activation data has been captured. Section 2.5 reviews the recent successes and ongoing challenges faced when using machine learning and artificial neural networks. Section 2.6 reviews the use of machine learning to play games. Section 2.7 reviews the previous use of machine learning in device control. Finally, Section 2.8 reviews previous research in sensor data fusion.

2.2 *Learning Human Hand Gestures*

Learning human gestures is hard, because the human hand is extremely complex. A hand has 22 degrees of freedom, which are controlled by about 38 muscles, and it incorporates about 17,000 tactile units (Zecca *et al.*, 2002). The mechanical control of the hand is further complicated by the wrist, which is controlled by the same muscles that cause the fingers to grip. Furthermore, the orientation of the hand is limited by the need to avoid awkward elbow and shoulder motions. Therefore, when creating artificial hands, it is necessary to simplify the design by limiting the available motions (Williams, 2011).

To decide which motions to make available, Peerdeman et al. first determined that grasps are more important than wrist movements. The three most important grasps are: the lateral – placing the

thumb on top of the fore-finger; the cylindrical – gripping a cylinder; and the tripod – a three finger pinch. In addition to these three grips, the index finger point gesture is also considered important for other activities such as typing. Once these basic grasps and gestures are available, the next most important motion requirement is for the flexing and extending of the wrist; and then rotating the wrist (Peerdeman *et al.*, 2011).

Modern electrical prosthetic hands have as many as 14 different grip positions (RSLSteeper, 2014). However, amputees often choose not to use them. The main reasons given for not using them include: they have limited dexterity; they only have one active degree of freedom (Bitzer & van der Smagt, 2006); the control is not intuitive; they lack sufficient feedback; they have insufficient functionality (Peerdeman *et al.*, 2011); and they do not react quickly to perturbations or changes in user intentions (Novak & Riener, 2015). Many amputees would rather use a simple cable-operated split hook or a voluntary close hook-type device, because they are more responsive and provide natural feedback. The body's proprioception enables users to sense how open the hook is, even in the dark. Whereas the best electrical prosthesis users get a feel for position by listening to the motors and sensing how long the device is taking to change position. This is not a good substitute for real proprioception (Williams, 2011). Better interfaces between the user and their devices are still greatly needed (Mamun *et al.*, 2012).

Feedback enables users to handle the inevitable uncertainty from feed forward control, and it is required to continually improve control through learning (Saunders & Vijayakumar, 2011).

Feedback on the position of the fingers significantly reduces the attention required and allows for more intuitive grasping. Continuous and proportional feedback improves a user's ability to handle the device intuitively, which makes it significantly easier to use. However, feedback should be unobtrusive and adjustable by the user (Peerdeman *et al.*, 2011). Even basic tactile feedback has been shown to significantly improve performance (Saunders & Vijayakumar, 2011). Force is considered the most important type of feedback, because it is impossible to determine through

visual inspection. Unfortunately, force sensitive resistors do not provide sufficient feedback, give poor proportional control, require an appreciable distance to be activated (Williams, 2011), and suffer from hysteresis (Castellini & van der Smagt, 2008).

In 1974, Simpson introduced the concept of extended physiological proprioception (EPP). EPP is the use of other joints to provide the feedback required for adequately controlling prostheses (Simpson, 1974). When prosthetic arm technology moved from body-powered to externally-powered systems, the control approach changed from position control with proprioceptive feedback, to open-loop velocity control with the only feedback being visual feedback. This is probably because the typical actuator is a DC electric motor, which is a rate-controlled device and does not draw power to maintain a particular position (Weir, 2003). Although Pistohl *et al.* have recently tried using the magnitude of EMG signals for proportional control to provide flexibility and a continuum of possible hand postures (Pistohl *et al.*, 2013), feedback remains visual.

2.3 Requirements for Real-Time Device Control

Device control is broken into multiple stages: sensors measure an input signal, which is filtered before features are extracted. Features are used to either generate a discrete or continuous control signal through classification or regression respectively (Novak & Riener, 2015). The control signal itself can be filtered to smooth the output, used as input into the next stage of a multi-stage controller, or used to directly control the required device.

Novak and Riener confirm that the main sensors used in HMIs that control exoskeletons and prosthetics are EMG, brain activity measurements, IMUs and pressure or force sensors (Novak & Riener, 2015).

The application of robotic technology to powered prosthetics, exoskeletons and remote-operated robot arms is still limited by the lack of a robust control strategy (Hargrove *et al.*, 2013; Gopura *et*

al., 2013; Alcaide-Aguirre, Morgenroth & Ferris, 2013). Users want more functions, lower reaction and execution times, and intuitive control and feedback systems (Peerdeman *et al.*, 2011). Given the choice, users prefer more functionality and a slower system over less functionality and a faster system (Hargrove *et al.*, 2010).

When the control of multiple degrees of freedom is non-intuitive, it places an excessive mental load on the user, which greatly diminishes any benefits the added functionality might offer (Weir, 2003). Therefore, devices should be designed to make control as predictable and repeatable as possible (Saunders & Vijayakumar, 2011). Peerdeman *et al.* suggest dividing the control of a device into three parts: identify the user intention, control the execution, and provide feedback. Users want to control the grasp selection and force, wrist movement speed, and initiate the actions (Peerdeman *et al.*, 2011). However, the actual execution of the actions should be performed automatically.

Artificial reflexes can be used to remove the operator from the control loop through the use of onboard intelligence (Weir, 2003). In addition, the device should continue holding an object and prevent it from slipping once grasped without continual user input (Williams, 2011).

It's also important to understand the impact of making an incorrect classification vs. rejecting a classification due to uncertainty. A rejected classification should not result in the same impact as an incorrect classification. Instead, a rejected classification should only result in a delay (Mace *et al.*, 2013).

Multiple controllers can be used in parallel to provide consensus (Englehart & Hudgins, 2003); control different devices; different components of the same device; or at different times – even using another controller to determine which controller to use (Novak & Riener, 2015).

2.4 Capturing Muscle Activation Data

Electromyographic (EMG) signals have been used to identify user motion intention since 1948 (Reiter, 1948). Maier and van der Smagt have shown how the accurate placement of only 10 EMG sensors can be used to detect the opening and closing, and sideways movements of all human fingers (Maier & van der Smagt, 2008).

However, not only do EMG signals vary with physical and physiological conditions, but also with the location of electrodes, with the shift of electrodes (Gopura *et al.*, 2013), with conductivity changes due to perspiration or humidity, with cognitive intent variations or contraction intensity changes (Sensinger, Lock & Kuiken, 2009), between subjects, with arm posture, with muscle fatigue (Stokes & Dalton, 1991; Castellini & van der Smagt, 2008), between sessions, with the size of the muscle, when the hand is rotated (Bitzer & van der Smagt, 2006), with body movement (Alcaide-Aguirre, Morgenroth & Ferris, 2013), with heel contact (Hargrove *et al.*, 2013), with time (Zecca *et al.*, 2002), and with muscle depth (Williams, 2011). There is also a limited number of independent control sources (Weir, 2003), they often require repeated calibration (Pilarski *et al.*, 2011), and result in delays of 100 to 125 ms² (Farrell & Weir, 2007). These problems, in addition to low speeds (Williams, 2011) and high costs, have led to their limited acceptance in prosthetic control (Alcaide-Aguirre, Morgenroth & Ferris, 2013).

Although EMGs are used to control commercial prostheses, there has not been any revolutionary development since the 1960s. The main changes include moving from simple on-off control to identifying muscle contraction rates (Asghari Oskoei & Hu, 2007).

There are two types of EMG signal: when the muscle transitions from rest to contraction; and steady-state during constant contraction. The transition signal requires the muscle to be initially at

² Delays of over 300 ms are considered unacceptable (Hudgins, Parker & Scott, 1993)(Hudgins, Parker & Scott, 1993).

rest, which is a problem when switching between states intuitively (Asghari Oskoei & Hu, 2007). The steady-state signal is more accurately classified and can use shorter segment lengths (Englehart & Hudgins, 2003). This makes the steady-state signal both better and faster (Asghari Oskoei & Hu, 2007).

Muscles produce sounds when they contract. The sounds are thought to be caused by the lateral oscillations of fibres and reflect the intrinsic mechanical activity of the muscle which is positively correlated with the force produced. The frequency of the sounds produced by voluntary muscle contraction is between 5 and 30 Hz (Stokes & Dalton, 1991); although these are the result of discrete, synchronous 400 μ s bursts from high-velocity, constant sarcomere length shortening periods (Orizio, 1993).

A mechanomyogram (MMG) is the signal detectable on the surface of a muscle when the muscle is contracted (Orizio, 1993), which can be recorded using a microphone. MMG recordings are known to be susceptible to contamination from limb movements, but pressure-based transducers can be used to minimise the impact of these movements (Posatskiy & Chau, 2012).

Posatskiy and Chau investigated the preferred geometry of transducer chambers and found that a rigid conical chamber 7 mm in diameter and 5 mm in height produced the highest gain and flattest response. They recommend combining this sized conical chamber with a low-frequency microelectromechanical system (MEMS) microphone and a 4 μ m aluminised Mylar membrane (Posatskiy & Chau, 2012). Woodward et al. have created an MMG sensor based on this design and connected it to the IMUs to simultaneously record limb orientation, acceleration and muscle activity (Woodward, Shefelbine & Vaidyanathan, 2014).

Although MMGs share many of the same problems experienced by EMGs, they have a number of advantages over using EMGs: MMGs are cheaper to produce, they can be used multiple times, and they do not require adhesives or electrical conductive gel (Woodward, Shefelbine & Vaidyanathan,

2014). Furthermore, EMG signals have been shown to increase in fatigued muscles, which changes the correlation between the EMG signal and force over time. In comparison, the correlation between MMG signals and force remains constant (Stokes & Dalton, 1991).

MMGs also have disadvantages: In addition to the contamination from limb movements (Posatskiy & Chau, 2012), the tissue layer through which the signal must pass may act as a low pass filter (Orizio, 1993).

To date, all approaches to EMG (and hence MMG) signal pattern recognition consist of four stages: data pre-processing, data windowing, feature extraction and classification (Scheme & Englehart, 2011).

Pre-processing of the data involves removing unwanted interference. In EMG signals, the most common sources are power line harmonics and motion artefacts due to sensor movement (Scheme & Englehart, 2011). MMG signals also suffer from motion artefacts due to sensor movement, but, more significantly, from noise due to physical contact being transmitted through the body.

Real-time constraints limit data segmentation to 300 ms or less; with bias and variance increasing as segment length decreases (Asghari Oskoei & Hu, 2007). Englehart and Hudgins showed that, by using continuous segmentation of a steady state signal, segment length can be reduced to 32 ms without a considerable decrease in accuracy (Englehart & Hudgins, 2003). Conversely, Farina and Merletti showed that a segment length less than 125 ms, has high variance and bias in the frequency domain (Farina & Merletti, 2000). While Hudgins, Parker and Scott showed that classification accuracy was highest when extracting features from a 200 ms segment divided into four to five sub-segments of 40 to 50 ms (Hudgins, Parker & Scott, 1993); Smith et al. identified a window size of 150 to 250 ms as the optimal balance between accuracy and delay (Smith *et al.*, 2011).

Using overlapped data windowing has two advantages: the processor is not idle during the time between segments; although clearly it must finish the classification before the next segment arrives (Asghari Oskoei & Hu, 2007). However, Farina and Merletti have shown that overlapped segments do not provide a significant improvement in the accuracy of spectral features (Farina & Merletti, 2000). To make smooth reliable decisions in continuous segmentation, the mode of a certain number of the previous and next classifications, dependent on the acceptable delay and segment length, can be used (Englehart & Hudgins, 2003).

The optimal selection of features is important; especially in noisy environments (Mamun *et al.*, 2012). Features can be categorised as time, frequency or time-scale domain features (Zecca *et al.*, 2002). The two main signal characteristics used for extracting time domain features are the amplitude, which represents the signal energy, activation level, duration and force, and the power spectrum. Time domain features include the mean absolute value, mean absolute value slope, zero crossing, slope sign changes, and waveform length (Asghari Oskoei & Hu, 2007; Gopura *et al.*, 2013). Using the mean absolute value of a signal works better at high levels of contraction, while the root mean square of the signal works better at low levels of contraction. However, Clancy, Morin and Merletti showed that there is very little difference between the two (Clancy, Morin & Merletti, 2002), and Chan *et al.* showed that slope sign changes do not improve classification performance and can cause it to deteriorate (Chan *et al.*, 2000).

Frequency domain analysis is mostly used to study muscle fatigue, because it is time variant and depends on the contraction force and hence muscle fatigue. Frequency domain analysis can also be used to measure firing rate using frequencies between 20 and 40 Hz, and the signal structure using frequencies between 40 Hz and 500 Hz (Novak & Riener, 2015). Hudgins, Parker and Scott showed that signal structure can be used to identify different contraction types (Hudgins, Parker & Scott, 1993). Additionally, the power spectral density, defined as the Fourier transform of the autocorrelation function of a signal, has two characteristic variables: the mean and median

frequency. However, it is affected by windowing and short-time segments (Asghari Oskoei & Hu, 2007).

Time-scale analysis is used for signal de-noising, identifying fatigue in long-term activity and isolating coordinated muscle activities. To obtain time-scale domain features, the signal is transformed into a two-dimensional function of time and frequency. This can be done with a short-time Fourier transform, a wavelet transform or a wavelet packet transform. The primary difference is in the way the time and frequency domains are partitioned. The short-time Fourier transform has fixed partitioning in both the time and frequency domains, the wavelet transform has fixed time domain and variable frequency domain partitioning and the wavelet packet transform has variable time and frequency domain partitioning. (Asghari Oskoei & Hu, 2007).

Feature evaluation can be approached either: structurally, using physical or physiological models; or phenomenologically, using mathematical models (Asghari Oskoei & Hu, 2007). Classification can be done using neural networks, Bayesian classifiers, fuzzy logic, linear discriminant analysis, support vector machines, hidden Markov models, neuro-fuzzy logic (Gopura *et al.*, 2013), Euclidean distance, matched filters, decision fusion, auto-regression, discrete Fourier transforms, discrete Karhunen–Loeve transforms, or discrete cosine transforms (Mace *et al.*, 2013).

If the resultant set of features is large, principal component analysis (PCA), independent components analysis (ICA) or linear projection can be used to reduce the number of feature dimensions (Scheme & Englehart, 2011). However, not only does this kind of statistical analysis add an additional hyper-parameter³ (Bengio, 2012b), multi-channel amplitude coded systems and multi-function control systems, that use statistical approaches to merge multiple channels, have been shown to have many deficiencies (Hudgins, Parker & Scott, 1993).

3 Either the number of principal components used, or the proportion of variance to be explained.

2.5 Recent Successes and Ongoing Challenges with Using Machine Learning and Artificial Neural Networks

An overview of machine learning and artificial neural networks is provided in Appendix B. This section reviews the recent successes and ongoing challenges faced in the use of machine learning and artificial neural networks.

Machine learning has received significant attention recently; especially deep neural networks, convolutional neural networks (CNN) and long short-term memory (LSTM). This is largely due to the recent publicity following successes in computer vision (He *et al.*, 2015), speech recognition (Seide, Li & Yu, 2011) and games (Silver *et al.*, 2016). Using artificial neural networks may appear to be an obvious approach now, but this was not the case at the start of this research. As late as 2007, there was little agreement on the best machine learning algorithm to use for classification (Kotsiantis, Zaharakis & Pintelas, 2007). Much of the recent success of using deep neural networks in machine learning is driven by Hinton *et al.*'s initial success in training deep neural networks (Hinton, Osindero & Teh, 2006) and his use of unsupervised learning to pre-train networks (Hinton & Salakhutdinov, 2006) and the subsequent improvements made to the approach (Ranzato *et al.*, 2006; Bengio *et al.*, 2007; Vincent *et al.*, 2010; Krizhevsky & Hinton, 2011); even if their approach to training deep neural networks is no longer the approach used (Silver *et al.*, 2016). Training deep neural networks is dependent on large datasets, which have only become available more recently (Krizhevsky, Sutskever & Hinton, 2012), and require significant processing power (Chilimbi *et al.*, 2014). Recent successes have also used CNNs to assist artificial neural network based natural language processors and image classifiers extract relevant information from data values located close to each other (Collobert *et al.*, 2011; Cireşan, Meier & Schmidhuber, 2012). LSTM modules allow historic data to be maintained while it is relevant (Hochreiter & Schmidhuber, 1997); in an attempt to overcome problems with other recurrent neural network (RNN) designs (Lipton,

Berkowitz & Elkan, 2015). Reinforcement learning has also recently been applied to deep neural networks (Lillicrap *et al.*, 2015). Finally, researchers are looking at new policies such as Soft Q Learning to improve reinforcement learning approaches (Schulman, Abbeel & Chen, 2017).

One of the biggest challenges faced by using machine learning and artificial neural networks is the large number of hyper-parameters that affect the rate of learning (Bengio, 2012b). Bengio suggests the following values for the learning hyper-parameters: A default initial learning rate of 0.01; keep the learning rate constant; keep the mini-batch size between 1 and a few hundreds with 32 being a good default value; instead of setting the number of training iterations, keep track of the test error and stop when it increases; do not use gradient momentum. (Bengio, 2012b).

The choice of learning rate is crucial. If the learning rate is larger than twice the largest eigenvalue of the average loss Hessian matrix, the average cost will increase. The optimal learning rate is usually within a factor of two of the largest learning rate that does not cause the average cost to increase (Bengio, 2012b). Learning rates should be explored in the log-domain, and are typically below 1. To efficiently search for a good learning rate, Bengio uses the following strategy: Start with a large learning rate that causes the cost to increase and then reduce it by a factor 3 until the costs decrease. The largest learning rate that does not cause the costs to increase is usually a very good choice of learning rate (Bengio, 2012a).

Bengio has found that having more than the optimal number of hidden units does not generally reduce the network's learning results, only the learning computation requirements. Furthermore, using the same size for all layers generally works better or the same as decreasing or increasing the size in each layer. The first hidden layer should be bigger than the input layer, and the optimal size is an order of magnitude larger when using unsupervised pre-training. Unsupervised pre-training requires more hidden units to carry information that will be irrelevant to the actual, subsequent supervised learning (Bengio, 2012b).

Bias weights can be initialized to zero, but neuron weights need to be initialized carefully to break the symmetry between hidden units of the same layer. If hidden units in the same layer have the same input and output weights, they will compute the same output and receive the same gradient, and hence their updates will be identical too. The feature represented will simply be spread across multiple neurons (Bengio, 2012b). Output units receive different gradient signals. Therefore, their symmetry does not need to be broken and the weights to the final layer can also be set to zero. The general recommendation is to use initial weights selected from a uniform distribution that is proportional to the inverse square root of the size of the weights' input and output layers.⁴ This removes the need to fine-tune this hyper-parameter. The random seeds themselves only have a slight effect on the result, therefore they can be ignored (Bengio, 2012b).

When using TD learning, Tesauro suggests using a high prediction weighting (setting λ close to 1) initially, because it may help the network get reasonable initial values early, and decreasing the weighting when the predictions become more accurate (setting λ closer to 0) (Tesauro, 1992).

To ensure computing something interesting, overly linear units need to be avoided (Glorot & Bengio, 2010). Samples used for selecting hyper-parameter values should not be used to estimate generalisation performance (Bengio, 2012b). One approach to reducing the number of hyper-parameters is to use Gaussian processes, which explores large multi-dimensional state-spaces by identifying the locations where the greatest improvement can be made (Rasmussen, 2006a).

Another key challenge faced when using artificial neural networks is that they are not guaranteed to converge and they frequently diverge, even when trying to approximate simple linear functions (Boyan & Moore, 1995). However, non-local changes provide the generalisation that is the main benefit of using artificial neural networks (Riedmiller, 2005). Combining TD learning with artificial

⁴ $\sqrt{\frac{6}{n^l + n^{l+1}}}$ for hyperbolic tangent units.
 $4\sqrt{\frac{6}{n^l + n^{l+1}}}$ for sigmoid units.

neural networks further increases the problems. Although using TD learning works amazingly well, there remain a number of serious theoretical and practical problems (Tesauro, 1992).

In linear functions, instabilities are overcome by using on-line sampling to ensure that the samples consist of an actual sequence of visited states obtained (Tsitsiklis & Van Roy, 1997). In non-linear value functions such as artificial neural networks, instabilities are caused by correlations in the input or target values, or small changes in the input causing large changes in the value function (Mnih *et al.*, 2015). While convergence of the TD algorithm has only been proved for linear networks and linearly independent sets of input patterns (Tesauro, 1992), Riedmiller showed that an artificial neural network value function can be stably updated by using an off-line batch learning approach (Riedmiller, 2005). Mnih *et al.* showed that a deep artificial neural network value function can be stabilised by randomising the input data to remove input correlations, and use iterative, periodic updates to remove target correlations (Mnih *et al.*, 2015). In addition, the sigmoid non-linearity has been shown to slow down learning because of its non-zero mean that induces important singular values in the Hessian (Glorot & Bengio, 2010).

The way in which the input and output data are represented has been found to be one of the most important factors in successful practical learning algorithms (Tesauro, 1992). Furthermore, Markovian states depend only on the current state. Therefore, they need to include all relevant information from previous states (Bremaud, 1999).

The main criticisms of neural networks are that they are difficult to interpret, they are task specific, and they require large amounts of training data (Novak & Riener, 2015). Conversely, their main advantage is that they can be used to create a generic controller. However, the best example found for using a neural network for creating a generic controller is by Mnih *et al.*, who created a generic controller for learning to play multiple computer games (Mnih *et al.*, 2013) (Mnih *et al.*, 2015).

2.6 Using Machine Learning to Play Games

Game theory's min-max solution will fail to find a solution to Tic-Tac-Toe, because it assumes an opponent will play in a particular way. A min-max player would never reach a game state from which it could lose, even if it always won from that position due to incorrect play by the opponent. Dynamic programming can compute an optimal solution for any opponent, but it requires a complete specification of that opponent, including the probabilities with which the opponent makes each move in each board state. On the other hand, a reinforcement learner can achieve the effects of planning and lookahead without using a model of the opponent and without carrying out an explicit search over possible sequences of future states and actions (Sutton & Barto, 1998).

In 1995, Tesauro used a TD learning network to successfully train a computer to play Backgammon. Tesauro was the first to demonstrate the ability for a single hidden layer with 80 neurons using TD learning to play Backgammon at World Cup Championship level with only 1,500,000 iterations. Tesauro's solution was successfully used to achieve near-parity against, and even to introduce new techniques now currently used by, world-class Backgammon players (Tesauro, 1995).

However, to be successful Tesauro used a number of hand-crafted data features. Instead of simply using a raw board description, Tesauro used features known to be relevant for good play. This was limited to having separate values to identify the number of pieces at each location. He did not use any additional pre-computed features, or specify any intermediate goals (Pollack & Blair, 1997).

The game of Backgammon has a possibility of winning a gammon – if the opponent has not removed any piece by the time the winner has removed all fifteen pieces – which counts for double a normal win. Therefore, a game can result in one of four possible outcomes; instead of just the normal two. Tesauro used a four-component reward signal, with each component corresponding to the probability of each of the four outcomes. To calculate the value of a game state, Tesauro used a

combination of the predicted probabilities: If p_1 , p_2 , p_3 , and p_4 are the network's estimates for player 1 winning, player 1 winning a gammon, player 2 winning, player 2 winning a gammon respectively, then the value of the game is calculated as $p_1 + 2 \cdot p_2 - p_3 - 2 \cdot p_4$ (Tesauro, 1992).

During a full game, the average sequence length from start to finish is around 50-60 time turns. However, if moves are selected at random, games can last several hundred or even thousands of turns. Critically, the optimal strategy used by a player depends on the opponent's strategy. This means that a network adapting to its own play has a more serious possibility of getting stuck in self-consistent but non-optimal strategies. To simplify the initial training, Tesauro first trained the network on end games. Not only are the number of turns reduced, they are also limited – there is no way for a player to go backwards – and the best move is independent of the opponent's strategy.

For each of the approximately 54,000 bear-off configurations the exact probability of removing all the pieces in a specific number of dice rolls can be computed and used to compute the exact probability that a player will win. For a randomly generated board state created by distributing all of each player's 15 pieces with uniform probability in the six home board locations, the average sequence length, for good play, is only about 14 turns. Tesauro used a state vector with 16 values: 12 values for the number of each player's piece in the final six locations, and 2 values to indicate the player to play next. He tried two different networks: one with no hidden layers, and one with a single hidden layer with varying numbers of hidden units.

Tesauro found that a learning rate of $\alpha = 0.1$ gave good results. Lower learning rates reduced the fluctuations in learning performance, but they did not improve the overall performance. While higher learning rates significantly reduced learning performance. He found that the value weight preference parameter λ had almost no effect on the overall performance, but he did find that learning was quicker with larger values of λ . He also found that the initial random weight range was not important.

Tesauro found that the artificial neural networks were only able to estimate the probability of winning to within 10% accuracy. While good play requires an accuracy of 1% for accurate move selection. He found that, although training reduced the prediction error, the move-selection performance usually reached a maximum after tens of thousands of games, and then decreased thereafter. He attributed this to over-training that could either be due to the difference between the training and test positions, or be due to the difference between the function minimized by the learning algorithm (prediction error) and the function that the user cares about (move-selection ability) (Tesauro, 1992). Over-fitting can be overcome by using a regularisation factor (Bishop, 2006).

The networks with the hidden layer did better. This demonstrated that the hidden layer was identifying the non-linear aspects of the problem, and are not just simply maximising the number of pieces removed. Learning also improved with the number of hidden neurons in the hidden layer, which were increased from 0 to 80. However, he did note that prediction error got worse nearer the end of the game, because states far from the end can be accurately approximated by counting the total dice count required, but not when there are only a few pieces left (Tesauro, 1992).

Both over-training and over-fitting may occur, however, if the error function minimized during training does not correspond to the performance function that the user cares about the network could produce fairly accurate predictions but not select very good moves; especially when the best move is only slightly better than other alternatives. Bad predictions might also occur if the distribution of input training and testing patterns are different. For example, the game-playing network might be trained on its own play, but have to perform against other players. The differing styles of play would lead to different distributions of input patterns (Tesauro, 1992).

Tesauro trained the network on race situations – when the players' pieces have all passed each other. He used a state vector with 52 values: 24 values for the number of each player's piece in the final

twelve locations, 2 values to indicate the number of pieces off the board, and 2 values to indicate the player to play next. The initial board configurations were generated by randomly picking a location on the board that splits the two players' pieces, and then, from a uniform distribution, randomly distributing each player's pieces on their side of the divider. The average number of time steps from start to finish is now approximately 20 (compared with 14 in the bear-off learning). The learning parameters were kept at $\alpha = 0.1$ and $\lambda = 0.7$; while the number of hidden neurons were varied between 0 and 40. Using a test set of 248 starting positions, he achieved a performance of 65% with 40 hidden neurons. Compared with only 5% without learning (Tesauro, 1992).

Tesauro's TD learning networks reached peak performance after only tens of thousands of games, and there was no clear evidence of over-training, which suggested that further training may result in even better performance. He identified that the network continued to waste opportunities when there was both a small chance of winning and a small chance of losing a gammon. A good player would attempt to avoid losing the gammon, whereas the network aimed to win (Tesauro, 1992).

For the full game, Tesauro used an expanded vector to describe the state of the game to 198. Instead of using a single value to represent the number of pieces a player has at each of the 24 locations, he used four values: Three values are binary values indicating whether there is a single, two or three pieces at the location, and the fourth to indicate the number of pieces greater than 3. He found that increasing the number of binary representations up to 5 or 6 pieces improved performance, but increased learning times. The final six values included values for indicating the number of pieces on the bar, off the board, and the player to move. Once again he kept the same learning parameters: $\alpha = 0.1$ and $\lambda = 0.7$. Although he did experiment with different weight preference (λ) values, larger values tended to decrease performance, and smaller values had no significant impact.

At the time, computer game and machine learning research believed that significant levels of performance in game-playing could only be achieved through the use of hand-crafted features in the

evaluation function. Therefore, Tesauro did not expect a single network to be able to learn anything useful. However, significant learning did take place. He stated that the network weights learned showed a lot of spatially organised structure, which could be interpreted by an experienced backgammon player as useful features, and developed a playing style that emphasised running and tactical play. Tesauro tested the network against the Sun Microsystems' "Gammontool" program, against which random play would result in winning only 1% of the time, beginner players can expect to win about 40% of the time, while intermediate-level players would win about 60% of the time, and expert players about 75% of the time. The testing procedure involved playing the game using the network until it becomes a race, and then use the Gammontool algorithm to move both sides until the end. Tesauro's network, learning to play by playing itself, was able to learn to beat the Gammontool up to 66.2% of the time (Tesauro, 1992).

The number of training games required to reach the performance levels reported were ranged from 50,000 games, for the networks with 0 to 10 hidden neurons, 100,000 games for networks with 20 hidden neurons, and 200,000 games for the network with 40 hidden neurons. The number of training games required was linear in the number of weights in the network. Since, the simulation time per game on a serial computer also scales linearly with the number of weights, the total computer time scaled quadratically with the number of weights. Tesauro used an IBM RS/6000 workstation, which required several hours to train the smallest network and the largest network required two weeks (Tesauro, 1992).

Tesauro's derived weights are considered intellectual property, and the success of the final version also incorporates a number of hand-crafted expert-knowledge features, which are not highlighted in the original paper (Pollack & Blair, 1997). However, others have reported that they have replicated the results (Boyan, 1992).

Until recently, the success of Tesauro's TD learning network had not led to similar impressive breakthroughs using a TD learning network in other applications or even other games. Pollack and Blair suggested that the success had more to do with the structure of the learning task and the dynamics of the backgammon game itself. They highlighted that self-training systems, with weak or non-existent internal representations, were abandoned by the field of AI, because, aside from problems of scale, self-playing learners are prone to develop strategies that allow them to draw each other, but perform poorly against humans and other programs (Pollack & Blair, 1997).

However, recent research has shown that this is a valid approach. Various researchers have used deep neural networks and significantly more processing power to achieve success in an ever increasing number of areas (Mnih *et al.*, 2013, 2015; Silver *et al.*, 2016; Vinyals *et al.*, 2017).

2.7 Using Machine Learning in Device Control

Real-world environments provide a number of new challenges: The environment can never be completely known – epistemic uncertainty; there will be sensor measurement errors – aleatoric uncertainty (Kendall & Gal, 2017); the environment is continuous in both the state and time domains; and it's difficult to provide accurate and precise feedback and even more difficult to provide automatic feedback (Peerdeman *et al.*, 2011).

To control the prosthetic state, researchers have used various approaches, such as machine learning, decoders, pattern recognition, and proportional control (Alcaide-Aguirre, Morgenroth & Ferris, 2013). However, there is a difference between learning to predict and learning to control. Machine learning does not address the issue of simultaneously learning both (Tesauro, 1992). To minimise the initial bias requires an a priori model. To eliminate the ongoing variance requires the controller to become more versatile as more data becomes available (Geman, Bienenstock & Doursat, 1992).

Williams confirms that pattern recognition still looks like the most promising way to extract intuitive control out of EMG signals, but he acknowledges that applying pattern recognition to quickly accomplish simple tasks is still difficult. Although pattern recognition seems to be approaching intuitive control of the wrist and a single grip, modern commercial prosthetic hands, that offer a number of grip positions, still require combinations of contractions, software switches and/or hardware switches to switch between those grip positions. This is a long way from intuitive control of individual fingers and thumbs. Therefore, there is a need for hands that require more control information (Williams, 2011).

The most popular approach is to use a single time-invariant classifier based on linear discriminant analysis, support vector machines, nearest neighbour classifiers, neuro-fuzzy classifiers and artificial neural networks. However, the choice of classifier has a limited impact on accuracy. The most important disadvantages with time-invariant classifiers are: they cannot improve; they result in reduced accuracy over time due to changes in the sensors' signals over time; and they have limited transferability between users (Novak & Riener, 2015). State-classifying control methods have two additional drawbacks: They only allow the subject to perform predetermined movements, and they limit the user's ability to control the magnitude of torque production (Alcaide-Aguirre, Morgenroth & Ferris, 2013).

Classifiers that are used to control prostheses must adapt with the user in order to be clinically viable over time. To overcome the disadvantages with time-invariant controllers, adaptive control is required. Adaptation can be supervised or unsupervised. Supervised adaptation involves either retraining or incremental corrections, but both involve cumbersome user input. Unsupervised adaptation does not require user input, but is less accurate. Sensinger, Lock and Kuiken compared various adaptive pattern recognition approaches. They found that unsupervised adaptation approaches – using confidence levels to decide whether or not to add data points to the control set and what the correct mapping was – did not reduce the error as much as supervised adaptation

approaches – not using confidence levels to decide whether or not to add data points to the control set, but manually selecting the data points with the correct mapping (Sensinger, Lock & Kuiken, 2009). In other words, the likelihood of including an incorrectly classified data point reduces the effectiveness of unsupervised adaption approaches, and manually selecting high confidence data increases the effectiveness of supervised adaption approaches. Fukuda et al. have developed an alternative unsupervised adaption approach. Instead of using confidence levels, they proposed using low entropy classifications to identify when to recalculate the classifier's boundaries (Fukuda *et al.*, 2003). Pilarski et al. have used a continuous actor-critic reinforcement learning (ACRL) system to detect and adapt to the slight mechanical and biological changes that occur during day to day use (Pilarski *et al.*, 2011). They subsequently used ACRL for real-time prediction and control (Pilarski *et al.*, 2013).

In 1993, Hudgins, Parker and Scott used an artificial neural network with a single hidden layer with four to twelve neurons to classify five states from features extracted from a single channel of EMG data in the initial phase of a muscle contraction with an error rate of around 10% (Hudgins, Parker & Scott, 1993). Thirteen years later, in 2006, Zhao et al. used an artificial neural network with a single hidden layer with up to 35 neurons. They used 15 wavelet and sample entropy parameters extracted from three EMGs to classify six states: flexion and extension of the thumb, the index finger and the middle finger. The result was an error rate of around 5% (Zhao *et al.*, 2006).

Castellini and van der Smagt have combined machine learning and force measurement to control a robotic hand. Machine learning with a down-sampling algorithm are used to classify finger position, which is used to select the grip type. Separately, the force measured on the finger is used to trigger opening or closing of the hand (Castellini & van der Smagt, 2008). Wojtczak et al. have used neural networks to control multi-function prostheses (Wojtczak *et al.*, 2009).

Feed-forward neural networks were believed to be largely inadequate for high dimension inputs, because they result in extrapolating rather than interpolating the intentions. Extrapolation results in

unpredictable results (Geman, Bienenstock & Doursat, 1992). Furthermore, the neural network learning process is considered too slow for large networks (Abe & Lan, 1995). However, since 2006, deep networks have been used in classification, regression, dimensionality reduction, modelling textures, modelling motion, object segmentation, information retrieval, robotics, natural language processing, and collaborative filtering (Bengio, 2009). Deep learning is still mostly used for various recognition or classification tasks, and, more recently, in the area of detection, such as face and pedestrian detection. In 2015, deep learning was used for the first time to detect robotic grasps (Lenz, Lee & Saxena, 2015).

2.8 *Sensor Data Fusion*

Mitchell defines the aim of sensor fusion as being: to improve the precision, accuracy and completeness of the information obtained; and warns that sensor fusion does not always achieve this aim (Mitchell, 2007). Hall and McMullen state that the aim of sensor fusion is to achieve inferences not possible from a single sensor, and involves using techniques from signal processing, statistical estimation, pattern recognition, artificial intelligence, cognitive psychology and information theory. Data fusion design should be focused on the output requirements, rather than efficiently using all the sensor input (Hall & McMullen, 2004). Liggins, Hall and Llinas recently refined this aim to achieving more specific inferences, and adding that new sensors, processing techniques, hardware and increased communication bandwidth have made real-time data fusion increasingly viable (Liggins, Hall & Llinas, 2017). Dumas, Lalanne and Oviatt state that multi-modal systems have been shown to only increase efficiency by 10%. Therefore, the objective of sensor fusion is to offer more ways to interact with a system, and to make the interaction more natural and transparent (Dumas, Lalanne & Oviatt, 2009).

Fusing data from multiple sources provides significant advantages over a single data source. In addition to the statistical advantage gained by additional data, the use of multiple types of sensor

may increase the accuracy (Hall & Llinas, 1997), reduce uncertainty (Oviatt, 2003), increase flexibility, improve error handling (Dumas, Lalanne & Oviatt, 2009), improve robustness and system reliability, extend spatial and temporal coverage, improve spatial and temporal resolution, increase confidence, reduce ambiguity (Hall & McMullen, 2004), improve estimates, and increase dimensionality, for example through triangulation (Liggins, Hall & Llinas, 2017).

Hall and Steinberg identified seven limitations to data fusion:

1. Data fusion cannot substitute for a good sensor measuring the required phenomenon.
2. Data fusion cannot correct for errors in individual sensors or errors in processing sensor data.
3. An incorrect sensor performance model will corrupt the fused data.
4. No sensor fusion algorithm is optimal under all conditions
5. Insufficient training data makes pattern recognition ineffective.
6. It is difficult to quantify the value of a data fusion system.
7. Data fusion is not static, it is an iterative dynamic process.

(Hall & Steinberg, 2001)

The easiest fusion technique is to simply concatenate the inputs and use dimensionality reduction such as Principal Component Analysis (PCA) to reduce the input length (Jaimes & Sebe, 2007). Performance further improves by incorporating weights in the fusion rules (Gupta *et al.*, 2005). Dumas, Lalanne and Oviatt identified three levels of fusion: data, feature and decision levels. Decision level fusion is the most common type of fusion, because data fusion requires data from similar sensors, and feature fusion requires temporal synchronisation (Dumas, Lalanne & Oviatt,

2009). Data fusion is typically done using Kalman filtering, while feature fusion is typically done using a concatenated vector and a pattern recognition system such as neural networks, statistical classifiers and vector machines. Decision fusion methods include weighted outputs, classical inference, Bayesian inference and Dempster-Shafer's method (Liggins, Hall & Llinas, 2017).

Novak and Riener identified four ways in which two types of sensor are used simultaneously. First, the features from each sensor type are input into a single controller. Second, one sensor type is used to determine the controller to use and the second sensor type is used as input to that controller. Third, both sensor types are used as inputs into multiple controllers and one sensor type is used to determine which controller to use. Finally, both sensor types are used as inputs into multiple controllers and one sensor type is used to determine how to merge the outputs of the controllers (Novak & Riener, 2015).

There are a number of examples where previous researchers have combined similar sensors to those used in this research: Huang et al. combined EMG with a 6 degree of freedom (DOF) load cell on a passive hydraulic knee prosthetic to predict locomotion modes and transitions from multiple gait phase dependent classifiers (Huang *et al.*, 2011).

Yin, Fan and Xu have created a bi-directional HMI to control a lower limb exoskeleton. They use EMG sensor data from a healthy leg to control an exoskeleton driving the paralysed leg. They provide a form of proprioception by sending exoskeleton joint angle and torque feedback through haptic stimulation. The haptic stimulation is in the form of air pressure applied to two gas bags contained in the exoskeleton (Yin, Fan & Xu, 2012). In this example, data fusion is done by the user's brain instead of the interface.

Frisoli et al. have combined an eye tracker, a 3D scene camera and an electroencephalography (EEG) brain computer interface (BCI). The eye tracker and the 3D scene camera are used to identify the location of the object the user is looking at. The EEG signal is used to identify the user's

intention to control the exoskeleton (Frisoli *et al.*, 2012). Aside from the data fusion being performed between the eye tracker and the scene camera, the data from the BCI is not being fused. Rather, two different classifiers are used to control two aspects of the exoskeleton: a classifier to determine the target location and a classifier to determine whether or not to move the exoskeleton to that location.

Assad *et al.* combined a large dense array of electromyographic sensors with an IMU (Assad *et al.*, 2013), but they don't use the IMU signal for anything.

Corbett, Kording and Perreault combined EMG signals with a gaze tracker to obtain a 2D target location to improve the control of a robot used to guide a user's hand to the target (Corbett, Kording & Perreault, 2013). The authors acknowledge that the gaze tracker provided near perfect target information, and the EMG data did not provide a statistically significant improvement.

Although using more than two types of sensor is rare (Novak & Riener, 2015), Gallego *et al.* have combined EEG, EMG and IMUs to drive a neuro-prosthesis that directly influences muscle activation to manage tremors. The EEG is used to monitor the central nervous system's intention to move and adjusts the threshold for the detection of a tremor, which determines whether or not management is required. Both the EMG and IMU signals are then used to detect and parametrise the tremors (Gallego *et al.*, 2012).

Hargrove *et al.* have demonstrated the effectiveness of fusing data by including thirteen mechanical sensors (a three-axis accelerometer, a three-axis gyroscope, and sensors for vertical load, knee and ankle position, torque, and velocity) in the mechanical design of a prosthesis. The design reduced control errors from 12.9% to 2.2%, and eliminated all critical errors – those deemed to possibly cause the subject to fall (Hargrove *et al.*, 2013).

Finally, in their research, Mace et al. point out that sensor data fusion and classifier ensembles still remain a relatively under-explored area (Mace *et al.*, 2013), and Liggins, Hall and Llinas confirm that data fusion algorithms still lack rigorous testing and evaluation (Liggins, Hall & Llinas, 2017).

2.9 Summary

The literature highlighted a number of things. First, learning human gestures is hard. Therefore, most researchers choose to identify the important gestures or grips, and then they try to capture them for use as control signals. However, advanced prosthetic users state the main problems with the control system are that it is not intuitive, it is not responsive, and it doesn't give the user feedback.

Second, trying to control multiple degrees of freedom with non-intuitive gestures places an excessive mental load on the user. Users want to select the actions and initiate the actions, but the actual execution should be automatic. Furthermore, classification uncertainty should only result in a delay in the control signal.

Third, despite the multitude of issues with EMGs, most researchers still use them to capture muscle activation signals. Although MMGs share many of the problems associated with EMGs, they are cheaper, reusable and easier to apply. Capturing muscle activation signals requires between 125 and 300 ms of data. A muscle activation signal can be captured using an amplitude trigger for a minimum duration based on firing frequencies of between 20 and 40 Hz.

Fourth, when training artificial neural networks, start with a default initial learning rate of 0.01 and explore learning rates in the log-domain. Use the largest learning rate that does not reduce the error being minimised. Using more than the optimal number of hidden units does not reduce the network's learning results, only the learning computation requirements. The first hidden layer should be bigger than the input layer, and the optimal size is an order of magnitude larger when

using unsupervised pre-training. Bias weights can be initialized to zero, but neuron weights need to be initialized carefully to break the symmetry between hidden units of the same layer.

Fifth, Tesauro's success with training an artificial neural network to play Backgammon was dependent on a number of hand-crafted expert-knowledge features and training focused on the requirements of Backgammon. Only recently have deep neural networks and significantly more processing power been able to achieve similar success in other areas. Training a deep neural networks requires large datasets and requires significant processing power.

Sixth, although numerous attempts have been made to use machine learning in device control, we are far from an intuitive control system that can control individual fingers and thumbs. There is a need for more control information. Controllers must adapt with the user in order to be clinically viable over time. Small artificial neural networks have been used to classify up to six pre-defined gestures.

Finally, sensor data fusion has many benefits including increasing accuracy, flexibility, reliability and error handling, while reducing uncertainty and ambiguity. However, the primary aim of sensor data fusion should be to achieve more specific inferences. Data fusion is typically done using Kalman filtering, while feature fusion is typically done using a concatenated vector and a pattern recognition system such as an artificial neural network. However, sensor data fusion remain a relatively explored area and algorithms still lack rigorous testing and evaluation.

This review further supports the need to create a Generic, Self-Improving Controller that can fuse multiple, different sensors' data and enable users to select the gestures they want to use to control a variety of devices.

Chapter 3

Source-Independent Hardware

Data Capture, Processing, Fusion and Storage

3.1 Introduction

A generic controller requires standardised input. At a software level, the simplest way to provide a standardised, multi-dimensional input is to use a vector of a specific length. In addition to a standardised input, the controller should not be dependent on the rate at which input is generated; or even whether or not input is being generated. To be able to use multiple, different, independent

hardware devices, each device needs to send data at the rate it is designed or configured for. In other words, the output not only needs to combine the data from multiple, different, independent sources into a single vector, it also needs to make the vector available when required not when it is generated.

Dynamic data stores information in the evolution of the data instead of the instantaneous values. To capture dynamic data, the standardised vector is extended to include multiple historic versions of the data. Once again, the historic copies need to be independent of the rate the data is generated by the multiple, different, independent sources. In other words, the output needs to include a specified number of copies of historical data stored at a specific period and made available when required independent of the rate at which the data is generated.

In addition, raw hardware data is noisy and needs to be filtered before it is added to the output vector or stored in the historical data. To provide digital instead of analogue data, the raw hardware data also needs to be normalised⁵ to represent the proportional value between its expected minimum and maximum⁶.

Since multiple types of IMU hardware were used in this research it was also necessary to create a standard interface for capturing the raw hardware data, before it was processed. This was required to ensure that the input is not only independent of the types of sensor used, but also independent of the type of IMU used. In other words, in addition to providing a standard, multi-dimensional input, a standard interface was required to capture the same type of hardware data. The standard hardware data captured from all the IMUs were accelerometer data, gyroscope data, magnetometer data, a quaternion representing the IMU's rotation from a reference orientation, and up to eight channels of analogue data.

⁵ Normalised data requires all data to be converted into a value between 0 and 1.

⁶ Values outside of the minimum and maximum values are clamped.

Section 3.2 details the design and construction of the IMU type independent data capture software interface. Section 3.3 explains the application of this interface to the two types of IMU hardware used. Section 3.4 describes the additional IMU specific software created to manage the configuration of the two types of IMU and the updates made to the firmware to improve their stability. Section 3.5 shows how the IMU type independent data capture software interface was applied to the controller of a Virtual Robot. Section 3.6 details the creation and testing of the real-time digital signal filters and triggers used to pre-process the raw hardware data before it is fused and stored. Finally, Section 3.7 describes how the data is fused, current and historical data is stored and updated, and how the input state vector containing all the data is made available when required.

3.2 IMU Type Independent Data Capture Software Interface

Multiple types of IMU were used in this research. All IMUs contain a Magnetic, Angular Rate and Gravity (MARG) sensor which includes a three axis accelerometer, a three axis gyroscope, and a three axis magnetometer. All the IMUs used in this research also contain eight auxiliary ports for connecting additional external analogue sensors. Furthermore, each IMU also includes a low powered embedded device that calculates a quaternion that represents the rotation from a reference orientation. This quaternion, called an orientation quaternion, is calculated from the MARG data using an AHRS orientation quaternion algorithm in real-time. The IMUs send the various sensor and quaternion data via an emulated serial port at regular intervals.

Since all the types of IMU used capture the same data, but present it in different ways, it was necessary to create a software interface to ensure the data captured was presented in a standardised form. Each type of IMU then simply needs to convert the data captured into this standardised form. This enables an application to use multiple different types of IMU simultaneously without needing to know the type of IMU.

Figure 3.1 provides an overview of the design of the IMU type independent data capture software interface that was developed. Multiple IMUs connect to the controller computer through a serial port. On the controller computer, the IMU Device Interface continually polls the serial port until data is received. When data is received, it passes the data to any IMU Listeners attached. Applications attach the IMU Listeners to IMU Device Interfaces connected to IMUs they're interested in. IMU Listeners are configured to only respond to data the application is interested in.

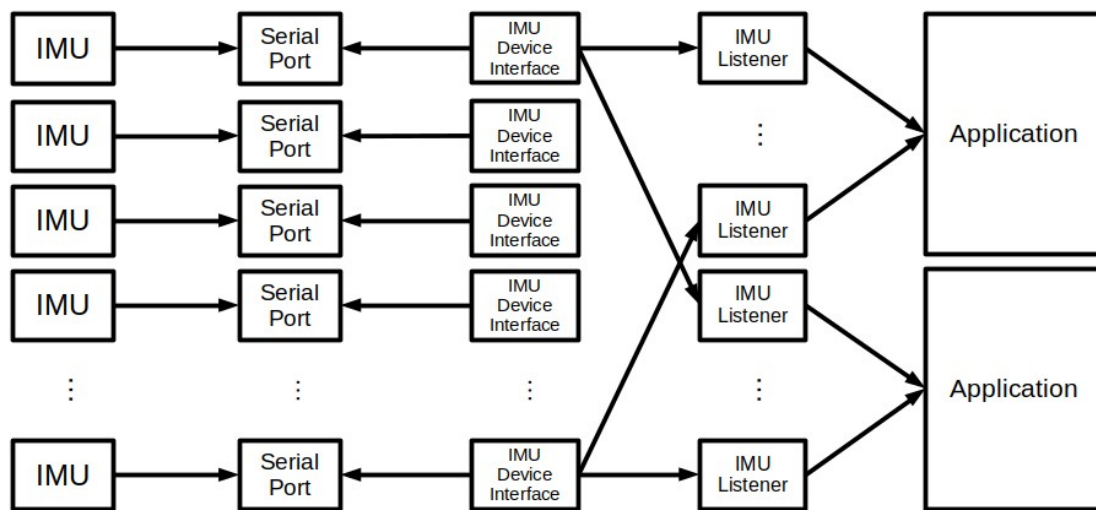


Figure 3.1: Data capture software interface overview for simultaneously capturing data from multiple, different IMUs and presenting only the required data to multiple applications in a standardised form.

This approach allows the IMU Device Interfaces to be independent of the application and the application to be independent of the type of IMU. Furthermore, multiple applications can be attached to the same IMUs simultaneously, and applications can choose which data to listen to without reconfiguring the IMU.

Each operating system uses its own approach for connecting to serial ports. Therefore, to create a cross-platform IMU type independent data capture interface, a flexible cross-platform serial port controller was also required. Each operating system provides ample, detailed documentation for

accessing the libraries used to access a computer's serial ports. Although there are open source code examples available for connecting to serial ports, they are generally very complex, because they try to incorporate all the, often unused, nuances, and are specific to a particular operating system. To the best of the author's knowledge, a simple, cross-platform C++ class for accessing serial ports did not exist. Therefore, a simple cross-platform C++ Serial class for reading and writing to a serial port was created.

To connect an instance of the Serial class to a required serial port, a set port method was created. The set port method takes the name of the serial port as an argument. The format of the name of the serial port depends on the operating system. Depending on the operating system used to compile the library, the method uses the operating system specific name format to call the correct operating system library. Then the correct operating system library connects the instance of the class to the required port. The method returns whether or not the system call to open the file associated with the named serial port was successful.

Similarly, to configure the serial port, a set parameters method was created. The set parameters method takes the baud rate in bits per second, the number of bits in a packet, the number of parity bits, the number of stop bits, and whether or not to block until bytes are available for reading as parameters. The defaults are 9600 bits per second, 8 bits per packet, no parity bits, one stop bit and not to block until bytes are available when reading. The method returns whether or not the system call to set the parameters was successful.

Once an instance of the Serial class has been connected to a serial port, data sent through the serial port by a device connected to the serial port can be read. To read data from the serial port, a read method, which takes a pointer to an array of bytes and its length as parameters, was created. The method calls the operating system specific function that copies the required number of bytes into the array. If the parameter to block until bytes are available for reading is set, a call to this method

will block until the required number of bytes is available. Since this will hang an application using this call, the default: not to block until bytes are available setting, is used, and the method only copies the bytes available into the array. To ensure the data in the array of bytes is not undefined, if the number of bytes read is less than the number of bytes required, the remaining bytes in the array are set to 0. The method returns the actual number of bytes read. An application calling this method must check this return value to ensure that the zero bytes are not used unintentionally.

Similarly, to send data to a device connected to the serial port connected to an instance of the Serial class, a write data method was created. The write data method also takes a pointer to an array of bytes and its length as parameters. The method calls the operating system specific function that writes the bytes in the array to the connected serial port. The method similarly returns the number of bytes actually written.

To manage the Serial class through a GUI, the Serial Panel shown in Figure 3.2 was created. Serial Panel is a wxWidgets panel. wxWidgets was selected for its cross-platform support. To connect the underlying Serial class to a serial port, all that is required is the name used by the operating system. The status light will reflect the connection status: blue indicates a new connection is in progress; red indicates the connection attempt was unsuccessful; and green indicates the connection attempt was successful. Multiple Serial Panels can be inserted into an application, which enables an application to easily connect to multiple devices via multiple serial ports.



Figure 3.2: Cross-platform Serial Panel used to manage serial ports

The generic IMU Device Interface class was designed so that it could be extended to accept data from any type of IMU. The IMUs connect to a computer via a serial port emulator using either Bluetooth or USB. The serial port emulator connects to a logical serial port on the computer. The IMU Device Interface uses the Serial class described above to connect to the serial port associated with the IMU.

When each type of IMU extends the generic IMU Device Interface, it is required to implement a two functions. First, it must implement a function that uses the underlying Serial class's functions to read bytes, and, when a packet is received, return it. Second, it must implement a function that converts a packet into an IMU Data item. Requiring each IMU type to separately implement the functions that return the next IMU Data item is necessary, because, each IMU type defines its own packet structure; including, but not limited to different headers, tails, compression, encodings, checksums. Furthermore, each IMU type also has packets that represent different data types.

The generic IMU Device Interface defines a minimum number of required IMU Data item types that each IMU type must be able to return. The IMU Data Types include: Message Data, Accelerometer Data, Gyroscope Data, Magnetometer Data, Quaternion Data and Analogue Data. Each type of IMU can also define additional data types.

Any number of applications can simultaneously use the data received by the IMU Device Interface through an IMU Listener. The base IMU Listener class receives an IMU Data item and calls the function associated with the IMU Data Type. Each application extends the base listener class. They override the IMU Data Type specific functions that define what happens when data that the application is interested in is received. IMU Listeners can be attached and removed from the IMU Device Interface online.

Each IMU Device Interface runs in its own thread. This thread calls the IMU type's implementation of the function that returns the next packet received from the IMU it is connected to. Each interface

continually calls this function until the next packet from the serial port that the IMU is connected to is received. The thread then calls the IMU type's implementation of the interface then converts the packet into the specified IMU Data item. Finally, the IMU Device Interface passes the IMU Data item to all the IMU Listeners currently attached. The IMU Listener calls the application specific function for that data type.

This design achieves three key things. First, since each IMU runs in its own thread, as long as there are sufficient threads, each IMU will not impact the performance of the other IMUs. Second, the application is unaware of the type of IMU used. Third, multiple applications can connect to the same IMU.

3.3 *IMU Device Interfaces*

Multiple different types of IMU were used during this research. Initially, IMUs from x-io Technologies, called x-IMUs, were used. These IMUs were chosen, because they had previously been successfully used by other members of the research group and the source code for interfacing with the x-IMU was available.

However, a number of limitations with x-IMU were identified. First, the auxiliary ports only have a single power port; so, without physically wiring all the external sensors together to enable them to use the single power port, only one external sensor could be attached. Second, although x-io technologies originally provided source code for interfacing with its IMU, they no longer make it available via their website. Third, later versions of the x-IMU firmware contained a bug that resulted in the auxiliary port data not being sent. The only way to overcome this issue was to use an older version of the firmware. Finally, to use the improved AHRS orientation quaternion algorithm described in Chapter 4, access to the firmware source code was required, but this was not available.

Therefore, the research group decided to develop its own IMU, the nuIMU, which became the second type of IMU used.

Both the x-IMU and the nuIMU have eight auxiliary ports for connecting additional external analogue sensors. As discussed in Chapter 2, muscle activation data provides a useful second, concurrent data signal for measuring body motion. MMGs were selected over EMGs, because they are significantly cheaper and easier to use, and this outweighed any disadvantages identified. To further minimise costs, the research group developed and now 3D-prints the MMGs shown in Figure 3.3. Although the IMUs were also combined with force sensors (Angeles *et al.*, 2016), the research group's MMGs were the primary external analogue sensor used.



Figure 3.3: MMG external sensor used to capture muscle activity.

The development of the IMU Device Interface implementations for each of the IMUs used is described below.

3.3.1 x-IMU

Figure 3.4 shows the first type of IMU used: the x-io technologies x-IMU.

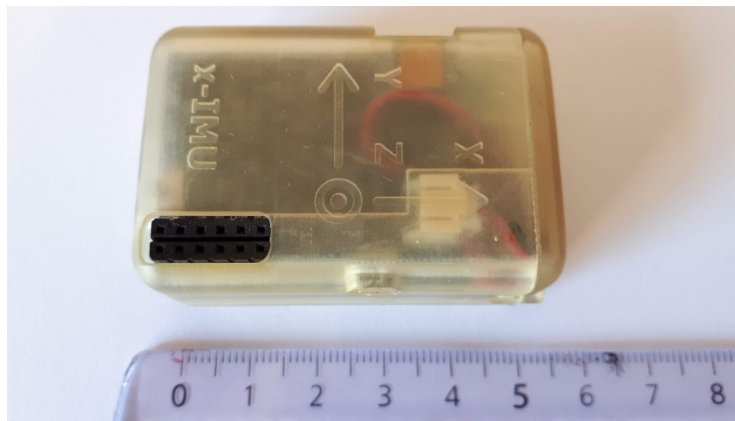


Figure 3.4: x-io Technologies x-IMU.

To apply the IMU Device Interface described above to the x-IMU, the source code for interfacing with the x-IMU was ported from C# to C++. This required understanding the encoding scheme used to send packets to and from the IMU.

Instead of using a documented header to indicate the start of a packet and define its length, the x-IMU uses its own encoding scheme. First, all the bits of the packet to be sent are concatenated and divided into groups of 7-bits. Second, each 7-bit group is stored in a byte with the most significant bit of all the bytes except for the last one being set to zero. The most significant bit of the last byte is set to one and is used to identify the last byte of the packet. This new, longer series of bytes is then sent using the default serial port transmission parameters: 8 bits, one stop bit and no parity bit.

This non-header based encoding scheme has the advantage that small packets of eight bytes or less only require a single extra byte⁷. However, to ensure all the packets sent are received, it does require the receiver to read a single byte at a time and cache all the bytes received until a byte with the most significant bit set is received. This cache is either of limited size or requires memory to be continuously allocated. The original C# code uses a variable length cache. Since there is a maximum packet size, a fixed cache size of 256 bytes was chosen. In the unlikely event that 256

⁷ Each byte uses one extra bit; so longer packets will require more bytes: one for each unencoded eight bytes.

bytes without the most significant bit set are received, the cache is discarded and an invalid packet signal is sent.

When a byte with the most significant bit set is received, it must be assumed that all the bytes since the previous byte with the most significant bit set form an encoded packet. The receiver must then decode this packet by dropping the most significant bit of each byte, concatenate all the bits, and then divide the bits into 8-bit bytes to create a decoded packet.

The final byte in the decoded packet is a checksum. However, again, instead of using a documented checksum, a simple modular sum type checksum is used: All the bytes, except the checksum are added, discarding any overflow. This is used to confirm that all the previous bytes do form a packet and that no single bit has been changed in transmission. Other checks achieved by well known, documented checksums are not detected. If the calculated checksum does not match the byte value, an invalid packet signal is sent.

Once a packet has been identified, decoded and confirmed using the checksum, the first byte of the packet is used to identify the data type. The data type defines the significance of the remaining bytes. The x-IMU defines data types for commands, errors, reading and writing register values, reading and writing the device's date and time, raw and calibrated battery and temperature data, raw and calibrated inertial and magnetic data, quaternions, digital input-output (IO) data, raw and calibrated analogue data, pulse width modulated output data, and raw and calibrated accelerometer⁸ bus data. These definitions are used to create the x-IMU specific IMU Data Types and functions for creating new IMU Data items.

The IMU Device Interface was extended to create the x-IMU Device Interface. The packet decoding and IMU Data item creation functions were used to implement the functions that returns the next IMU Data item required to by the IMU Device Interface. Since the x-IMU defines its own data

⁸ The x-IMU uses the Analog Devices' ADXL345 digital accelerometer and uses this product name in the data type.

types, the IMU Listener class was extended to support all the x-IMU Data Types. The x-IMU Listener calls functions associated with each x-IMU Data Type. In addition, when the x-IMU Data item contains data from multiple sensors, for example the inertial and magnetic data, that the IMU Listener defines separately, it extracts each sensor's data, creates multiple IMU Data items, and calls each of the functions associated with the IMU Data Types in the base IMU Listener. This enables both applications that want to work with all the x-IMU Data items to attach x-IMU Listeners to the IMU Device Interface and applications that want to work with any type of IMU to simultaneously attach IMU Listeners to the same IMU Device Interface.

3.3.2 nuIMU

Due to issues and limitations experienced with the x-IMUs, the research group chose to develop its own IMU: the nuIMU. Figure 3.5 shows the nuIMU developed by other members of the research group during the course of this research and was made available to this research about half-way through. The key advantages to the nuIMU include the lower cost, smaller size, access to the firmware source code, power port for each external sensor, and ability to send up to 1,000 packets per second.

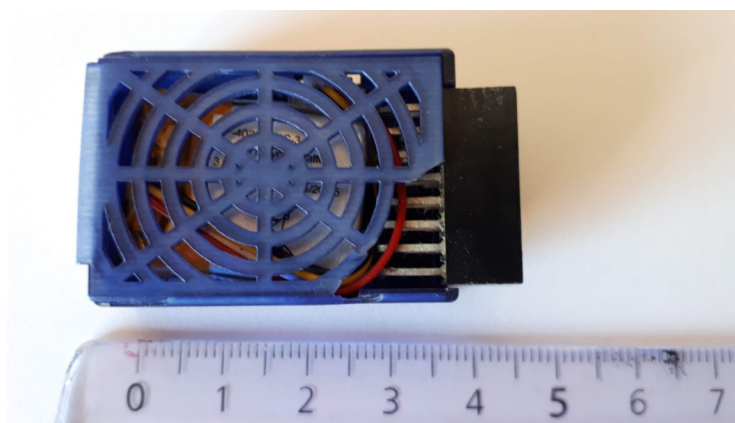


Figure 3.5: nuIMU

The nuIMUs use the LSM9DS0 MARG (accelerometer, gyroscope and magnetometer combined sensor). The accelerometer is able to measure linear accelerations up to 16 g at a rate of 1,600 Hz. The gyroscope is able to measure angular rotation rates up to 2000 degrees per second at a rate of 760 Hz. The magnetometer is able to measure magnetic fields up to 12 gauss at a rate of 100 Hz (ST, 2013).

The accelerometer on the LSM9DS0 was configured to measure linear accelerations between ± 2 g, which has a specified precision of ± 0.051 mg and a zero offset accuracy of ± 60 mg. The gyroscope was configured to measure angular rotations between ± 500 degrees per second, which has a specified precision of 17.5 millidegrees per second and a zero offset accuracy of ± 15 degrees per second. The magnetometer was configured to measure magnetic fields between ± 2 gauss, which has a specified precision of ± 0.08 mgauss (ST, 2013). The specified precisions are very close to the highest possible precision available when a 16 bit number is used to specify values in the required range:

Table 3.1: Comparison between 16 bit (2 byte) precision available and the specified precision of the LSM9DS0 accelerometer, gyroscope and magnetometer components.

Range	Precision Available	Precision Specified	Ratio
2 g	0.031 mg	0.061 mg	0.50
4 g	0.061 mg	0.122 mg	0.50
6 g	0.092 mg	0.183 mg	0.50
8 g	0.122 mg	0.244 mg	0.50
16 g	0.244 mg	0.732 mg	0.33
245 dps	3.74 mdps	8.75 mdps	0.43
500 dps	7.6 mdps	17.5 mdps	0.44
2000 dps	31 mdps	70 mdps	0.44
2 gauss	0.03 mgauss	0.08 mgauss	0.38
4 gauss	0.06 mgauss	0.16 mgauss	0.38
8 gauss	0.12 mgauss	0.31 mgauss	0.39
12 gauss	0.18 mgauss	0.48 mgauss	0.38

The nuIMUs use different packet structures and different data types than those used by the x-IMU. Therefore, it required its own extensions of the IMU Device Interface, IMU Listener and IMU Data Types.

Packets sent from the IMU have a three byte header: 0xDD, 0xAA, 0x55 followed by a byte that specifies the length of the remainder of the packet, which contains the data followed by another byte that also specifies the length. Packets sent to the IMU have a single header byte: 0x07 and a

single tail byte: 0xB. A key disadvantage to these packet structures is that there is no checksum. Therefore, any transmissions errors (that are not in the header, tail or length bytes) will not be detected. Furthermore, not only is a different packet structure used for sending and receiving data, the header doesn't include the packet length, because all the packets currently sent to the IMU have a fixed size: four, plus the header and tail byte.

Once a packet sent from the nuIMU has been identified, the first byte is used to identify the data type. The data type defines the significance of the remaining bytes. The nuIMU defines data types for message data, number data, final data, inertial, magnetometer and analogue data, quaternion data and register data. These definitions are used to create nuIMU specific IMU Data Types and functions for creating nuIMU Data items.

As with the x-IMU, the IMU Device Interface was extended to create the nuIMU Device Interface. The packet decoding and IMU Data item creation functions are used to implement the functions that returns the next IMU Data item. The IMU Listener class was also extended to create the nuIMU Listener. The nuIMU Listener, calls functions associated with each nuIMU Data Type. Data items that contain data from multiple sensors are split into multiple IMU Data items. Finally, the functions associated with standard IMU Data Types in the base IMU Listener are called.

3.3.3 nuIMU Recorded Data

Other research group members were conducting their own experiments using the nuIMUs. Their data was typically processed offline; so the live data was being recorded. This data was made available to this research too. To process this data as if it were real-time data, an IMU Device Interface was created for nuIMU recorded data.

The first step was to create a serial port emulator that uses data read from a file instead of received via Bluetooth or USB. The cross-platform Serial class described in Section 3.2 was extended to

create the cross-platform Serial File class. The underlying functions were then overridden. Instead of connecting to a serial port, the set port command opens the specified file. As there is no underlying serial port, the set parameters function does nothing. The read and write data functions simply read and write data to the file specified in the set port command. As the incoming byte data was recorded in comma separated files, the read command converted the comma separated byte data back into the original byte data format before copying the data into the byte buffer. The nuIMU Recorded Data Interface extends the nuIMU Device Interface, by using the Serial File class described above instead of the original Serial class.

Since it's not possible to query the original IMU to determine the configuration, the specified configuration is saved to a file too. This file is automatically updated when updated configuration data is "sent to the IMU" through this IMU Device Interface. This configuration data is important for two reasons. First, the values of the data received from the accelerometer, gyroscope and magnetometer are scaled depending on the IMU configuration. Second, and more importantly, the frequency at which data is sent cannot be determined by the frequency at which the data is received. The frequency at which the data is sent is an IMU configuration setting. Instead of sending data as it is received (or in this case read from the file) an artificial delay, equivalent to the period of the frequency setting, needs to be inserted between data packets to emulate data being sent in real-time.

Although all the data recorded was stored in comma separated files, the actual data provided in the files was inconsistent. In some cases, the raw byte data that was received from the IMU was recorded. In other cases, the processed, scaled data for the specific sensors of interest were recorded. Furthermore, in some cases the recorded data included additional information about each packet. When more than one IMU was used, the data included the number of the IMU that the packet was received from. Sometimes, a code for the particular gesture that was being recorded was usefully included. In other files a time span was included, which enabled verifying the specified frequency setting.

To facilitate the multiple data formats, two additional IMU Data Interfaces were created that extended the generic nuIMU Recorded Data Interface described above. One for files that recorded raw byte data. Another for files that recorded processed data. From these two nuIMU Recorded Interfaces, new nuIMU Recorded Data Interfaces can be created that only need to map the data values (columns) to the respective fields needed to create IMU Data packets to be sent to IMU Listeners.

For files that included a code for when the user was asked to perform the specified gesture, a new IMU Data packet was created and sent every time the code changed.

3.4 IMU Specific Software

The design of the IMU Type Independent Data Capture Software Interface allows applications to use data sent by any type of sensor and specifically any type of IMU. However, there is still a need for IMU type specific applications to manage the IMUs themselves. What follows is a description of the IMU type specific software created as well as fixes made to the nuIMU firmware to make it more reliable.

3.4.1 x-IMU GUI

The first application created was a GUI for configuring the x-IMU. The GUI created was a cross-platform port of the x-IMU configuration tool called the x-IMU GUI. This was required, because, despite the claim that the source-code was provided, the source code used both libraries for which the source code is not provided and libraries that are specific to a single operating system. These libraries needed to be reverse-engineered. The x-IMU GUI consists of five tabs: the serial port, registers, data and time, commands, and sensor data tabs.

The serial port tab, shown in Figure 3.6, uses the simple Serial Panel described in Section 3.2 and shown in Figure 3.2 to connect to the required x-IMU.

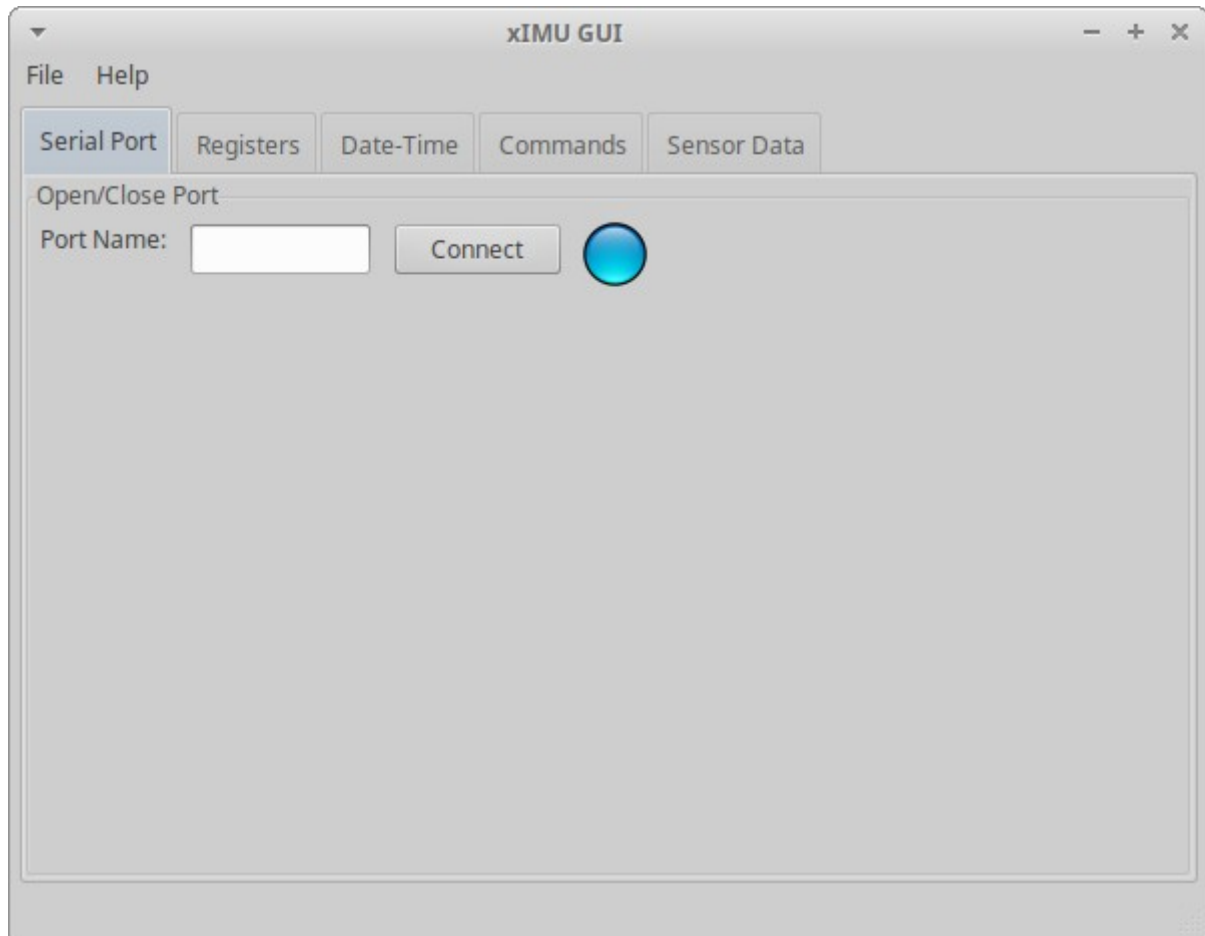


Figure 3.6: Cross-platform x-IMU GUI's serial port connection panel

The x-IMU is configured using 112 registers. The registers are displayed and modified using the tool's register tab shown in Figure 3.7. Each register is classified as either read-only or modifiable. If the register is modifiable, the specific values or range of values the register can take also need to be defined. The specific values a register can take depend on and are specified by the components used by the device.

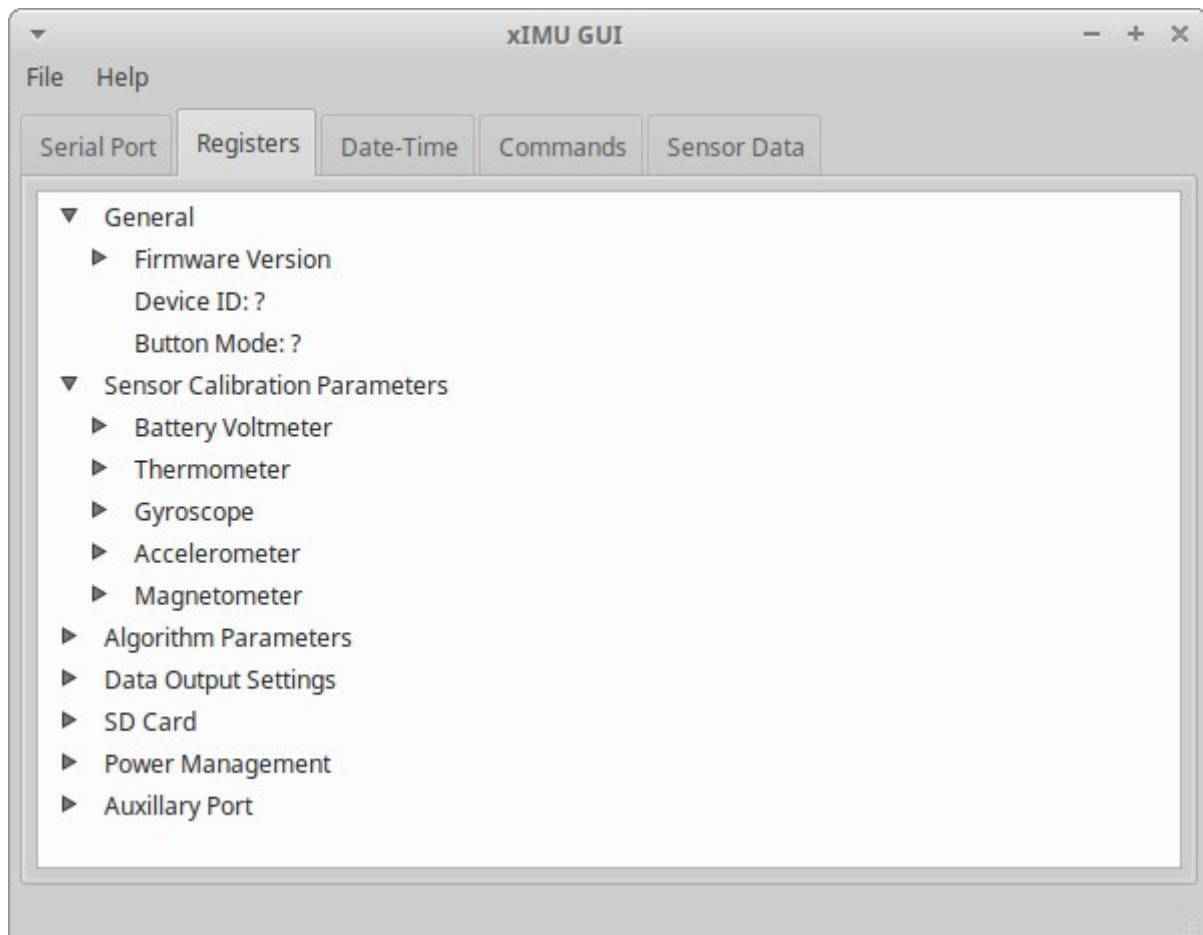


Figure 3.7: Cross-platform x-IMU GUI's register display and modification panel

A tree structure is used to group, display and, if allowed, change each of the registers. When a branch of the tree is expanded, the tool automatically sends read register data commands to the device for each of the registers that become displayed and populates the respective register values with the values returned. If a value is not returned, a question mark is displayed. To facilitate specifying the correct value, the interpretation of the value provided by the component's documentation is displayed. The current value of a specific register or all the values in an entire branch can be refreshed manually, by right-clicking on the register or branch and selecting refresh or refresh all respectively. If allowed, a register can be edited, by right-clicking on the specific register and selecting edit. The description associated with the allowed values are displayed in a drop down list. When the required option is selected and confirmed, the tool sends a write register

command to the device. To confirm that the register was set correctly, the new value will only be displayed if the device returns confirmation of the value that the register was set to.

The register tree has seven branches: general, sensor calibration parameters, algorithm parameters, data output settings, SD card, power management and, auxiliary port. The general branch provides the device's id and firmware version, and enables specifying the function of the button on the device. The button is not used in this research, therefore it was disabled.

The sensor calibration parameters branch's registers are used to convert raw sensor integer values to calibrated float values before being sent or recorded. Although these register values are modifiable, they are not changed here. Some values are specific to the actual sensors used and are defined by the manufacturer. Others are set automatically using sensor specific calibration commands.

The algorithm parameters branch contains the registers used to calculate the orientation quaternion. As described in Chapter 4, the algorithm that only uses the gyroscope and accelerometer does not provide a reliable orientation quaternion. Therefore, only the AHRS algorithm is used. The remaining registers are stored in three sub-branches: gains, quaternion tare and valid magnetic field magnitude. The gain sub-branch registers store the algorithm's feedback parameters. The quaternion tare sub-branch registers store the reference quaternion used by the algorithm and is set using the tare command. The valid magnetic field magnitude sub-branch registers store the minimum and maximum measured magnetic field values that will be considered valid. The stated purpose of these values is to avoid the effects of magnetic distortion or interference by not using the magnetometer if the measured value is too high or too low. However, this would result in the algorithm, at best, using the algorithm that only uses the gyroscope and accelerometer measurements, which, as already stated, is unreliable. Although not impossible, it's extremely unlikely that a net magnetic field less than the earth's fixed magnetic field would ever be present. Therefore, there is no reason to set a minimum valid magnetic field value. A strong, but fixed, magnetic field is exactly what is

required by the algorithm. Therefore, even if there is a strong additional magnetic field present, there is no reason to set a maximum valid magnetic field value.

The data output settings branch controls what data is sent by the device. It contains five registers: sensor data mode, date-time data rate, battery and thermometer data rate, inertial and magnetic data rate, and quaternion data rate. The sensor data mode register controls whether the device sends raw or calibrated data. To take advantage of the device's calibration tools and built-in correction algorithms, the calibrated data mode is used. The date-time data rate, and battery and thermometer data rate are set to disabled, because the date and time, and battery and thermometer data are not used. If only the quaternion data is used, the quaternion data rate register can be set to the maximum 512 Hz. However, experience has shown that the device can only send one data type at the maximum rate of 512 Hz. Attempting to send quaternion data and inertial and magnetic data at 512 Hz resulted in data packets being lost and unreliable data being received.

The SD card branch contains a single register that specifies the filename to be used by the device when recording data to the SD card. Setting this register creates a new file on the SD card on the device. Since changing this register has the additional side effect of overwriting the previous file, it would have been better to have two commands: one to create a new file, and one to define the filename. This change was not implemented, because storing data to the SD card is not used during this research.

The power management branch contains four registers: battery shutdown voltage, sleep timer, motion trigger wakeup, and Bluetooth power. The battery shutdown voltage is kept at the default value set by the manufacturer. The sleep timer is set to zero to ensure the device does not enter sleep mode when it is not moving but still being used. Although the device is never expected to enter sleep mode, the motion trigger wakeup is set to disabled to prevent the device from powering on

just because it is moved. Bluetooth power is set to enabled, because all data transfer in this research is done via Bluetooth.

The auxiliary port branch has two registers and four sub-branches. The first register sets the auxiliary port mode. The auxiliary port can be disabled or used for digital input and output, analogue input, Pulse Width Modulation (PWM) output, direct access to the accelerometer bus, as a Universal Asynchronous Receiver Transmitter (UART), or to receive an external sleep/wake signal. When an MMG was connected to the auxiliary port, the auxiliary port was mode was set to analogue input, otherwise the auxiliary port was set to disabled.

The other sub-branches and register of the auxiliary port branch are specific to the auxiliary port mode selected. The analogue input sub-branch is used to configure the auxiliary port when the analogue input mode is selected. It consists of four registers: data mode, data rate, sensitivity and bias. The data mode register is used to set whether raw or calibrated data is sent. If the calibrated data mode is set, the raw data is converted to calibrated data using the values in the sensitivity and bias registers. When both quaternion and analogue data are being used, both the quaternion data rate register described above, and the auxiliary port, analogue input, data rate are set to 256 Hz to ensure reliable data is received. The other auxiliary port modes were not used. Therefore, the other auxiliary port registers were not set.

The date and time of the device can be displayed and set in the cross-platform x-IMU configuration tool's date-time tab shown in Figure 3.8.

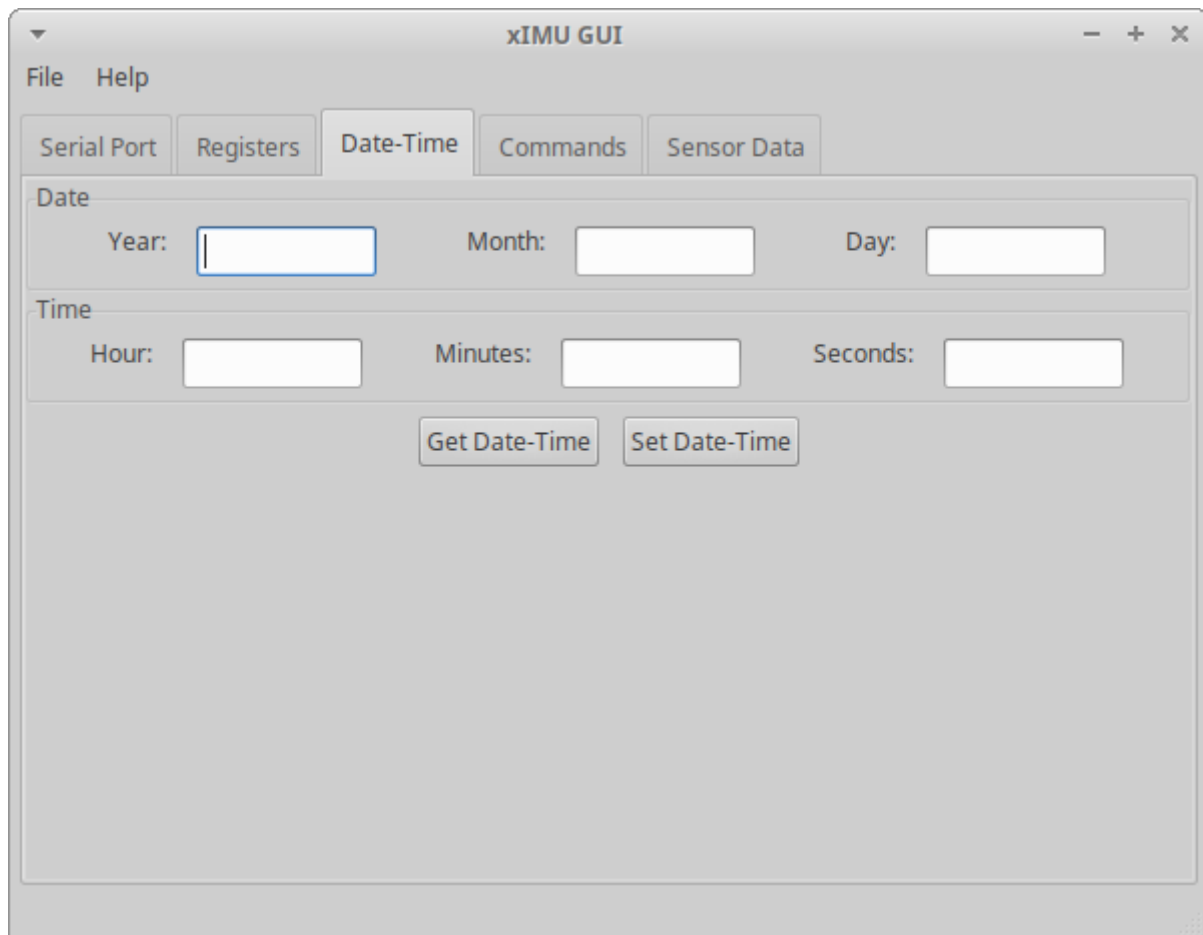


Figure 3.8: Cross-platform x-IMU GUI's date and time display and modification panel

In addition to being able to update the registers individually, the xIMU has a number of standard commands that can be used to control, configure and calibrate the device. The commands are grouped into five categories: general, gyroscope, accelerometer, magnetometer and algorithm. Figure 3.9 shows the cross-platform x-IMU configuration tool's command tab.

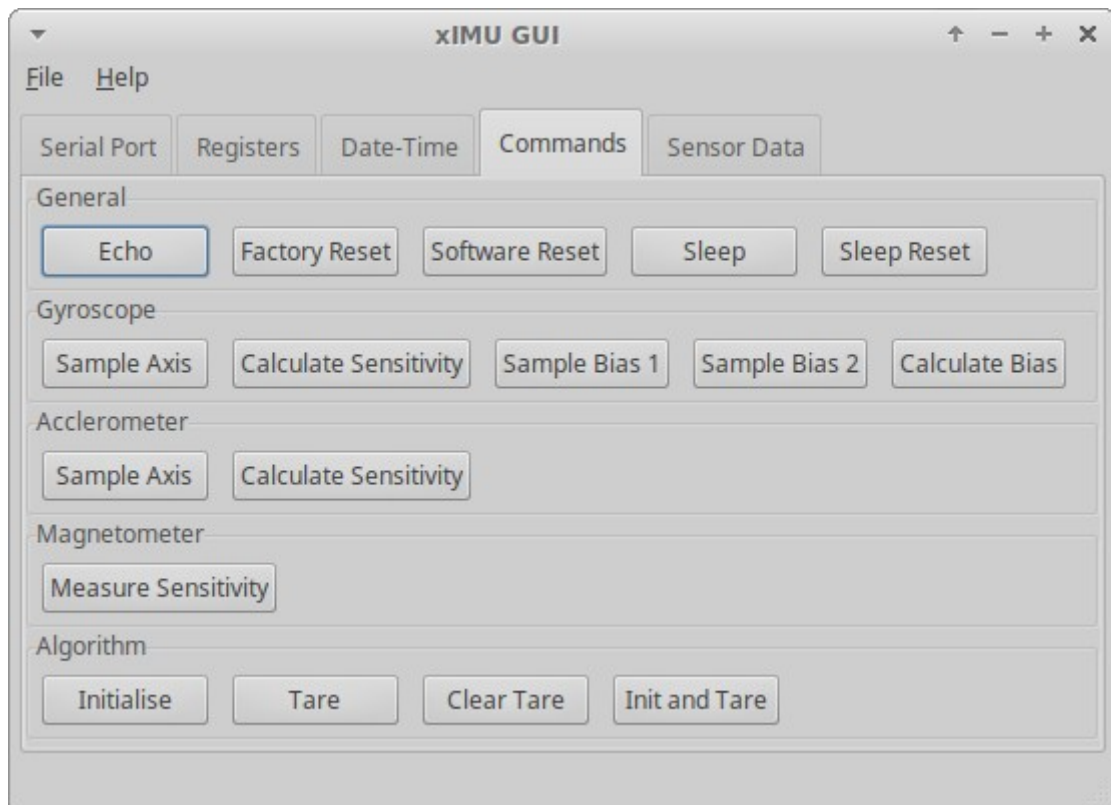


Figure 3.9: Cross-platform x-IMU GUI's command panel

The most basic general command is the echo command. The echo command provides a simple way to test connectivity to the device and makes no changes. The other general commands include a number of reset options. The factory reset command will reset all registers to their default values and will require the device to be recalibrated. The software reset will not affect the registers and will simply restart the device as if it had been powered off and on again. If the appropriate registers are set, the device will enter a low-power mode after a period of inactivity. The sleep reset command will reset the timer measuring the period of inactivity. The final general command is the sleep command, which will force the device into low-power mode.

The gyroscope, accelerometer and magnetometer category commands are used to calibrate the sensors after a factory reset. Since calibration requires additional specific equipment not available to this research, none of these commands are used.

The algorithm commands include: initialise, tare, clear tare, and initialise and tare. The initialise command will cause the algorithm to reinitialise i.e. reset the gyroscope integration. The tare command is used to set the current orientation as the starting orientation that the calculated orientation quaternions are relative to. The clear tare command will reset the stored starting quaternion to the default value. The final command, the initialise and tare command is the same as sending an initialise command followed by a tare command.

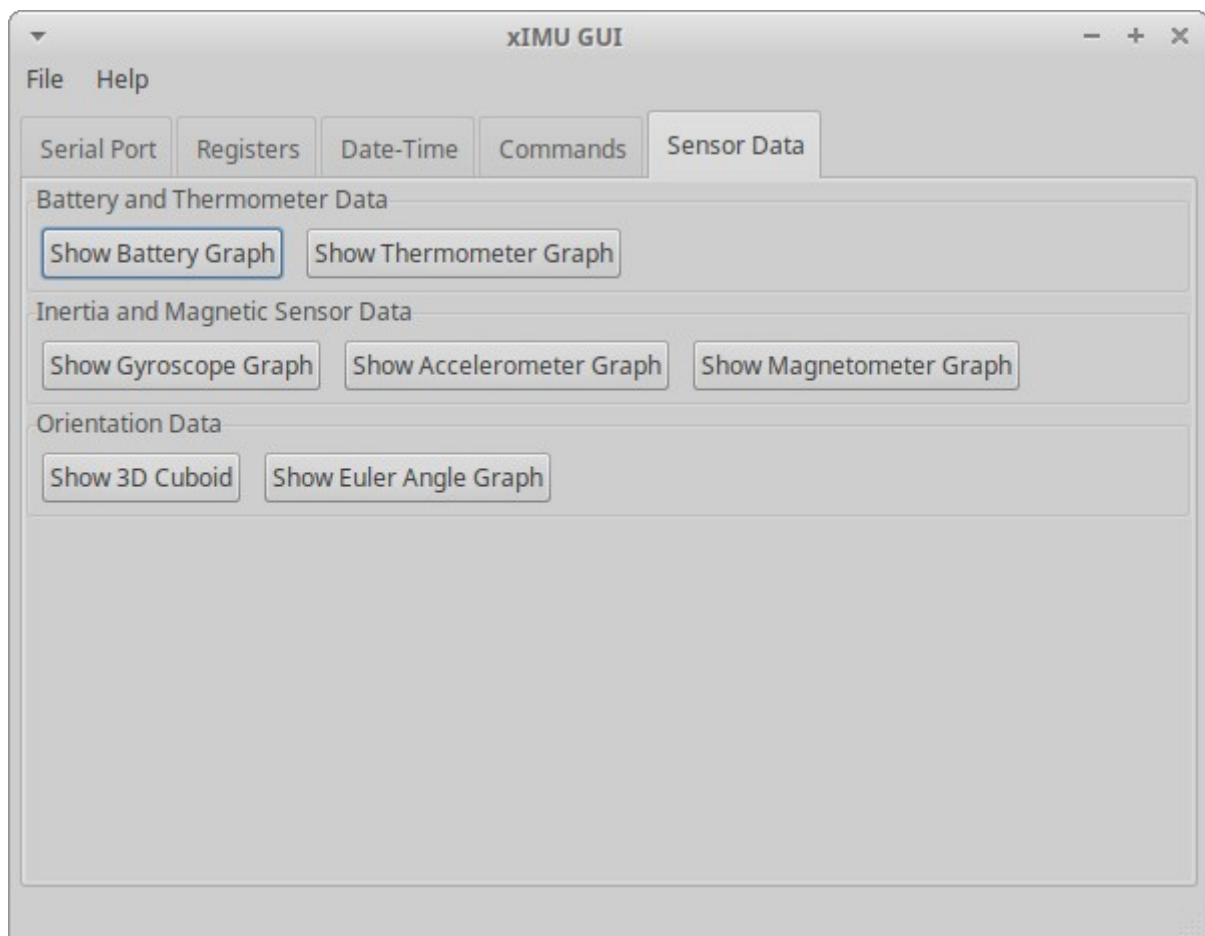


Figure 3.10: Cross-platform x-IMU GUI's sensor data panel

The cross-platform x-IMU configuration tool's final tab is the sensor tab shown in Figure 3.10. If the sensor data mode register has been set to send calibrated measurements, the sensor panel can be used to test whether data is being received. If the battery and thermometer data rate register is not set to disabled, the show battery graph and show thermometer graph buttons will display digital

oscilloscopes that show the calibrated battery and thermometer data that is being received from the connected device. Similarly if the inertial and magnetic data rate is not set to disabled, the show gyroscope graph, show accelerometer graph and show magnetometer graph buttons will respectively display digital oscilloscopes that show the calibrated gyroscope, accelerometer and magnetometer calibrated data being received from the connected device. An example of the digital oscilloscope graph is shown in Figure 3.11.

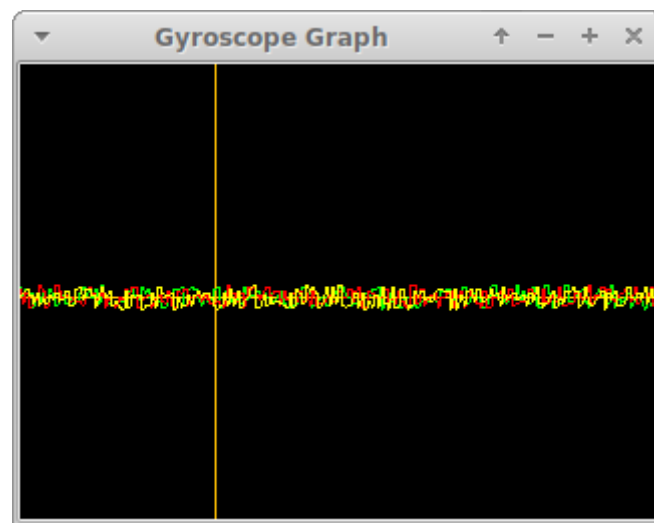


*Figure 3.11: Cross-platform x-IMU GUI's
Gyroscope Graph*

The best way to visualise the orientation quaternion, is to rotate a three dimensional representation of the device from the expected starting orientation using the orientation quaternion. If the quaternion data rate register is not set to disabled, the show 3D cuboid button will display the cuboid of Figure 3.12. The cuboid rotates as the connected device is rotated, which is used to confirm the operation and tare of the orientation quaternion. The orientation displayed will be relative to the reference quaternion stored using the tare command. If a tare command is sent when the device's axes are parallel and perpendicular to the display screen, the cuboid orientation will continue to reflect the device's orientation.

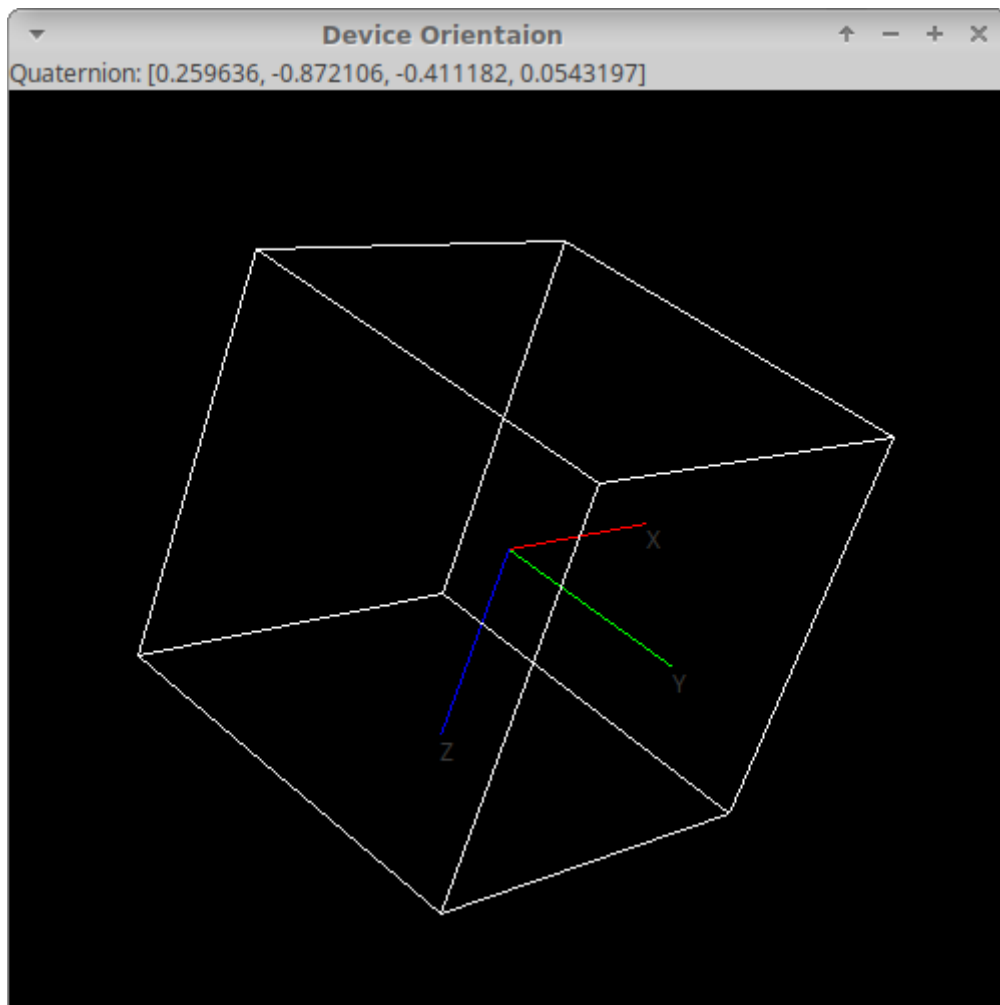


Figure 3.12: Orientation quaternion visualisation

The final result is a cross-platform x-IMU GUI that now allows the x-IMUs to be configured and tested from any operating system.

3.4.2 nuIMU GUI

A cross-platform nuIMU GUI was also created that allows the nuIMUs to be configured and tested. Figure 3.13 shows the principal nuIMU GUI interface. It simultaneously displays real-time data received from each of the three dimensions of the accelerometer, gyroscope and magnetometer, the eight external sensors and the four dimensions of the orientation quaternion. The frequency at which data packets associated with each of the sensor types is also displayed.

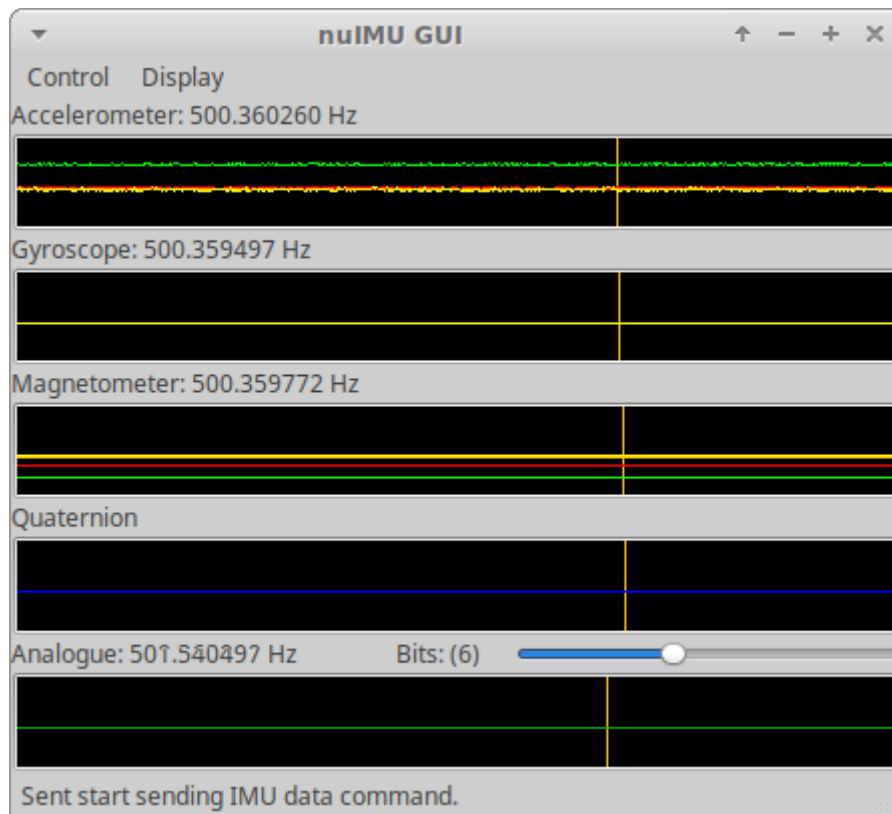


Figure 3.13: Cross-platform nuIMU GUI

The scale used for displaying the accelerometer, gyroscope and magnetometer data is the scale set in the configuration of the IMU. This provides a useful visualisation for confirming that the correct scale configuration setting is being used. The quaternion data is normalised, Therefore, they do not need to be scaled. The external sensors are independent of the IMU; so the scale of the analogue signals received from the external sensors cannot be set via the IMU configuration. Instead, the analogue signal display includes a slider for specifying the scale in the form of the number of significant bits to use when displaying the data.

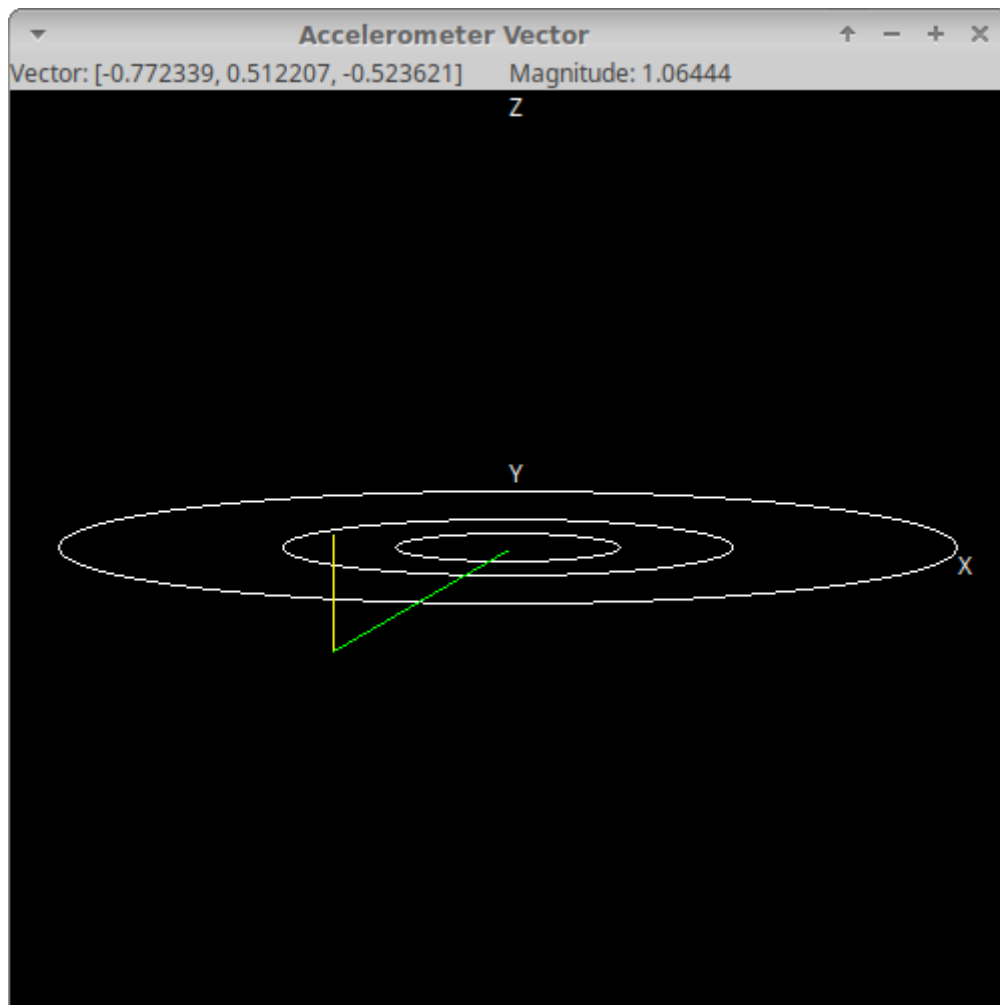


Figure 3.14: Three-dimensional vector data visualisation

Although viewing the raw three-dimensional data received from the accelerometer, gyroscope and magnetometer, and four-dimensional data of the orientation quaternion is useful, the nuIMU GUI also provides the actual three-dimensional view of each source. Figure 3.14 shows the how three-dimensional data from, in this case, the accelerometer is displayed. The three-dimensional values are displayed as the green line from the centre to its projection onto the X-Z plane. However, when using a simple projection of a variable length three dimensional vector on to a two dimensional plane, it's not possible to differentiate between a change in the length of the vector and the extent to which it extends into or out of the third dimension. Therefore, to properly visualise a three dimensional vector drawn as a line on a two dimensional surface, two things are done. First, the X-

Z plane is tilted and the X-Y plane is shown as a series of concentric circles. Second, a vertical, yellow line is drawn from the end of the three dimensional vector to the a point where it would be projected on the tilted X-Y plane.

Similarly, the best way to visualise the orientation quaternion, is to rotate a three dimensional representation of the device from the expected starting orientation using the orientation quaternion. The same cuboid representation used for the x-IMU shown in Figure 3.12 is used to represent the nuIMU device that has been rotated using the orientation quaternion calculated by the onboard AHRS orientation quaternion algorithm. This can be used to confirm the calculated orientation quaternion matches the rotation required from the starting orientation to the current orientation of the device.

As with the x-IMU GUI, the nuIMU GUI can switch between different IMUs by selecting the required serial port, or in the case of recorded data, the required file. However, as shown in Figure 3.15, the nuIMU GUI can also switch online between nuIMU Data Interfaces.

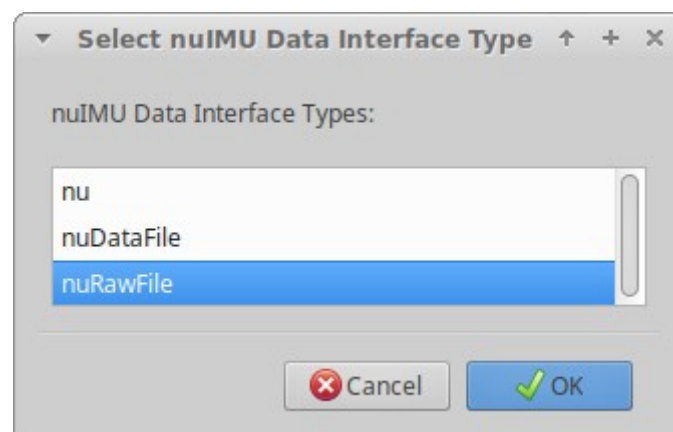


Figure 3.15: Cross-platform nuIMU GUI's online selection of the nuIMU Data Interface Type panel

In addition to being able to send any arbitrary command to the connected IMU, the nuIMU GUI can be used to send various pre-programmed commands to the connected IMU. Pre-programmed

commands include those used to test the gyroscope and accelerometer, start and stop sending MARG and quaternion data, and setting a specific configuration or register value. However, the most useful feature is the ability to display, modify and set all the built-in configuration settings.

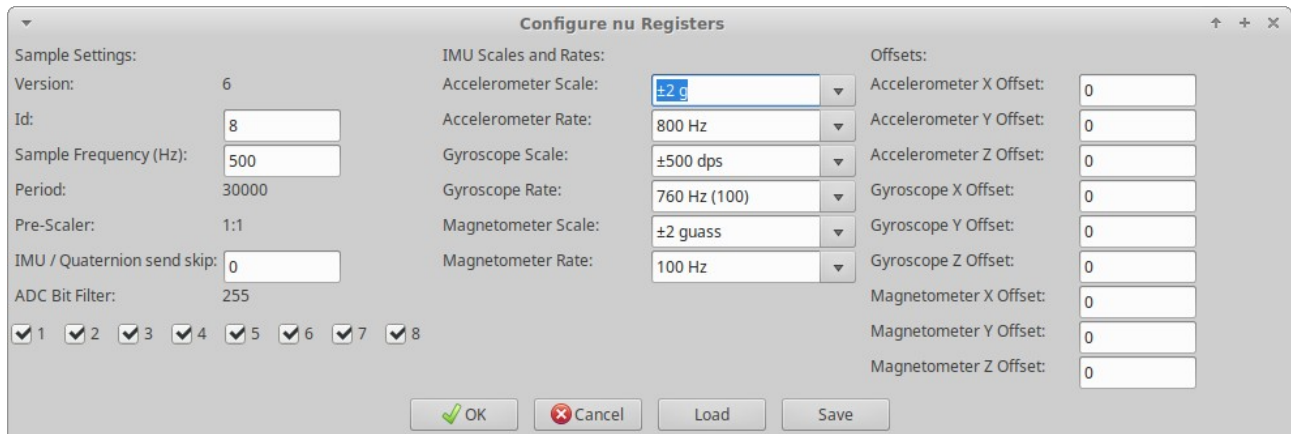


Figure 3.16: nuIMU GUI's register configuration panel

The nuIMU uses multiple registers. The byte value of each register is associated with a particular configuration setting. To change a configuration setting, the required byte value is obtained manually from a lookup table in the documentation provided by the component manufacturer. To change a specific setting, a command to change that register is sent with the required byte value. If the register and the byte value is recognised the register is changed. Obviously this is error-prone and a time consuming process. The nuIMU GUI enables all the component and configuration registers to be quickly and easily set using human readable values. The human readable values are mapped to the required register and byte values. When required, drop-down lists are used to ensure only valid values are selected. Furthermore, when multiple register values are used to determine a single value, only the single value needs to be entered. Likewise, when multiple settings are configured with a single register, each setting can be modified in isolation.

3.4.3 nuIMU Firmware Updates

The key challenge with using the nuIMU was the frequency with which it would stop responding. Fortunately, the firmware was available, and the problem could be resolved.

The first step was to clearly document the original code, while identifying and deleting unused code. Next, the main code loop was converted into a state machine. Flags are used to control which parts of the main loop will run. Each part was then moved into a sub-routine, where it can be managed, improved and simplified independently.

Next a number of changes were made to identify the source of the instability. First, to ensure there are no buffer overruns, all writes to buffers now first check to ensure the buffer is not full. If the buffer is full, the new data is dropped and a flag is set to signal the main loop to discard the unprocessed data and clear the buffer. This code also resets the state machine as required by resetting the appropriate flags.

Next, to ensure that interrupts are not missed, the interrupts were reduced to do just two things. First copy the data from the hardware buffer to the software buffer. Second set a flag to signal to the state machine that data has been received. This approach also allows interrupts to be interrupted.

Allowing interrupts to be interrupted ensures all interrupts are processed. Processing the data stored in the software buffer is done by a sub-routine that is launched from the main loop. The sub-routine runs, because the flag is set. When the sub-routine is finished, it clears the software buffer and resets the flag.

The initial firmware used the fact that all incoming packets were of the same length. Bytes were saved to a buffer as they were received. Once the expected number of bytes had been received, they would be processed. The result was that if a byte was ever last in transmission, the first byte of the next packet would be assumed to be the last byte of the previous packet. All subsequent packets

would then both lose the first byte of their header and be one byte short. They would be malformed and the problem would perpetuate. The result was that the device became unresponsive to commands being sent to it.

A second state machine was created to control incoming serial data. The serial data control state controls how incoming data bytes are treated. The default state is the waiting state. In the waiting state, all data is ignored until a byte representing the first byte of a header is received. When a byte representing the first byte of a header is received, the serial data control state is switched to the receiving header state. If any of the next bytes are not the expected next bytes of the header, the serial data control state switches back to the waiting state. If the full header is received, the serial data control state switches to the receiving data state. The device will remain in the receive data state until the expected number of bytes are received. Once the number of bytes has been received, the serial data control state returns to the waiting state.

With this approach, any problems in transmission will result in the serial data control state being reset to the waiting state when the next packet is received. This will result in the next packet also being lost, but sending any packet will result in the device returning to a responsive state. To overcome the loss of the next packet, the main loop was configured to reset the serial data control state to the waiting state if the serial data control state is in a receiving state and no bytes are being received.

Registry values are saved to the firmware to ensure that, once a device is configured as required, these settings are loaded every time the device is powered on. To save time, writing to firmware only sets bits. It assumes that all bits are cleared. This is a valid assumption the first time data is written, but not on subsequent writes. To correctly write to the firmware that has previously been written to, the bits must first be cleared. The code that saves registry values to the firmware needed to be updated to ensure that it is correctly cleared bits before new data was written.

The code that loads registry values saved to the firmware was updated to ensure that the data structures are correctly reconstructed. Most of the registry values require two or more bytes of data. Data is read from the firmware one byte at a time. The original code assumed that sequential memory bytes were used. Multi-byte type variables would then be “recreated” by simply accessing the multiple bytes that just happened to be next to each other. However, it cannot be assumed that sequential memory bytes are used. Therefore, the code was updated to not use this assumption. Multi-byte types are reconstructed by correctly shifting the more significant bytes before adding the byte components.

Finally, the code that calculates the quaternions was updated to use the improved AHRS orientation quaternion algorithm described in Chapter 4.

3.5 IMU Type Independent Control of a Virtual Robot

To test the IMU type independent data capture, a body motion tracker was created. The output of the body motion tracking was then used to control a Virtual Robot.

The Virtual Robot has fifteen controllable body segments: the left and right, upper and lower arm segments, the left and right, upper and lower leg segments, the left and right foot, the torso, the perpendicular shoulder segment, the perpendicular hip segment, and the upper and lower parts of the head. To visualise the position of each segment, the Virtual Robot shown in Figure 3.17 was created. Each segment is represented by a vector with a default orientation. The Virtual Robot is created by connecting the vectors end to end relative to a starting point.

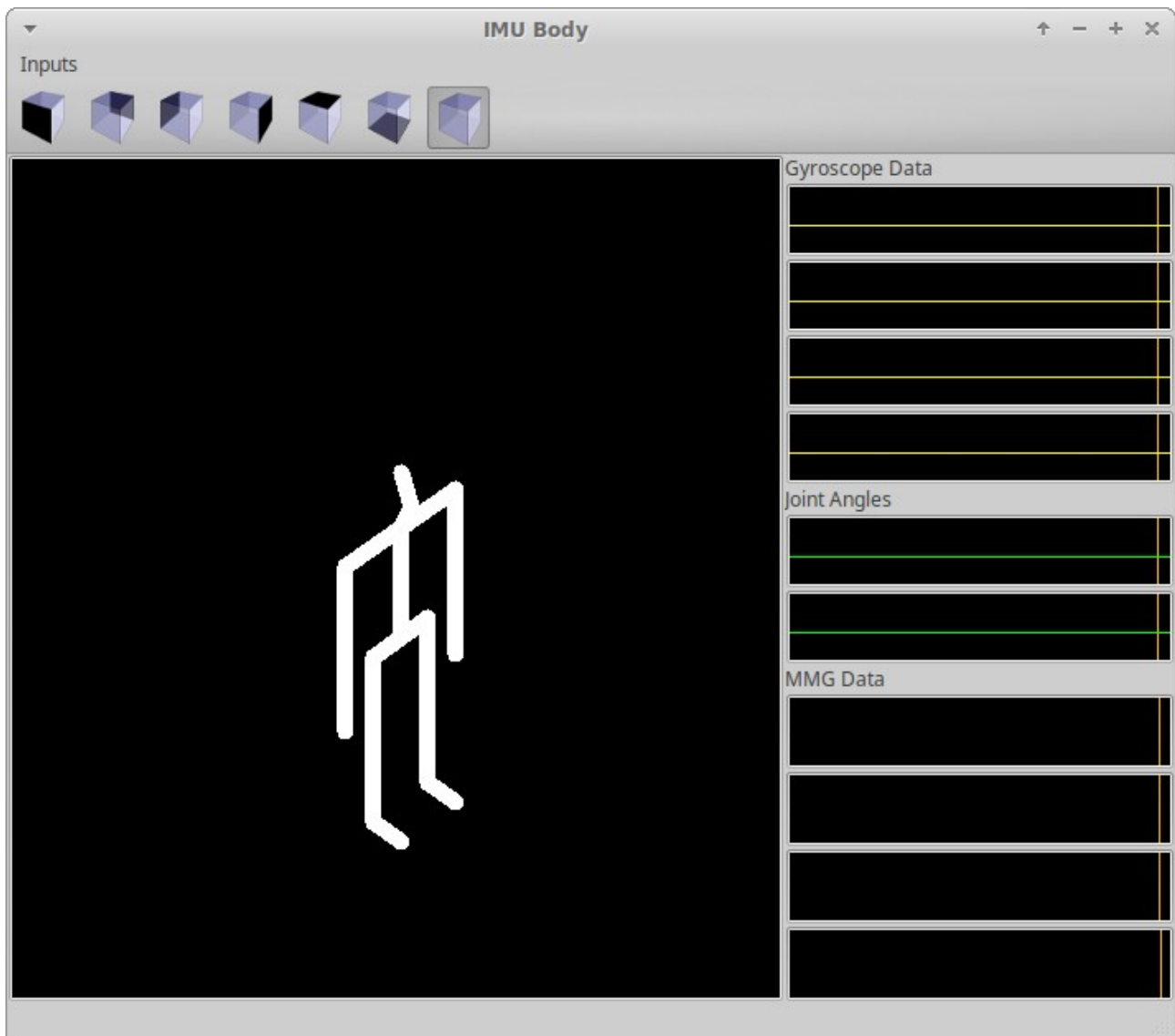


Figure 3.17: IMU controlled Virtual Robot

Two modes are available. The mode selected depends on the expectation of the movements to be captured. The first mode assumes that the top of the body segment is stationary. The second mode assumes the lowest point of either foot is stationary. In the first mode, bending of the knees will be seen as a jump with the feet coming up in the display. In the second mode, bending of the knees will be seen as a squat with the feet remaining stationary. Detecting which mode was required by, for example, using the accelerometer, was considered outside the scope of this work.

The orientation quaternions specify the rotation of the IMU⁹ relative to a specified starting orientation. For the x-IMUs the starting orientation is the orientation of the IMU the last time the xIMU tare command was sent to the IMU. This value is saved between power-cycles. For the nuIMU the starting orientation is the orientation of the IMU at the time the IMU is powered on.

The Virtual Robot orients the body segment by rotating the vector associated with the body segment by the orientation quaternion of the IMU attached to that body segment. For the orientation of the vector associated with the body segment to match the orientation of the IMU, the starting orientation of the IMU must match the starting orientation of the vector associated with the body segment. To enable IMUs to be placed in an orientation that is convenient and not prescribed, the application allows the user to specify the starting orientation of each IMU. The vector associated with the body segmented is first rotated into the starting orientation of the IMU before being rotated by the current orientation quaternion of the IMU. Finally, the vector is rotated back using the conjugate of the starting orientation of the IMU to provide the correct orientation of the vector associated with the body segment as provided by the current orientation of the IMU attached to that body segment.

IMU	Segment	X	Y	Z
IMU 0:	LeftUpperArm	Up	Forward	Left
IMU 1:	RightUpperArm	Up	Back	Right
IMU 2:	LeftLowerArm	Up	Forward	Left
IMU 3:	RightLowerArm	Up	Back	Right

Figure 3.18: Specifying the number, placement and starting orientations of the Virtual Robot's IMUs.

⁹ The quaternion calculated by the AHRS is the orientation of the world with respect to the IMU. To get the orientation of the IMU with respect to the world, the conjugate of the quaternion is used.

To facilitate the specification of the number of IMUs used, the body segment to which each IMU is attached and the starting orientation of that IMU, the GUI shown in Figure 3.18 was created. The starting orientation of an IMU is fully specified by the direction of two axes. The application will automatically calculate the direction of the third axis, when the direction of any of the other two axes is changed. The application will then calculate the rotation quaternion required to rotate the IMU from the default starting orientation to the specified starting orientation. The application also enables the specification of the placement and orientation of the IMUs to be saved and other configurations to be loaded. Although the GUI only allows specifying the direction of each axis in the six axes directions, the file format used is human readable. Therefore, the file can be updated, and any rotation quaternion can be used to specify the starting orientation of each IMU.



Figure 3.19: Tracking body motion using IMUs.

The result is a low-cost, real-time, flexible body motion tracker that is used to control the Virtual Robot. Figure 3.19 shows the application being used in a live demonstration with the full video available at the following URL: <https://www.youtube.com/watch?v=f1w4YPqB32c>. The application is now being developed into a commercial application for boxing training (Burr, 2017).

3.6 Real-Time Digital Signal Processing

In addition to ensuring that the inputs have a zero mean and are divided by their standard deviation, all the real-time filters and triggers created convert the raw sensor data from the IMUs and external sensors (including the MMGs) into digital signals that can be used as input into the Generic, Self-Improving Predictor described in Chapter 6.

Although no new digital signal processing algorithms have been created, most of the existing digital signal processing algorithms assume that the data sampling period is known and constant, and many of the algorithms require all the data to be available. Therefore, there was a requirement to convert the existing, static digital processing algorithms into real-time algorithms.

Real-time digital signal algorithms that allow synchronous samples to be received with variable sampling periods were formulated. To enable the processing of the same real-time data through multiple configurations, learning algorithms and filters, the real-time algorithms also needed to be able to process recorded real-time data played back at both real-time and delay-free speeds. In other words, they need to be processed as fast as possible, as if they were being received in real-time.

All filters created and described below produce the same result in any of three modes. First, all the data can be provided with a constant, known sample period. Second, the data can be provided synchronously in real-time. Third, the data can be passed synchronously together with the time that has passed since the last sample. Furthermore, all filters can filter multiple channels of data

simultaneously. Multiple channel data can be provided individually, if each channel is sampled independently, or concurrently, if all channels are sampled together.

To achieve this, an abstract Filter class was created. All the filters created are extensions of this Filter class. The abstract Filter class requires each derived Filter to define a filter function that, given the current input value, returns the filtered value. By default the function assumes that the time between function calls is the actual time passed; unless the actual time passed is specified. Each filter function works regardless of whether or not the previous call specified the actual time passed. This enables two additional features. First, the playback of recorded data can be switched online between real-time and without delays i.e. as fast as possible. Second, the play-back of recorded data can be paused. The same function is also used to filter static data. Therefore, the filtering of static data produces exactly the same results as the original digital signal processing function upon which it is based. This not only enables testing and comparison of the filter function against well-tested implementations, it also enables static viewing of recorded data that is filtered through the filter in real-time or filtered without delays. Being able to view the filter output facilitates the selection of the filters that will be applied.

Many of the filters rely on the values of previous inputs and outputs. To support this, a Temporal Data store that stores any number of channels of historical data for any specified length of time was created. Samples are stored using a period that does not need to match the period between received data. To support both real-time and delay-free play-back, the time passed between samples can be specified or, by default, use the actual time passed. The store uses linear interpolation to store the expected values at the store's period points in time. This provides the values required by static filters that depend on a fixed sample period. Each or all of the channels' data can be retrieved at any time. Either the real-time shifted data, the specified time shifted data, or the data as it was the last time it was added can be retrieved. The first approach enables real-time access to the data that is independent of the data addition. The second approach is required for fast-as-possible playback. The

last approach is required for algorithms that use historical data. Currently, the default minimum and specified accuracy of the user interrupt period on Microsoft based operating systems is 1 ms. Therefore, this is the minimum store period supported by these algorithms. However, a 1 ms period enables data sampling at 1 kHz, which is also the maximum sampling rate used by the nuIMU.

3.6.1 Simple Moving Average Filter

The first filter created was a simple moving average filter. The filter returns the running average of the specified number of previous samples:

$$\bar{x}_i = \bar{x}_{i-1} + \frac{x_i}{n} - \frac{x_{i-n}}{n} \quad (3.1)$$

(Winder, 2002)

Instead of specifying the number of samples, in this implementation of the simple moving average filter, the time over which the data should be averaged is specified. A Temporal Data store is used to store the data as it arrives and returns the mean of the required time length of historical data. If the data is received at regular intervals, the output is equivalent. However, this approach has the advantage of, not only supporting data received at irregular intervals, but it is not affected by missing samples.

3.6.2 Median Filter

A median filter is used to remove noise. It is a simple way of removing data anomalies or spikes, by ensuring the largest or smallest value in a series is not used¹⁰. It has the side effect of further flattening the maxima and minima encountered through discrete sampling.

¹⁰ (Winder, 2002)

This implementation also uses the quick select algorithm¹¹ to find any specified ordinal sorted value. To find the median value, half the total number of values is used for the ordinal value in the quick select algorithm.

Finally, similar to the simple moving average filter, this median filter also uses a Temporal Data store to ensure robustness in the event of missing samples and support samples that arrive at irregular intervals, while providing an equivalent filter for regularly spaced samples.

3.6.3 Short-Term Energy Filter

The short-term energy filter is used with a threshold trigger to detect the beginning and end points of relevant data in the presence of noise where the signal-to-noise ratio is 30 dB or better¹². It is calculated by taking the simple sum of the squares of a specified number of sample values, which can be calculated by using a running total:

$$E_i = E_{i-1} + x_i^2 - x_{i-n}^2 \quad (3.2)$$

(Mace *et al.*, 2013)

Again, for robustness, a Temporal Data store with a specified time length was used, instead of a specified number of samples. The short-term energy is calculated using the sum of the squares of the required number of samples appropriately scaled by the store's period.

¹¹ (Cormen *et al.*, 2009)

¹² (Rabiner & Sambur, 1974)

3.6.4 Simple Low-Pass Filter

In electronics, a simple low pass filter can be created using a single resistor and a single capacitor with the resistor in series and the output taken across the capacitor. The cut-off frequency is defined by:

$$f_c = \frac{1}{2\pi RC} \quad (3.3)$$

This simple filter can be simulated in a discrete environment where the filtered output is a function of the input and the previous output:

$$y_i = y_{i-1} + \alpha (x_i - y_{i-1}) \quad (3.4)$$

(Winder, 2002)

α is dependent on the time passed since the last input and the RC value:

$$\alpha = \frac{\Delta t}{RC + \Delta t} \quad (3.5)$$

In this implementation, either the RC value or the required cut-off frequency can be specified, and the time passed Δt is either the real-time passed or it can be specified.

3.6.5 Simple High-Pass Filter

Similarly, in electronics, a simple high-pass filter can be created using a single resistor and a single capacitor with the capacitor in series and the output taken across the resistor. The cut-off frequency is also defined by equation (3.3).

In this case, the discrete, simulated version is a function of the input and both the previous input and the previous output:

$$y_i = \alpha (y_{i-1} + x_i - x_{i-1}) \quad (3.6)$$

(Winder, 2002)

Again α is defined by equation (3.5).

This implementation also allows the user to either specify the RC value or the required cut-off frequency, and the time passed Δt is either the real-time passed or specified.

3.6.6 Butterworth Difference Filters

The simple low and high-pass filters described above have the limitation that the frequency response rolls off at only -20 dB per decade. To achieve higher roll-offs, higher order filters are required. Unlike other higher order filters, a Butterworth filter has no ripples in the pass band and rolls off to zero in the stop band.

In digital signal processing, Butterworth filters can be implemented synchronously using the difference equation:

$$a_0 y_i = b_0 x_n + b_1 x_{i-1} + \dots + b_{n_b} x_{i-n_b} - a_1 y_{i-1} - \dots - a_{n_a} y_{i-n_a} \quad (3.7)$$

(Winder, 2002)

The $\mathbf{a}$ vector values are the Butterworth output parameters and the $\mathbf{b}$ vector values are the Butterworth input parameters. This is easy to implement, the challenge was calculating the Butterworth parameters used to create any order high-pass, low-pass, band-pass or band-stop filter.

The Octave code created by Paul Kienzle, Doug Stewart and Alexander Klein was ported to C++.

The Octave code was based on Proakis and Manolakis' Digital and Signal Processing book¹³. The

13 (Proakis & Manolakis, 1996)

parameters are calculated using the Laplace transform on the s-plane and then converting them to the z-plane using the bilinear transform¹⁴.

It is based on the principle that a low-pass filter specified by its poles and zeros in the s-plane, can be converted to the required low-pass, high-pass, band-pass or band-stop filter by transforming each of the original poles and zeros, depending on the type of filter and cut-off or edge frequencies required. The original poles and zeros used, are the known s-plane poles for a low-pass cut-off frequency of 1 Hz, which are the required order number of equally distributed points on the complex unit circle.

The Butterworth output parameters ***a*** are the real components of the z-plane zeros multiplied by the gain, while the Butterworth input parameters ***b*** are the real components of the z-plane zeros.

Combining this code with the Temporal Data store, stable, real-time and accelerated, high-order, low-pass, high-pass, band-pass and band-stop filters that can be instantiated by simply specifying the type, the required cut-off or edge frequencies and the order were created.

3.6.7 Envelope Filters

An envelope filter that uses the magnitude of the analytic signal to obtain the instantaneous amplitude was also implemented.

¹⁴ The s-plane is the complex plane used to calculate equations in the frequency domain that are specified in the time domain using the Laplace transform, while the z-plane is the discrete-time equivalent (Graf, 2012).

The analytic signal is a complex-valued function that has no negative frequency components. The analytic signal of a real valued function $x(t)$ is calculated by, first, obtaining its Fourier transform $X(f)$, then, by definition, doubling the values for all the positive frequencies, keeping the offset the same, and setting the values for all the negative frequencies to zero:

$$X_a(f) \stackrel{\text{def}}{=} \begin{cases} 2 X(f) & \text{for } f > 0 \\ X(f) & \text{for } f = 0 \\ 0 & \text{for } f < 0 \end{cases} \quad (3.8)$$

(Smith, 2007)

Finally, the inverse Fourier transform of this modified Fourier transform $X_a(f)$ is calculated.

The in-place, breadth-first, decimation-in-frequency Cooley-Tukey Fast Fourier Transform algorithm¹⁵ for calculating the discrete Fourier transform and discrete inverse Fourier transform was used, while manually applying the value changes from the definition (3.8).

The Temporal Data store was used to capture the input values. To ensure the Fourier transform functions correctly, the length is set to an exponent of two, and is increased if necessary. The real-time filter then simply returns the last value of the analytic function.

3.6.8 Trigger Filters

Once the input data has been filtered to obtain the signal required, two trigger filters that convert the signal to a digital signal were created.

The binary trigger filter returns 1 if the input signal has been above a specified threshold value for a specified amount of time, and zero if the signal has fallen below the specified threshold at any time in the previous specified time duration.

¹⁵ (Rosetta Code, n.d.)

The frame trigger filter extends the binary trigger by, in addition to checking that the signal has exceeded a minimum threshold, also checking that the signal hasn't exceeded a specified maximum threshold during the specified time period. Furthermore, it puts a maximum length on the trigger, after which the signal must either drop below the minimum threshold or exceed the maximum threshold, before it can trigger again.

3.6.9 Multi-Filter

Finally, a Multi-Filter that can combine any number of the above filters, including the trigger filters and other Multi-Filters in the required order was created.

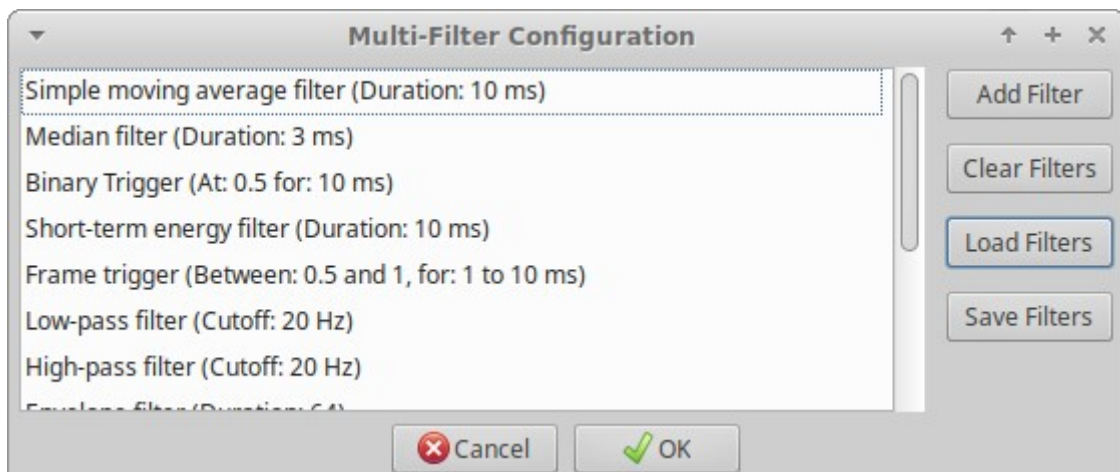


Figure 3.20: Multi-Filter configuration tool.

Figure 3.20 shows the GUI tool created for configuring each of the above filters and combining them into a Multi-Filter. After the user has selected a filter, the GUI prompts the user for the parameters required to specify that filter. Once the Multi-Filter has been configured, it can be saved, and loaded into any application that uses the filters.

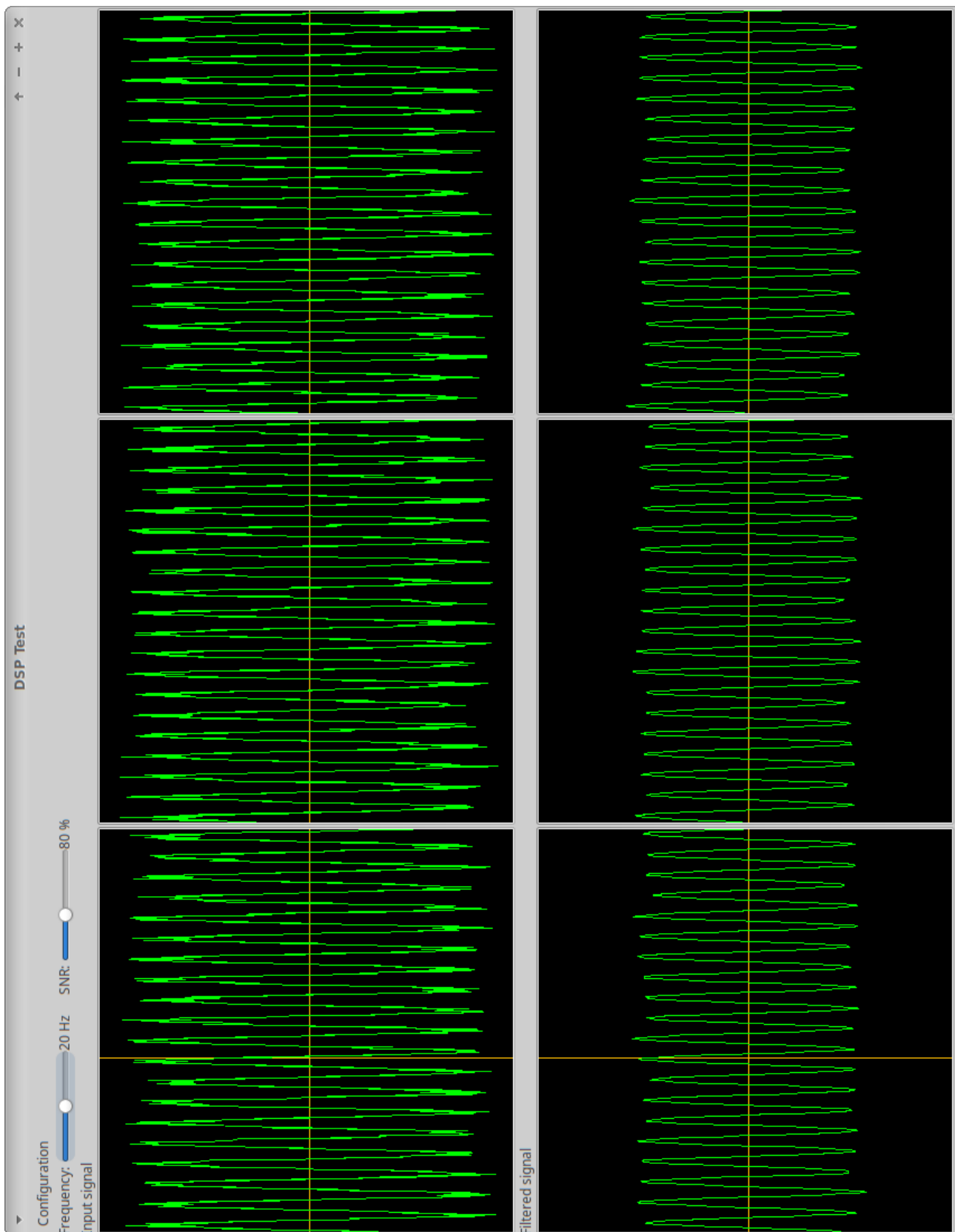


Figure 3.21: Testing and comparing real-time, real-time recorded and full-speed recorded filters.

3.6.10 Testing the Digital Signal Filters

To test and confirm the real-time, real-time recorded and full-speed recorded versions of each of the filters produces the exact same output, the Filter Test Suite shown in Figure 3.21 was created.

The filter (or Multi-Filter combination of filters) to be tested is created using the Multi-Filter configuration tool. A noisy, sinusoidal signal is used as input to each of the three versions of the filter. The frequency of the sinusoid and the signal to noise ratio can be adjusted online. The input to each of the three versions of the currently configured filter is shown in the top row and the output in the bottom row. The left-hand column shows the real-time data filter. The middle column shows the real-time recorded data filter. The right-hand column shows the full-speed recorded data filter. Each time any of the settings changes, the data sent to the recorded data filters is resent. The data sent to the real-time filter is continuous, but the display can be frozen.

All the digital filters were tested in turn to ensure that the output from all three versions of the filter were the same.

3.7 *Fusing Multiple Sensors' Data*

To provide a single vector that can be used by the Generic, Self-Improving Predictor, the data from multiple IMUs of different types and analogue data from the MMGs need to be combined. To facilitate this, an IMU Data Collector, which uses IMU Data Collector Listeners to capture the data, was created.

To capture data from the multiple IMUs of different types, the IMU Data Collector uses a generic IMU Data Collector Listener that virtually extends the IMU Listener described in Section 3.2. The generic IMU Data Collector Listener defines the functions for receiving accelerometer, gyroscope, magnetometer and analogue data. The generic IMU Data Collector Listener also virtually extends

the specific IMU type listeners: the x-IMU Listener described in Section 3.3.1 and the nuIMU Listener described in Section 3.3.2. This enables the IMU Data Collector to also capture x-IMU and nuIMU specific data types if required. The specific IMU Data Collector Listeners are assigned to the specific IMU Device Interfaces. The specific IMU Data Collector Listeners call the generic IMU Data Collector Listener functions as new data arrives.

The IMU Data Collector Listeners are created with a pointer to the IMU Data Collector and an index for the IMU with which it is associated. The IMU Data Collector Listener data type defined functions call matching functions in the IMU Data Collector and specify the IMU index to enable the IMU Data Collector to collect data from as many IMUs of different types as required.

In addition to supporting multiple IMU types, the IMU Data Collector has a Multi-Filter attached to each data type for each IMU. As described in Section 3.6.9, each Multi-Filter can contain any combination of filters or triggers required.

For each data type, the IMU Data Collector can also be configured to store a specified length of historical data.

The data collected by each IMU can either be retrieved individually, returning a vector or even a quaternion representing the last data received from the specified IMU, or be collated into a single vector representing all the historical data from all the sensors from all the IMUs. When additional IMUs are added, or removed, the collated data vector is automatically extended, or truncated.

Similarly, when the length of any historical data stored is increased, or decreased, additional values are inserted, or removed, extending, or truncating, the overall length of the collated data vector.

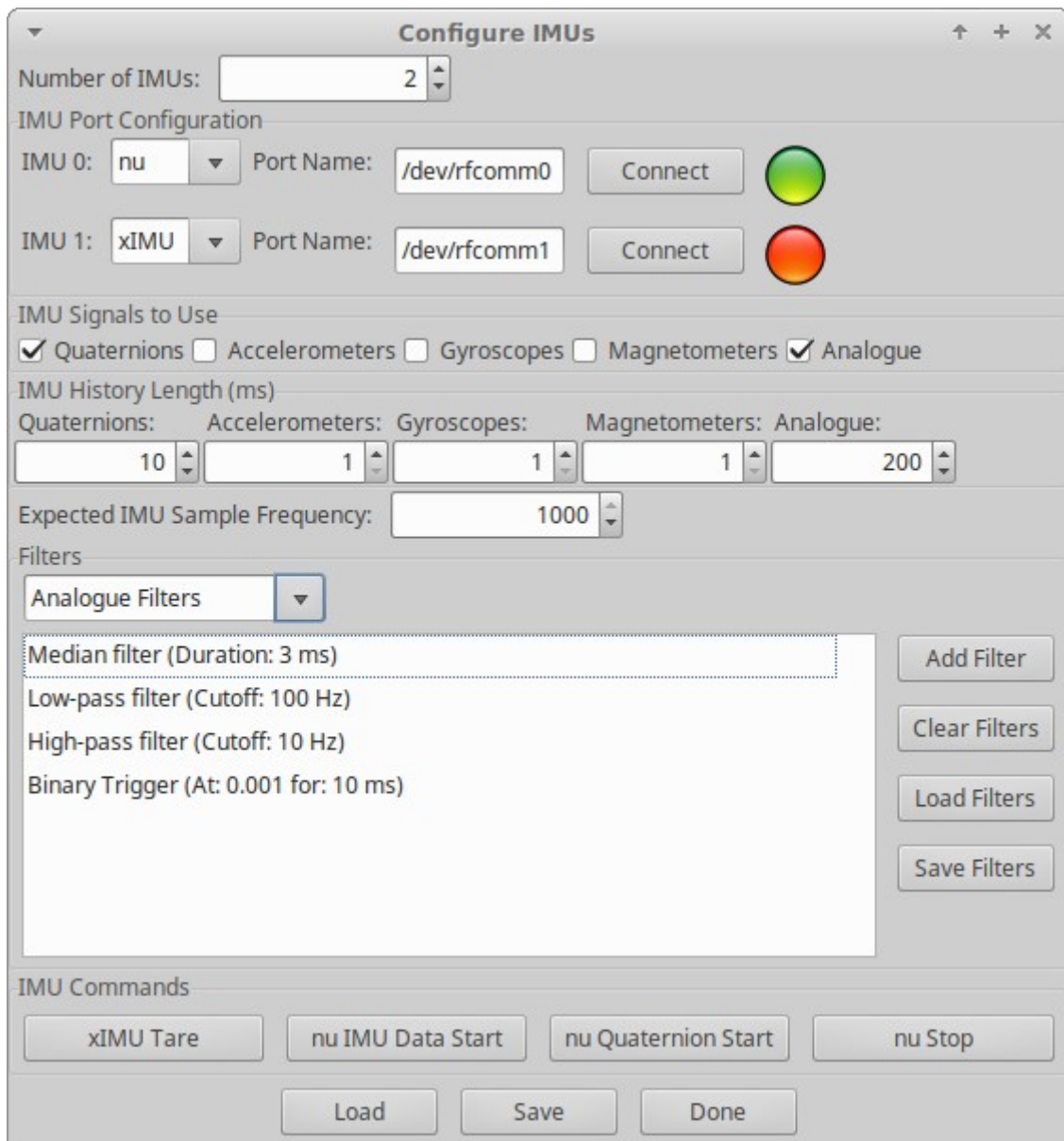


Figure 3.22: Interface for configuring the IMU Data Collector.

To facilitate configuring and controlling the IMUs connected to an IMU Data Collector, the IMU Data Collector Configuration GUI shown in Figure 3.22 was created. The IMU Data Collector Configuration GUI enables selecting the number of IMUs connected, which automatically adjusts the number of IMUs that can be configured. For each IMU, it uses the Serial Panel described in Section 3.2 and shown in Figure 3.2 to connect the IMU to a specific serial port. The IMU Data

Collector Configuration GUI also enables the user to select the type of IMU. Each IMU is connected to, their IMU Device Interface type.

For each data type, the user can specify whether or not to use the data signal, and how much history to keep. The expected sample frequency is used to determine the length of the history buffers required. Each data type can also have its own set of filters. Filter sets can be saved and previously configured filters can be loaded. Specific, commonly used commands available in the IMU GUIs described in Sections 3.4.1 and 3.4.2, have also been included. Finally, the entire configuration can be saved for future use.

The IMU Data Collector can also be configured to record the data that is received, and played-back at a later date. The recorded data is stored without any of the data settings applied. This enables the same data received from multiple IMUs to be used multiple times by the same application with different settings: different data types, different historical lengths and different filters. Furthermore, different applications can use the same data, and data recorded by different applications in a different format can, once converted into the recorded format, be used by an application using the IMU Data Collector. Finally, the recorded data does not need to be replayed in real-time. If required, the IMU Data Collector can collect, filter and deliver the data as fast as the data can be processed. This is particularly useful when using the same data processed with different settings, or when collating data that was collected over long periods of time.

3.8 Summary

An IMU Data Collector abstracts the data collection and processing from the application. Changes can be made to the IMU Data Collector online. Data retrieval from multiple IMUs and other external sensors is independent of the types of IMU or external sensor used. The IMUs and MMGs used can be added or removed as required. The data from each sensor can be enabled or disabled

without affecting the applications that are using the data. Similarly, the amount of historical data kept can be increased or decreased as needed. Each sensor's data can be filtered using any combination of the digital filters created. Multiple applications can simultaneously retrieve the data as and when each application requires it, and not when it is generated. The data can be retrieved in real-time or recorded. Recorded data can be played-back in either real-time or as fast as possible. Each application can choose which data to receive, and from which IMUs, irrespective of type. Each application can also choose whether to retrieve all the data a single vector, each sensor's data independently, or both.

The collated data collected by the IMU Data Collector and retrieved as a single vector can be used as the input state vector used by the Generic, Self-Improving Predictor, which is described in Chapter 6.

Chapter 4

An Improved AHRS Orientation Quaternion Algorithm

4.1 Introduction

To calculate the orientation of an IMU, an attitude and heading reference system (AHRS) orientation quaternion algorithm is used. The AHRS orientation quaternion algorithm uses the data captured by a gyroscope, accelerometer and magnetometer to calculate an orientation quaternion: the rotation quaternion that rotates the device from a specified starting orientation to the current orientation. Madgwick et al. created an AHRS orientation quaternion algorithm that corrects for cumulative errors (Madgwick, Harrison & Vaidyanathan, 2011). This algorithm, known as the

Madgwick AHRS algorithm, was the original algorithm used in both IMUs used in this research. However, when reviewing the algorithm a number of issues were identified. To address these issues, the Madgwick AHRS algorithm was reformulated to create an Improved AHRS orientation quaternion algorithm.

Section 4.2 provides an overview of why quaternions are used to describe orientations, how orientation quaternions are calculated using MARG data, and how multiple methods can be combined to provide corrections. Section 4.3 evaluates the Madgwick AHRS algorithm, details its limitations, and justifies the need for reformulating the algorithm. Section 4.4 derives the Improved AHRS orientation quaternion algorithm. Section 4.5 describes the creation of an IMU device simulation environment for comparing implementations of the AHRS orientation quaternion algorithms. Finally, Section 4.6 presents the results from comparing the AHRS orientation quaternion algorithms in the simulation environment.

The improved formulation of the AHRS orientation quaternion algorithm and the results presented in this chapter were published in the Proceedings of the 2017 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (Admiraal, Wilson & Vaidyanathan, 2017).

4.2 Overview of Quaternions, Orientation and Correction

Euler angles: yaw, pitch and roll, iteratively describe the rotation around three axes. They provide an intuitive way to represent orientation. However, when the pitch angle reaches $\pm 90^\circ$, the roll axis will be aligned with the yaw axis, resulting in the loss of one degree of freedom. This loss of the degree of freedom is known as a gimbal lock from the mechanical equivalent. When the pitch is reduced from 90° the direction is either constrained by the current position of the other axes or requires a step change in the other axes. Using Euler angles to represent orientation and gyroscopes

to measure the rate of change, results in mathematical singularities. This can be avoided by adding a fourth rotational axis that is deliberately changed to avoid a pitch of $\pm 90^\circ$ ¹⁶. However, a more convenient approach is to use quaternions¹⁷.

Euler's rotation theorem states that any rotation or sequence of rotations is equivalent to a single rotation about a single axis. Similarly, an orientation can be described relative to a starting orientation irrespective of the rotations made using a single rotation about a single axis. Quaternions can be used to represent a rotation about a single axis and hence orientations relative to a starting orientation. Unlike Euler angles, which also depend on the order the angles are provided, quaternions provide a consistent representation, and they avoid the problem of gimbal lock. Quaternions are also more compact than rotation matrices, and have been shown to be more numerically stable¹⁸.

Quaternions extend the complex number system and are represented in the form:

$$\mathbf{q} = w + x\mathbf{i} + y\mathbf{j} + z\mathbf{k} \quad (4.1)$$

where $\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1$, and $\mathbf{ij} = \mathbf{k}$, $\mathbf{ji} = -\mathbf{k}$, $\mathbf{ik} = -\mathbf{j}$, $\mathbf{ki} = \mathbf{j}$, $\mathbf{jk} = \mathbf{i}$, $\mathbf{kj} = -\mathbf{i}$.

The product of two quaternions is non-commutative and is defined as:

$$\begin{aligned} \mathbf{q}_1 \cdot \mathbf{q}_2 = & w_1 w_2 - x_1 x_2 - y_1 y_2 - z_1 z_2 + \\ & (w_1 x_2 + x_1 w_2 + y_1 z_2 - z_1 y_2) \mathbf{i} + \\ & (w_1 y_2 - x_1 z_2 + y_1 w_2 + z_1 x_2) \mathbf{j} + \\ & (w_1 z_2 + x_1 y_2 - y_1 x_2 + z_1 w_2) \mathbf{k} \end{aligned} \quad (4.2)^{19}$$

The conjugate of a quaternion is analogous to the conjugate of a complex number and is defined as:

$$\mathbf{q}^* = w - x\mathbf{i} - y\mathbf{j} - z\mathbf{k} \quad (4.3)$$

16 (Lovell & Kluger, 2006)

17 (Harrison, 1999)

18 (Lorusso *et al.*, 1995)

The norm of a quaternion is defined as the square root of the product of a quaternion and its conjugate:

$$\|q\| = \sqrt{qq^*} \quad (4.4)^{20}$$

The norm allows the reciprocal of a quaternion to be defined as:

$$q^{-1} = \frac{q^*}{\|q\|^2} \quad (4.5)$$

$$\begin{aligned}
 19 \quad q_1 \cdot q_2 &= (w_1 + x_1 i + y_1 j + z_1 k)(w_2 + x_2 i + y_2 j + z_2 k) \\
 &= w_1(w_2 + x_2 i + y_2 j + z_2 k) + \\
 &\quad x_1 i(w_2 + x_2 i + y_2 j + z_2 k) + \\
 &\quad y_1 j(w_2 + x_2 i + y_2 j + z_2 k) + \\
 &\quad z_1 k(w_2 + x_2 i + y_2 j + z_2 k) \\
 &= w_1 w_2 + w_1 x_2 i + w_1 y_2 j + w_1 z_2 k + \\
 &\quad x_1 w_2 i + x_1 x_2 i i + x_1 y_2 i j + x_1 z_2 i k + \\
 &\quad y_1 j w_2 + y_1 x_2 j i + y_1 y_2 j j + y_1 z_2 j k + \\
 &\quad z_1 k w_2 + z_1 x_2 k i + z_1 y_2 k j + z_1 z_2 k k \\
 &= w_1 w_2 + w_1 x_2 i + w_1 y_2 j + w_1 z_2 k + \\
 &\quad x_1 w_2 i - x_1 x_2 + x_1 y_2 k - x_1 z_2 j + \\
 &\quad y_1 w_2 j - y_1 x_2 k - y_1 y_2 + y_1 z_2 i + \\
 &\quad z_1 w_2 k + z_1 x_2 j - z_1 y_2 i - z_1 z_2 \\
 &= w_1 w_2 - x_1 x_2 - y_1 y_2 - z_1 z_2 + \\
 &\quad w_1 x_2 i + x_1 w_2 i + y_1 z_2 i - z_1 y_2 i + \\
 &\quad w_1 y_2 j - x_1 z_2 j + y_1 w_2 j + z_1 x_2 j + \\
 &\quad w_1 z_2 k + x_1 y_2 k - y_1 x_2 k + z_1 w_2 k \\
 &= w_1 w_2 - x_1 x_2 - y_1 y_2 - z_1 z_2 + \\
 &\quad (w_1 x_2 + x_1 w_2 + y_1 z_2 - z_1 y_2) i + \\
 &\quad (w_1 y_2 - x_1 z_2 + y_1 w_2 + z_1 x_2) j + \\
 &\quad (w_1 z_2 + x_1 y_2 - y_1 x_2 + z_1 w_2) k
 \end{aligned}$$

$$\begin{aligned}
 20 \quad q \cdot q^* &= (w + x i + y j + z k)(w - x i - y j - z k) \\
 &= w(w - x i - y j - z k) + \\
 &\quad x i(w - x i - y j - z k) + \\
 &\quad y j(w - x i - y j - z k) + \\
 &\quad z k(w - x i - y j - z k) \\
 &= w w - w x i - w y j - w z k + \\
 &\quad x i w - x i x i - x i y j - x i z k + \\
 &\quad y j w - y j x i - y j y j - y j z k + \\
 &\quad z k w - z k x i - z k y j - z k z k \\
 &= w^2 - w x i - w y j - w z k + \\
 &\quad w x i + x^2 - x y k + x z j + \\
 &\quad w y j + x y k + y^2 - y z i + \\
 &\quad w z k - x z j + y z i + z^2 \\
 &= w^2 + x^2 + y^2 + z^2 + \\
 &\quad -w x i + w x i - y z i + y z i + \\
 &\quad -w y j + w y j + x z j - x z j + \\
 &\quad -w z k + w z k - x y k + x y k \\
 &= w^2 + x^2 + y^2 + z^2
 \end{aligned}$$

Having a reciprocal enables quaternions to be divided.

To calculate the angle of rotation θ of a body with constant angular velocity ω , the angular velocity is multiplied by the duration Δt :

$$\theta \mathbf{u} = \omega \cdot \Delta t \quad (4.6)$$

A rotation through an angle of θ around an axis defined by a unit vector $\mathbf{u} = u_x \mathbf{i} + u_y \mathbf{j} + u_z \mathbf{k}$ can be represented by a quaternion using an extension of Euler's formula:

$$\mathbf{q} = e^{\frac{\theta}{2}(u_x \mathbf{i} + u_y \mathbf{j} + u_z \mathbf{k})} = \cos \frac{\theta}{2} + (u_x \mathbf{i} + u_y \mathbf{j} + u_z \mathbf{k}) \sin \frac{\theta}{2} \quad (4.7)$$

(Shepperd, 1978)

The quaternion $1 + 0\mathbf{i} + 0\mathbf{j} + 0\mathbf{k}$ represents no rotation.

Vectors can then be rotated using this rotation quaternion by first converting the vectors to quaternions with a real coordinate equal to zero: $\mathbf{p} = 0 + u_x \mathbf{i} + u_y \mathbf{j} + u_z \mathbf{k}$, and evaluating the conjugation of $\mathbf{p}$ by $\mathbf{q}$:

$$\mathbf{p}' = \mathbf{q} \cdot \mathbf{p} \cdot \mathbf{q}^{-1} \quad (4.8)$$

(Diebel, 2006)

Both $\mathbf{q}$ and $-\mathbf{q}$ represent the same rotation²¹. The reverse rotation is achieved by rotating vectors using the reciprocal: $\mathbf{q}^{-1}$. Since a rotation in a rotating body's own reference frame is the reverse of a rotation in the external reference frame, the transformation from the body's reference frame and the external reference frame and visa versa is also achieved by using the reciprocal of the quaternion. Furthermore, both the vectors that define an orientation's axes and the points of an object in three dimensional space relative to an orientation's axes, can be rotated using the same rotation quaternions.

21 $s\mathbf{q}$ for any s represents the same rotation: $s\mathbf{q} \cdot \mathbf{p} \cdot (s\mathbf{q})^{-1} = s\mathbf{q} \cdot \mathbf{p} \cdot s^{-1}\mathbf{q}^{-1} = ss^{-1}\mathbf{q} \cdot \mathbf{p} \cdot \mathbf{q}^{-1} = \mathbf{q} \cdot \mathbf{p} \cdot \mathbf{q}^{-1}$.

If $\mathbf{q}$ is normalised i.e. $\|\mathbf{q}\|^2 = 1$, the reciprocal can be replaced with the conjugate and the rotation of a vector equation can be replaced with:

$$\mathbf{p}' = \mathbf{q} \cdot \mathbf{p} \cdot \mathbf{q}^* \quad (4.9)$$

Using a normalised quaternion significantly simplifies rotation calculations.

Multiple rotations can be combined by first multiplying the quaternions together and then evaluating the conjugation. If each subsequent rotation is made in the rotating body's current reference frame, the quaternion multiplications are made in the forward order:

$$\mathbf{q}^B = \mathbf{q}_1 \cdot \mathbf{q}_2 \quad (4.10)$$

If each subsequent rotation is made in the same external reference frame, the quaternions need to be multiplied in reverse order:

$$\mathbf{q}^E = \mathbf{q}_2 \cdot \mathbf{q}_1 \quad (4.11)^{22}$$

This can be used to iteratively determine the rotation quaternion as a function of time of a rigid body undergoing constant angular velocity ω ; depending on whether the angular velocity is measured in the rotating body's reference frame:

$$\mathbf{q}(t) = \mathbf{q}_0 \cdot (\mathbf{q}_\omega^B)^t \quad (4.12)^{23}$$

or the external reference frame:

$$\mathbf{q}(t) = (\mathbf{q}_\omega^E)^t \cdot \mathbf{q}_0 \quad (4.13)$$

where $\mathbf{q}_0$ is the rotation of the body at time $t = 0$ and $\mathbf{q}_\omega^B$ and $\mathbf{q}_\omega^E$ are the quaternions of the rotation during a single time step.

²² $\mathbf{q}_2 \cdot (\mathbf{q}_1 \cdot \mathbf{p} \cdot \mathbf{q}_1^{-1}) \cdot \mathbf{q}_2^{-1} = \mathbf{q}_2 \cdot \mathbf{q}_1 \cdot \mathbf{p} \cdot \mathbf{q}_1^{-1} \cdot \mathbf{q}_2^{-1} = (\mathbf{q}_2 \cdot \mathbf{q}_1) \cdot \mathbf{p} \cdot (\mathbf{q}_2 \cdot \mathbf{q}_1)^{-1}$.

²³ If the rotation of the body at time $t = 0$, $\mathbf{q}(0) = \mathbf{q}_0$, then the rotation after one time step $\mathbf{q}(1) = \mathbf{q}_0 \cdot \mathbf{q}_\omega^B$ and after two time steps $\mathbf{q}(2) = \mathbf{q}_0 \cdot \mathbf{q}_\omega^B \cdot \mathbf{q}_\omega^B = \mathbf{q}_0 \cdot (\mathbf{q}_\omega^B)^2$, etc.

The derivatives of the functions of the rotation of a body with respect to time are:

$$\dot{\mathbf{q}}(t) = \frac{1}{2} \mathbf{q}(t) \boldsymbol{\omega}^B \quad (4.14)^{24}$$

$$\dot{\mathbf{q}}(t) = \frac{1}{2} \boldsymbol{\omega}^E \mathbf{q}(t) \quad (4.15)^{25}$$

If the angular velocity is not constant, the derivative can be used to quickly estimate the rotation of a body relative to an original orientation by repeatedly sampling the angular velocity and using finite sums:

$$\mathbf{q}(t + \Delta t) \approx \mathbf{q}(t) + \dot{\mathbf{q}}(t) \Delta t \quad (4.16)$$

The rotation quaternion that describes the rotation relative to a starting orientation is known as the orientation quaternion. The error of this estimated orientation quaternion will accumulate with time. Therefore, it is necessary to continually correct the estimation.

24 Using the polar form of the rotation quaternion $\mathbf{q}_\omega = e^{\frac{\theta}{2}\mathbf{u}}$, $(\mathbf{q}_\omega)^t = e^{t\frac{\theta}{2}\mathbf{u}}$:

$$\begin{aligned} \dot{\mathbf{q}}(t) &= \frac{d}{dt} \mathbf{q}_0 (\mathbf{q}_\omega)^t \\ &= \frac{d}{dt} \mathbf{q}_0 e^{t\frac{\theta}{2}\mathbf{u}} \\ &= \mathbf{q}_0 e^{t\frac{\theta}{2}\mathbf{u}} \mathbf{u} \frac{\theta}{2} \\ &= \mathbf{q}(t) \frac{\theta}{2} \mathbf{u} \\ &= \frac{1}{2} \mathbf{q}(t) \boldsymbol{\omega} \end{aligned}$$

$$\begin{aligned} 25 \quad \dot{\mathbf{q}}(t) &= \frac{d}{dt} (\mathbf{q}_\omega)^t \mathbf{q}_0 \\ &= \frac{d}{dt} e^{t\frac{\theta}{2}\mathbf{u}} \mathbf{q}_0 \\ &= \frac{\theta}{2} \mathbf{u} e^{t\frac{\theta}{2}\mathbf{u}} \mathbf{q}_0 \\ &= \frac{\theta}{2} \mathbf{u} \mathbf{q}(t) \\ &= \frac{1}{2} \boldsymbol{\omega} \mathbf{q}(t) \end{aligned}$$

Another method for estimating the orientation quaternion is to minimise the difference between a reference vector in the starting orientation rotated by the previous estimated orientation quaternion and a measurement of that vector:

$$\min_q f(\mathbf{q}, \mathbf{v}_r, \mathbf{v}_m) \quad (4.17)$$

where

$$f(\mathbf{q}, \mathbf{v}_r, \mathbf{v}_m) = \mathbf{q} \cdot \mathbf{v}_r \cdot \mathbf{q}^* - \mathbf{v}_m \quad (4.18)$$

and where $\mathbf{q}$ is the normalised, estimated orientation quaternion, $\mathbf{v}_r$ is the reference vector and $\mathbf{v}_m$ is the measured vector. There are infinite solutions to a rotation of a single vector – all rotations around the axis of the vector itself. Therefore, multiple reference vectors are required.

To minimise the difference between two vectors with respect to $\mathbf{q}$, either the length of the difference vector needs to be minimised:

$$\|f(\mathbf{q}, \mathbf{v}_r, \mathbf{v}_m)\| = 0 \quad (4.19)$$

or the non-linear equations that make each component of the vector zero need to be solved simultaneously:

$$f(\mathbf{q}, \mathbf{v}_r, \mathbf{v}_m) = \mathbf{0} \quad (4.20)$$

Both the minimisation of the length and the simultaneous non-linear equation approaches can be solved iteratively using gradient descent:

$$\mathbf{q}_{k+1} = \mathbf{q}_k - \alpha \nabla_{\mathbf{q}} F(\mathbf{q}) \quad (4.21)$$

where

$$F(\mathbf{q}) = \|f(\mathbf{q}, \mathbf{v}_r, \mathbf{v}_m)\|^2 \quad (4.22)$$

when minimising the length, and

$$F(\mathbf{q}) = \frac{1}{2} \mathbf{f}^T(\mathbf{q}, \mathbf{v}_r, \mathbf{v}_m) \mathbf{f}(\mathbf{q}, \mathbf{v}_r, \mathbf{v}_m) \quad (4.23)$$

with

$$\nabla_{\mathbf{q}} F(\mathbf{q}) = J_{\mathbf{q}}(\mathbf{f}(\mathbf{q}, \mathbf{v}_r, \mathbf{v}_m)) \cdot \mathbf{f}(\mathbf{q}, \mathbf{v}_r, \mathbf{v}_m) \quad (4.24)$$

when solving the simultaneous equations.

The accuracy of the orientation quaternion derived from minimising the difference between a reference vector rotated by the previous estimated orientation quaternion and a measurement of that vector depends on the accuracy of both the known reference vector and the measurement vector. In particular, when gravity is used as the reference vector and an accelerometer is used as the measured vector, the measured vector will be the vector sum of gravity and any lateral acceleration of the device containing the accelerometer. High accelerations will have a significant impact on the calculated orientation quaternion.

Both methods for estimating the orientation quaternion can be combined: A single iteration of the gradient descent difference minimisation algorithm can be used to improve the finite sums estimate of the orientation quaternion at each time step, where the rotation at $t = 0$ and zero angular velocity is the rotation of the body's reference vectors relative to the measured vectors.

A gyroscope is typically used to measure the angular velocity. Earth's gravity and magnetic field can be used as the reference vectors and compared with the measured vectors obtained from an accelerometer and magnetometer respectively.

The calculation of the gradient of the difference function can be simplified by setting one, or better two, of the axes of the reference vectors to zero. For example, earth's gravity can be assumed to only operate in the vertical, z-axis direction. Similarly, earth's magnetic field can be assumed to be

zero in the z-axis, and the known direction of earth's magnetic field at the current location can be specified.

To counter the error in the corrective step when the device containing the accelerometer is undergoing high accelerations, the corrective step can be skipped above a specified threshold.

4.3 *Evaluation of the Madgwick AHRS Algorithm*

Madgwick et al. created an AHRS orientation quaternion algorithm that uses equation (4.20) to correct for cumulative errors in equation (4.16). They use earth's gravity as the first reference vector, and they assume that it only operates in the z-axis direction. However, they don't use the direction of earth's magnetic field as the second reference vector. They mention using the declination of the earth's magnetic field instead of the direction, but they don't use the known declination at the current location. Instead, the reference vector is actually calculated from the current measured vector. However, there is no clear explanation as to how a reference vector, calculated from a measured vector, can be compared with the same measured vector to correct the quaternion. Therefore, how the algorithm was implemented is explained below.

The calculation of the reference vector involves first rotating the measured vector by the current orientation quaternion. Conversely, the gradient descent algorithm first rotates the reference vector by the inverse of the orientation quaternion and then compares it with the measured vector. At this point in the calculation of the reference vector, the difference will always be zero, and no quaternion updates will be made.

The second step in the calculation of the reference vector constrains it, but not completely. If the constraint was complete, i.e. a constant, the reference vector could be specified and there would be no need to constrain it. Madgwick et al. constrain it by rotating the vector around the z-axis on to

the two-dimensional plane perpendicular to the y-axis; and reflecting negative x values to the positive-x half of the plane:

$$\mathbf{v}'_r = [\sqrt{v_{xr}^2 + v_{ry}^2}, 0, v_{rz}] \quad (4.25)$$

(Madgwick, Harrison & Vaidyanathan, 2011)

Constraining the calculated reference vector to the positive-x half of the plane has no impact on vectors with positive x components. Vectors with negative x components will have vector differences proportional to the size of the rotated measured vector's x component, which does not provide orientation information.

Constraining the calculated reference vector to the plane perpendicular to the y-axis by rotating it around the z-axis is the same as fixing the reference vector for rotations around the z-axis.

Therefore, minimisation updates will be made for rotations around the z-axis. Further constraining the calculated reference vector to the positive-x half of the plane does not constrain the vector further and can actually result in ambiguous initial conditions depending on the initial orientation with respect to the measured magnetic field.

However, constraining the calculated reference vector to the plane perpendicular to the y-axis has no impact on rotations around the y-axis, because differences will be zero and again no correction will be made. Rotations around the x-axis will, depending on the vector, generally result in oscillations around the y-axis, which combines the problems of no impact on rotations around the y-axis with differences proportional to the rotated measured vector's z component.

When the calculated reference vector is further constrained to the positive-x half of the plane, the calculated reference vector rotated around the y-axis will eventually hit the z-axis. At this point, further rotations will only reduce the size of the calculated reference vector z-component. This fixes the direction of the calculated reference vector, which is required for updating the rotation vector.

Continued rotation around the y-axis will update the orientation quaternion in sync with, but offset from, the actual rotation, and this keeps the calculated reference vector direction fixed. However, this is only temporary. A rotation around the y-axis in the opposite direction will eventually release the calculated reference vector. Again, this will result in no updates until it hits the opposite z-axis.

In summary, the calculated reference vector does not provide a useful reference vector. It is not stationary, and when it is, it provides updates to the quaternion that do not reflect orientation.

However, the calculated reference vector is not used on its own. Two reference vectors are required to specify direction. The other reference vector is gravity, which is fixed to the z-axis. This reference vector correctly updates the quaternion for all rotations around the x- and y-axes. As discussed above, constraining the calculated reference vector to the plane perpendicular to the y-axis by rotating it around the z-axis is the same as fixing the reference vector for rotations around the z-axis.

The combination of using a fixed reference vector in one direction and a calculated reference vector rotated around the axis of the first reference vector on to a fixed plane, provides two fixed reference vectors. As long as the measured vector is not parallel with the fixed reference vector, the amount it needs to be rotated around the axis of the fixed reference vector can be determined. The actual angle of the measured vector away from the fixed reference vector is irrelevant.

When gravity is used as the fixed reference vector and assumed to only operate in the (negative) z-axis direction, the calculated reference vector can be obtained from the magnetometer reading, because its inclination will not be $\pm 90^\circ$ i.e. parallel to gravity. By rotating the magnetometer reading around the z-axis on to a fixed plane, its declination becomes irrelevant.

The actual inclination of the earth's magnetic field i.e. the direction of magnetic north remains relevant. The plane and direction that the calculated reference vector is rotated on to will be offset

from the actual inclination. However, a fixed reference vector will have the same problem, as well as the offset from the actual declination.

In conclusion, calculating the magnetometer's reference vector from the measured magnetometer reading and constraining it by rotating it around the axis of the accelerometer's fixed reference vector on to a plane, fixes the magnetometer's reference vector around the axis that cannot be fixed when just using the accelerometer's fixed reference vector. The calculated magnetometer's reference vector's declination is equal to the actual declination. The calculated magnetometer's reference vector will remain offset from the actual inclination of the earth's magnetic field in the same way a fixed reference vector would.

This analysis also revealed that the Madgwick et al. equations do not use the gradient descent algorithm (4.21), because the step sizes are not proportional to the gradient. Nor do they use the actual reverse rotation difference equation (4.18) and its gradient. Therefore, the Jacobian used in equation (4.24) to solve the simultaneous non-linear equations for iteratively updating the rotation unit quaternions is inaccurate. For these reasons the AHRS orientation quaternion algorithm was reformulated.

4.4 Improved AHRS Orientation Quaternion Algorithm

Formulation

A gyroscope is used to measure the angular velocity ω , which is multiplied by half the previous, corrected estimate of the orientation quaternion $\mathbf{q}_{t-1}$ using equation (4.14) to estimate the current derivative of the orientation quaternion with respect to time $\dot{\mathbf{q}}(t)$. The estimated derivative is multiplied by the period of each time step Δt and added to the previous, corrected estimate of the orientation quaternion using equation (4.16) to obtain the new uncorrected, estimate of the orientation quaternion $\mathbf{q}$.

To correct the estimate of the orientation quaternion, the difference between a reference vector $\mathbf{v}_r$ rotated by the new estimated orientation quaternion $\mathbf{q}$ and a measurement of that vector $\mathbf{v}_m$ is calculated using equation (4.18). This is achieved as follows:

First, to rotate the reference vector $\mathbf{v}_r$ by the estimated rotation quaternion $\mathbf{q}$, equation (4.9) is expanded:

$$\begin{aligned} \mathbf{v}_r \cdot \mathbf{q}^* = & \left(0q_w + v_{rx}q_x + v_{ry}q_y + v_{rz}q_z \right) + \\ & \left(0q_x + v_{rx}q_w - v_{ry}q_z + v_{rz}q_y \right) \mathbf{i} + \\ & \left(0q_y + v_{rx}q_z + v_{ry}q_w - v_{rz}q_x \right) \mathbf{j} + \\ & \left(0q_z - v_{rx}q_y + v_{ry}q_x + v_{rz}q_w \right) \mathbf{k} \end{aligned} \quad (4.26)$$

$$\begin{aligned} \mathbf{q} \cdot \mathbf{v}_r \cdot \mathbf{q}^* = & \left(q_w(v_{rx}q_x + v_{ry}q_y + v_{rz}q_z) - q_x(v_{rx}q_w - v_{ry}q_z + v_{rz}q_y) \right. \\ & \left. - q_y(v_{rx}q_z + v_{ry}q_w - v_{rz}q_x) - q_z(-v_{rx}q_y + v_{ry}q_x + v_{rz}q_w) \right) + \\ & \left(q_w(v_{rx}q_w - v_{ry}q_z + v_{rz}q_y) + q_x(v_{rx}q_x + v_{ry}q_y + v_{rz}q_z) \right. \\ & \left. + q_y(-v_{rx}q_y + v_{ry}q_x + v_{rz}q_w) - q_z(v_{rx}q_z + v_{ry}q_w - v_{rz}q_x) \right) \mathbf{i} + \\ & \left(q_w(v_{rx}q_z + v_{ry}q_w - v_{rz}q_x) - q_x(-v_{rx}q_y + v_{ry}q_x + v_{rz}q_w) \right. \\ & \left. + q_y(v_{rx}q_x + v_{ry}q_y + v_{rz}q_z) + q_z(v_{rx}q_w - v_{ry}q_z + v_{rz}q_y) \right) \mathbf{j} + \\ & \left(q_w(-v_{rx}q_y + v_{ry}q_x + v_{rz}q_w) + q_x(v_{rx}q_z + v_{ry}q_w - v_{rz}q_x) \right. \\ & \left. - q_y(v_{rx}q_w - v_{ry}q_z + v_{rz}q_y) + q_z(v_{rx}q_x + v_{ry}q_y + v_{rz}q_z) \right) \mathbf{k} \end{aligned} \quad (4.27)$$

$$\begin{aligned} = & \left(+v_{rx}q_wq_x + v_{ry}q_wq_y + v_{rz}q_wq_z - v_{rx}q_wq_x + v_{ry}q_xq_z - v_{rz}q_xq_y \right. \\ & \left. - v_{rx}q_yq_z - v_{ry}q_wq_y + v_{rz}q_xq_y + v_{rx}q_yq_z - v_{ry}q_xq_z - v_{rz}q_wq_z \right) + \\ & \left(+v_{rx}q_wq_w - v_{ry}q_wq_z + v_{rz}q_wq_y + v_{rx}q_xq_x + v_{ry}q_xq_y + v_{rz}q_xq_z \right. \\ & \left. - v_{rx}q_yq_y + v_{ry}q_xq_y + v_{rz}q_wq_y - v_{rx}q_zq_z - v_{ry}q_wq_z + v_{rz}q_xq_z \right) \mathbf{i} + \\ & \left(+v_{rx}q_wq_z + v_{ry}q_wq_w - v_{rz}q_wq_x + v_{rx}q_xq_y - v_{ry}q_xq_x - v_{rz}q_wq_x \right. \\ & \left. + v_{rx}q_xq_y + v_{ry}q_yq_y + v_{rz}q_yq_z + v_{rx}q_wq_z - v_{ry}q_zq_z + v_{rz}q_yq_z \right) \mathbf{j} + \\ & \left(-v_{rx}q_wq_y + v_{ry}q_wq_x + v_{rz}q_wq_w + v_{rx}q_xq_z + v_{ry}q_wq_x - v_{rz}q_xq_x \right. \\ & \left. - v_{rx}q_wq_y + v_{ry}q_yq_z - v_{rz}q_yq_y + v_{rx}q_xq_z + v_{ry}q_yq_z + v_{rz}q_zq_z \right) \mathbf{k} \end{aligned} \quad (4.28)$$

$$\begin{aligned}
&= (v_{rx}(q_w q_x - q_w q_x - q_y q_z + q_y q_z) + v_{ry}(q_w q_y + q_x q_z - q_w q_y - q_x q_z) \\
&\quad + v_{rz}(q_w q_z + q_x q_y - q_x q_y - q_w q_z)) + \\
&\quad (v_{rx}(q_w q_w + q_x q_x - q_y q_y - q_z q_z) + v_{ry}(-q_w q_z + q_x q_y + q_x q_y - q_w q_z) \\
&\quad + v_{rz}(q_w q_y + q_x q_z + q_w q_y + q_x q_z)) \mathbf{i} + \\
&\quad (v_{rx}(q_w q_z + q_x q_y + q_x q_y + q_w q_z) + v_{ry}(q_w q_w - q_x q_x + q_y q_y - q_z q_z) \\
&\quad + v_{rz}(-q_w q_x - q_w q_x + q_y q_z + q_y q_z)) \mathbf{j} + \\
&\quad (v_{rx}(-q_w q_y + q_x q_z - q_w q_y + q_x q_z) + v_{ry}(q_w q_x + q_w q_x + q_y q_z + q_y q_z) \\
&\quad + v_{rz}(q_w q_w - q_x q_x - q_y q_y + q_z q_z)) \mathbf{k}
\end{aligned} \tag{4.29}$$

$$\begin{aligned}
&= 0 + \\
&\quad (v_{rx}(q_w q_w + q_x q_x - q_y q_y - q_z q_z) + v_{ry}(-2q_w q_z + 2q_x q_y) + v_{rz}(2q_w q_y + 2q_x q_z)) \mathbf{i} + \\
&\quad (v_{rx}(2q_w q_z + 2q_x q_y) + v_{ry}(q_w q_w - q_x q_x + q_y q_y - q_z q_z) + v_{rz}(-2q_w q_x + 2q_y q_z)) \mathbf{j} + \\
&\quad (v_{rx}(-2q_w q_y + 2q_x q_z) + v_{ry}(2q_w q_x + 2q_y q_z) + v_{rz}(q_w q_w - q_x q_x - q_y q_y + q_z q_z)) \mathbf{k}
\end{aligned} \tag{4.30}$$

Next, the measured vector is subtracted from the rotated reference vector:

$$\begin{aligned}
\mathbf{q} \cdot \mathbf{v}_r \cdot \mathbf{q}^* - \mathbf{v}_m &= (v_{rx}(q_w q_w + q_x q_x - q_y q_y - q_z q_z) + v_{ry}(2q_x q_y - 2q_w q_z) \\
&\quad + v_{rz}(2q_w q_y + 2q_x q_z) - v_{mx}) \mathbf{i} + \\
&\quad (v_{rx}(2q_w q_z + 2q_x q_y) + v_{ry}(q_w q_w - q_x q_x + q_y q_y - q_z q_z) \\
&\quad + v_{rz}(-2q_w q_x + 2q_y q_z) - v_{my}) \mathbf{j} + \\
&\quad (v_{rx}(-2q_w q_y + 2q_x q_z) + v_{ry}(2q_w q_x + 2q_y q_z) \\
&\quad + v_{rz}(q_w q_w - q_x q_x - q_y q_y + q_z q_z) - v_{mz}) \mathbf{k}
\end{aligned} \tag{4.31}$$

The partial derivative of the difference with respect to each component of the estimated orientation quaternion is derived:

$$\begin{aligned}
\frac{\partial}{\partial q_w} (\mathbf{q} \cdot \mathbf{v}_r \cdot \mathbf{q}^* - \mathbf{v}_m) &= (2v_{rx}q_w - 2v_{ry}q_z + 2v_{rz}q_y) \mathbf{i} + \\
&\quad (2v_{rx}q_z + 2v_{ry}q_w - 2v_{rz}q_x) \mathbf{j} + \\
&\quad (-2v_{rx}q_y + 2v_{ry}q_x + 2v_{rz}q_w) \mathbf{k}
\end{aligned} \tag{4.32}$$

$$\begin{aligned}
\frac{\partial}{\partial q_x} (\mathbf{q} \cdot \mathbf{v}_r \cdot \mathbf{q}^* - \mathbf{v}_m) &= (2v_{rx}q_x + 2v_{ry}q_y + 2v_{rz}q_z) \mathbf{i} + \\
&\quad (2v_{rx}q_y - 2v_{ry}q_x - 2v_{rz}q_w) \mathbf{j} + \\
&\quad (2v_{rx}q_z + 2v_{ry}q_w - 2v_{rz}q_x) \mathbf{k}
\end{aligned} \tag{4.33}$$

$$\begin{aligned}
\frac{\partial}{\partial q_y} (\mathbf{q} \cdot \mathbf{v}_r \cdot \mathbf{q}^* - \mathbf{v}_m) &= (-2v_{rx}q_y + 2v_{ry}q_x + 2v_{rz}q_w) \mathbf{i} + \\
&\quad (2v_{rx}q_x + 2v_{ry}q_y + 2v_{rz}q_z) \mathbf{j} + \\
&\quad (-2v_{rx}q_w + 2v_{ry}q_z - 2v_{rz}q_y) \mathbf{k}
\end{aligned} \tag{4.34}$$

$$\begin{aligned} \frac{\partial}{\partial q_z} (\mathbf{q} \cdot \mathbf{v}_r \cdot \mathbf{q}^* - \mathbf{v}_m) = & (-2 v_{rx} q_z - 2 v_{ry} q_w + 2 v_{rz} q_x) \mathbf{i} + \\ & (2 v_{rx} q_w - 2 v_{ry} q_z + 2 v_{rz} q_y) \mathbf{j} + \\ & (2 v_{rx} q_x + 2 v_{ry} q_y + 2 v_{rz} q_z) \mathbf{k} \end{aligned} \quad (4.35)$$

The partial derivatives are combined to form the Jacobian:

$$\mathbf{J}_q^T (\mathbf{q} \cdot \mathbf{v}_r \cdot \mathbf{q}^* - \mathbf{v}_m) = \begin{bmatrix} 2 v_{rx} q_w - 2 v_{ry} q_z + 2 v_{rz} q_y & 2 v_{rx} q_z + 2 v_{ry} q_w - 2 v_{rz} q_x & -2 v_{rx} q_y + 2 v_{ry} q_x + 2 v_{rz} q_w \\ 2 v_{rx} q_x + 2 v_{ry} q_y + 2 v_{rz} q_z & 2 v_{rx} q_y - 2 v_{ry} q_x - 2 v_{rz} q_w & 2 v_{rx} q_z + 2 v_{ry} q_w - 2 v_{rz} q_x \\ -2 v_{rx} q_y + 2 v_{ry} q_x + 2 v_{rz} q_w & 2 v_{rx} q_x + 2 v_{ry} q_y + 2 v_{rz} q_z & -2 v_{rx} q_w + 2 v_{ry} q_z - 2 v_{rz} q_y \\ -2 v_{rx} q_z - 2 v_{ry} q_w + 2 v_{rz} q_x & 2 v_{rx} q_w - 2 v_{ry} q_z + 2 v_{rz} q_y & 2 v_{rx} q_x + 2 v_{ry} q_y + 2 v_{rz} q_z \end{bmatrix} \quad (4.36)$$

The gradient of the difference between a reference vector $\mathbf{v}_r$ rotated by the new estimated orientation quaternion $\mathbf{q}$ and a measurement of that vector $\mathbf{v}_m$ is calculated using equation (4.24): multiplying the Jacobian of the difference by the difference:

$$\begin{aligned} \nabla \mathbf{F}_q = \mathbf{J}_q^T (\mathbf{q} \cdot \mathbf{v}_r \cdot \mathbf{q}^* - \mathbf{v}_m) \cdot (\mathbf{q} \cdot \mathbf{v}_r \cdot \mathbf{q}^* - \mathbf{v}_m) = & \begin{bmatrix} 2 v_{rx} q_w - 2 v_{ry} q_z + 2 v_{rz} q_y & 2 v_{rx} q_z + 2 v_{ry} q_w - 2 v_{rz} q_x & -2 v_{rx} q_y + 2 v_{ry} q_x + 2 v_{rz} q_w \\ 2 v_{rx} q_x + 2 v_{ry} q_y + 2 v_{rz} q_z & 2 v_{rx} q_y - 2 v_{ry} q_x - 2 v_{rz} q_w & 2 v_{rx} q_z + 2 v_{ry} q_w - 2 v_{rz} q_x \\ -2 v_{rx} q_y + 2 v_{ry} q_x + 2 v_{rz} q_w & 2 v_{rx} q_x + 2 v_{ry} q_y + 2 v_{rz} q_z & -2 v_{rx} q_w + 2 v_{ry} q_z - 2 v_{rz} q_y \\ -2 v_{rx} q_z - 2 v_{ry} q_w + 2 v_{rz} q_x & 2 v_{rx} q_w - 2 v_{ry} q_z + 2 v_{rz} q_y & 2 v_{rx} q_x + 2 v_{ry} q_y + 2 v_{rz} q_z \end{bmatrix} \cdot \\ & \begin{bmatrix} v_{rx} (q_w q_w + q_x q_x - q_y q_y - q_z q_z) + v_{ry} (-2 q_w q_z + 2 q_x q_y) + v_{rz} (2 q_w q_y + 2 q_x q_z) - v_{mx} \\ v_{rx} (2 q_w q_z + 2 q_x q_y) + v_{ry} (q_w q_w - q_x q_x + q_y q_y - q_z q_z) + v_{rz} (-2 q_w q_x + 2 q_y q_z) - v_{my} \\ v_{rx} (-2 q_w q_y + 2 q_x q_z) + v_{ry} (2 q_w q_x + 2 q_y q_z) + v_{rz} (q_w q_w - q_x q_x - q_y q_y + q_z q_z) - v_{mz} \end{bmatrix} \end{aligned} \quad (4.37)$$

Similarly, when deriving the equations for the reverse rotations i.e. when the reference vector is in the external reference frame and is measured by the rotating body:

$$\begin{aligned} \mathbf{v}_r \cdot \mathbf{q} = & (0 q_w - v_{rx} q_x - v_{ry} q_y - v_{rz} q_z) + \\ & (0 q_x + v_{rx} q_w + v_{ry} q_z - v_{rz} q_y) \mathbf{i} + \\ & (0 q_y - v_{rx} q_z + v_{ry} q_w + v_{rz} q_x) \mathbf{j} + \\ & (0 q_z + v_{rx} q_y - v_{ry} q_x + v_{rz} q_w) \mathbf{k} \end{aligned} \quad (4.38)$$

$$\begin{aligned}
\mathbf{q}^* \cdot \mathbf{v}_r \cdot \mathbf{q} = & (q_w(-v_{rx}q_x - v_{ry}q_y - v_{rz}q_z) + q_x(v_{rx}q_w + v_{ry}q_z - v_{rz}q_y) \\
& + q_y(-v_{rx}q_z + v_{ry}q_w + v_{rz}q_x) + q_z(v_{rx}q_y - v_{ry}q_x + v_{rz}q_w)) + \\
& (q_w(v_{rx}q_w + v_{ry}q_z - v_{rz}q_y) - q_x(-v_{rx}q_x - v_{ry}q_y - v_{rz}q_z) \\
& - q_y(v_{rx}q_y - v_{ry}q_x + v_{rz}q_w) + q_z(-v_{rx}q_z + v_{ry}q_w + v_{rz}q_x)) \mathbf{i} + \\
& (q_w(-v_{rx}q_z + v_{ry}q_w + v_{rz}q_x) + q_x(v_{rx}q_y - v_{ry}q_x + v_{rz}q_w) \\
& - q_y(-v_{rx}q_x - v_{ry}q_y - v_{rz}q_z) - q_z(v_{rx}q_w + v_{ry}q_z - v_{rz}q_y)) \mathbf{j} + \\
& (q_w(v_{rx}q_y - v_{ry}q_x + v_{rz}q_w) - q_x(-v_{rx}q_z + v_{ry}q_w + v_{rz}q_x) \\
& + q_y(v_{rx}q_w + v_{ry}q_z - v_{rz}q_y) - q_z(-v_{rx}q_x - v_{ry}q_y - v_{rz}q_z)) \mathbf{k}
\end{aligned} \tag{4.39}$$

$$\begin{aligned}
= & (-v_{rx}q_wq_x - v_{ry}q_wq_y - v_{rz}q_wq_z + v_{rx}q_wq_x + v_{ry}q_xq_z - v_{rz}q_xq_y \\
& - v_{rx}q_yq_z + v_{ry}q_wq_y + v_{rz}q_xq_y + v_{rx}q_yq_z - v_{ry}q_xq_z + v_{rz}q_wq_z) + \\
& (v_{rx}q_wq_w + v_{ry}q_wq_z - v_{rz}q_wq_y + v_{rx}q_xq_x + v_{ry}q_xq_y + v_{rz}q_xq_z \\
& - v_{rx}q_yq_y + v_{ry}q_xq_y - v_{rz}q_wq_y - v_{rx}q_zq_z + v_{ry}q_wq_z + v_{rz}q_xq_z) \mathbf{i} + \\
& (-v_{rx}q_wq_z + v_{ry}q_wq_w + v_{rz}q_wq_x + v_{rx}q_xq_y - v_{ry}q_xq_x + v_{rz}q_wq_x \\
& + v_{rx}q_xq_y + v_{ry}q_yq_y + v_{rz}q_yq_z - v_{rx}q_wq_z - v_{ry}q_zq_z + v_{rz}q_yq_z) \mathbf{j} + \\
& (v_{rx}q_wq_y - v_{ry}q_wq_x + v_{rz}q_wq_w + v_{rx}q_xq_z - v_{ry}q_wq_x - v_{rz}q_xq_x \\
& + v_{rx}q_wq_y + v_{ry}q_yq_z - v_{rz}q_yq_y + v_{rx}q_xq_z + v_{ry}q_yq_z + v_{rz}q_zq_z) \mathbf{k}
\end{aligned} \tag{4.40}$$

$$\begin{aligned}
= & (v_{rx}(-q_wq_x + q_wq_x - q_yq_z + q_yq_z) + v_{ry}(-q_wq_y + q_xq_z + q_wq_y - q_xq_z) \\
& + v_{rz}(-q_wq_z - q_xq_y + q_xq_y + q_wq_z)) + \\
& (v_{rx}(q_wq_w + q_xq_x - q_yq_y - q_zq_z) + v_{ry}(q_wq_z + q_xq_y + q_xq_y + q_wq_z) \\
& + v_{rz}(-q_wq_y + q_xq_z - q_wq_y + q_xq_z)) \mathbf{i} + \\
& (v_{rx}(-q_wq_z + q_xq_y + q_xq_y - q_wq_z) + v_{ry}(q_wq_w - q_xq_x + q_yq_y - q_zq_z) \\
& + v_{rz}(q_wq_x + q_wq_x + q_yq_z + q_yq_z)) \mathbf{j} + \\
& (v_{rx}(q_wq_y + q_xq_z + q_wq_y + q_xq_z) + v_{ry}(-q_wq_x - q_wq_x + q_yq_z + q_yq_z) \\
& + v_{rz}(q_wq_w - q_xq_x - q_yq_y + q_zq_z)) \mathbf{k}
\end{aligned} \tag{4.41}$$

$$\begin{aligned}
= & 0 + \\
& (v_{rx}(q_wq_w + q_xq_x - q_yq_y - q_zq_z) + v_{ry}(2q_wq_z + 2q_xq_y) + v_{rz}(-2q_wq_y + 2q_xq_z)) \mathbf{i} + \\
& (v_{rx}(-2q_wq_z + 2q_xq_y) + v_{ry}(q_wq_w - q_xq_x + q_yq_y - q_zq_z) + v_{rz}(2q_wq_x + 2q_yq_z)) \mathbf{j} + \\
& (v_{rx}(2q_wq_y + 2q_xq_z) + v_{ry}(-2q_wq_x + 2q_yq_z) + v_{rz}(q_wq_w - q_xq_x - q_yq_y + q_zq_z)) \mathbf{k}
\end{aligned} \tag{4.42}$$

$$\begin{aligned}
\mathbf{q}^* \cdot \mathbf{v}_r \cdot \mathbf{q} - \mathbf{v}_m = & (v_{rx}(q_wq_w + q_xq_x - q_yq_y - q_zq_z) + v_{ry}(2q_wq_z + 2q_xq_y) \\
& + v_{rz}(-2q_wq_y + 2q_xq_z) - v_{mx}) \mathbf{i} + \\
& (v_{rx}(-2q_wq_z + 2q_xq_y) + v_{ry}(q_wq_w - q_xq_x + q_yq_y - q_zq_z) \\
& + v_{rz}(2q_wq_x + 2q_yq_z) - v_{my}) \mathbf{j} + \\
& (v_{rx}(2q_wq_y + 2q_xq_z) + v_{ry}(-2q_wq_x + 2q_yq_z) \\
& + v_{rz}(q_wq_w - q_xq_x - q_yq_y + q_zq_z) - v_{mz}) \mathbf{k}
\end{aligned} \tag{4.43}$$

$$\begin{aligned}\frac{\partial}{\partial q_w}(\mathbf{q}^* \cdot \mathbf{v}_r \cdot \mathbf{q} - \mathbf{v}_m) = & (2v_{rx}q_w + 2v_{ry}q_z - 2v_{rz}q_y)\mathbf{i} + \\ & (-2v_{rx}q_z + 2v_{ry}q_w + 2v_{rz}q_x)\mathbf{j} + \\ & (2v_{rx}q_y - 2v_{ry}q_x + 2v_{rz}q_w)\mathbf{k}\end{aligned}\quad (4.44)$$

$$\begin{aligned}\frac{\partial}{\partial q_x}(\mathbf{q}^* \cdot \mathbf{v}_r \cdot \mathbf{q} - \mathbf{v}_m) = & (2v_{rx}q_x + 2v_{ry}q_y + 2v_{rz}q_z)\mathbf{i} + \\ & (2v_{rx}q_y - 2v_{ry}q_x + 2v_{rz}q_w)\mathbf{j} + \\ & (2v_{rx}q_z - 2v_{ry}q_w - 2v_{rz}q_x)\mathbf{k}\end{aligned}\quad (4.45)$$

$$\begin{aligned}\frac{\partial}{\partial q_y}(\mathbf{q}^* \cdot \mathbf{v}_r \cdot \mathbf{q} - \mathbf{v}_m) = & (-2v_{rx}q_y + 2v_{ry}q_x - 2v_{rz}q_w)\mathbf{i} + \\ & (2v_{rx}q_x + 2v_{ry}q_y + 2v_{rz}q_z)\mathbf{j} + \\ & (2v_{rx}q_w + 2v_{ry}q_z - 2v_{rz}q_y)\mathbf{k}\end{aligned}\quad (4.46)$$

$$\begin{aligned}\frac{\partial}{\partial q_z}(\mathbf{q}^* \cdot \mathbf{v}_r \cdot \mathbf{q} - \mathbf{v}_m) = & (-2v_{rx}q_z + 2v_{ry}q_w + 2v_{rz}q_x)\mathbf{i} + \\ & (-2v_{rx}q_w - 2v_{ry}q_z + 2v_{rz}q_y)\mathbf{j} + \\ & (2v_{rx}q_x + 2v_{ry}q_y + 2v_{rz}q_z)\mathbf{k}\end{aligned}\quad (4.47)$$

$$\begin{aligned}\mathbf{J}_q^T(\mathbf{q}^* \cdot \mathbf{v}_r \cdot \mathbf{q} - \mathbf{v}_m) = & \begin{bmatrix} 2v_{rx}q_w + 2v_{ry}q_z - 2v_{rz}q_y & -2v_{rx}q_z + 2v_{ry}q_w + 2v_{rz}q_x & 2v_{rx}q_y - 2v_{ry}q_x + 2v_{rz}q_w \\ 2v_{rx}q_x + 2v_{ry}q_y + 2v_{rz}q_z & 2v_{rx}q_y - 2v_{ry}q_x + 2v_{rz}q_w & 2v_{rx}q_z - 2v_{ry}q_w - 2v_{rz}q_x \\ -2v_{rx}q_y + 2v_{ry}q_x - 2v_{rz}q_w & 2v_{rx}q_x + 2v_{ry}q_y + 2v_{rz}q_z & 2v_{rx}q_w + 2v_{ry}q_z - 2v_{rz}q_y \\ -2v_{rx}q_z + 2v_{ry}q_w + 2v_{rz}q_x & -2v_{rx}q_w - 2v_{ry}q_z + 2v_{rz}q_y & 2v_{rx}q_x + 2v_{ry}q_y + 2v_{rz}q_z \end{bmatrix}\end{aligned}\quad (4.48)$$

$$\begin{aligned}\nabla F_q = \mathbf{J}_q^T(\mathbf{q}^* \cdot \mathbf{v}_r \cdot \mathbf{q} - \mathbf{v}_m) \cdot (\mathbf{q}^* \cdot \mathbf{v}_r \cdot \mathbf{q} - \mathbf{v}_m) = & \begin{bmatrix} 2v_{rx}q_w + 2v_{ry}q_z - 2v_{rz}q_y & -2v_{rx}q_z + 2v_{ry}q_w + 2v_{rz}q_x & 2v_{rx}q_y - 2v_{ry}q_x + 2v_{rz}q_w \\ 2v_{rx}q_x + 2v_{ry}q_y + 2v_{rz}q_z & 2v_{rx}q_y - 2v_{ry}q_x + 2v_{rz}q_w & 2v_{rx}q_z - 2v_{ry}q_w - 2v_{rz}q_x \\ -2v_{rx}q_y + 2v_{ry}q_x - 2v_{rz}q_w & 2v_{rx}q_x + 2v_{ry}q_y + 2v_{rz}q_z & 2v_{rx}q_w + 2v_{ry}q_z - 2v_{rz}q_y \\ -2v_{rx}q_z + 2v_{ry}q_w + 2v_{rz}q_x & -2v_{rx}q_w - 2v_{ry}q_z + 2v_{rz}q_y & 2v_{rx}q_x + 2v_{ry}q_y + 2v_{rz}q_z \end{bmatrix} \cdot \\ & \begin{bmatrix} v_{rx}(q_wq_w + q_xq_x - q_yq_y - q_zq_z) + v_{ry}(2q_wq_z + 2q_xq_y) + v_{rz}(-2q_wq_y + 2q_xq_z) - v_{mx} \\ v_{rx}(-2q_wq_z + 2q_xq_y) + v_{ry}(q_wq_w - q_xq_x + q_yq_y - q_zq_z) + v_{rz}(2q_wq_x + 2q_yq_z) - v_{my} \\ v_{rx}(2q_wq_y + 2q_xq_z) + v_{ry}(-2q_wq_x + 2q_yq_z) + v_{rz}(q_wq_w - q_xq_x - q_yq_y + q_zq_z) - v_{mz} \end{bmatrix}\end{aligned}\quad (4.49)$$

When the reference vector is gravity and assumed only to operate in the negative vertical direction, the above gradient equation is simplified by setting the reference vector $\mathbf{v}_r = [0, 0, -1]$:

$$\nabla \mathbf{F}_q = \mathbf{J}_q^T (\mathbf{q}^* \cdot \mathbf{v}_r \cdot \mathbf{q} - \mathbf{v}_m) \cdot (\mathbf{q}^* \cdot \mathbf{v}_r \cdot \mathbf{q} - \mathbf{v}_m) =$$

$$\begin{bmatrix} 2q_y & -2q_x & -2q_w \\ -2q_z & -2q_w & 2q_x \\ 2q_w & -2q_z & 2q_y \\ -2q_x & -2q_y & -2q_z \end{bmatrix} \cdot \begin{bmatrix} 2q_w q_y - 2q_x q_z - v_{mx} \\ -2q_w q_x - 2q_y q_z - v_{my} \\ -q_w q_w + q_x q_x + q_y q_y - q_z q_z - v_{mz} \end{bmatrix} \quad (4.50)$$

When the reference vector is the earth's magnetic field and only its declination is considered, the gradient equation is simplified by setting $\mathbf{v}_r = [v_{rx}, 0, v_{rz}]$:

$$\nabla \mathbf{F}_q = \mathbf{J}_q^T (\mathbf{q}^* \cdot \mathbf{v}_r \cdot \mathbf{q} - \mathbf{v}_m) \cdot (\mathbf{q}^* \cdot \mathbf{v}_r \cdot \mathbf{q} - \mathbf{v}_m) =$$

$$\begin{bmatrix} 2v_{rx} q_w - 2v_{rz} q_y & -2v_{rx} q_z + 2v_{rz} q_x & 2v_{rx} q_y + 2v_{rz} q_w \\ 2v_{rx} q_x + 2v_{rz} q_z & 2v_{rx} q_y + 2v_{rz} q_w & 2v_{rx} q_z - 2v_{rz} q_x \\ -2v_{rx} q_y - 2v_{rz} q_w & 2v_{rx} q_x + 2v_{rz} q_z & 2v_{rx} q_w - 2v_{rz} q_y \\ -2v_{rx} q_z + 2v_{rz} q_x & -2v_{rx} q_w + 2v_{rz} q_y & 2v_{rx} q_x + 2v_{rz} q_z \end{bmatrix} \cdot$$

$$\begin{bmatrix} v_{rx}(q_w q_w + q_x q_x - q_y q_y - q_z q_z) + v_{rz}(-2q_w q_y + 2q_x q_z) - v_{mx} \\ v_{rx}(-2q_w q_z + 2q_x q_y) + v_{rz}(2q_w q_x + 2q_y q_z) - v_{my} \\ v_{rx}(2q_w q_y + 2q_x q_z) + v_{rz}(q_w q_w - q_x q_x - q_y q_y + q_z q_z) - v_{mz} \end{bmatrix} \quad (4.51)$$

Madgwick et al. use equation (4.25) to constrain the calculated magnetometer's reference vector.

This not only constrains the vector to the x-z plane, but further constrains it to the positive half of the plane. This not only increases the error of the offset from the actual inclination, it also makes the initial conditions ambiguous.

To ensure that the constraint provided does not also constrain the calculated reference vector to the right half of the plane, the sign of the x-component of the constrained, calculated reference vector is made equal to the sign of the unconstrained calculated reference vector using:

$$\mathbf{v}'_r = \begin{cases} [\sqrt{v_{xr}^2 + v_{ry}^2}, 0, v_{rz}] & \text{if } v_{xr} \geq 0 \\ [-\sqrt{v_{xr}^2 + v_{ry}^2}, 0, v_{rz}] & v_{xr} < 0 \end{cases} \quad (4.52)$$

The speed of the algorithm will limit the minimum time delta in the finite sums algorithm (4.16).

Ultimately the speed of the algorithm is dependent on the processor speed, but within the constraints of a particular processor, for example the processor onboard an embedded device, the speed depends on the total number of clock cycles required.

To minimise the time required to normalise the vectors and quaternions, the so called 0x5f3759df fast inverse square root algorithm²⁶ is used, which avoids both the need to find the square root and dividing. The fast inverse square root uses the relationship:

$$\log_b\left(\frac{1}{\sqrt{x}}\right) = -\frac{1}{2}\log_b(x) \quad (4.53)$$

and that IEEE 754 single-precision binary floating-point formats approximate $\log_2(x)$ when aliased to an integer²⁷.

Depending on the processor, using shifted 32-bit integers to represent real numbers instead of 32-bit floating-point numbers can significantly reduce the total number of clock cycles required²⁸. On the PIC24FJ running at 30 MHz, the floating-point version of the algorithm cannot be run faster than 320 Hz. However, a fixed-point, shifted-integer scaled by 2^{14} version of the algorithm can be run at over 750 Hz.

The 0x5f3759df fast inverse square root algorithm cannot be used when using the shifted-integer approximation. Therefore, a bit-wise approximation of the square root²⁹ is used, which eliminates the memory requirements of a maths library for calculating the square root, but not the requirement for performing a division.

26 The algorithm is called the 0x5f3759df fast inverse square root algorithm, because it uses that hexadecimal number but the origin of this number is unknown (Blinn, 1997).

27 (Blinn, 1997)

28 (Padgett & Anderson, 2009)

29 (Blinn, 1997)

In summary, the improved formulation of the AHRS orientation quaternion algorithm uses the Jacobian and the difference equations (4.49), (4.50) and (4.51). The non-linear equations are defined by the difference between reference vectors and reversely rotated measured vectors (4.18). It uses a single step of the gradient descent algorithm (4.21) with an α of 0.1, to simultaneously solve the non-linear equations (4.20). The vertically constrained gravity reference vector is used with this research's updated constraint on the calculated magnetic north reference vector (4.52) for iteratively updating the rotation unit quaternions. The floating-point version of the algorithm uses the 0x5f3759df fast inverse square root algorithm and the, and the fixed-point, shifted-integer version uses the bit-wise approximation of the square root.

4.5 Algorithm Simulation Environment

To test and compare the various algorithms, the simulation environment shown in Figure 4.1 was created.

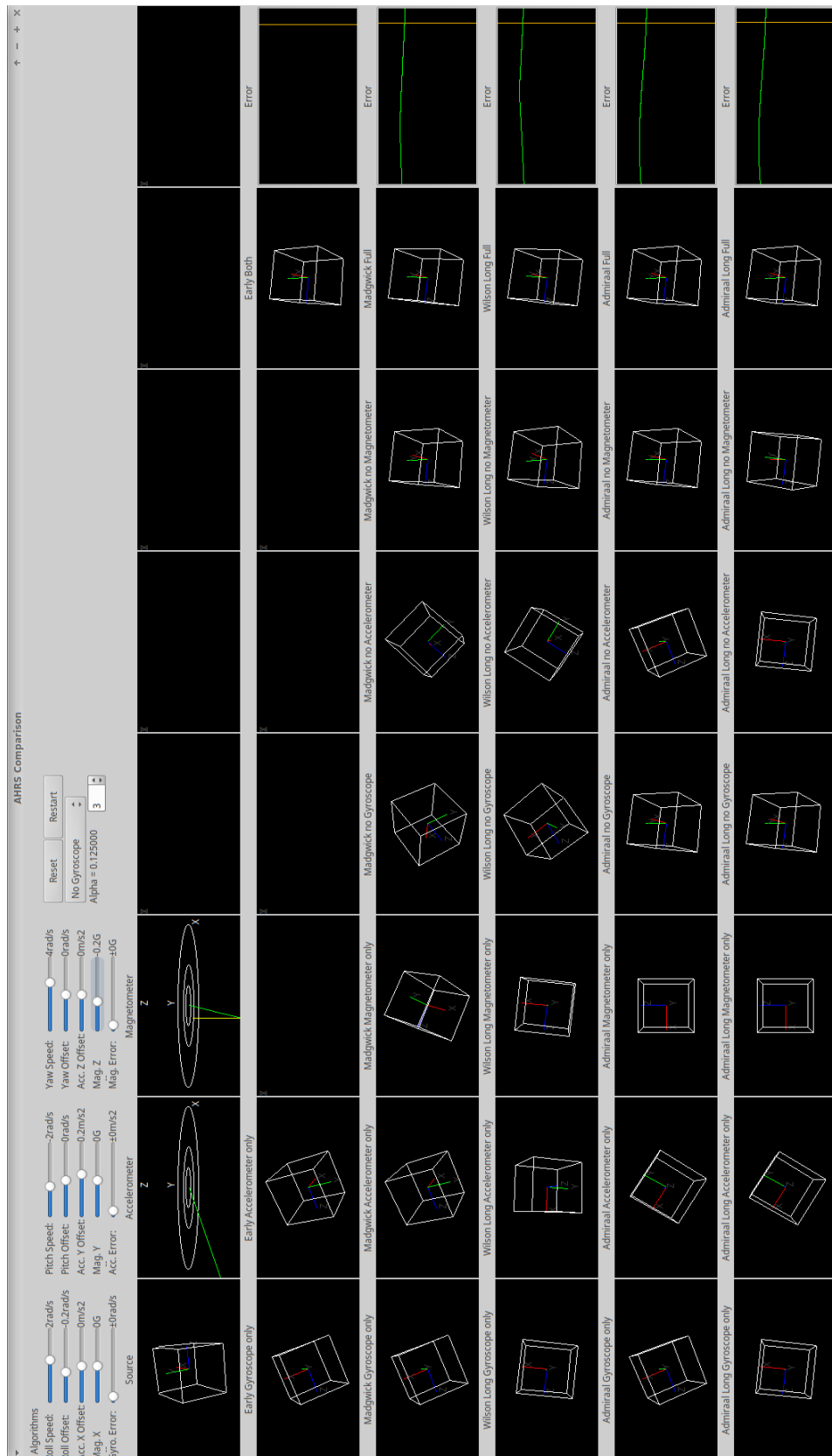


Figure 4.1: Simultaneously comparing the orientation quaternion calculated by multiple algorithms using various combinations of sensors.

The control panel is enlarged in Figure 4.2. The first row of the control panel enables changing the speed of rotation. The speed of rotation can be varied between ± 10 radians per second, and each axis is controlled independently. These rotations are used to create the source orientation, which is displayed in real-time in the left-hand window of Figure 4.2. Based on the source orientation, the vectors that an accelerometer and a magnetometer would yield are displayed in the next two windows.

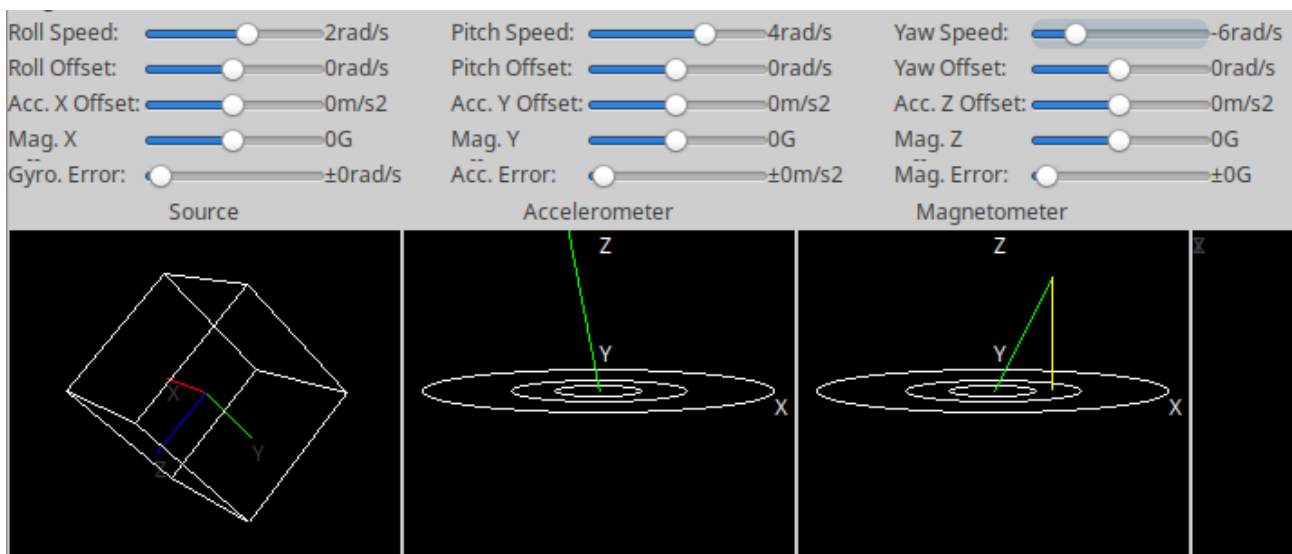


Figure 4.2: Gyroscope, accelerometer and magnetometer device simulation environment.

The next three rows of the control panel enable changing the fixed offset error of each of the sensors in each direction that will be sent to the algorithms. The final row enables changing the maximum size of the random error that is added to each component of each sensor.

The algorithms to be compared can be added online and are shown in separate rows. Figure 4.1 shows the output created when comparing five different algorithms. The algorithms being compared are the early version of the Madgwick AHRS algorithm that did not use the magnetometer, the Madgwick AHRS algorithm, the original fixed-point, shifted-integer version of the Madgwick

AHRS algorithm, the reformulated AHRS orientation quaternion algorithm and the fixed-point, shifted-integer version of the reformulated AHRS orientation quaternion algorithm.

To visualise the impact of each sensor on each selected AHRS orientation quaternion algorithm, and to assist with understanding how the AHRS orientation quaternion algorithm works, the simulator also displays the orientation calculated using any combination of sensors. The first seven columns of each row display the orientation calculated by that row's AHRS orientation quaternion algorithm using:

- Just the gyroscope;
- Just the accelerometer;
- Just the magnetometer;
- The accelerometer and magnetometer, but no gyroscope;
- The gyroscope and magnetometer, but no accelerometer;
- The gyroscope and accelerometer, but no magnetometer; and
- All the sensors;

The final column shows the error between the simulated source orientation and the orientation calculated using a selected combination of sensors.

The output of AHRS orientation quaternion algorithms are controlled through the extension to the control panel shown in Figure 4.3. The Reset button resets all the simulation controls to their initial values. Whereas, the Restart button sets all the simulated and AHRS orientation quaternion algorithms' calculated orientations to zero. The Compare button begins recording the output data used to compare the algorithms.

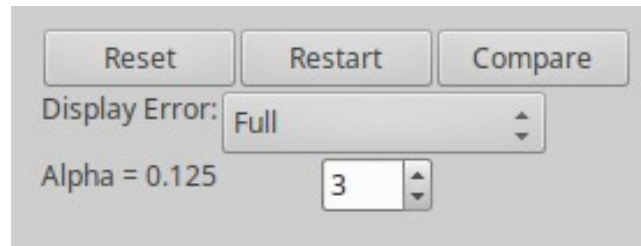


Figure 4.3: AHRS orientation quaternion algorithm controls.

The Display Error drop-down list enables selecting which combination of sensors the error between the simulated source orientation and the orientation calculated column should display.

Finally, the α value used by the gradient descent algorithm (4.21) can be changed. To enable comparisons between the floating-point and fixed-point, shifted-integer versions of the algorithms, the size of alpha is specified using the number of bits the fixed-point, shifted-integer algorithm shifts the calculation by during the fast division. The equivalent floating-point division value is calculated and displayed.

In addition to manually varying the each of the above parameters, the exact changes required were scripted. This enabled taking multiple samples of different sets of parameters for making consistent comparisons between the various algorithms. It also provided the ability to precisely control the duration of each sample. With multiple samples using the same parameters for the same amount of time, statistical comparisons were made.

4.6 Algorithm Comparison

As discussed above, the maximum algorithm iteration frequency possible will depend on both the algorithm and the embedded processor used. However, to maximise the accuracy of the algorithms, and to compare them on their merits and not the processor used, the period of the iterations of the

algorithm was set to 1 ms, i.e. run at 1,000 Hz, and the simulations were run on a desktop with a processor that could easily manage all the calculations in real-time.

For each set of parameters, the algorithm is run for 1,000 iterations, i.e. one second in real-time. Each time the parameters are changed the simulation is restarted, i.e. both the simulated and AHRS orientation quaternion algorithms' calculated orientations are set to $1 + 0\mathbf{i} + 0\mathbf{j} + 0\mathbf{k}$. The average and standard deviation of ten simulation samples is used for each data point plotted.

A direct comparison was made between the Madgwick AHRS algorithm and the improved formulation of the AHRS orientation quaternion algorithm, hereafter called the Improved AHRS algorithm. Both algorithms were configured to use an α value of 0.25 in the gradient descent algorithm (4.21)³⁰. The speed of rotation along each three-dimensional axis was changed independently. The speed of rotation was changed on each axis independently to highlight any potential differences generated by the different approaches used to obtain the reference vectors used in the corrective step. The speed of rotation on an axis was increased from 0 to 10 radians per second in increments of 0.25 radians per second.

Figures 4.4, 4.5 and 4.6 plot the results of the comparison between the errors in the Madgwick AHRS algorithm and the Improved AHRS algorithm. Figure 4.4 plots the errors for rotations around the x-axis. Figure 4.5 plots the errors around the y-axis, and Figure 4.6 around the z-axis. To visualise the errors in the Improved AHRS algorithm, Figures 4.7, 4.8 and 4.9 plot the errors in the Improved AHRS algorithm at a scale five-orders of magnitude greater. Figure 4.7 plots the errors around the x-axis, Figure 4.8 around the y-axis and Figure 4.9 around the z-axis.

³⁰ The selection of an α value of 0.25 is based on the analysis below, where a value of 0.25 is shown to be one of the best values to use.

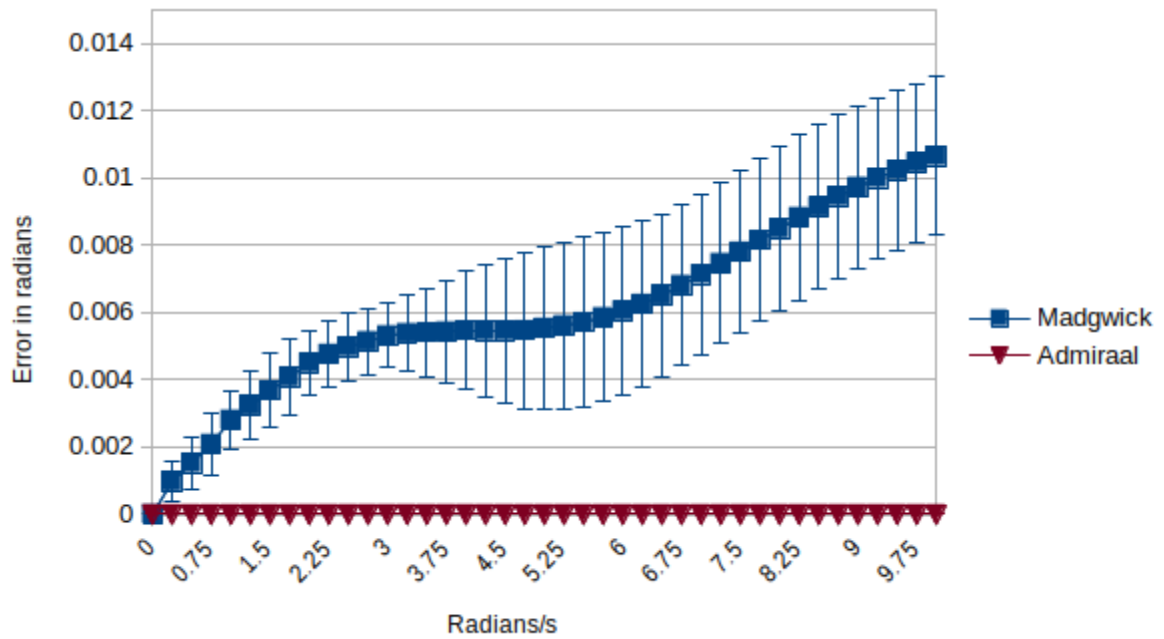


Figure 4.4: Comparison between errors in the Madgwick AHRS algorithm and the Improved AHRS algorithm after 1,000 iterations under different rotational speeds around the x-axis

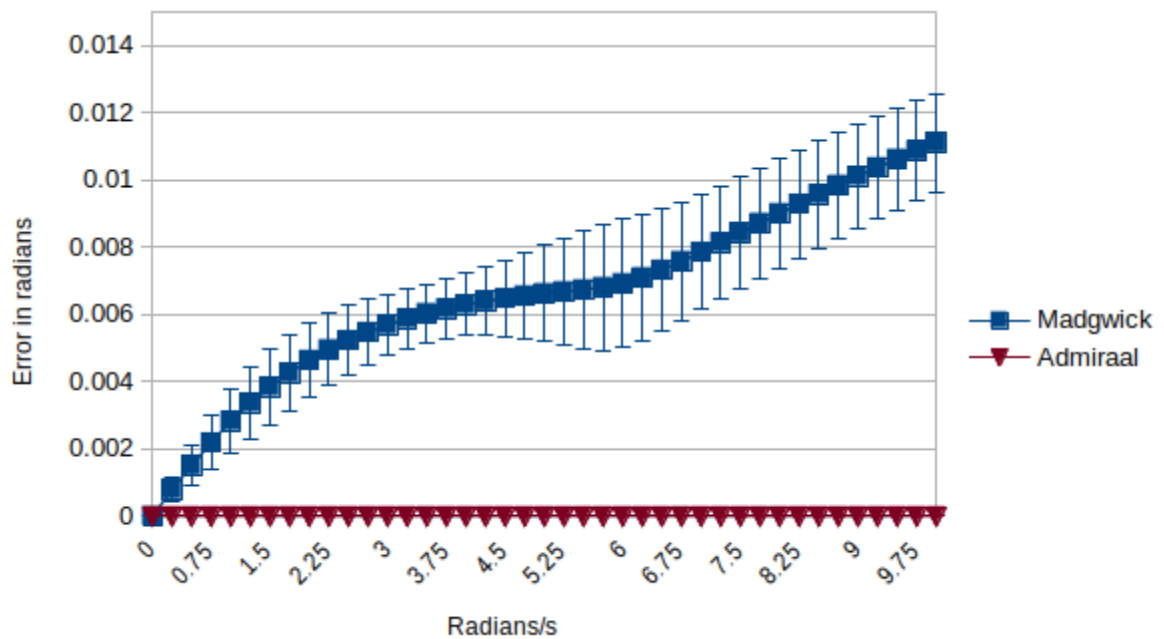


Figure 4.5: Comparison between errors in the Madgwick AHRS algorithm and the Improved AHRS algorithm after 1,000 iterations under different rotational speeds around the y-axis

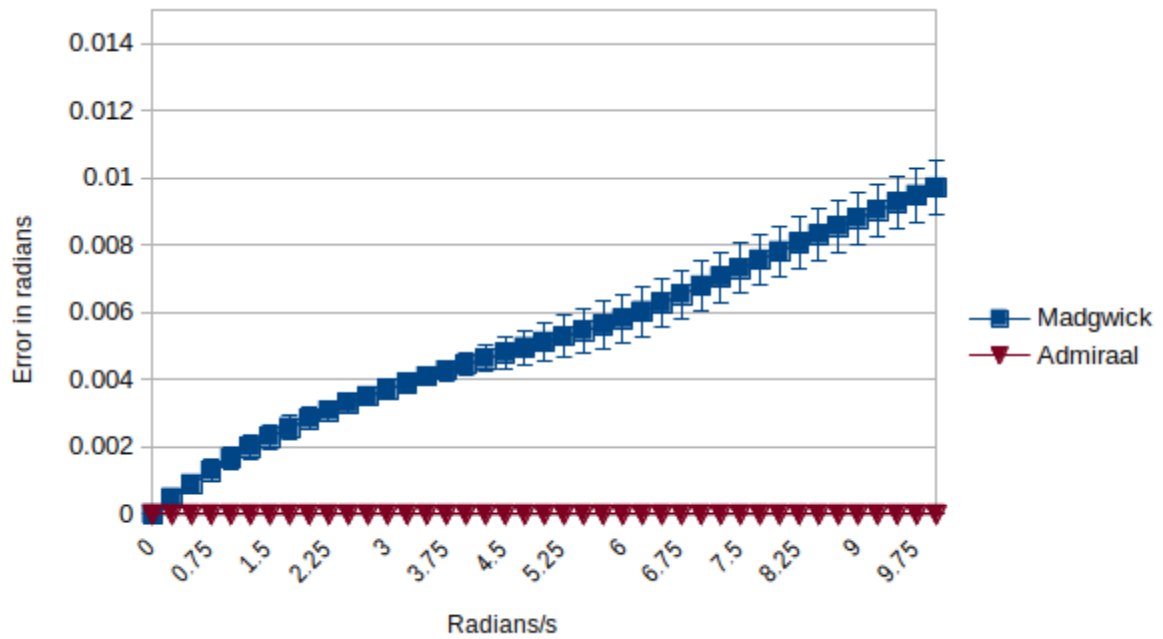


Figure 4.6: Comparison between errors in the Madgwick AHRS algorithm and the Improved AHRS algorithm after 1,000 iterations under different rotational speeds around the z-axis

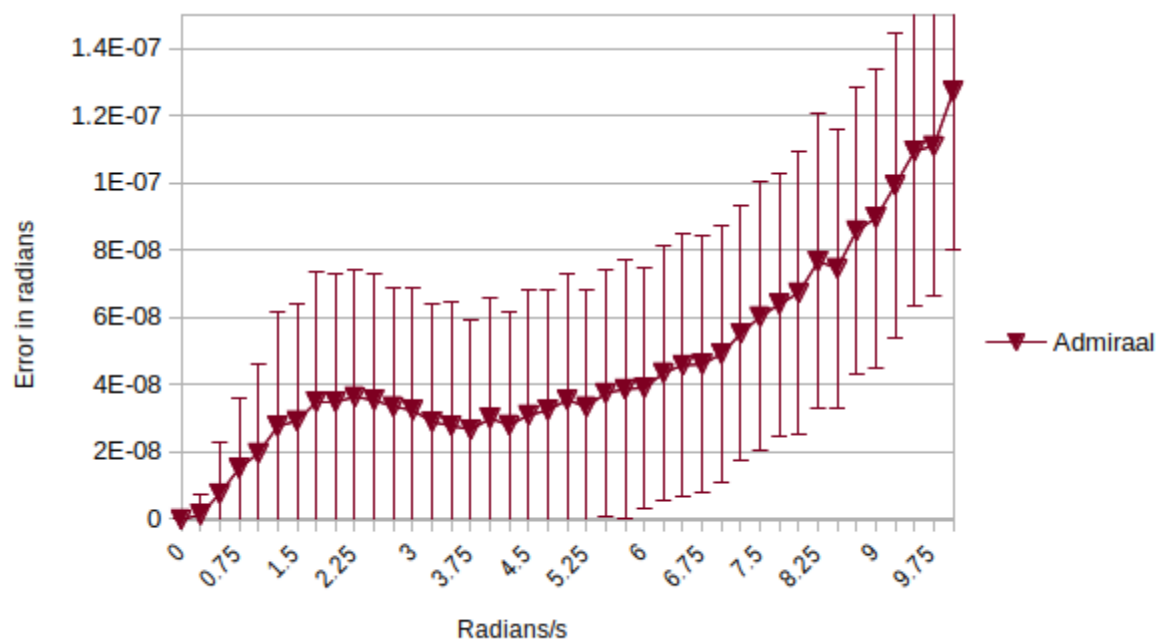


Figure 4.7: Errors in the Improved AHRS algorithm after 1,000 iterations under different rotational speeds around the x-axis at five-orders of magnitude greater resolution

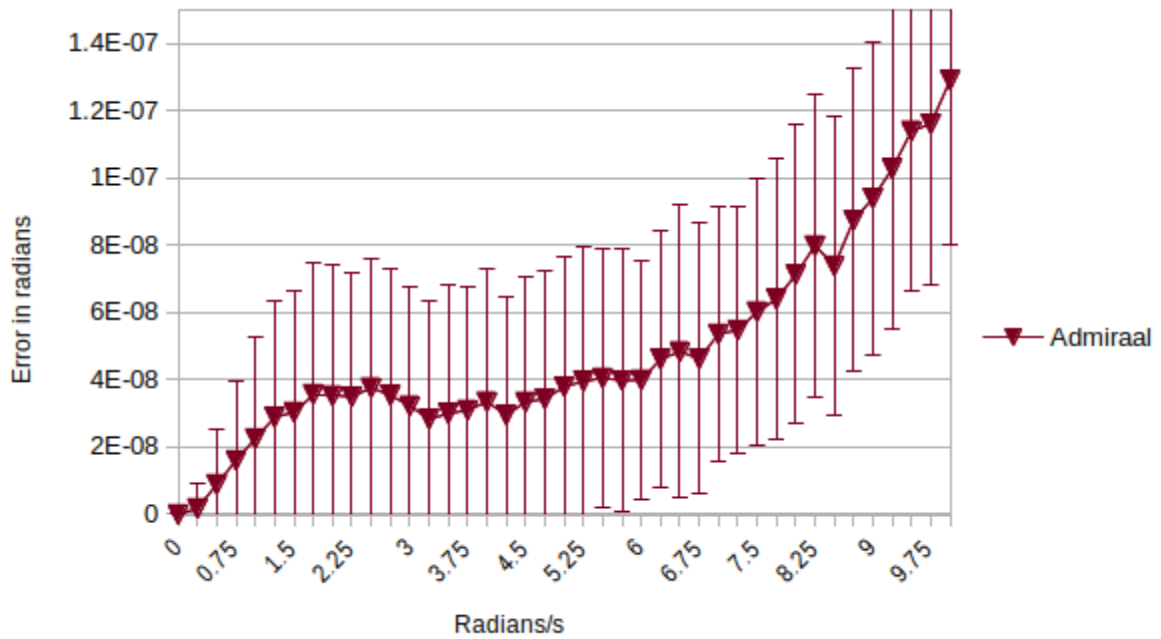


Figure 4.8: Errors in the Improved AHRS algorithm after 1,000 iterations under different rotational speeds around the y-axis at five-orders of magnitude greater resolution

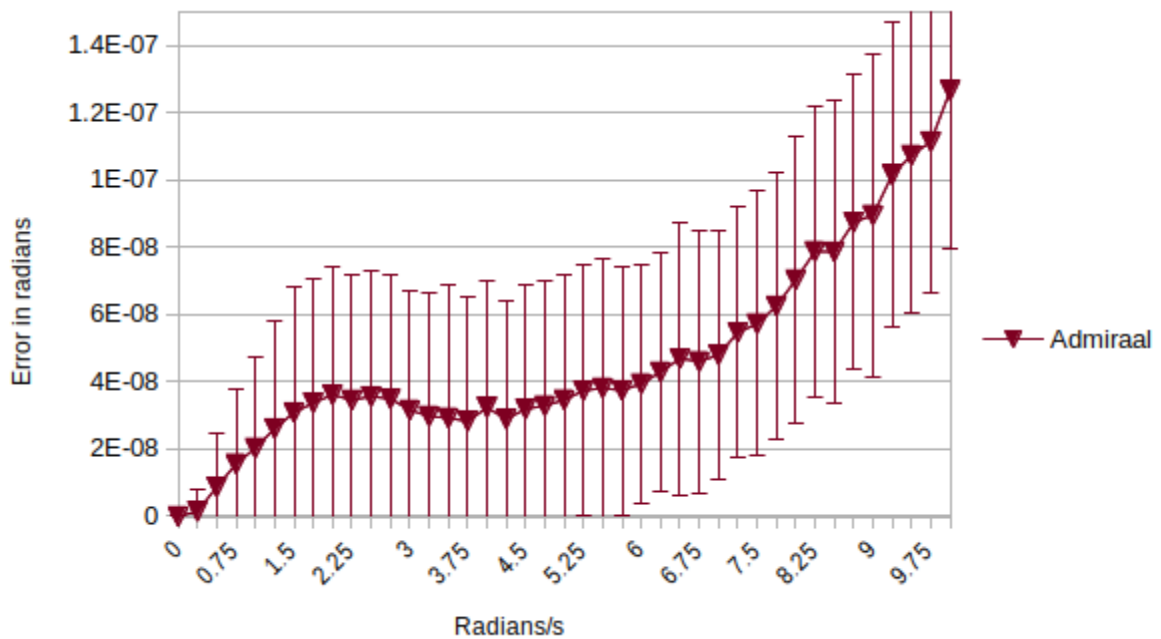


Figure 4.9: Errors in the Improved AHRS algorithm after 1,000 iterations under different rotational speeds around the z-axis at five-orders of magnitude greater resolution

Irrespective of the axis of rotation, after 1 second at 10 radians per second, the Madgwick AHRS algorithm shows an error of around 0.01 radians between the correct simulated orientation and the AHRS calculated orientation. After the same time, the Improved AHRS algorithm has an error of around 0.0000001 or 10^{-7} radians.

The Madgwick AHRS algorithm has been shown to correct for errors in the iterative estimated orientation quaternion calculation of equation (4.16). These results show that the Improved AHRS algorithm provides a five orders of magnitude improvement on the estimation of the orientation quaternion in dynamic scenarios. Similarly, these results are also five orders of magnitude better than alternative approaches that have been used to calculate orientation quaternions such as the linear Kalman filter, which have been shown to be both marginally worse (Madgwick, Harrison & Vaidyanathan, 2011) and better (Valenti, Dryanovski & Xiao, 2016) than the Madgwick AHRS algorithm.

The precision of the algorithm is dependent on the size of α in the gradient descent algorithm (4.21). The values of α used were: 0.5, 0.25, 0.12, 0.062, 0.031, 0.016³¹. Again, the speed of rotation along each three-dimensional axis was changed independently from 0 to 10 radians per second in increments of 0.25 radians per second.

³¹ As discussed above, to enable comparisons with the fixed-point versions of the algorithms, the values of α available in the simulation are limited to values equal to $\frac{1}{2^n}$ with n a positive integer.

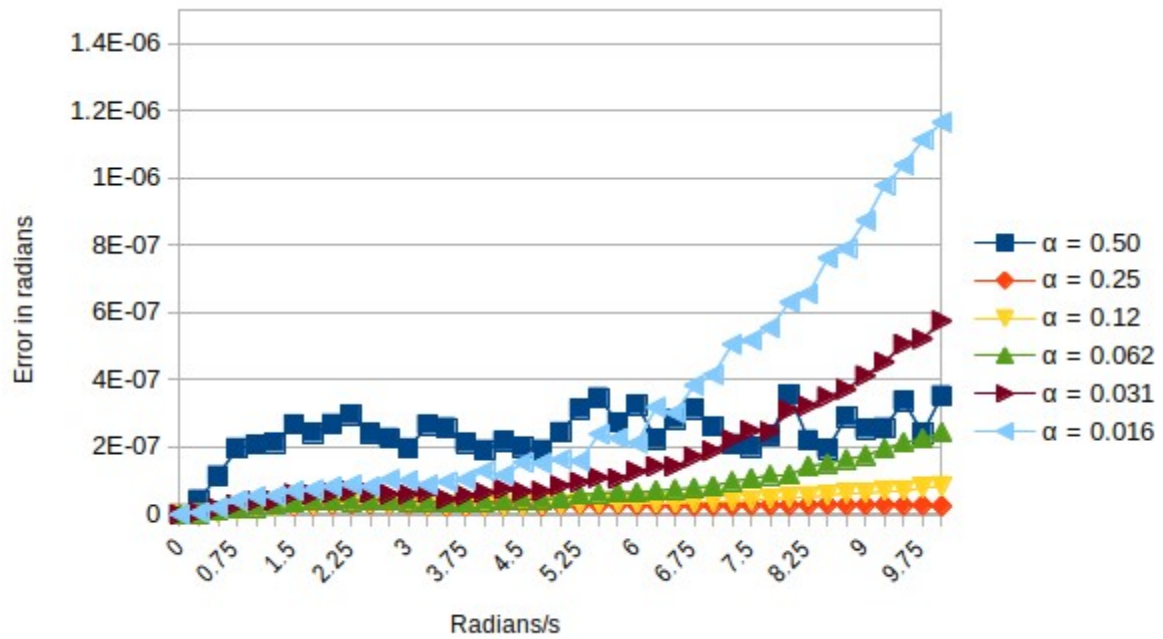


Figure 4.10: Impact of the step size of the gradient descent algorithm (α) on the precision of the Improved AHRS algorithm under different rotational speeds around the x-axis

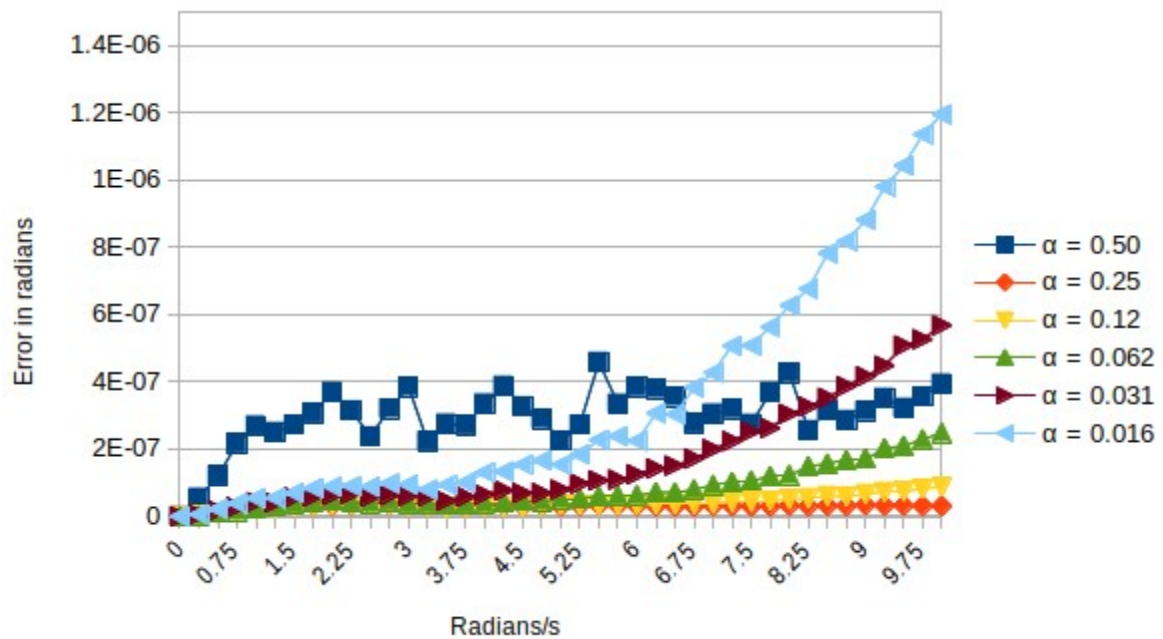


Figure 4.11: Impact of the step size of the gradient descent algorithm (α) on the precision of the Improved AHRS algorithm under different rotational speeds around the y-axis

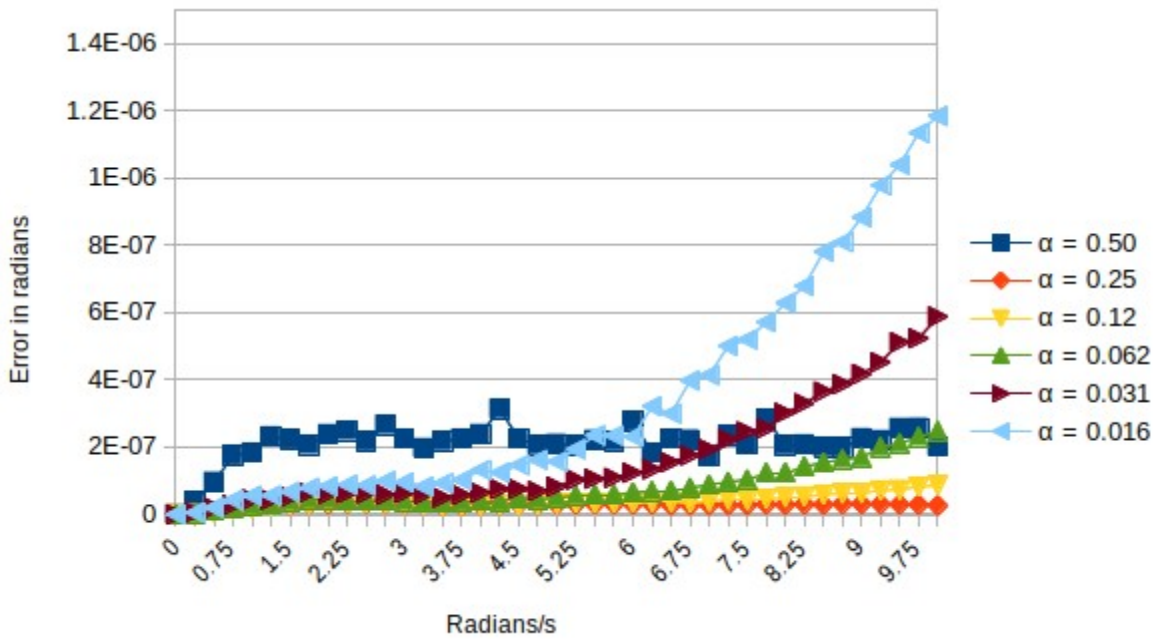


Figure 4.12: Impact of the step size of the gradient descent algorithm (α) on the precision of the Improved AHRS algorithm under different rotational speeds around the z-axis

Figures 4.10, 4.11 and 4.12 show the impact of varying α on the precision of the Improved AHRS algorithm at different rotational speeds. The precision of the fixed-point, shifted-integer version of the algorithm with a gradient descent step size α value of 0.50 is an order of magnitude worse, therefore it is not shown. Furthermore, for clarity in the presentation of the results, the standard deviations are not shown. Figure 4.10 shows the impact of varying α on the precision of the Improved AHRS algorithm at increasing rotational speeds around the x-axis. Figure 4.11 shows the impact at increasing rotational speeds around the y-axis, and Figure 4.12 shows the impact around the z-axis.

The best step size α for the gradient descent algorithm is 0.25. It's worth noting that the precision of a 32-bit floating-point number is limited by the 23 bits³² allocated to the fraction component of the IEEE 754 single-precision binary floating-point format, which equates to 7.2 decimal digits³³.

³² The actual precision of the fraction component is 24 bits, because the most significant bit can always be configured to 0 by increasing the exponent by 1.

³³ $\log_{10}(2^{24}) = 7.2$

Therefore, most of the errors in the Improved AHRS algorithm with a step size of 0.25 can be attributed to the limits of the number representation.

To improve the speed of each iteration on an embedded processor to enable increasing the sensor sample frequency, a fixed-point, shifted-integer version of the algorithm was created. Therefore, the next comparison made was the precision of the fixed-point, shifted-integer version of the algorithm with varying step sizes α in the gradient descent.

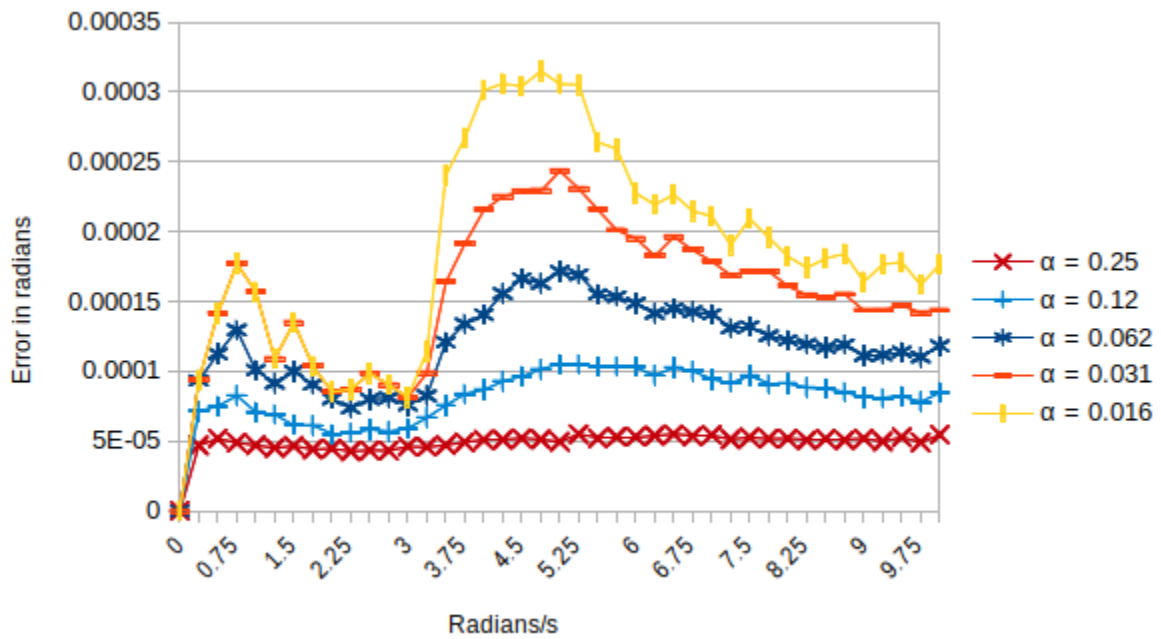


Figure 4.13: Impact of varying α on the precision of the fixed-point, shifter-integer version of the Improved AHRS algorithm under different rotational speeds around the x-axis

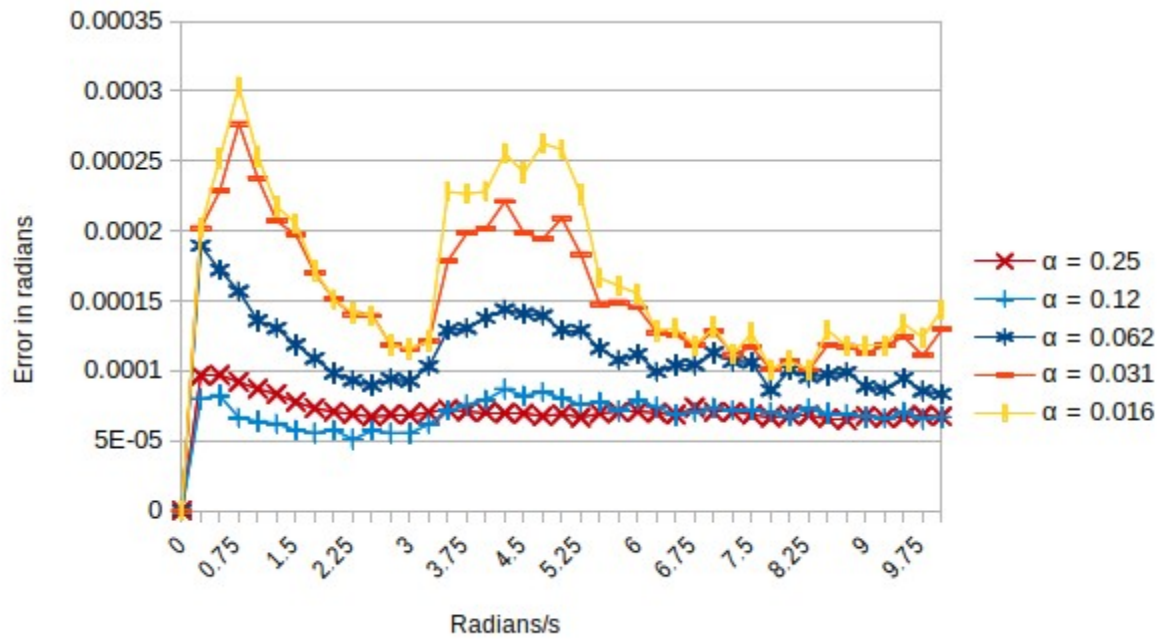


Figure 4.14: Impact of varying α on the precision of the fixed-point, shifter-integer version of the Improved AHRS algorithm under different rotational speeds around the y-axis

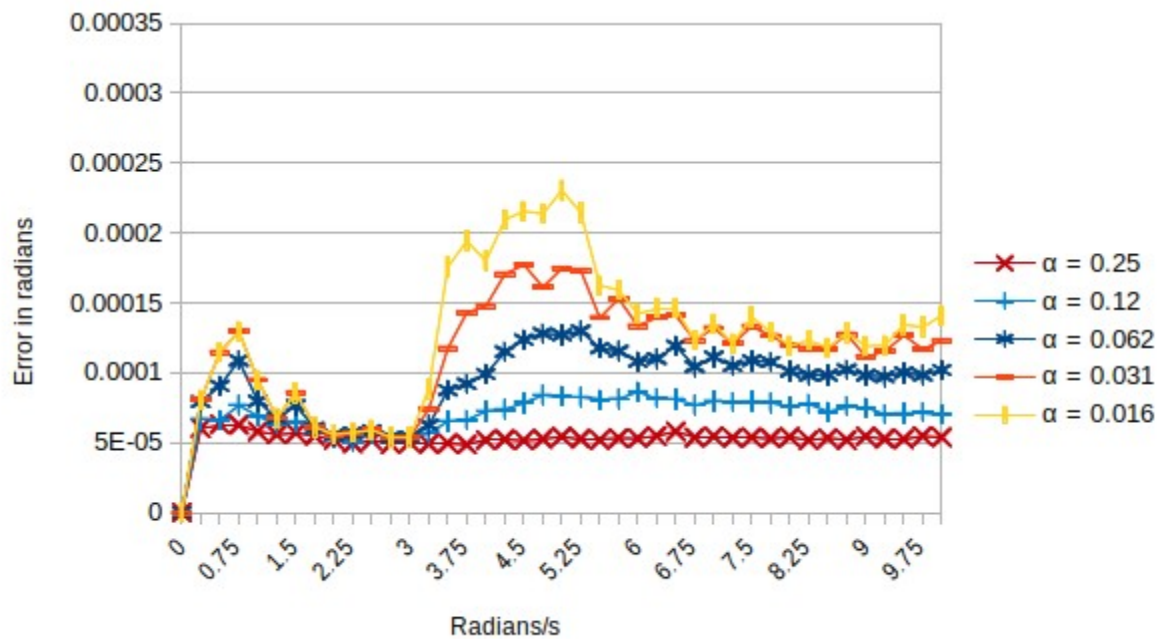


Figure 4.15: Impact of varying α on the precision of the fixed-point, shifter-integer version of the Improved AHRS algorithm under different rotational speeds around the z-axis

Figure 4.13 shows the impact of varying the step size of the gradient descent algorithm (α) on the precision of the fixed-point, shifted-integer version of the Improved AHRS algorithm at increasing rotational speeds around the x-axis. Figure 4.14 shows the impact at increasing rotational speeds around the y-axis, and Figure 4.15 shows the impact around the z-axis.

The results once again show that an α of 0.25 provides the best precision across a range of rotational speeds for the fixed-point, shifted-integer version of the Improved AHRS algorithm.

As with all gradient descent algorithms, if the value of α is too small, the corrective step will not be sufficient to quickly correct for the increasing accumulative error. If the value of α is too large the corrective step will actually increase the error. Therefore, the intermediate value of 0.25 provides the best results.

The precision available when working with fixed-point, shifted-integers scaled by 2^{14} is only 4.2 decimal digits³⁴ or 10^{-4} , which again accounts for most of the error in the precision of the floating-point and fixed-point, shifted-integer version of the Improved AHRS algorithm. It also explains the odd shape of the curves. At low rotational speeds the error is higher than at higher rotational speeds. The error is a rounding error. Below a threshold rotational speed the algorithm is actually calculating no rotation from the gyroscopes. Only when the accelerometer and magnetometer readings are sufficiently far from the reference vectors does the corrective step change the calculated orientation.

All the above comparisons are made using data from precise measurements obtained from simulated sensors. However, the precision of real sensors is limited. To test the sensitivity of the algorithms, a random error was introduced into the simulated measurements. The error introduced was taken from a uniform distribution in the range of the precision of the LSM9DS0 MARG. The LSM9DS0 is the MARG used in the in-house developed device described in Section 3.3.2. The

³⁴ $\log_{10}(2^{14}) = 4.2$

manufacturer specified precisions listed in Table 3.1 were converted to relative standard errors. The relative standard error for the gyroscope is 0.03%, and 0.005% for both the accelerometer and magnetometer.

Figures 4.16, 4.17 and 4.18 show the impact of the relative standard error obtained from the sensitivity of the sensors specified by the manufacturer. To show the impact of the precision of each sensor, the errors were introduced to each sensor independently. Figure 4.16 shows the impact of the stated sensitivity of the gyroscope on the floating-point version of the Improved AHRS algorithm across a range of rotational speeds with varying gradient descent corrective step size α values. Figure 4.17 shows the impact of the stated sensitivity of the accelerometer. Figure 4.18 shows the impact of the stated sensitivity of the magnetometer. Again, for clarity, the standard deviations are not shown.

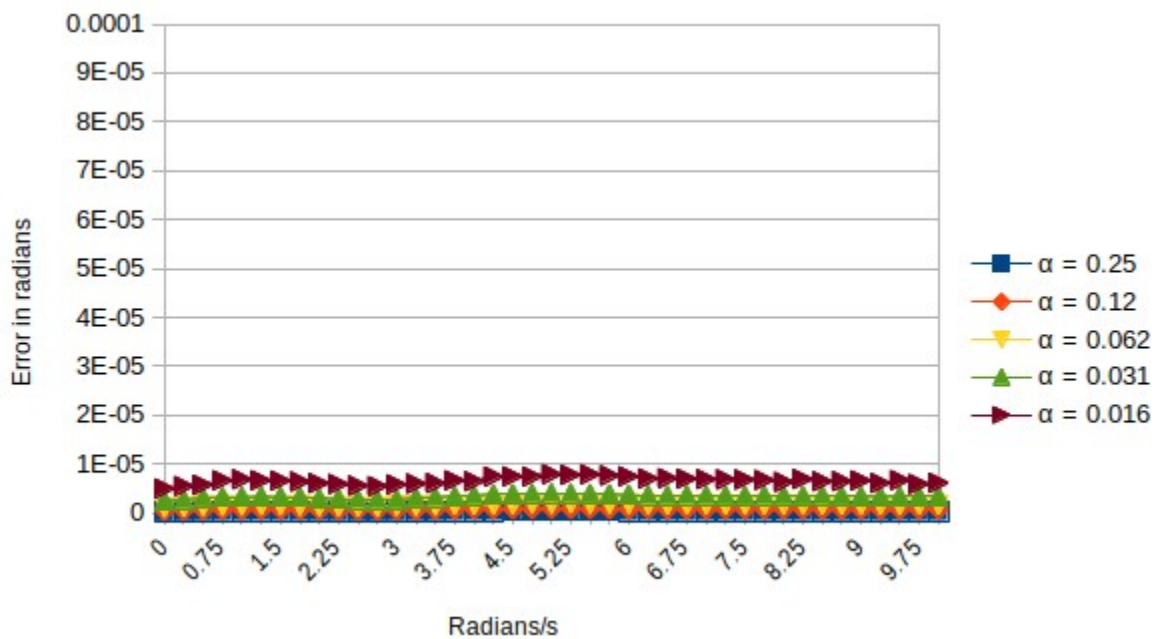


Figure 4.16: Impact of the stated precision of the gyroscope on the Improved AHRS algorithm with different gradient descent step-sizes (α)

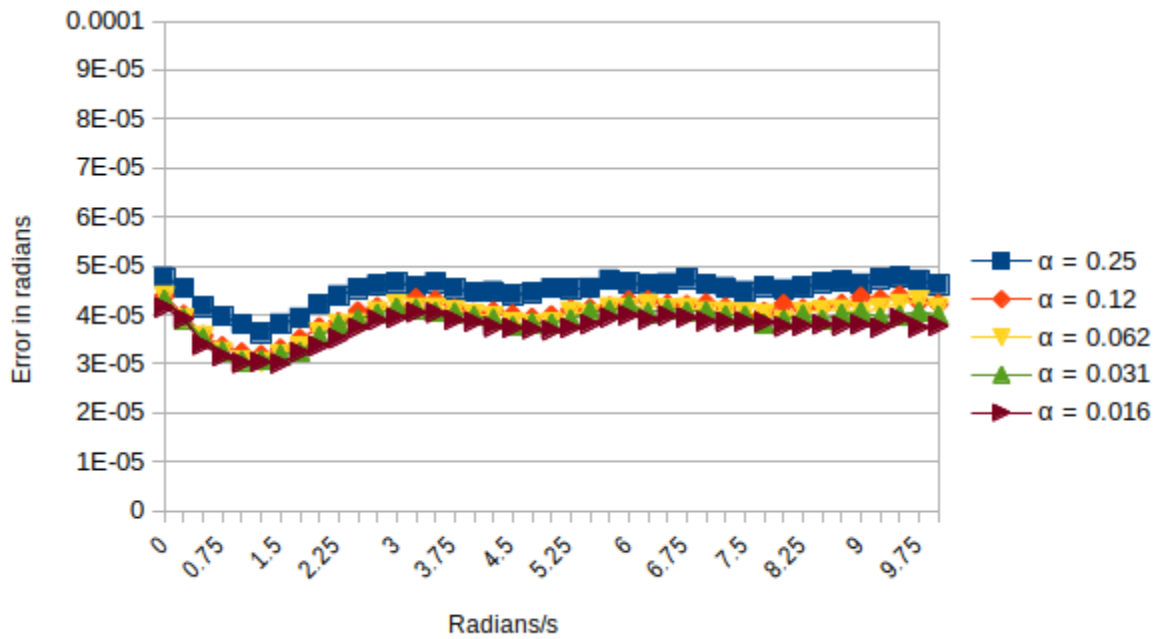


Figure 4.17: Impact of the stated precision of the accelerometer on the Improved AHRS algorithm with different gradient descent step-sizes (α)

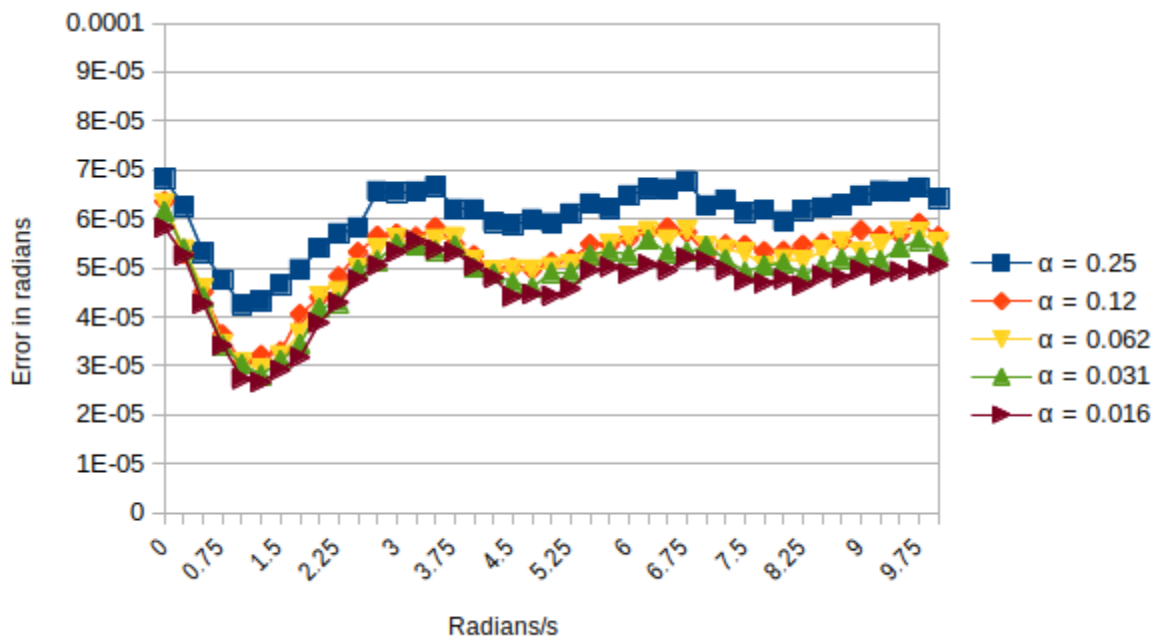


Figure 4.18: Impact of the stated precision of the magnetometer on the Improved AHRS algorithm with different gradient descent step-sizes (α)

Although the gyroscope has the largest relative standard error, it has the smallest impact on the error of the floating-point version of the Improved AHRS algorithm. The magnetometer, with a five to six-fold smaller relative standard error has the largest impact on the error of the Improved AHRS algorithm.

The manufacturer specified relative standard error of the gyroscope has an impact of less than 1×10^{-5} radians across the range of rotational speeds and α values. The accelerometer, with a six-fold smaller manufacturer specified relative standard error, generates an average error of between 3×10^{-5} and 5×10^{-5} radians across the range of rotational speeds and α values. The precision of the magnetometer, with the same manufacturer specified relative standard error as the accelerometer, has an impact of up to 7×10^{-5} radians on the floating-point version of the Improved AHRS algorithm.

In addition to the stated precision of the LSM9DS0 MARG, it was noticed that the precision of measurements taken from a stationary LSM9DS0 MARG were significantly lower. The measured precision was calculated using the average of the range of 1000 measurements, obtained from ten samples, across all three axis, of three different stationary MARGs. The relative standard error for the gyroscopes was found to be 1.0%, 0.1% for the accelerometers, and 0.2% for the magnetometers. The measured relative standard error of the accelerometer, gyroscope and magnetometer are 20 to 40 times greater than the relative standard error specified by the manufacturer.

The larger measured relative standard error can be accounted for by actual variances in the experimental environment. The threshold for detection of linear accelerations is between 6.5 and 8.5 cm/s²³⁵. Similarly, the earth's magnetic field varies between 0.25 to 0.65 gauss. Both these

35 (Kingma, 2005)

ranges are significantly greater than the measured variances, and would easily account for all the variances measured by a stationary MARG on a lab bench.

To test the sensitivity of the algorithms to the measured sensitivity of the MARG sensors the above simulations were rerun with new random errors. The error introduced was taken from a uniform distribution in the range of the measured precisions. Figures 4.19, 4.20 and 4.21 show the impact of the measured relative standard errors of the three sensors. Again, to show the impact of the precision of each sensor, the errors were introduced to each sensor independently. Figure 4.19 shows the impact of the measured sensitivity of the gyroscope on the Improved AHRS algorithm across a range of rotational speeds with varying gradient descent corrective step size α values. Figure 4.20 shows the impact of the measured sensitivity of the accelerometer. Figure 4.21 shows the impact of the measured sensitivity of the magnetometer. Again, for clarity, the standard deviations are not shown.

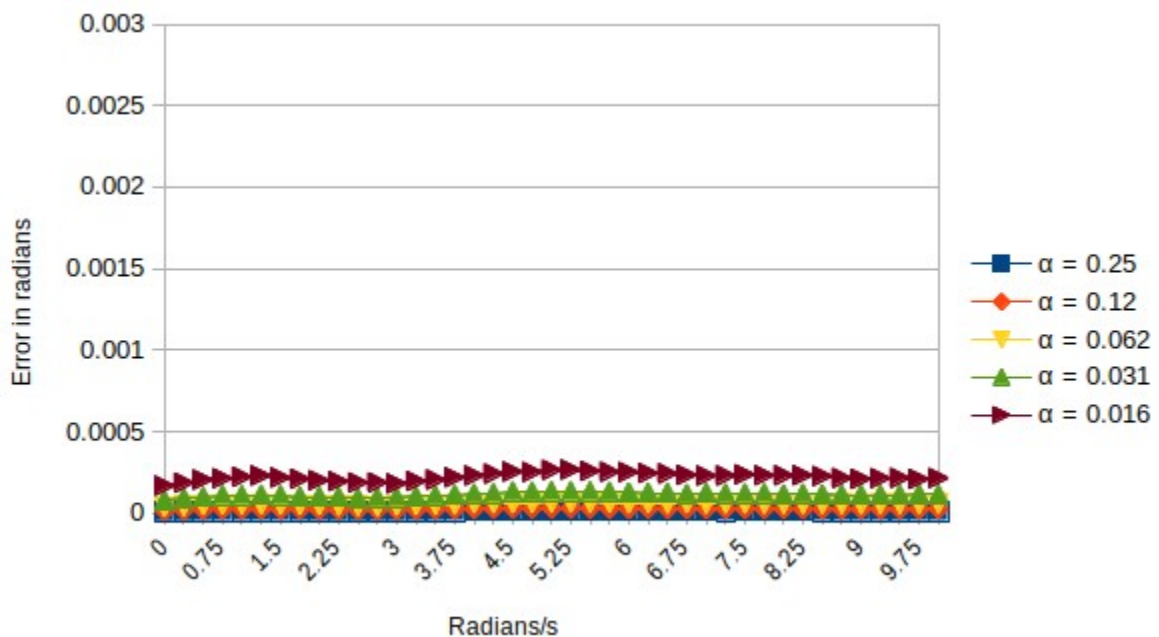


Figure 4.19: Impact of the measured precision of the gyroscope on the Improved AHRS algorithm with different gradient descent step-sizes (α)

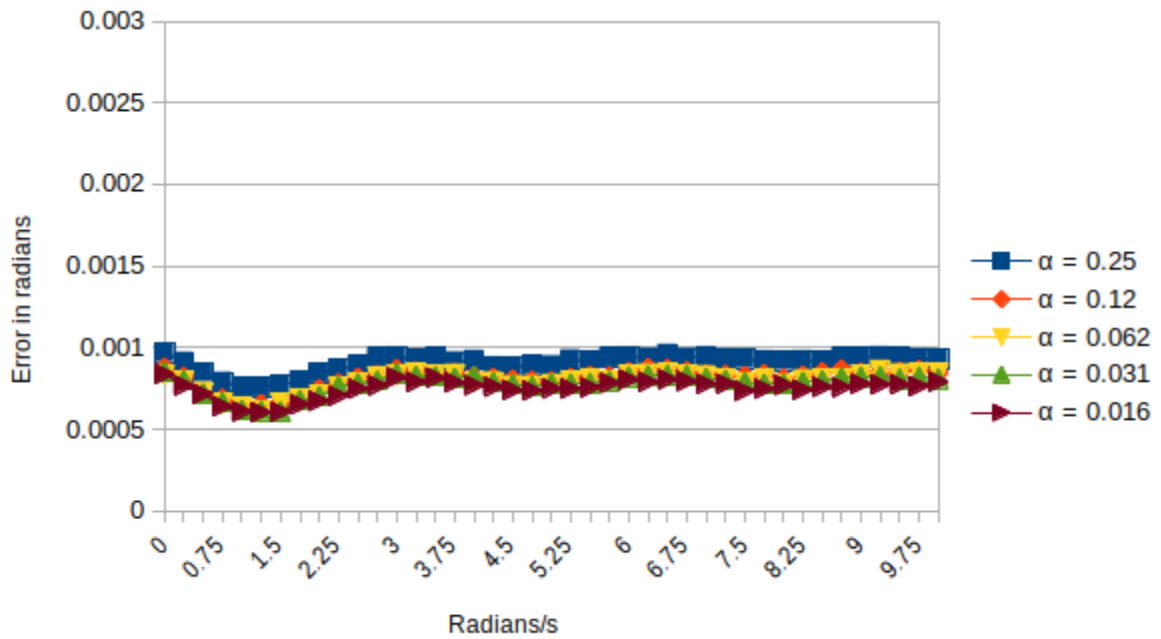


Figure 4.20: Impact of the measured precision of the accelerometer on the Improved AHRS algorithm with different gradient descent step-sizes (α)

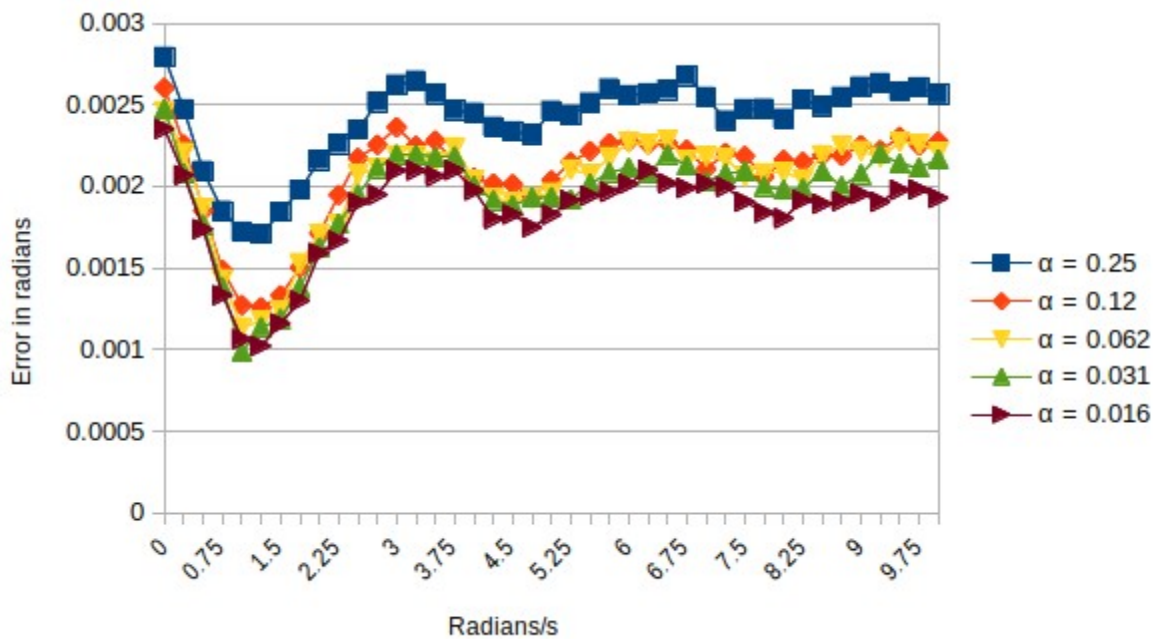


Figure 4.21: Impact of the measured precision of the magnetometer on the Improved AHRS algorithm with different gradient descent step-sizes (α)

The impact of the measured relative standard errors are orders of magnitude larger, but the gyroscope, despite having the largest measured relative standard error, still has the smallest impact on the error of the algorithm. The impact of the gyroscope measured relative standard error is significantly less than 0.5×10^{-3} radians across the range of rotational speeds and α values. The accelerometer, with a ten-fold smaller measured relative standard error, generates an average error of up to 1.0×10^{-3} radians across the range of rotational speeds and α values. However, the measured precision of the magnetometer, despite being five times smaller than the measured precision of the gyroscope, but twice the measured precision of the accelerometer, has the largest impact on the error of the fixed-point, shifted-integer version of the Improved AHRS algorithm. The measured precision of the magnetometer has an impact of up to 3.0×10^{-3} radians across the range of rotational speeds and α values.

The impact of the precision of the sensors, especially the precision of the accelerometer and the magnetometer used during the corrective step of the Improved AHRS algorithm, also has an impact on the best α value to be used in the gradient descent algorithm in the corrective step. The results suggest that to minimise the impact, a smaller α value of between 0.12 and 0.016 should be used.

A final consideration is that large lateral accelerations cannot be distinguished from gravitational acceleration. Lateral accelerations will affect the accuracy of the measured acceleration against which the reference vector is measured, which will affect the accuracy of the correction step of the algorithm. To minimise the impact of lateral accelerations, the corrective step can be skipped when the lateral acceleration is greater than a specified threshold value.

4.7 Summary

A number of issues were identified with the original AHRS orientation quaternion algorithm used to calculate the orientation quaternion from MARG sensors. First, the algorithm does not use the

gradient descent algorithm (4.21). Second, they do not use the actual reverse rotation difference equation (4.18) and its gradient; so the resulting Jacobian used in equation (4.24) to solve the simultaneous non-linear equations for iteratively updating the rotation unit quaternions is inaccurate. Finally, the calculated magnetometer's reference vector, not only constrains the vector to the x - z plane, it unnecessarily constrains it to the positive half of the plane. This not only increases the error of the offset from the actual inclination, it also makes the initial conditions ambiguous.

To address these issues, the Improved AHRS algorithm was formulated. A simulation environment was created to compare the AHRS orientation quaternion algorithms, and the Improved AHRS algorithm was found to be five orders of magnitude better than previous approaches. The errors in the Improved AHRS algorithm under various simulated conditions were shown to be orders of magnitude smaller than the measurement errors of the MARG sensors used in this research; therefore any further improvement would be academic only.

The fixed-point, shifted-integer version of the Improved AHRS algorithm is now used in the nuIMU described in Section 3.3.2.

Chapter 5

Formulation of New Machine Learning Algorithms and Policies

5.1 *Introduction*

This research is about putting artificial intelligence into wearable human machine interfaces.

Machine learning techniques are used to provide the artificial intelligence. The key machine learning technique used is reinforcement learning. Reinforcement learning enables self-learning. Temporal difference learning allows for delayed learning.

Delayed learning is difficult, because there is no explicit signal that indicates the correct output at each time step, and when the final reward signal is received, the learner must also solve the

temporal credit assignment problem. Temporal Difference (TD) learning solves the temporal credit assignment problem by apportioning the credit using the difference between temporally successive predictions. Although TD learning has been applied to neural networks³⁶, and regularisation has been applied to TD learning³⁷, to the best of the author's knowledge there are no published algorithms for regularised, TD learning applied to the back-propagation algorithm.

TD learning solves the temporal credit assignment problem, and cost minimisation is used to provide the machine learning policy. However, reinforcement learning requires exploration to learn. Unfortunately, the standard approach of using random exploration results in actions known to be bad being selected occasionally. This is unacceptable in an operating environment. Therefore, there is a requirement for a new, operational reinforcement learning policy that avoids exploring actions that are known to be bad: a Prioritised Reinforcement Learning Policy.

Similarly, there is a requirement for a control policy that can use a self-improving controller to steadily improve the control of the device. The fidelity of control should improve as the accuracy of the predicted control signal improves. This chapter describes the formulation of new algorithms and the reformulation of algorithms that, as far as the author is aware, have not been previously published.

Section 5.2 provides an overview of TD learning and the equations upon which the new algorithms are based. Section 5.3 shows how regularisation has been applied to the TD learning equations. Section 5.5 formulates the application of these regularised TD learning equations to the artificial neural network back-propagation algorithm. Section 5.6 describes the development of the Prioritised Reinforcement Learning Policy. Finally, Section 5.7 describes the creation of a Blended Control Policy that takes advantage of a controller that improves while being used.

³⁶ (Anderson, 1986; Tesauro, 1995)

³⁷ (Hoffman *et al.*, 2011; Liu, Mahadevan & Liu, 2012)

5.2 Overview of TD Learning

An overview of machine learning, artificial neural networks and the derivation of the equations used in this research can be found in Appendix B. To facilitate the description of the derivation of the algorithms formulated in this Chapter, the original TD learning algorithm is outlined below.

A prediction function f is based on a set of weight parameters $\mathbf{w}$ that takes an input $\mathbf{x}$ to predict an output value p : $f_{\mathbf{w}}(\mathbf{x}) = p$. A cost function J quantifies the error between the predicted output p and the desired output y : $J_{f_{\mathbf{w}}(\mathbf{x})}(p, y)$. Machine learning uses the gradient descent method to iteratively update the weights to minimise the cost function with respect to the weights $\mathbf{w}$:

$$\mathbf{w} := \mathbf{w} - \alpha \nabla J(\mathbf{w}) \quad (5.1)$$

(Widrow & Hoff, 1960)

Given a sequence of observational inputs $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \mathbf{x}^{(3)}, \dots, \mathbf{x}^{(m)}$, with predicted outputs $p^{(1)}, p^{(2)}, p^{(3)}, \dots, p^{(m)}$, and a final outcome y ; the standard half, average quadratic difference cost function is used:

$$J(\mathbf{w}) = \frac{1}{2m} \sum_{t=1}^m (p^{(t)} - y)^2 \quad (5.2)$$

whose gradient with respect to the weights is:

$$\nabla J(\mathbf{w}) = \frac{1}{m} \sum_{t=1}^m (p^{(t)} - y) \nabla p^{(t)} \quad (5.3)$$

Using the fact that

$$p^{(t)} - y = \sum_{h=t}^m (p^{(h)} - p^{(h+1)}) \quad (5.4)$$

where $p^{(m+1)} \equiv y$ then:

$$\nabla J(\mathbf{w}) = \frac{1}{m} \sum_{t=1}^m \sum_{h=t}^m (p^{(h)} - p^{(h+1)}) \nabla_{\mathbf{w}} p^{(t)} \quad (5.5)$$

$$= \frac{1}{m} \sum_{h=1}^m \sum_{t=1}^h (p^{(h)} - p^{(h+1)}) \nabla_{\mathbf{w}} p^{(t)} \quad (5.6)^{38}$$

$$= \frac{1}{m} \sum_{t=1}^m (p^{(t)} - p^{(t+1)}) \sum_{h=1}^t \nabla_{\mathbf{w}} p^{(h)} \quad (5.7)$$

Using this version of the gradient of the cost function with respect to the weights, the update procedure (5.1) becomes:

$$\mathbf{w} := \mathbf{w} - \frac{\alpha}{m} \sum_{t=1}^m (p^{(t)} - p^{(t+1)}) \sum_{h=1}^t \nabla_{\mathbf{w}} p^{(h)} \quad (5.8)$$

Equation (5.8) can be processed iteratively at each time step using:

$$\mathbf{w} := \mathbf{w} - \beta (p^{(t)} - p^{(t+1)}) \sum_{h=1}^t \nabla_{\mathbf{w}} p^{(h)} \quad (5.9)$$

where $\beta = \frac{\alpha}{m}$ is a parameter that needs to be selected.

38 Equation (5.5) was not obviously equal to equation (5.6), but their equality was verified by expanding the sums and simplifying:

$$\begin{aligned} \frac{1}{m} \sum_{t=1}^m \sum_{h=t}^m (p^{(h)} - p^{(h+1)}) \nabla_{\mathbf{w}} p^{(t)} &= \frac{1}{m} \left(\sum_{h=1}^m (p^{(h)} - p^{(h+1)}) \nabla_{\mathbf{w}} p^{(1)} + \sum_{h=2}^m (p^{(h)} - p^{(h+1)}) \nabla_{\mathbf{w}} p^{(2)} + \dots + \sum_{h=3}^m (p^{(h)} - p^{(h+1)}) \nabla_{\mathbf{w}} p^{(m)} \right) \\ &= \frac{1}{m} \left((p^{(1)} - p^{(2)}) \nabla_{\mathbf{w}} p^{(1)} + (p^{(2)} - p^{(3)}) \nabla_{\mathbf{w}} p^{(1)} + \dots + (p^{(m)} - p^{(m+1)}) \nabla_{\mathbf{w}} p^{(1)} + \right. \\ &\quad \left. (p^{(2)} - p^{(3)}) \nabla_{\mathbf{w}} p^{(2)} + \dots + (p^{(m)} - p^{(m+1)}) \nabla_{\mathbf{w}} p^{(2)} + \dots + (p^{(m)} - p^{(m+1)}) \nabla_{\mathbf{w}} p^{(m)} \right) \\ \frac{1}{m} \sum_{h=1}^m \sum_{t=1}^h (p^{(h)} - p^{(h+1)}) \nabla_{\mathbf{w}} p^{(t)} &= \frac{1}{m} \left(\sum_{t=1}^1 (p^{(1)} - p^{(2)}) \nabla_{\mathbf{w}} p^{(t)} + \sum_{t=1}^2 (p^{(2)} - p^{(3)}) \nabla_{\mathbf{w}} p^{(t)} + \dots + \sum_{t=1}^m (p^{(m)} - p^{(m+1)}) \nabla_{\mathbf{w}} p^{(t)} \right) \\ &= \frac{1}{m} \left((p^{(1)} - p^{(2)}) \nabla_{\mathbf{w}} p^{(1)} + (p^{(2)} - p^{(3)}) \nabla_{\mathbf{w}} p^{(1)} + (p^{(2)} - p^{(3)}) \nabla_{\mathbf{w}} p^{(2)} + \dots + \right. \\ &\quad \left. (p^{(m)} - p^{(m+1)}) \nabla_{\mathbf{w}} p^{(1)} + (p^{(m)} - p^{(m+1)}) \nabla_{\mathbf{w}} p^{(2)} + \dots + (p^{(m)} - p^{(m+1)}) \nabla_{\mathbf{w}} p^{(m)} \right) \\ &= \frac{1}{m} \left((p^{(1)} - p^{(2)}) \nabla_{\mathbf{w}} p^{(1)} + (p^{(2)} - p^{(3)}) \nabla_{\mathbf{w}} p^{(1)} + \dots + (p^{(m)} - p^{(m+1)}) \nabla_{\mathbf{w}} p^{(1)} + \right. \\ &\quad \left. (p^{(2)} - p^{(3)}) \nabla_{\mathbf{w}} p^{(2)} + \dots + (p^{(m)} - p^{(m+1)}) \nabla_{\mathbf{w}} p^{(2)} + \dots + (p^{(m)} - p^{(m+1)}) \nabla_{\mathbf{w}} p^{(m)} \right) \end{aligned}$$

To increase the weight of more recent predictions, exponential weighting is used. By discounting predictions occurring k steps in the past using λ^k for $0 < \lambda < 1$, the iterative update function (5.9) becomes:

$$\mathbf{w} := \mathbf{w} - \beta (p^{(t)} - p^{(t+1)}) \sum_{h=1}^t \lambda^{t-h} \nabla_{\mathbf{w}} p^{(h)} \quad (5.10)$$

This is simplified to:

$$\Delta \mathbf{w}^{(t)} = -\beta \delta^{(t)} \mathbf{e}^{(t)} \quad (5.11)$$

with δ at time t defined as the difference between the current prediction and the previous prediction:

$$\delta^{(t)} = p^{(t)} - p^{(t+1)} \quad (5.12)$$

and the *eligibility trace* $\mathbf{e}$ at time t defined as:

$$\mathbf{e}^{(t)} = \sum_{h=1}^t \lambda^{t-h} \nabla_{\mathbf{w}} p^{(h)} \quad (5.13)$$

(Sutton, 1988)

5.3 Regularised TD Learning Algorithm

To prevent over-fitting and the clamping of neurons in an artificial neural network, the size of the weights needs to be kept low. Therefore, a regularised form of the TD learning algorithm is needed. Although other research describes the use of a regularised TD learning algorithm, as far as the author is aware, there is no known documented formulation of the regularised form of the TD learning algorithm. Therefore, this section describes how the TD learning algorithm is extended to create a regularised TD learning algorithm.

Instead of using the standard half, average quadratic difference cost function (5.2), a regularised version of the cost function is used:

$$J(\mathbf{w}) = \frac{1}{2m} \left(\sum_{t=1}^m (p^{(t)} - y)^2 + \gamma \sum_{j=1}^n w_j^2 \right) \quad (5.14)$$

where γ is the regularisation factor.

The size of the weights will exponential increase the cost function. To minimise the cost function, the size of the weights need to be kept small too. The partial derivatives with respect to the weights are:

$$\frac{\partial J(\mathbf{w})}{\partial w_j} = \frac{1}{m} \sum_{t=1}^m (p^{(t)} - y) \frac{\partial}{\partial w_j} p^{(t)} + \frac{\gamma}{m} w_j \quad (5.15)$$

Using equation (5.4): $p^{(t)} - y = \sum_{h=t}^m (p^{(h)} - p^{(h+1)})$ these regularised partial derivatives were then

converted into the TD form:

$$\frac{\partial J(\mathbf{w})}{\partial w_j} = \frac{1}{m} \sum_{t=1}^m \sum_{h=t}^m (p^{(h)} - p^{(h+1)}) \frac{\partial}{\partial w_j} p^{(t)} + \frac{\gamma}{m} w_j \quad (5.16)$$

$$= \frac{1}{m} \sum_{h=1}^m \sum_{t=1}^h (p^{(h)} - p^{(h+1)}) \frac{\partial}{\partial w_j} p^{(t)} + \frac{\gamma}{m} w_j \quad (5.17)$$

$$= \frac{1}{m} \sum_{t=1}^m (p^{(t)} - p^{(t+1)}) \sum_{h=1}^t \frac{\partial}{\partial w_j} p^{(h)} + \frac{\gamma}{m} w_j \quad (5.18)^{39}$$

Using these regularised TD versions of the partial derivatives of the cost function with respect to the weights, the incremental update procedure (5.1) becomes:

$$w_j := w_j - \frac{\alpha}{m} \sum_{t=1}^m (p^{(t)} - p^{(t+1)}) \sum_{h=1}^t \frac{\partial}{\partial w_j} p^{(h)} - \frac{\gamma}{m} w_j \quad (5.19)$$

39 The variables t and h used in the sums have been swapped, for greater clarity in the following steps.

The incremental update procedure can be processed iteratively at each time step using:

$$w_j := w_j - \alpha (p^{(t)} - p^{(t+1)}) \sum_{h=1}^t \frac{\partial}{\partial w_j} p^{(h)} - \gamma w_j \quad (5.20)^{40}$$

The sum of the derivatives of the predictions with respect to the weights is also calculated iteratively using:

$$\sum_{h=1}^{t+1} \frac{\partial}{\partial w_j} p^{(h)} = \frac{\partial}{\partial w_j} p^{(t+1)} + \sum_{h=1}^t \frac{\partial}{\partial w_j} p^{(h)} \quad (5.21)$$

Earlier predictions can also be discounted. Predictions occurring k steps in the past are exponentially discounted using λ^k for $0 < \lambda < 1$, by updating the iterative update to:

$$w_j := w_j - \alpha (p^{(t)} - p^{(t+1)}) \sum_{h=1}^t \lambda^{t-h} \frac{\partial}{\partial w_j} p^{(h)} - \frac{\gamma}{m} w_j \quad (5.22)$$

The simplified form of the update function (5.11) becomes:

$$\Delta w_j^{(t)} = -\alpha \delta^{(t)} e^{(t)} - \gamma w_j^{(t)} \quad (5.23)$$

The prediction difference δ at time t is defined as:

$$\delta^{(t)} = p^{(t)} - p^{(t+1)} \quad (5.24)$$

The eligibility trace e at time t defined as:

$$e^{(t)} = \sum_{h=1}^t \lambda^{t-h} \frac{\partial}{\partial w_j} p^{(h)} \quad (5.25)$$

40 The α in equation (5.20) is the α in equation (5.19) divided by m . Similarly, the γ in equation (5.20) is the γ in equation (5.19) divided by m . However, since m may be unknown and both α and γ are parameters that need to be determined, for readability, the same variable name is used.

The exponentially discounted sum of the derivatives of the predictions with respect to the weights is calculated iteratively using:

$$\sum_{h=1}^{t+1} \lambda^{t+1-h} \frac{\partial}{\partial w_j} p^{(h)} = \frac{\partial}{\partial w_j} p^{(t+1)} + \lambda \sum_{h=1}^t \lambda^{t-h} \frac{\partial}{\partial w_j} p^{(h)} \quad (5.26)$$

which can be simplified to:

$$e^{(t+1)} = \frac{\partial}{\partial w_j} p^{(t+1)} + \lambda e^{(t)} \quad (5.27)$$

5.4 TD Learning Applied to the Artificial Neural Network

Back-Propagation Algorithm

Although other research describes the use of the TD learning algorithm applied to the artificial neural network back-propagation algorithm, as far as the author is aware, there is no known documented formulation of the TD learning algorithm applied to the artificial neural network back-propagation algorithm. Therefore, below is a description of how these algorithms are formulated.

First, the simplified form of the TD weight update function (5.11) is applied to each of the weights in the artificial neural network:

$$\Delta w_{kj}^{(l)} = -\beta \delta_k^{(l,t)} e_k^{(l,t)} \quad (5.28)$$

where $w_{kj}^{(l)}$ is the weight from neuron j to neuron k in layer l .

Using equation (5.28) simplifies the goal to finding the appropriate functions to apply to the difference $\delta_k^{(l,t)}$ and eligibility trace $e_k^{(l,t)}$ values for each neuron k in each layer l at each time step t .

The output of an artificial neural network is calculated iteratively one layer at a time. The output $p_k^{(l)}$, of each of the $s^{(l)}$ neurons in layer l , is calculated using the weighted sum of the non-linear modified outputs of the previous layer:

$$p_k^{(l+1)} = \sum_{j=1}^{s^{(l)}} w_{kj}^{(l+1)} g_j^{(l)}(p_j^{(l)}) + b^{(l+1)} \quad (5.29)$$

where $g(x)$ is the non-linear output activation function for a neuron, and b is the bias for the layer.

The partial derivatives of the output of a neuron with respect to the weights is the modified output of the previous neuron i.e. after the activation function:

$$\frac{\partial p_k^{(l)}}{\partial w_{kj}^{(l)}} = g_j^{(l-1)}(p_j^{(l-1)}) \quad (5.30)$$

Equation (5.30) is calculated iteratively, layer by layer.

The output of each neuron is a prediction. Therefore, given a sequence of observational inputs $\mathbf{x}^{(1)}$, $\mathbf{x}^{(2)}$, $\mathbf{x}^{(3)}$, ..., $\mathbf{x}^{(m)}$, each neuron will have a series of predicted outputs $p_k^{(l,1)}$, $p_k^{(l,2)}$, $p_k^{(l,3)}$, ..., $p_k^{(l,m)}$.

Applying the partial derivatives (5.30) to the iterative calculation of the eligibility trace (5.27) gives:

$$e^{(t+1)} = g_j^{(l-1)}(p_j^{(l-1)}) + \lambda e^{(t)} \quad (5.31)$$

Since each neuron is a prediction, it is tempting to simply extend equation (5.24) to:

$$\delta_k^{(l,t)} = p_k^{(l,t)} - p_k^{(l,t+1)} \quad (5.32)$$

However, (5.32) cannot be used to calculate δ , because it is based on equation (5.4), which requires the desired output of $p_k^{(l,m+1)}$ to be known. The correct outputs of hidden neurons are not known. During supervised learning, only the desired outputs of the final layer neurons at the end of the sequence y_k will be known.

The sequence of observational inputs $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \mathbf{x}^{(3)}, \dots, \mathbf{x}^{(m)}$, will produce a series of predicted outputs to the artificial neural network: $f_w(\mathbf{x}^{(1)})_k, f_w(\mathbf{x}^{(2)})_k, f_w(\mathbf{x}^{(3)})_k, \dots, f_w(\mathbf{x}^{(m)})_k$; with $f_w(\mathbf{x}^{(m+1)})_k \equiv y_k$. The predicted outputs at time $t+1$ are used to calculate the temporal differences for the output layer (L) neurons at time t :

$$\delta_k^{(L)} = f_w(\mathbf{x}^{(t)})_k - f_w(\mathbf{x}^{(t+1)})_k \quad (5.33)$$

The artificial neural network back-propagation algorithm defines the error attributed to neurons in lower layers in terms of the weighted errors of the neurons in the layer above:

$$\delta_j^{(l)} = \sum_{k=1}^{s^{(l+1)}} \left(\delta_k^{(l+1)} w_{kj}^{(l+1)} \right) \frac{\partial g_j^{(l)}}{\partial p_j^{(l)}} \quad (5.34)$$

(Nielsen, 2015)

The input errors are defined as zero: $\delta_j^{(0)} = 0$.

The back-propagation algorithm (5.34) is extended to support the TD algorithm:

$$\delta_j^{(l,t)} = \sum_{k=1}^{s^{(l+1)}} \left(\delta_k^{(l+1,t)} w_{kj}^{(l+1)} \right) \frac{\partial g_j^{(l,t)}}{\partial p_j^{(l,t)}} \quad (5.35)$$

Finally, the incremental form of the artificial neuron, non-linear output activation function, eligibility trace equation (5.31) and the temporal error back-propagation algorithm (5.35) are applied to the TD, iterative, artificial neuron weight change equation (5.28) to create the TD learning applied to the artificial neural error back-propagation algorithm:

$$\Delta w_{kj}^{(l,t)} = -\beta \left(\sum_{i=1}^{s^{(l+1)}} \left(\delta_i^{(l+1,t)} w_{ik}^{(l+1)} \right) \frac{\partial g_k^{(l,t)}}{\partial p_k^{(l,t)}} \left(g_j^{(t)} + \lambda e_j^{(t-1)} \right) \right) \quad (5.36)$$

5.5 Regularised TD Learning Applied to the Artificial Neural Network Back-Propagation Algorithm

As far as the author is aware, there is no known formulation of a regularised TD learning equation applied to the artificial neural network back-propagation algorithm. Below is a description of how this algorithm was formulated.

First, the simplified form of the regularised TD weight update function (5.23) was applied to each of the weights in the artificial neural network:

$$\Delta w_{kj}^{(l)} = -\alpha \delta_k^{(l,t)} e_k^{(l,t)} - \gamma w_{kj}^{(l,t)} \quad (5.37)$$

where $w_{kj}^{(l)}$ is the weight from neuron j to neuron k in layer l .

As with equation (5.28), equation (5.37) simplifies the goal to finding the appropriate functions to apply to the difference $\delta_k^{(l,t)}$ and eligibility trace $e_k^{(l,t)}$ values for each neuron k in each layer l at each time step t , but equation (5.37) also includes the regularisation component.

The incremental form of the artificial neuron, non-linear output activation function, eligibility trace equation (5.31) and the temporal error back-propagation algorithm (5.35) were applied to the regularised, TD, iterative, artificial neuron weight change equation (5.37) to create the regularised TD learning applied to the artificial neural error back-propagation algorithm:

$$\Delta w_{kj}^{(l,t)} = -\alpha \left(\sum_{i=1}^{s^{(l+1)}} \left(\delta_i^{(l+1,t)} w_{ik}^{(l+1)} \right) \frac{\partial g_k^{(l,t)}}{\partial p_k^{(l,t)}} \left(g_j^{(t)} + \lambda e_j^{(t-1)} \right) + \gamma w_{kj} \right) \quad (5.38)$$

5.6 Prioritised Reinforcement Learning Policy

A reinforcement learner needs a policy for selecting an action from the current state. The most basic policy is the greedy policy: selecting the action associated with the current highest estimated value.

However, this may result in always doing the same thing; potentially selecting a non-optimal action from a state. To ensure that potentially better actions that are currently valued lower are explored, the ϵ -greedy policy uses a greedy policy most of the time, and a random action a specified percentage of the time. However, this approach results in actions known to be bad being selected occasionally.

A policy based on preferences picks an action randomly with the probability of an action being selected being proportional to its preference. Pursuit methods update the policy by increasing the preference of the action associated with the current highest estimated value, and decreasing the preferences of all the other actions. However, this approach requires storing the preferences of every state-action; an approach that is not generalisable. To overcome the issue of exploring actions known to be bad, or requiring storing preferences for every state-action, a Prioritised Reinforcement Learning Policy was created.

First, instead of storing action preferences, this new policy uses the estimated values of each action, by using the post action state as the input. Then, as with the preference based policy, the action selected is picked at random with the probability of an action being selected being proportional to the estimated value. Second, instead of estimating the value of an action, the learner is trained to predict the probability that a particular action is correct. Without any learning, the learners will, on average, initially select each of the available actions with equal probability. Adjusting the probability of an action being correct simultaneously adjusts its probability of being selected. The main advantage of this approach is that it is more likely to explore alternative actions that are good, but not the best, and less likely to explore actions that have proven to be bad.

This approach had two challenges. First, it required that the cumulative rewards be normalised. Second, the estimated probability that an action is correct is independent of the number of actions;

this was especially true with the learners created during this research, which are designed to enable the number of actions to be increased without impacting existing learning.

In simple right or wrong environments normalising rewards is easy. If the correct action is known, the problem is reduced to a supervised learning problem. A final reward of 1 is awarded for a correct answer and 0 for an incorrect answer. The predictions are normalised using logistic activation functions on the output neurons. Similarly, if the policy is being used to play a game, the final rewards are simply 1 for a victory, $\frac{1}{2}$ for a draw and 0 for a loss. Again, the logistic activation function on the output neurons is used.

More complex environments will have cumulative rewards. If the maximum cumulative reward is known, all rewards can be normalised by dividing by this maximum. However, in an iterative learning environment, there is potentially no maximum cumulative reward. Even if there is a maximum cumulative reward, it may not be known. The hyperbolic tangent function can be used to convert the cumulative rewards to a value between -1 and 1. Its non-linear nature ensures that positive rewards are strongly favoured over negative rewards, but there is less differentiation between larger rewards.

To resolve this, the two approaches were combined. First, the cumulative value is divided by the maximum expected reward. Second, the divided value is passed through the hyperbolic tangent function to ensure that the final reward is between -1 and 1. This approach provides a balance between ensuring a close to linear differentiation between rewards up to a certain expected maximum cumulative reward value, while not putting an upper limit on the actual cumulative rewards.

To resolve the challenge that the estimated probability that an action is correct is independent of the number of actions, the soft-max or normalised exponential function was used:

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}} \quad (5.39)$$

(Sutton & Barto, 1998)

The soft-max function ensures that the predicted probabilities, regardless of number, always sum to 1.

However, a problem with the policy was subsequently discovered. Since there is often not a single best action – many environments; especially games, are symmetric – the probability distributions didn't converge on a single action. Equally effective actions would continually try to increase their probability of being selected without converging on a common, joint probability.

To overcome this, the way the probabilities are calculated was separated. Instead of using the predicted probabilities directly, the predictor predicts the value of an action as normal. The probability of selecting a particular action is then calculated using the fraction of the action's value and the sum of all the other actions' values. This enables multiple actions to have an equal probability of being selected without needing to know how many actions may need to have an equal probability of being selected.

However, in temporally delayed reward environments, for example games, the action probabilities for earlier actions still did not converge. The expectation was that, if it had learnt that one action would result in a reward of 1 – a win – and another action would result in a reward of ½ – a draw – the probability of selecting the action with a reward of 1 would be 2/3 and the probability of selecting the action with a reward of ½ would be 1/3. Therefore, the value of the previous set of inputs should converge on 5/6 (the weighted average). However, it doesn't converge and tends to oscillate. The reason could be that always using probabilities results in too much cumulative exploring, which creates too much noise in the feedback to earlier actions.

To overcome this, the standard ϵ -greedy policy approach was modified. The best action is still used most of the time, but when exploring ϵ percent of the time, instead of selecting a random action, the learnt probabilities are used. Using this approach, the predicted probabilities converge. If there is a single best action, both the greedy policy and the learnt probabilities will converge and return the same action. If there are multiple, equally good actions, the learnt probabilities will converge to ensure that all the good actions are selected.

In summary, the Prioritised Reinforcement Learning Policy will select the action with the currently predicted highest value most of the time. A default 10% of the time, the policy will select an action at random, with the probability of an action being selected being proportional to its predicted value relative to all the other values.

5.7 *Blended Control Policy*

Before applying the Generic, Self-Improving Controller to a real-world environment it was necessary to consider how to use the predictions and their confidence levels. In addition to the requirement for a Prioritised Reinforcement Learning Policy described in Section 5.6, there was a further need for a control policy that could take advantage of a controller that improves while being used.

The Generic, Iterative Learners are configured to produce an output vector with values between 0 and 1, which can be interpreted as the predictor's estimated confidence that the input state vector corresponds to each of a number of options. Since the estimated likelihood that any specific option is independent of the number of options, and the number of options can be changed online, the output is converted into probabilities that sum to 1. The standard approach is to use the soft-max function (5.39).

There are a number of ways to use the probabilities generated by the Generic, Self-Improving Predictor to make control decisions. The simplest approach is to simply select the option with the highest probability, which is similar to the reinforcement learning greedy policy. It is sufficient in non-critical environments where selecting the wrong action is not catastrophic, or when incorrect actions can be corrected. Furthermore, there is an expectation that the predictor will make incorrect predictions. Even when fully trained, the predictor is unable to correctly predict all the training data. If it does, it is likely that it has been over-fitted to the training data, and it is more likely to perform worse on input data it has never seen.

A simple approach is to configure the controller to nothing, or some other safe action, by default. Another approach is to ensure that the controller only performs an action when the confidence level of the predicted control action exceeds a threshold. This threshold can even be learnt, by training the controller with a do nothing option, and using a continuous do-nothing feedback signal until a known alternative option feedback signal is sent. The controller's prediction will now result in a high probability for doing nothing until the probability that another option exceeds the probability of doing nothing. This requires that the do something inputs are significantly different from the do nothing inputs. Furthermore, it's important to note that rarely occurring outcomes will take much longer to learn.

Alternatively, instead of only selecting the most likely control action, the confidence level can be used to provide an analogue control signal: the higher the probability the larger the control signal. Two control signals that provide opposite effects are combined. The resulting blended control signal is the effective difference between the probabilities.

The output of the soft-max function will ensure that the outputs sum to 1 and the order of predictions will remain consistent. However, it also tends to squash the values. The difference between the values is reduced. Therefore, to both use the predicted difference between the

probabilities and ensure that the predictions sum to one, a different approach was required. Instead of using the soft-max function, the predictions are linearly scaled to sum to 1. This is achieved by simply dividing each prediction by the sum of the predictions.

The Blended Control Policy uses desired outcomes that are paired so they can use an analogue signal. The analogue signal is derived from the difference between the paired predictions. To ensure that the control signals provide an output that can be interpreted as a confidence level, the outputs are linearly scaled so that they sum to 1.

This Blended Control Policy was implemented in the Blended Mouse Control Policy described in Section 7.5.

5.8 Summary

TD learning solves the temporal credit assignment problem. To keep the size of the weights used in an artificial neural networks low, regularisation is used. Since there is no known documented formulation of regularisation applied to TD learning, regularisation was applied to the TD learning algorithm and its formulation described.

Likewise, there is no known documented formulation of the application of TD learning to artificial neural networks. In addition, there is no known application of regularised TD learning to the artificial neural network back-propagation algorithm. The formulated, regularised TD learning equations were applied to the artificial neural network back-propagation algorithm and its formulation detailed.

Using known bad actions in an operating environment would be unacceptable. To avoid known bad actions being selected by the reinforcement learning policy, the new Prioritised Reinforcement Learning Policy that avoids exploring actions that are known to be bad was developed.

Similarly, a control policy that can use a self-improving controller to continuously improve the fidelity of the control was created. The Blended Control Policy blends the controller predictions; using the differences between the predicted probabilities to determine the required strength of the control signal.

Chapter 6

The Generic, Self-Improving Predictor

6.1 *Introduction*

As visualised in Figure 1.1 of Section 1.5, at the core of the Generic, Self-Improving Controller is the Generic, Self-Improving Predictor. The Generic, Self-Improving Predictor is required to convert a temporal sequence of inputs into predictions and adjust predictions using feedback.

As with the IMU Data Collector described in Chapter 3, the input and the output should be independent. The Generic, Self-Improving Controller will request predictions from the Generic, Self-Improving Predictor when required. The Generic, Self-Improving Predictor will then request

the latest input from the IMU Data Collector. In addition, feedback may not be instantaneous, or even expected. Feedback will be sporadic and delayed. When training the Generic, Self-Improving Controller a temporal sequence of inputs will be presented to the Generic, Self-Improving Controller. When feedback is made available, the Generic, Self-Improving Predictor's underlying learning algorithm is expected to use the temporal sequence of inputs and the feedback to improve future predictions.

This chapter describes the development of the Generic, Self-Improving Predictor. Artificial neural networks are used to allow analogue representations of the state space. Section 6.2 describes the development of a Flexible Artificial Neural Network that can grow without losing existing learning and can use multiple learning algorithms. To enable the Generic, Self-Improving Predictor to use any of a multiple number of iterative learners, a common learning interface is required. The Temporal Learning Interface developed is described in Section 6.3. Sections 6.4, 6.5 and 6.6 describe the application of three different iterative learning algorithms to the Temporal Learning Interface. Section 6.4 describes the Markov Chain, Monte-Carlo (MCMC) learner that uses a standard MCMC algorithm and digital input. Section 6.5 describes the combination of a Markov Chain with the Flexible Artificial Neural Network developed in Section 6.2 to create a MC Network Learner that supports analogue input. Finally, Section 6.6 details the combination of the regularised TD learning algorithm applied to the artificial neural network back-propagation algorithm developed in Section 5.5 with the Flexible Artificial Neural Network developed in Section 6.2 to create a TD Network Learner.

6.2 *Flexible Artificial Neural Network*

A Flexible Artificial Neural Network that can be instantiated by simply specifying the size of the input, the output and the size of each hidden layer in a vector of any length was created. This section describes the implementation of the Flexible Artificial Neural Network. An overview of

artificial neural networks and the derivation of the equations used below can be found in Appendix B.

To calculate the output of a neural network $f(\mathbf{x})$ for a particular input $\mathbf{x}$, the output of each neuron p_i in each layer l is calculated iteratively as the weighted sum of the modified outputs $g(p_i)$ of the neurons in the layer below using layer weights $\mathbf{w}^{(l)}$ and bias $b^{(l)}$:

$$p_i^{(l)} = w_0^{(l)} g(p_0^{(l-1)}) + w_1^{(l)} g(p_1^{(l-1)}) + \dots + w_n^{(l)} g(p_n^{(l-1)}) + b^{(l)} \quad (6.1)$$

In the case of the first layer, the weighted sum of the input $\mathbf{x}$ is used. To ensure a layer cannot be eliminated by replacing it with the linear sum of the previous layer, the function $g(p)$ used to modify the output of each neuron should be non-linear. Learning requires the function $g(p)$ to have a derivative. The neuron type is defined by the function used to modify the output. Multiple neuron types were implemented:

- **Normal:** The output of the neuron is not changed, the cost function is simply the quadratic difference between the output and the target value, and the derivative is 1. This is a linear function, therefore it should not be used. It was only created and used for testing.
- **Binary:** The modified output is either 0 or 1 depending on whether or not the unmodified output is greater than zero. The derivative is not defined; so it cannot be used with a learning algorithm.
- **Stochastic Binary:** The outputs of a layer's neurons are modified using the logistic function:

$$g(p) = \frac{1}{1 + e^{-p}} \quad (6.2)$$

The logistic function ensures the output is between 0 and 1. The value is then set to either 0 or 1 depending on whether or not the logistic output is greater than a random number between 0 or 1. The derivative is undefined; so it too cannot be used with a learning algorithm.

- **Logistic:** As with the stochastic binary neuron, the modified output is defined by the logistic function (6.2). Although the derivative for the stochastic binary neuron is not defined, the logistic function has a derivative:

$$g'(p) = g(p)(1 - g(p)) \quad (6.3)$$

Since the derivative is defined, the logistic neuron can be used with a learning algorithm.

- **Hyperbolic-Tangent:** The modified output is defined by the hyperbolic-tangent function:

$$g(p) = \tanh(p) = \frac{e^p - 1}{e^p + 1} \quad (6.4)$$

The hyperbolic-tangent function ensures the output values are between -1 and 1 . It too has a derivative:

$$g'(p) = 1 - (g(p))^2 \quad (6.5)$$

Since the derivative is defined, the hyperbolic-tangent neuron can also be used with a learning algorithm.

- **Threshold:** The modified output is either 0 or unchanged depending on whether or not the unmodified output is greater than zero. Similarly, the derivative for unmodified outputs less than zero is 0, and for unmodified outputs greater than zero is 1. Although technically the derivative is not defined at zero, the discrete nature of digital approximations allows this to be ignored and simply assigned either zero or 1. Therefore, it can be used with a learning algorithm.

To perform supervised learning, a cost function J with its derivative must be defined. The default cost function used is the weight regularised, half, average quadratic difference between the calculated output $f(\mathbf{x})$ and the known correct supervised output $\mathbf{y}$:

$$J(\mathbf{w}) = \frac{1}{2m} \left(\sum_{i=1}^m (f_w(\mathbf{x}^{(i)}) - y^{(i)})^2 \right) + \gamma \sum_{j=1}^n w_j^2 \quad (6.6)$$

(Bishop, 2006),

The regularisation factor γ must be specified separately.

When the Logistic or Hyperbolic-Tangent neuron output modification functions are used on the output layer for creating classifiers, the cost function used is the weight regularised cross-entropy error:

$$J(\mathbf{w}) = -\frac{1}{m} \sum_{i=1}^m y^{(i)} \log(f(\mathbf{x}^{(i)})) + (1 - y^{(i)}) \log(1 - f(\mathbf{x}^{(i)})) + \gamma \sum_{j=1}^n w_j^2 \quad (6.7)$$

(McClelland, 2015)

Both cost functions have the same partial derivatives with respect to the weights:

$$\frac{\partial J(\mathbf{w})}{\partial w_j} = \frac{1}{m} \sum_{i=1}^m (f_w(\mathbf{x}^{(i)}) - y^{(i)}) \frac{\partial}{\partial w_j} f_w(\mathbf{x}^{(i)}) + \frac{\gamma}{m} w_j \quad (6.8)$$

To calculate the error associated with each neuron, the back-propagation algorithm is implemented: First the output errors δ of the final layer (L) are calculated. If the weight regularised, half, average quadratic difference cost function is used, the output error δ is the partial derivative of the cost function with respect to the unmodified output of each neuron p_j i.e. before the activation function $g(p)$:

$$\delta^{(L)} = \frac{\partial J}{\partial \mathbf{p}^{(L)}} \quad (6.9)$$

If the output neuron type uses the cross entropy cost function, the error is defined simply as the difference between output values and the target values.

The error is then propagated down to each layer (l) using the weighted sum of the errors from the layer above multiplied by the derivative of the activation function defined for the type of neuron:

$$\delta^{(l)} = \sum_{k=1}^{S^{(l+1)}} \left[\delta_k^{(l+1)} \mathbf{w}_k^{(l+1)} \right] \frac{\partial g^{(l)}}{\partial p^{(l)}} \quad (6.10)$$

(Nielsen, 2015)

Finally, the input error is defined as zero: $\delta^{(0)} = \mathbf{0}$.

To calculate the cost gradient with respect to each of the weights, first the errors associated with each neuron are calculated as above, and then the gradient with respect to each weight is calculated. The gradient with respect to each weight is calculated by multiplying the output of the neuron before the weight and the error of the neuron after the weight. The gradients are then augmented using the regularisation term:

$$\frac{\partial J}{\partial \mathbf{w}_{kj}^{(l)}} = \frac{1}{m} \sum_{i=1}^m \delta_k^{(l)} g_j^{(l-1)} + \frac{\gamma}{m} \mathbf{w}_{kj}^{(l)} \quad (6.11)$$

(Nielsen, 2015)

Once instantiated, the sizes of the layers of the artificial neural network can be adjusted. If the size of a layer decreases, the associated weights are simply deleted. If the size of a layer increases, new weights are added, but the values of the existing weights are kept. Although both deleting weights and adding weights will impact the output of the values of their associated neurons; keeping the existing weights ensures that existing learning is not completely lost when adjusting the network size.

To enable different learning algorithms to be used, the weights can be changed manually. Furthermore, to enable the use of the correct regularisation term during batch training, the regularisation terms can be disabled during the cost and gradient calculations. To speed up

calculations, OpenMP was used to process each neuron in each stage of the back-propagation algorithm in parallel.

6.3 Temporal Learning Interface

To enable the Generic, Self-Improving Predictor to use any of a number of different types of iterative learner, the learners require a common interface. This interface defines the requirements of the iterative learners. Since the learners will accept a temporal sequence of inputs, this interface was called the Temporal Learning Interface.

The Temporal Learning Interface specifies that all learners are based on a minimum of four parameters: the length of the input state vector; the length of the output vector; the learning rate; and the temporal reward discount rate. The learners should be able to accept analogue or digital input state vectors. Although not required, if analogue data is used, it should be normalised i.e. all input data should be between 0 and 1. Similarly, the values generated for the output vector should be between 0 and 1; so they can be used for confidence or probability values, however, again, this is not required.

If the values of the output vector are configured to sum to 1, they can be used for probabilities. A common approach is the soft-max function (5.39). In addition to requiring that the learning and temporal reward discount rates be changeable, the interface also requires that the input and output vector lengths be changeable. This is to enable additional input data, for example the addition of a new sensor, and additional output control signals, e.g. the addition of a new device to be added to an existing learner. To the best of the learner's ability, this should not impact existing learning.

The learners accept two kinds of input: a vector representing the current input state along with any interim reward; and a final reward, which, depending on the type of learner, will also indicate the end of a sequence of inputs. The interface also allows for a sequence reset to instruct a learner that

no final reward is forthcoming. In addition, at anytime, the learners are required to return their current predicted output.

As state-value learners, the learners are expected to adjust their outputs to minimise the difference between the estimated value and the cumulative, temporally discounted rewards. The temporal discount rate should be used to reduce the value of earlier interim rewards. While the learning rate should be used to adjust the rate that the output changes.

Standard supervised learning is implemented using a batch trainer. The batch trainer takes a number of input and target output pairs and runs them through a minimisation algorithm. The minimisation algorithm takes a pointer to the regularised cost function and its derivative with respect to the weights, and minimises the cost function with respect to the weights.

The minimisation algorithm used is a port to C++ of Rasmussen's Matlab code⁴¹ for minimising a differentiable multivariate function, which uses the Polack-Ribiere flavour of conjugate gradients to calculate search directions, quadratic and cubic polynomial approximations for line searches, and the Wolfe-Powell stopping criteria to ensure descent.

6.4 MCMC Learner

The first learner implemented was the Markov Chain, Monte-Carlo (MCMC) Learner. The MCMC Learner stores the current estimated value for every input state in a lookup table. If an input state has not been previously encountered, it creates a new entry in the table, which is given a default output value. To create the Markov chain, the sequence of inputs are simultaneously stored in a separate table.

As each input can also be accompanied by an interim reward, each state in the Markov chain table also stores the cumulative, discounted rewards received to date. As each reward is received, the

⁴¹ (Rasmussen, 2006b)

temporal discount is applied before being added to the reward associated with each state in the chain.

After the final reward is received, and the Markov chain's rewards have been updated, each state in the MCMC Learner lookup table found in the Markov chain is updated with the difference between cumulative, discounted rewards received and the current estimated value, multiplied by the learning rate.

When the input state length is changed, the entire input state lookup table is updated. If the new input state length is shorter, the entries in the lookup table are truncated. If two truncated input states are now the same, only the first one is kept. If the new input state length is longer, the entries in the input state table are padded with zeros. If the output length is changed, the default value is adjusted too.

The key limitation to the input state lookup table approach is that it is not generalisable. If an input state has never been encountered, even if it is similar to other input states, it will create a new entry. Therefore, the MCMC Learner requires discretised input data. The data should either be digital or limited to a specific range of values using analogue to digital conversion techniques.

The limitations of the lookup table are particularly relevant when additional sensors are added. The input state is extended, and every input state would not have been encountered before. Although the learned values of shorter states can be used as the initial default value for longer states, this was not implemented.

6.5 MC Network Learner

To overcome both the discretisation requirement of the MCMC Learner and the associated large memory requirements of its lookup table, the value function is estimated. The second learner

implemented was the MC Network Learner. The MC Network Learner uses the Flexible Artificial Neural Network described in Section 6.2 to estimate the value function. The MC Network Learner then uses a Markov chain to store the sequence of input states; together with a cumulative, discounted reward received to date.

As with the MCMC Learner, as each reward is received, the temporal discount is applied before being added to the reward associated with each state in the chain. When the final reward has been received and added, each state in the Markov Chain is compared with the actual cumulative, discounted reward received to determine the error. The error is then back-propagated through the network. The gradients are calculated and the weights adjusted by the product of the negative gradient and the learning rate.

The key limitation to the MC Network Learner is that for learning to occur there needs to be a final reward, which can be zero, to indicate the end of a sequence of inputs⁴². This is generally not a problem, and it has a number of advantages: First, standard supervised learning can be implemented by simply following each input with a final reward i.e. the known target value. Second, if learning is not required or the reward is not known, a sequence reset signal can be sent to ensure that all the previous inputs are ignored.

The requirement for a final reward also implies the requirement to know when a sequence starts. When a final reward or reset sequence signal is sent to the MC Network and MCMC Learners, the implication is that a new sequence starts, and the discounted rewards are applied to all the inputs received thereafter. This assumes a final reward is ultimately received.

⁴² This limitation applies to all MC methods. Therefore, it is also a limitation of the MCMC Learner.

6.6 TD Network Learner

The third learner implemented combines the regularised TD learning algorithm applied to artificial neural networks derived in Section 5.5 with the Flexible Artificial Neural Network created in Section 6.2 to create the TD Network Learner.

As each input is received, the Flexible Artificial Neural Network is used to calculate the estimated value and any interim reward received is added. The result is compared with the output for the previous input and the error is calculated using equation (5.33). The errors are then back propagated using equation (5.35). The prediction gradient sum is calculated iteratively using equation (5.27). Finally, the weights are updated using equation (5.38). When a final reward is received, a flag is set to indicate that there is no previous input, and the gradient sum is set to zero. With the flag set, no update will take place when the next input is received.

Once again, standard supervised learning can be implemented by simply following each input with the known target value as a final reward.

The key limitation to the TD Network Learner is that learning occurs when new inputs are presented. There is no way to indicate that the previous inputs should be ignored, because learning has already been applied. A reset sequence signal will simply not use the next input to update the previous input's estimate. However, if the inputs are valid inputs, this should not be a problem, because the updates will simply reflect the known learning, and hence be a form of unsupervised learning.

6.7 Summary

A Flexible Artificial Neural Network was created that uses OpenMP to achieve the maximum level of performance available to a multi-core processor to implement the back-propagation algorithm.

The Flexible Artificial Neural Network can be expanded to use additional inputs, larger hidden layers and additional outputs when required; without losing already acquired learning. Finally, it provides the ability for the back-propagation algorithm to be updated. This enables a learning algorithm that uses the regularised TD learning algorithm applied to the artificial neural network back-propagation algorithm to be used.

The Generic, Self-Improving Predictor is required to convert a temporal sequence of inputs into predictions and adjust predictions using feedback. To achieve this, it produces vectors describing the probabilities for each of the specified number of desired outcomes that the current input is associated with. It uses a temporal sequence of inputs and sporadic, delayed feedback to improve those predictions. To achieve this, it uses a learner. To allow it to use multiple learners interchangeably, a Temporal Learning Interface was developed that each learner implements.

The Temporal Learning Interface defines four requirements: First, produce a prediction vector based on the current input state vector when required. Second, receive an input state vector describing the current state when provided. Third, receive a feedback signal: a vector describing the correct values. Fourth, receive a reset signal. The feedback signal is used to indicate that all the input state vectors (whether requested by the first requirement or provided through the second requirement) since the last feedback signal should have produced the prediction described by the current feedback signal. To allow for occasions when input state vectors since the last feedback signal will not be associated with the next feedback signal, the reset signal is used to indicate that all inputs since the last feedback should be ignored. Currently only direct output feedback has been implemented. To indicate that no foreseeable feedback is expected, the Temporal Learning Interface includes the option to disable learning to ensure that long, temporal input sequences are not being stored unnecessarily. The Temporal Learning Interface includes two basic learner parameters: the learning rate and the temporal discount rate. Finally, the size of the learner input and output vectors are adjustable.

The predictor can be configured online to use any learner that implements the Temporal Learning Interface. Currently three learners have been implemented. First, the MCMC Learner, which is based on the well known Markov Chain, Monte-Carlo algorithm. Second, the MC Network Learner, which uses a Markov Chain, to store the temporal sequence of inputs, and the Flexible Artificial Neural Network to enable the use of an analogue state space. Finally, the TD Network Learner, which uses the regularised TD learning algorithm applied to the artificial neural network back-propagation algorithm and implemented in the Flexible Artificial Neural Network.

Chapter 7

The Generic, Self-Improving Controller

7.1 *Introduction*

The normal approach when using machine learning to create a controller is to provide users with a number of gestures that they need to make; record multiple instances of each gesture using specific sensors; extract the relevant sensor data and pass it through a supervised learning algorithm until the algorithm can successfully identify the gestures; and map each gesture to a control signal that performs a desired outcome. Furthermore, the user must learn the gesture to control mapping.

Instead of providing users with the gestures that they need to learn, users want to decide what intuitive gestures they would like to use for each desired outcome. Furthermore, instead of learning offline and implementing a static controller, users want a controller whose fidelity improves with use; that learns from the user while being used. There should be no separate activity to extract the relevant data and repeatedly pass it through a supervised learning algorithm. The controller should continually predict desired outcomes and use feedback to continually improve those predictions.

This approach has two key advantages: First, users do not have to learn unintuitive gestures that have been mapped to control signals. Users select gestures that are intuitive and personal. They can even use multiple gestures for the same control signal. Second, feedback from incorrect predictions can be used to continually improve future predictions. This also enables the user to change the gesture associated with a particular desired outcome either actively or passively. As fidelity improves over time, users will be able to use more subtle gestures. Furthermore, there is a requirement that additional desired outcomes can be added without affecting existing learning.

The aim of this research was to create this Generic, Self-Improving Controller. The controller is expected to learn users' gestures without either the controller understanding what the source of the data is, or the user needing to learn pre-defined gestures to predict an increasing number of control signals. As illustrated in Figure 1.1, to create the Generic, Self-Improving Controller requires combining the source-independent hardware data capture, processing, fusion and storage module described in Chapter 3, the Generic, Self-Improving Predictor module described in Chapter 6, and using an implementation of the Blended Control Policy described in Chapter 5 to provide the controller output signals. For demonstration purposes, the output of the Generic, Self-Improving Controller is used to control the computer's mouse pointer.

Section 7.2 outlines the creation of the Generic, Self-Improving Controller. To manage the modules in the Generic, Self-Improving Controller, the Action Predictor GUI was created. Section 7.3

describes managing the Generic, Self-Improving Controller's modules through the Action Predictor GUI. Section 7.4 describes how the Action Predictor GUI is used to train the Generic, Self-Improving Controller's underlying Generic, Self-Improving Predictor. Section 7.5 describes the creation of a Cross-Platform Mouse Pointer Override System. Section 7.6 outlines the implementation of the Blended Control Policy in a Blended Mouse Control Policy that is built into the Action Predictor GUI and linked to the Cross-Platform Mouse Pointer Override System.

7.2 Creating the Generic, Self-Improving Controller

Due to the modular nature of the design, creating the Generic, Self-Improving Controller simply requires creating an application with the three modules contained in Figure 1.1 and controlling the flow of data between them. The three modules used are: the IMU Data Collector described in Chapter 3, the Generic, Self-Improving Predictor described in Chapter 6 and an implementation of the Blended Control Policy described in Chapter 5. Since the output generated by each module and the input required by each module has been designed to be a vector, controlling data flow simply involves passing vectors.

The IMU Data Collector uses a separate thread for collecting data from each IMU. When learning, the Generic, Self-Improving Controller uses an additional thread to periodically sample the data stored in the IMU Data Collector and pass it as a temporal sequence of inputs into the Generic, Self-Improving Predictor. Similarly, as will be described in Section 7.6, the Blended Mouse Control Policy runs in its own thread and periodically queries the Generic, Self-Improving Predictor for its current prediction. Querying the Generic, Self-Improving Predictor for its current prediction will generate a separate query to the IMU Data Collector for the current input. Finally, any feedback provided to the Generic, Self-Improving Controller is passed to the Generic, Self-Improving Predictor.

7.3 Managing the Generic, Self-Improving Controller

The modular nature of the design, makes creating and running the Generic, Self-Improving Controller simple. However, each module has numerous parameters that can be adjusted. To manage these parameters, the Action Predictor GUI shown in Figure 7.1, was created.

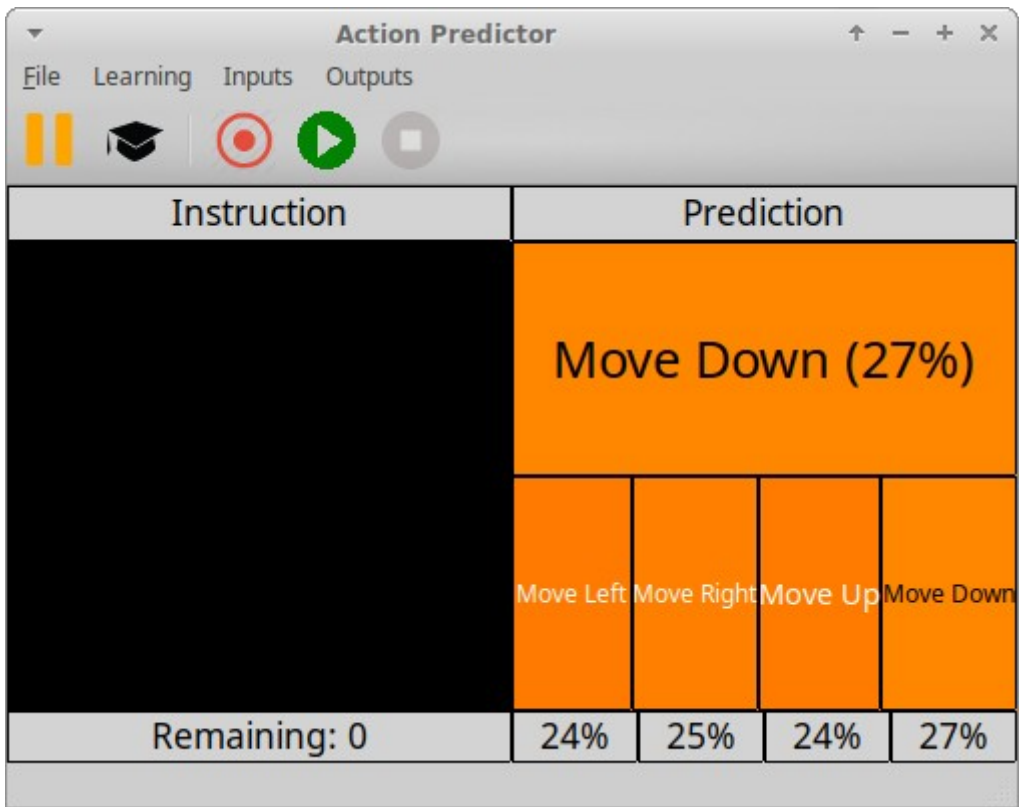


Figure 7.1: Action Predictor GUI used to manage and train the Generic, Self-Improving Controller

The left panel of Figure 7.1 provides instructions to the user when training the Generic, Self-Improving Controller. It includes an indicator for the number of instructions remaining in the current learning tranche. Training the Generic, Self-Improving Controller is covered in Section 7.4. The right panel of Figure 7.1 provides the current most likely predicted desired outcome for the current input; along with the predicted confidence levels of all the desired outcomes. The confidence levels are colour coded and range from red to green depending on the confidence level

required for a correct prediction. The confidence level required for a correct prediction depends on the number of desired outcomes. This provides a way of visualising the predictions in real-time, and it provides a sense of the state of the Generic, Self-Improving Predictor. The confidence levels should continually improve, because learning occurs while the controller is being used.

The Action Predictor GUI is used to simultaneously control all three modules of the Generic, Self-Improving Controller. Section 7.3.1 describes how the IMU Data Collector module is managed. Section 7.3.2 describes how the Generic, Self-Improving Predictor module is managed. Section 7.3.3 describes how the Blended Mouse Control Policy and the connection to the Cross-Platform Mouse Pointer Override System is managed.

7.3.1 Managing the IMU Data Collector

The IMU Data Collector is managed through the Inputs menu shown in Figure 7.2.

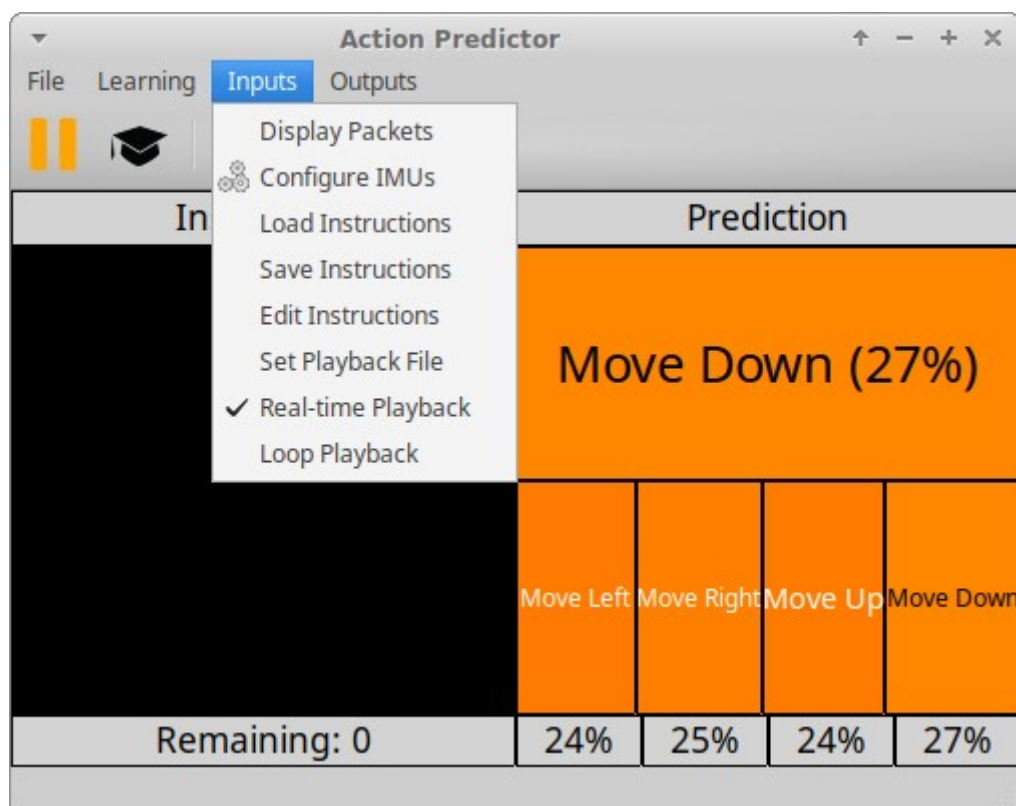


Figure 7.2: Action Predictor GUI's Inputs menu

During debugging, when the Action Predictor GUI is being run via a console, the data packets that are being received by the IMU Data Collector can be displayed to the console. The menu option Display Packets provides easy access to enable or disable this option. The Configure IMUs option presents the user with the IMU Data Collector control panel shown in Figure 3.22. Full details for managing the IMU Data Collector is provided in Chapter 3. The other Inputs menu's options are used for controlling the inputs used to train the Generic, Self-Improving Predictor and are described in Section 7.4.

7.3.2 Managing the Generic, Self-Improving Predictor

To manage the Generic, Self-Improving Predictor, the Learning menu shown in Figure 7.3 is used.

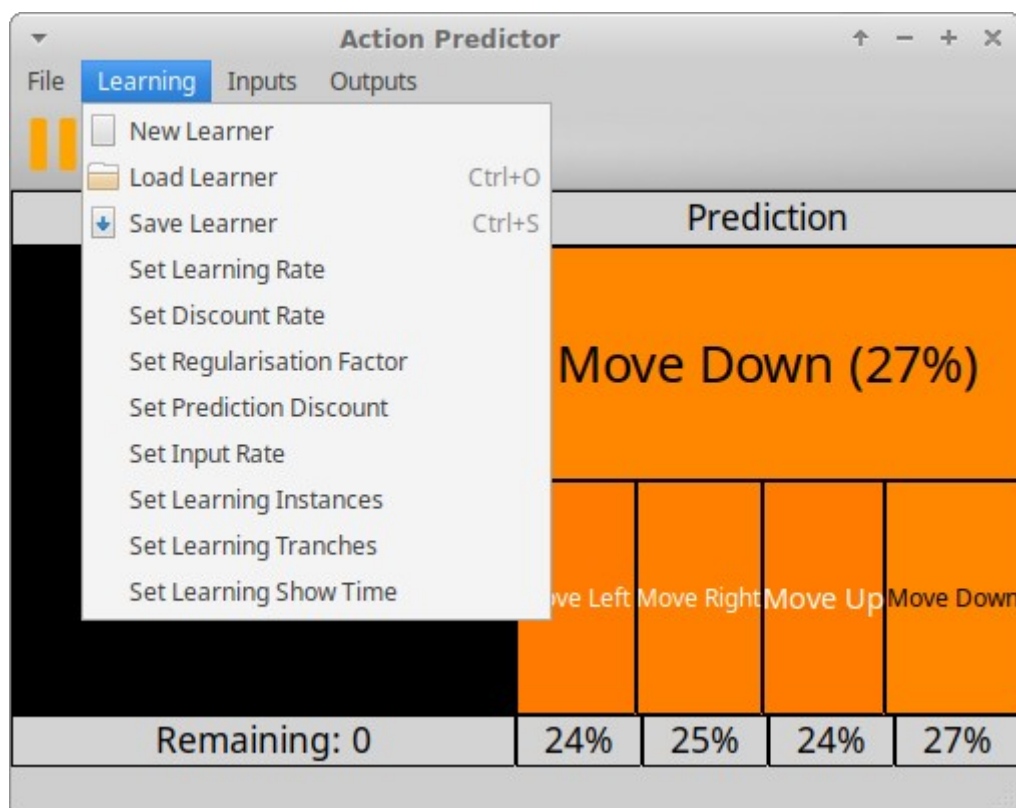


Figure 7.3: Action Predictor GUI's Learning menu

The New Learner menu option presents the user with the sequence of panels used to gather the parameters required to create a new learner. The first panel is shown in Figure 7.4. The first panel asks the user to select the type of Iterative Learner to be created. The types of learner available and their characteristics are described in Chapter 6.

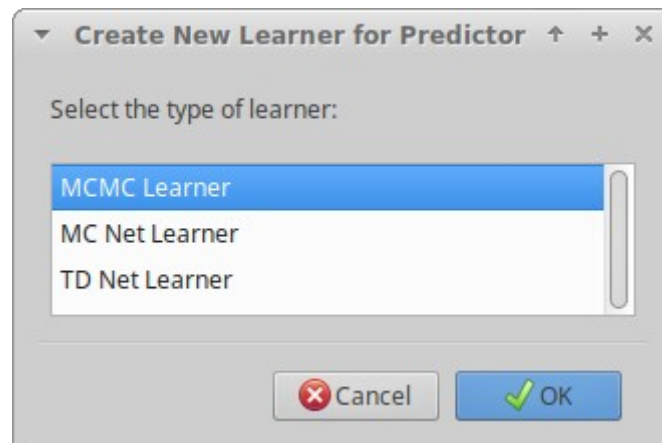


Figure 7.4: Action Predictor GUI's New Learner type selection panel

If a Flexible Artificial Neural Network based learner is selected, the user is prompted to provide the number and size of each of the Flexible Artificial Neural Network's hidden layers. The user can choose to bypass the specification of the Flexible Artificial Neural Network and use a default Flexible Artificial Neural Network. A default Flexible Artificial Neural Network has a single hidden layer with twice as many neurons as the size of the input state vector.

The size of the input state vector is determined by the IMU configuration. When additional IMUs are added or the history length is extended, the length of the input state vector is extended too. Similarly, when IMUs are removed, the length of the input state vector is truncated. This is done automatically. For the MCMC Learner this results in all learning being lost. However, the Flexible Artificial Neural Networks used in the MC Network Learner and the TD Network Learner attempt to save previous learning.

When the input state length is extended, new random weights are created. When the input state length is truncated, unused weights are deleted. Adding or removing weights will affect the output of the values of the layers above. However, the values of the weights that remained are kept. This ensures that existing learning is not lost.

Extending or truncating the input state vector adjusts the Flexible Artificial Neural Network based learners' input state length. However, the number of hidden layer's neurons is not changed.

Therefore, if a default Flexible Artificial Neural Network is to be used, the number of IMUs and the length of the history to be used should be specified first.

The Flexible Artificial Neural Network was designed to enable the number of neurons in each of the hidden layers to be likewise extended or truncated. However, the only way to adjust the number of neurons in each hidden layer via the Action Predictor GUI is by creating a new Iterative Learner. This is not a technical limitation, only a limitation in the current implementation.

Although the existing weights are kept, the new input state data values are inserted at their location in the input state vector and not just appended. Therefore, when the input state length is changed, the historical data entries kept for each sensor are lost, because subsequent values no longer align with their previous location. Again, this is not a technical limitation, only a limitation in the current implementation.

Likewise, when additional output options are added or removed, the output prediction vector is adjusted. Again, in the Flexible Artificial Neural Network based learners, when the output prediction length is extended, new random weights are created, and when the output prediction length is truncated, unused weights are deleted.

To preserve learning, the active Iterative Learner can be saved, and previously saved Iterative Learner can be loaded via the Save Learner and Load Learner Learning menu's options. Loading a

learner with a different input state vector or output state vector than the Action Predictor GUI is currently configured for, will also result in the loaded learner's input and output lengths being adjusted automatically.

The learning parameters available to all the learners can be configured via the next four Action Predictor GUI Learning menu's options: Set Learning Rate, Set Discount Rate, Set Regularisation Factor and Set Prediction Discount. If an attempt to modify a parameter, that is not available to the current Iterative Learner, the user will be notified with a simple warning. For example, attempting to modify the TD learning prediction discount rate or lambda parameter, when using the MCMC learner or MC Network Learner, will result in the warning shown in Figure 7.5.

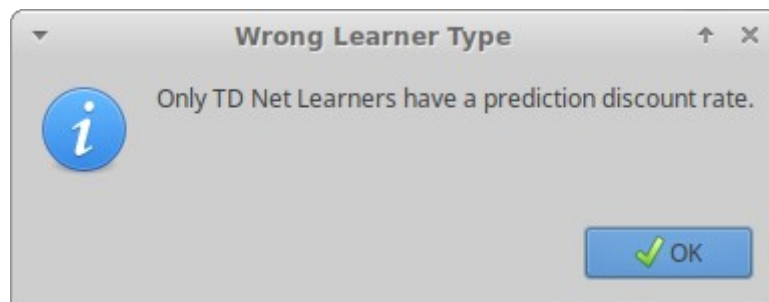


Figure 7.5: Warning presented when attempting to modify the TD algorithm's prediction discount rate when not using a TD learning algorithm

The next menu item enables setting the input rate. The input rate controls the period of the Generic, Self-Improving Predictor's temporal sequence data i.e. the frequency that the IMU Data Controller is sampled at. The period affects the temporal sequence of input data, that is passed to the Generic, Self-Improving Predictor.

The remaining Action Predictor GUI Learning menu's options are used for managing the training of the underlying Generic, Self-Improving Predictor and are discussed in Section 7.4.

7.3.3 Managing the Blended Mouse Control Policy

To manage the Blended Mouse Control Policy described in Section 7.6, the Outputs menu shown in Figure 7.6 is used.

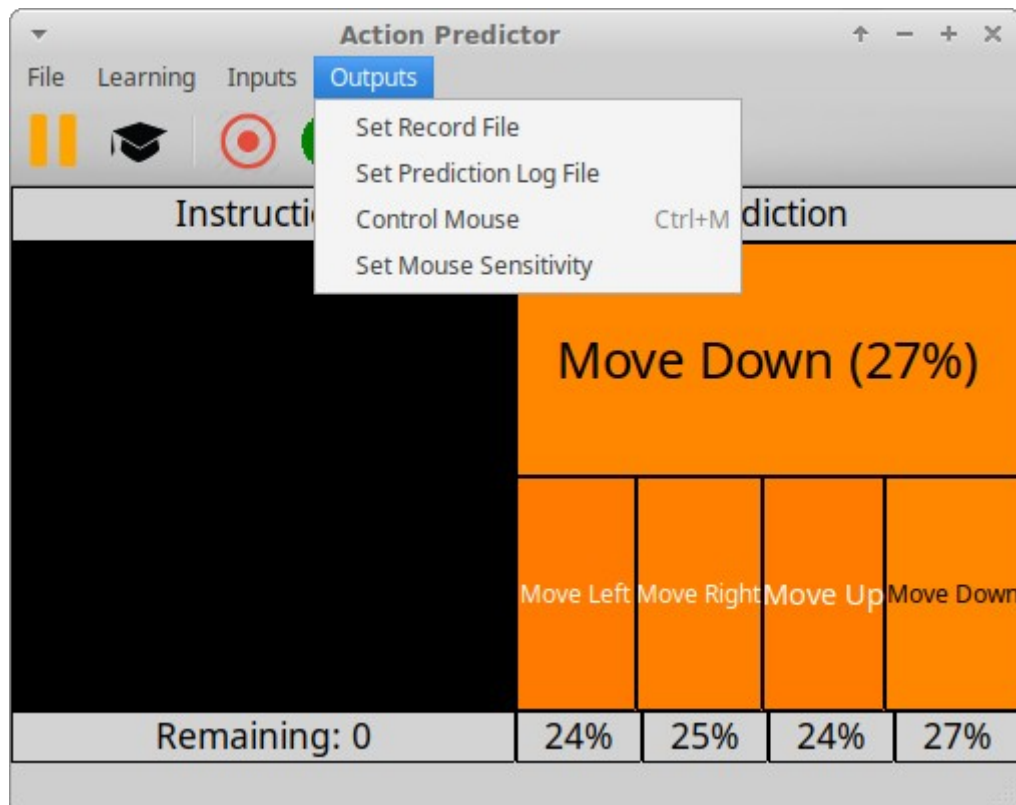


Figure 7.6: Action Predictor GUI's Outputs menu

The first two options of the Action Predictor GUI's Outputs menu are used for recording output generated during training of the underlying Generic, Self-Improving Predictor and are discussed in Section 7.4.

The Action Predictor GUI's Outputs menu's Control Mouse option is used to connect and disconnect the output of the Generic, Self-Improving Controller to the Cross-Platform Mouse Pointer Override System described in Section 7.5. As shown on the menu, the Action Predictor GUI is also configured with a short-cut key combination: Ctrl-M, which will also connect and disconnect

the output of the Generic, Self-Improving Controller to the Cross-Platform Mouse Pointer Override System.

The only parameter available to the Blended Mouse Control Policy is the Mouse Sensitivity, which is managed via the Output menu's Set Mouse Sensitivity option. The Mouse Sensitivity option controls the prediction sample period, and is described in Section 7.6.

7.4 Training the Generic, Self-Improving Controller

To enable users to train the Generic, Self-Improving Controller, the Action Predictor GUI also provides the tools for training the underlying Generic, Self-Improving Predictor.

The Generic, Self-Improving Controller can be configured with any list of instructions. The list of instructions should be a list of desired outcomes. Desired outcomes are the actions the device connected to the output of the Generic, Self-Improving Controller should perform. They should be a list of desired outcomes, because this allows the user to select the gestures they want to use: the gestures they find most intuitive. However, the list can also contain a list of gestures the user is expected to make. The user is then expected to learn the gesture to desired outcome mapping. During training, the list of instructions are presented to the user in the left panel of the Action Predictor GUI shown in Figure 7.1.

The third, fourth and fifth options of the Action Predictor's Inputs menu, shown in Figure 7.2, enable the list of instructions to be edited, saved and loaded. The simple text editor shown in Figure 7.7 was created to edit the list of instructions.

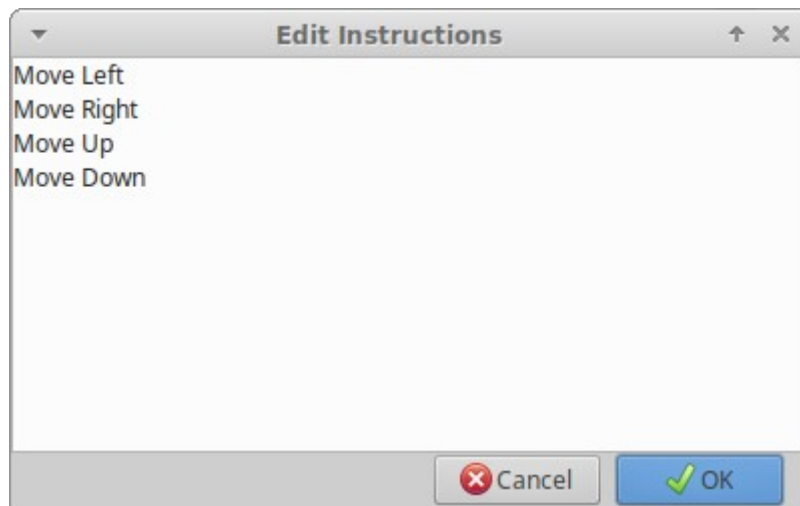


Figure 7.7: Action Predictor GUI's Instruction Editor

Changing the words or phrases used, while keeping the number of desired outcomes or gestures constant will not affect the Generic, Self-Improving Predictor. However, if the number of items in the list is increased or decreased, the number of outputs from the underlying learner will be updated automatically. Adjusting the number of outputs will have the same effect as adjusting the length of the input state vector. As discussed in Section 7.3.2, when modifying the number of IMUs or the historic data lengths, the length of the input state vector is automatically adjusted. The MCMC Learner will lose all learning. The Flexible Artificial Neural Networks used in the MC Network Learner and the TD Network Learner will attempt to save previous learning.

Learning is broken into tranches. At the end of each tranche, the Generic, Self-Improving Predictor is tested. The results of the testing are stored in a prediction log file. When the configuration of the IMU Data Collector or the underlying Iterative Learner are changed, the configurations are also saved into the prediction log file to enable associating the learning achieved with the configuration used. The prediction log filename can be changed via the Action Predictor GUI's Outputs menu's Set Prediction Log File menu item shown in Figure 7.6. During real-time learning, the end of a

learning tranche also provides a convenient breakpoint, when learning can be paused and restarted later without impacting the temporal state of the underlying Iterative Learner.

The number of learning tranches, the number of learning instances in each tranche and the period of time given to the user to perform the gesture during each instance are configured using the final three menu options of the Action Predictor GUI's Learning menu shown in Figure 7.3.

Although the Generic, Self-Improving Controller was initially designed to learn while being used, for testing purposes, it also became necessary to be able to record and replay the learning data using different learning parameters. Furthermore, the Action Predictor GUI needed to be able to use IMU data recorded by other members of the research group and their various means of recording the desired outcomes or gestures associated with the recorded data. This required additional learning modes to be developed. To control the learning mode, the Action Predictor GUI's File menu shown in Figure 7.8 is used.

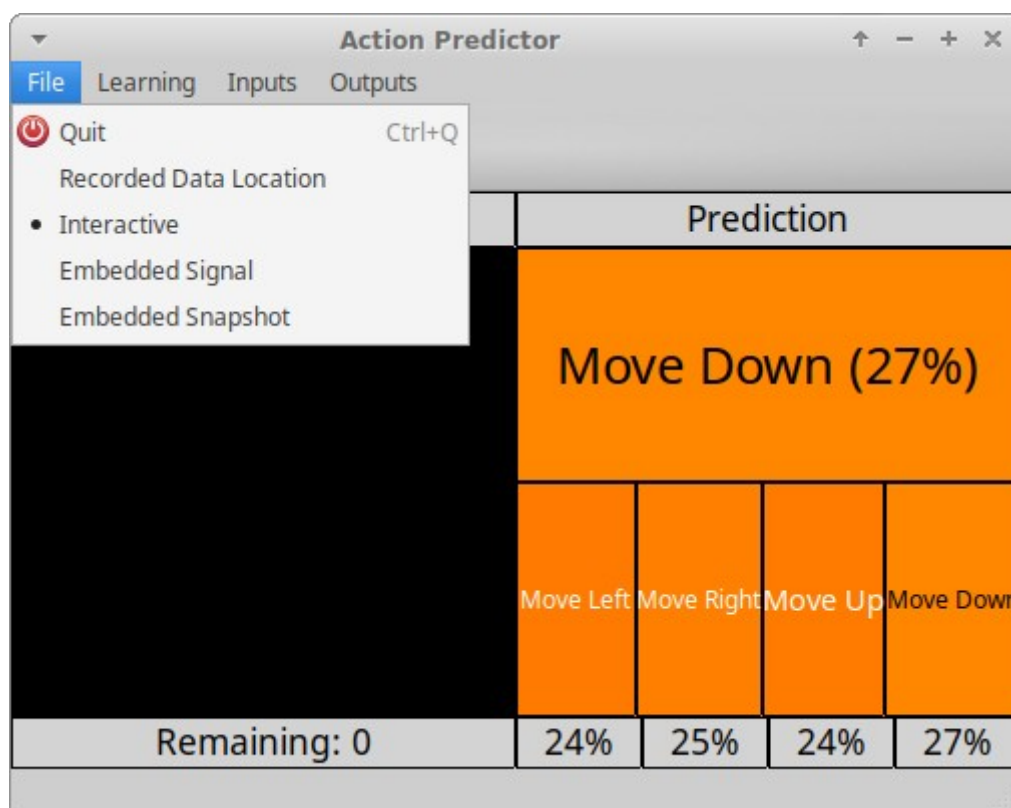


Figure 7.8: Action Predictor GUI's File menu

The Interactive learning mode is used for real-time learning. Users interact with the Action Predictor GUI in real-time. They are presented with instructions, and the underlying Generic, Self-Improving Predictor uses the data captured by the IMU Data Collector to iteratively improve the accuracy of the predictions while the Generic, Self-Improving Controller is being used. Although typically the output of the controller will be disconnected.

The first additional learning mode is the Embedded Signal mode. In this mode the IMU data was recorded with an additional value to represent when the user was asked to perform the required gesture. When reading the recorded IMU data file, this additional value is read and sent as a separate input signal. The filenames are used to identify which gesture the user was being asked to perform. Each file contains multiple instances of each gesture, and there are multiple files for each gesture. The Embedded Signal mode uses one file from each gesture to create a learning tranche. With the gesture instances in each file providing the individual learning instances. The file used for a learning instance is changed at the end of each learning instance. The next file used is selected at random from the list of files in that tranche that still contain gestures that have not been presented.

The files associated with a particular test subject are stored in individual folders. The folder containing the individual test subject's recorded data that will be used during learning needs to be selected. The folder that contains the files to be used is selected using the Action Predictor GUI's File menu's Recorded Data Location item. When switching to Embedded Signal mode or selecting a new folder for the location of the recorded data, the folder is scanned. The filenames are used to identify the number of gestures and number of tranches for each gesture. The files are then scanned to identify the number of gesture instances in each file. The minimum number of gestures, tranches and instances are then used for the learning parameters.

To record and replay any data, a generic logger and log playback utility combo where created. To ensure recorded data is not lost, the logger will automatically increment the number suffixed to the

filename any time a new logger with the same filename is created. The logger will record any single line of characters and prefix the line with a time stamp. The timestamp is the number of milliseconds since the logger was started. The timestamp can either be specified manually or calculated from the time logging was started. Manually timestamping the data is required when the data is not generated in real-time. Furthermore, the logger can be paused. The real-time data stamp will then reflect the time since the logging was started less any time the logger was paused. The logger has also been created to be thread safe. Therefore, the multiple threads running in the Generic, Self-Improving Controller can add entries to the logger without them overwriting each other.

The log playback utility will read the logfile and create the events that reflect the log entries using a simple lookup. The log playback utility can be extended to add events associated with log entries. Furthermore, the log playback includes an event that can be used to create new pseudo entries in the log file being read i.e. to generate delayed events. Finally, the log playback utility can be run in real-time or as fast as possible.

The Action Predictor GUI application extends the generic logger and playback utility combo. All the input received by the IMU Data Collector, as well as all the signals generated during training and testing, are recorded. During playback the same learning and input signals can be sent to a new learner with different settings. The same learning and input signals can be sent to a new learner in either real-time or as fast as possible. The same learning and input signals can also be sent multiple times. Finally, since the input to the IMU Data Collector is recorded, the IMU Data Collector can be reconfigured using the same input data to generate different output i.e. different input can be presented to the Generic, Self-Improving Predictor. Different data, but still associated with the same learning signals. This not only enables the sensors to be disabled, but, more importantly, for the history level for each sensor to be changed.

Although the logger will automatically create a new log file every time the logger is started, the log filename can be specified via the first option of the Action Predictor GUI's Outputs menu's Set Record File menu option. Similarly, although the log playback will automatically use the logger's default log filename, the log playback log filename can be specified using the Action Predictor GUI's Inputs menu's Set Playback File menu option. The speed of playback and the number of times the playback is looped are configured using the final two menu options of the Action Predictor GUI's Inputs menu.

To ensure that data is only logged during learning, the logger is automatically paused when it is created. The logger will be restarted at the correct time during learning as described below.

Starting, pausing, restarting and stopping the recording to the selected log file and starting, pausing, restarting and stopping the playback of the last or selected log file is done using the buttons on the right half of the Action Predictor GUI's toolbar. The toolbar automatically updates to reflect the current state of the logger and the log playback as shown in Figure 7.9.

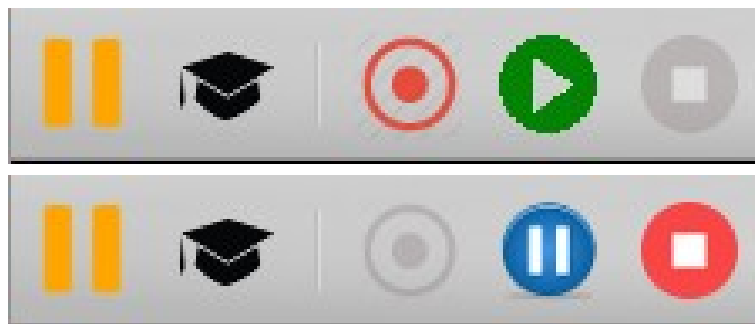


Figure 7.9: Action Predictor GUI's toolbar showing it updating to reflect the current state

Similarly, starting, pausing, restarting and stopping learning, is controlled using the left half of the Action Predictor GUI's toolbar.

To simplify the training irrespective of the learning mode, a simple state machine was created. The state machine consists of ten states: Learning Initialisation, Learning Tranche Initialisation, Countdown Initialisation, Learning Step Initialisation, Learning, Testing Initialisation, Testing Step Initialisation, Testing, and Done. The state machine enables the learning methodology to be consistent irrespective of the learning mode. The state machine model also enables the controller loop to yield thread control when switching between states or repeating states. Yielding thread control quickly ensures the application remains responsive to new control signals.

When learning starts, the Learning State is switched to the Learning Initialisation state. In the Learning Initialisation state the initial settings required by the selected learning mode are set according to the learning parameters previously configured. The Learning State is then switched to the Learning Tranche Initialisation state.

In Interactive learning mode, at the end of a learning tranche, the user has the option of pausing the learning. When learning is restarted, the Learning State is also switched to the Learning Tranche Initialisation state.

The first thing the Learning Tranche Initialisation state does is check if there are any more learning tranches. If there are no more learning tranches, the Learning State is switched to the Done state. If there are more learning tranches, the Learning Tranche Initialisation state initialises the learning tranche depending on the learning mode. In Interactive learning mode, the Learning State is simply switched to the Countdown Initialisation state. In Embedded Signal learning mode, the set of files used for the learning tranche are changed in the Learning Tranche Initialisation state. In Embedded Signal learning mode the state is switched directly to the Learning Step Initialisation state.

A countdown is provided in Interactive learning mode at the start of each learning tranche. The countdown provides users time between clicking the start learning button and the first instruction being presented. The countdown between tranches also provides an opportunity to click the pause

between without affecting the learning signals being sent to the underlying Iterative Learner. The countdown is skipped when using recorded data.

The Initialise Countdown state simply sets the length of the countdown and switches the state to the Countdown state. In the Countdown state, the Instruction Panel, the left-hand side of the Action Predictor GUI in Figure 7.1, is set to display the number of seconds remaining before the learning tranche is started. The countdown is decremented and then the learning loop thread is suspended for a second. When the learning loop restarts it will check if any control signals were received. If no control signals were received the Learning State machine will rerun the Countdown state. When the countdown reaches zero, the logger is restarted (if logging is enabled), and learning is enabled in Generic, Self-Improving Predictor. Finally, the Learning State is set to the Learning Step Initialisation state.

The Learning Step Initialisation state is used to set the random instruction to be displayed to the user. In Embedded Signal mode, the Learning Step Initialisation state is also used to switch randomly to the gesture file to be used for the next learning step. If all the learning instances have been completed the Learning State is switched to the Testing Initialisation state, otherwise the Learning State is switched to the Learning state.

In the Learning state, the sequence of events depends on the learning mode. In Interactive learning mode, the randomly selected instruction is presented to the user in the Instruction Panel. The display time is decremented and the learning loop thread is suspended for a second. As with the Countdown state, when the learning loop restarts it will check if any control signals were received. If no control signals were received the Learning State machine will rerun the Learning state, until the display time reaches zero. When the display time reaches zero, the selected instruction index is sent to the Generic, Self-Improving Predictor as feedback to indicate the desired correct prediction. Finally, the Learning State is set to Learning Step Initialisation state again.

In Embedded Signal mode, the sequence of events in the Learning state depends on whether or not no signals are being used. If no signals are not used, the message “No Signal” is presented in the Instruction Panel until the separate embedded signal switches on. When the embedded signal switches on, the logger is restarted; the Generic, Self-Improving Predictor learning is enabled; and the randomly selected instruction is presented in the Instruction Panel. When the embedded signal switches off again, the index for the randomly selected gesture is sent to the Generic, Self-Improving Predictor as the correct prediction; the message “No Signal” is presented again; the Generic, Self-Improving Predictor learning is stopped again; and the logger is paused again.

The option to use the no signal as an additional controller outcome has also been programmed. When this option is enabled, an additional outcome is added to the beginning of the list of instructions: Do nothing. When the embedded signal switches on or off the logger is not restarted or paused. Similarly, the Generic, Self-Improving Predictor learning is not started and stopped. Instead, when the embedded signal switches on, the index for Do nothing desired outcome is sent to the Generic, Self-Improving Predictor as the desired correct prediction.

In the Learning Step Initialisation state, if all the learning instances have been completed the Learning State is switched to the Testing Initialisation state. In the Testing Initialisation state, the Generic, Self-Improving Predictor learning is stopped; and the prediction meta data is recorded to the prediction log. Finally, the desired outcome testing index is set to zero, and the Learning State is switched to Testing Step Initialisation.

The Testing Step Initialisation state, first checks whether any more desired outcomes need to be tested. If not, the learning tranche is incremented and the Learning State is switched to the Learning Tranche Initialisation state. In Embedded Signal mode, the Testing Step Initialisation state also selects the correct gesture file to be used. Finally, the Learning State is switched to the Testing state.

The Testing state, is very similar to the Learning State. In Interactive learning mode, the desired outcome instruction to be tested is presented to the user in the Instruction Panel, the display time is decremented and the learning loop thread is suspended for a second. When the learning loop thread restarts, it will check if any control signals were received. If no control signals were received the Learning State machine will rerun the Testing state. When the display time reaches the testing trigger point, the Generic, Self-Improving Predictor is queried for its prediction. The prediction is recorded in the prediction log, and the learning loop thread is suspended again. When the display time reaches zero, the Learning State is set to the Testing Step Initialisation state again.

Similarly, in Embedded Signal mode, the message “No Signal” is presented in the Instruction Panel until the separate embedded signal switches on. The gesture to be tested instruction is presented to the Instruction Panel, and the trigger timer is started. When the timer reaches the testing trigger point, the Generic, Self-Improving Predictor is queried for its prediction, and the prediction is recorded in the prediction log. When the embedded signal switches off again, the index for the desired outcome to be tested is incremented; the message “No Signal” is presented again; and the Learning State is set to the Testing Step Initialisation state again.

Finally, in the Done state, the Instruction Panel is presented with the word “Done”.

7.5 Cross-Platform Mouse Pointer Override System

A Cross-Platform Mouse Pointer Override System provides a simple interface for overriding the operating system’s mouse pointer controller. The operating system’s mouse pointer controller takes control signals from the mouse and moves the mouse pointer’s location on a computer screen. Similarly, it sends click signals when the mouse buttons are pressed and released. Since each operating system has it’s own bespoke way of managing the mouse, the challenge was to create a

simple interface for sending new signals to control the mouse pointer irrespective of the underlying operating system. The result is a simple way for any application to control the mouse pointer.

The Cross-Platform Mouse Pointer Override System interface allows the mouse pointer to be moved in any direction by simply passing the number of pixels to move in that direction. The interface also supports the other typical mouse events: left and right-clicking. To achieve this, it provides a simple interface for doing four things: move the mouse a number of pixels in the left (negative) or right (positive) direction; move the mouse a number of pixels up (negative) or down (positive) directions; a left click (both a mouse button press and a mouse button release) at the current location; and, finally, a right-click at the current location.

To control the location of the mouse pointer or send click signals requires building event objects and sending them to the input system. The structure of the event object on each operating system is different. Therefore, as with the creation of the IMU Type Independent Data Capture Software Interface described in Section 3.2, the code adapts to the operating system the code is being compiled for.

The Windows SDK provides a hook to the operating system's input system to which mouse event signals can be sent. However, Linux based operating systems, including Apple, allow for multiple, generic input streams. Therefore, they require either creating a new input stream, or finding an existing input stream and inserting addition signals into it. To create a fully functioning, independent mouse pointer controller would require creating and implementing a new input stream. However, for demonstration purposes, an existing input stream is used. Currently, the first input stream found that receives mouse events is used. If no stream is found, the Cross-Platform Mouse Pointer Override System is disabled.

When the Cross-Platform Mouse Pointer Override System receives the simple pixel and click signals described above, it converts them into the event objects required by the operating system.

On Windows the event objects are sent to the Windows input interface. On Linux based operating systems the events are inserted into the selected input stream.

7.6 *Blended Mouse Control Policy*

The Blended Control Policy described in Section 5.7 was implemented in the Blended Mouse Control Policy that uses the output of the Generic, Self-Improving Controller described in Section 7.2 to send signals to the Cross-Platform Mouse Pointer Override System described in Section 7.5.

Ideally, the module that implements the control policy should be separate from the both the Generic, Self-Improving Controller and the device driver it is controlling (in this case the Cross-Platform Mouse Pointer Override System). However, since the Blended Mouse Control Policy implementation was created for demonstration purposes only, the Blended Mouse Control Policy is built into the Action Predictor GUI that is used to manage the Generic, Self-Improving Controller and train the underlying Generic, Self-Improving Predictor; as described in Sections 7.3 and 7.4 respectively.

The Blended Mouse Control Policy runs in a separate thread. When enabled, it periodically queries the Generic, Self-Improving Controller for its current prediction based on the current input provided by the IMU Data Collector. It has been configured to use the first four outputs to predict whether to move the mouse pointer left, right, up and down respectively. If there are less than four outputs, but more than two outputs, it has been configured to use the two outputs to predict whether to move the mouse pointer left or right respectively. If there are more than four outputs, the additional outputs are ignored. If there is only one output it is also ignored.

As with most mouse pointer controllers, the sensitivity of the Blended Mouse Control Policy can be adjusted. Normally this is done by adjusting the size of the step. However, to support maximum sensitivity, the size of the step is kept to a minimum: one pixel. Instead, the period between which

steps are taken is adjusted. The period between steps is the period between queries to the Generic, Self-Improving Controller for its current prediction.

As detailed in Chapter 9, by using the Blended Control Policy to blend the predictions for moving the mouse pointer left or right and blend the predictions for moving the mouse pointer up or down, it is possible to control the mouse pointer with very little training.

Obviously, if the Generic, Self-Improving Controller has not been adequately trained, it may be difficult to point the mouse to the menu and the menu option described in Section 7.3.3. Therefore, it can also be disabled (and enabled) using a the Ctrl-M shortcut key combination.

7.7 Summary

The Generic, Self-Improving Controller combines an implementation of the source independent hardware data capture, processing, fusion and storage module, the Generic, Self-Improving Predictor and an implementation of the Blended Control Policy. The Action Predictor GUI manages the Generic, Self-Improving Controller implementation that uses the IMU Data Collector implementation of the source independent hardware data capture, processing, fusion and storage module and the Blended Mouse Control Policy implementation of the Blended Control Policy. The Action Predictor GUI also manages the training of the Generic, Self-Improving Controller.

The Action Predictor GUI enables both real-time data and recorded data to be used. The Action Predictor GUI will also record all the learning signals used to enable the same data to be used with different learning settings, different Iterative Learners, and even different IMU Data Collector settings. The playback of recorded data can be done in real-time or as fast as possible, which enables rapid testing of different learning settings, learners and input data capture settings. The playback of recorded data can also be looped to simulate the capture of more data. However, this will reduce the generalisation ability of the learner.

Finally, the Action Predictor GUI enables connecting the output of the Generic, Self-Improving Controller to a new Cross-Platform Mouse Pointer Override System to enable a user to use their chosen gestures to control the computer mouse pointer.

Chapter 8

Testing the Generic, Self-Improving Predictor

8.1 *Introduction*

At the core of the Generic, Self-Improving Controller is the Generic, Self-Improving Predictor. The Generic, Self-Improving Predictor can use any of the Iterative Learners created. The artificial neural network based Iterative Learners use the Flexible Artificial Neural Network.

The correct functioning of the Generic, Self-Improving Controller requires the correct functioning of the Generic, Self-Improving Predictor and its components. Although not necessarily directly transferable, testing the components independently also provides suggestions for the correct

parameters to be used when incorporating the components into Generic, Self-Improving Predictor. Similarly, testing the Generic, Self-Improving Predictor independently provides suggestions for the correct parameters to be used when incorporating it into the Generic, Self-Improving Controller.

Section 8.2 describes how the Flexible Artificial Neural Network was tested using well-known supervised learning. Section 8.3 describes the testing of each of the Iterative Learners; including the Iterative Learners that incorporate the Flexible Artificial Neural Network. Section 8.4 describes the creation of the generic game interface that provides the controlled simulated environments used to test the Generic, Self-Improving Predictor. Section 8.5 describes testing the Generic, Self-Improving Predictor by teaching it to play Tic-Tac-Toe. Finally, Section 8.6 describes testing the Generic, Self-Improving Predictor by teaching it to play Backgammon.

8.2 *Testing the Artificial Neural Network*

The Flexible Artificial Neural Network was trained on the standard Modified National Institute of Standards and Technology (MNIST) handwritten digit database⁴³. The MNIST database is used, because it is the standard database used for testing learning techniques and pattern recognition methods on real-world data. The MNIST database contains 60,000 training samples and 10,000 test samples. Each sample consists of 400 values: 20x20 grey-scale hand-written numerals; and one of 10 potential outputs: 0 – 9. To test the functionality (and not the effectiveness) of the neural network, only 5,000 samples were used. Furthermore, only the training error was measured.

43 (LeCun, Cortes & Burges, n.d.)

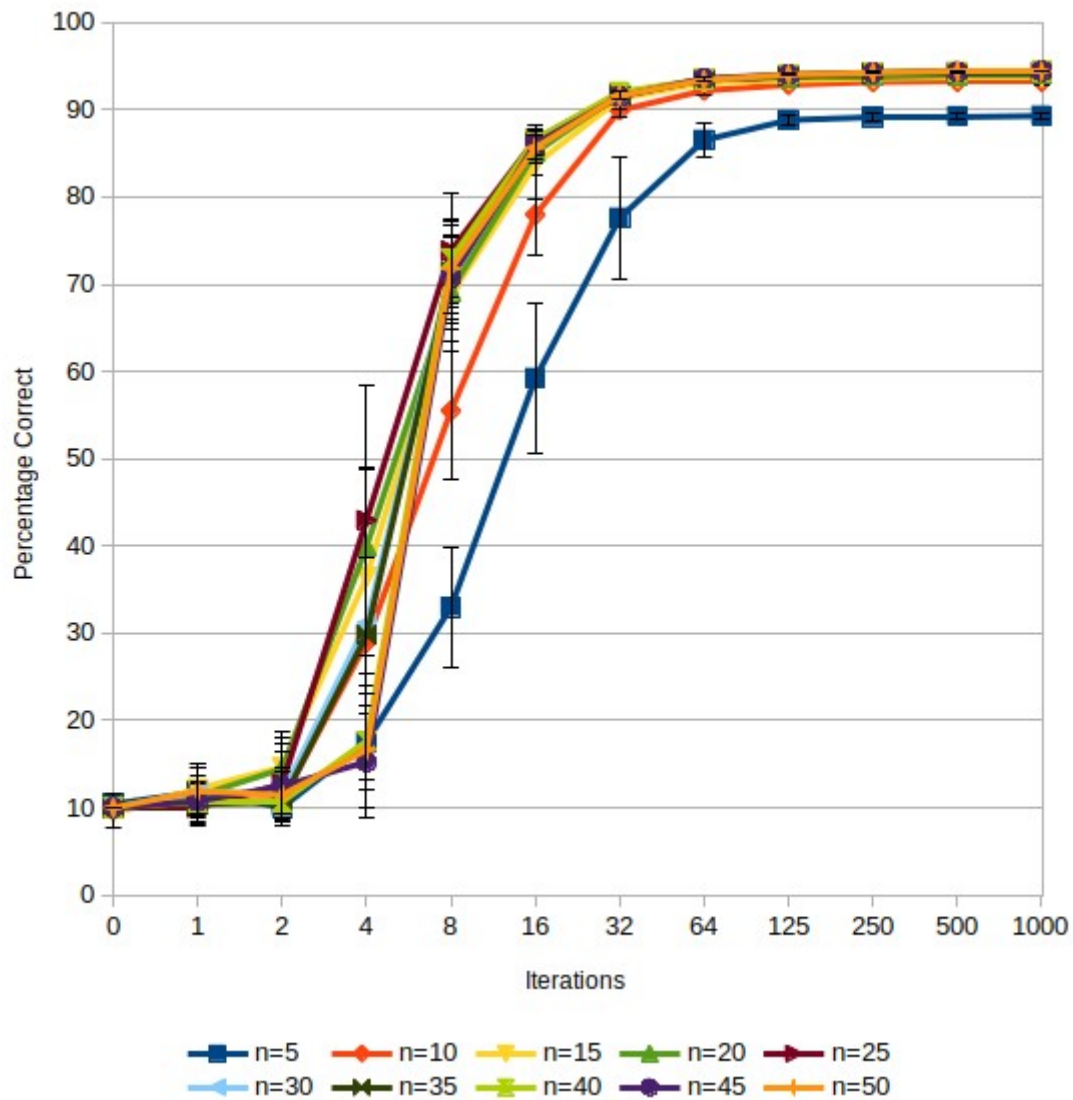


Figure 8.1: Impact of varying the number of hidden neurons (n) when batch training an artificial neural network to classify ten 20x20 grey-scale hand-written numerals from 5000 samples

For the initial tests of the Flexible Artificial Neural Network a batch trainer was created. A batch trainer based on the popular conjugate gradient descent method was used to enable the results to be compared with other standard, supervised learning tools⁴⁴. Figure 8.1 shows the results of batch training the Flexible Artificial Neural Network with different numbers of hidden neurons. The results are averaged across 10 simulations using a regularisation factor of 10 to prevent over fitting.

⁴⁴ (Wright & Nocedal, 1999)

There are a two things to note. First, there is very little difference in performance above 20 hidden neurons. Second, except for the network with only 5 hidden neurons, there is very little if any improvement after around 50 iterations. After 50 iterations, the performance is already over 90%. After 1,000 iterations, the best performance was just 94.4%.

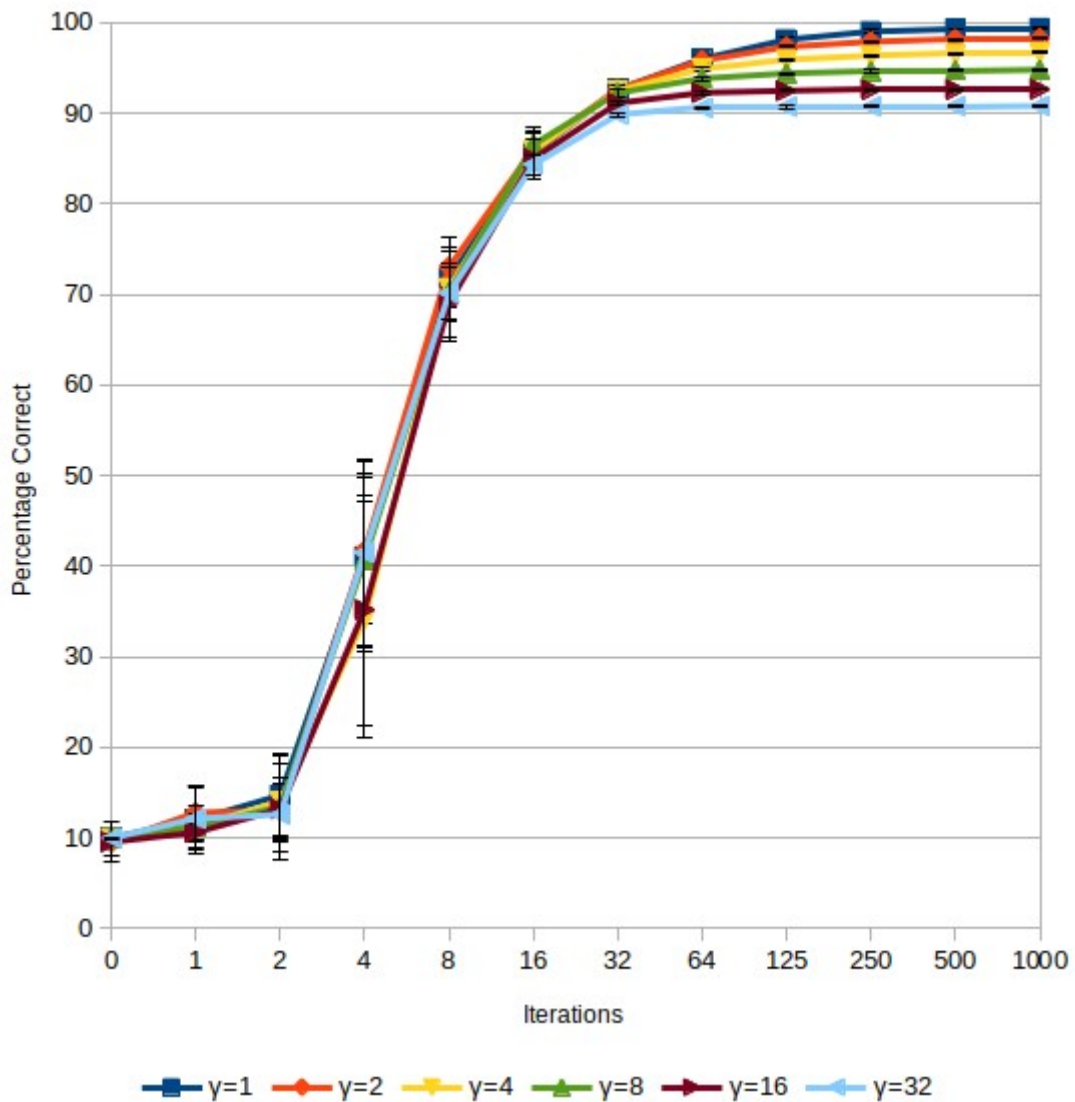


Figure 8.2: Impact of varying the regularisation factor (γ) when batch training an artificial neural network with 20 hidden neurons to classify ten 20x20 grey-scale handwritten numerals from 5000 samples

The reason for not achieving 100% classification of the training is due to the use of regularisation. The results presented in Figure 8.1 were achieved with a regularisation factor of 10. The results presented in Figure 8.2 show the impact of varying the regularisation factor when batch training the Flexible Artificial Neural Network with 20 hidden neurons averaged over 10 simulations. With a low regularisation factor, nearly 100% classification is achieved on the training data. As the regularisation factor is increased, the classification rate reduces. Determining the correct regularisation factor would require testing against separate data.

Iterative learners do not use batch training. Instead, learning takes place as each new sample is received. Ideally, each sample should be independent and identically distributed. However, not only will the samples be reused with each iteration, the samples are grouped together. Both of these factors will impact the learning, but at this point, the requirement is to test the applicability of the approach and compare the impact of varying the learning rate and the number of neurons in the hidden layer.

The Flexible Artificial Neural Network was configured with the twenty hidden neurons identified as the minimum number required to effectively classify the 20x20 grey-scale hand-written numerals. Figure 8.3 highlights the impact of varying the learning rate of the Flexible Artificial Neural Network averaged over 10 simulations. The most important point to note is that there is an optimal learning rate. A learning rate of 0.001 resulted in learning to classify the ten 20x20 grey-scale hand-written numerals in nearly the same number of iterations as the batch trainer.

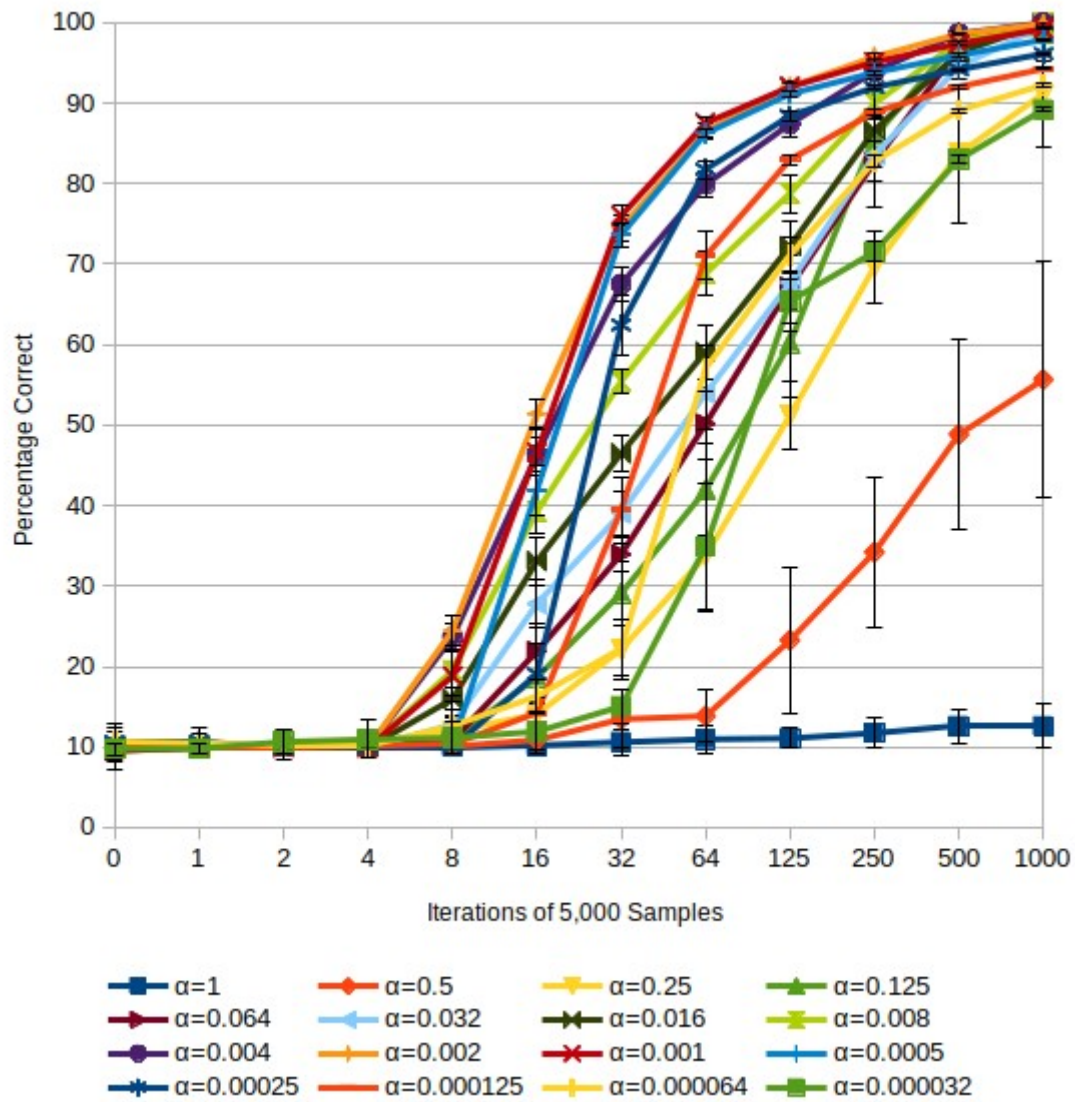


Figure 8.3: Impact of varying the learning rate (α) when using an artificial neural network with 20 hidden neurons to learn to classify ten 20x20 grey-scale hand-written numerals one image at a time

To ensure that the assumption that twenty hidden units is appropriate for classifying 20x20 grey-scale hand-written numerals when learning iteratively, the number of hidden neurons was varied while keeping the best learning rate identified of 0.001. Figure 8.4 shows the impact of varying the number of hidden neurons on the Flexible Artificial Neural Network's ability to classify the hand-written numerals averaged over 10 simulations. Once again, there is very little difference in performance above 20 hidden neurons. This supports the previous result; with the possible

exception that learning is slower at first with more hidden neurons. This is to be expected, because more weights need to be learnt.

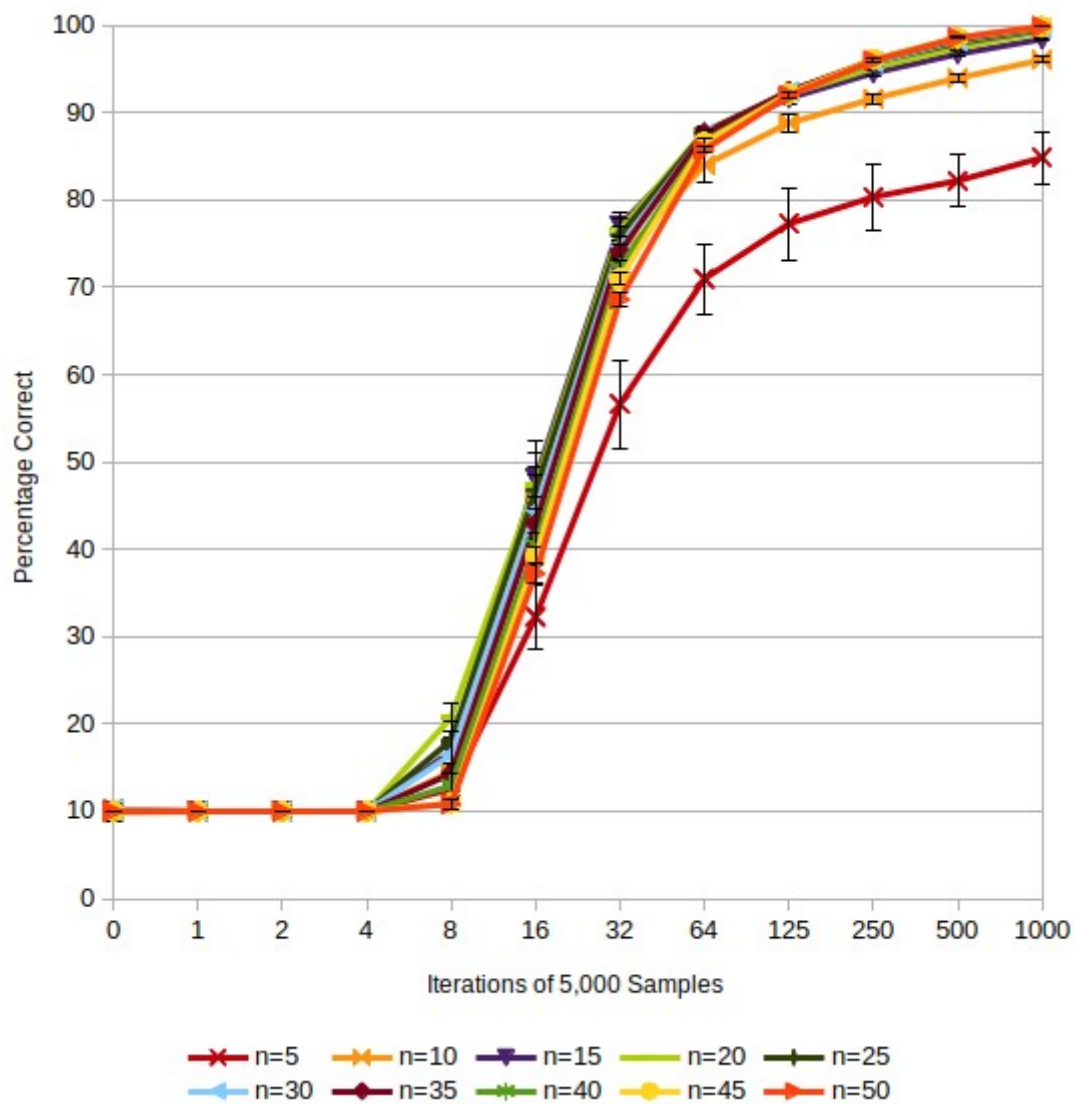


Figure 8.4: Impact of varying the number of hidden neurons (n) when using an artificial neural network with a learning rate of 0.001 to learn to classify ten 20x20 grey-scale hand-written numerals one image at a time

One issue with the results presented in Figures 8.3 and 8.4 is that, for learning rates of and below 0.125 and with more than 15 hidden neurons, classification of nearly 100% is achieved. Since

testing is being done against the training data, this suggests there is over-fitting taking place. To address this, regularisation is used.

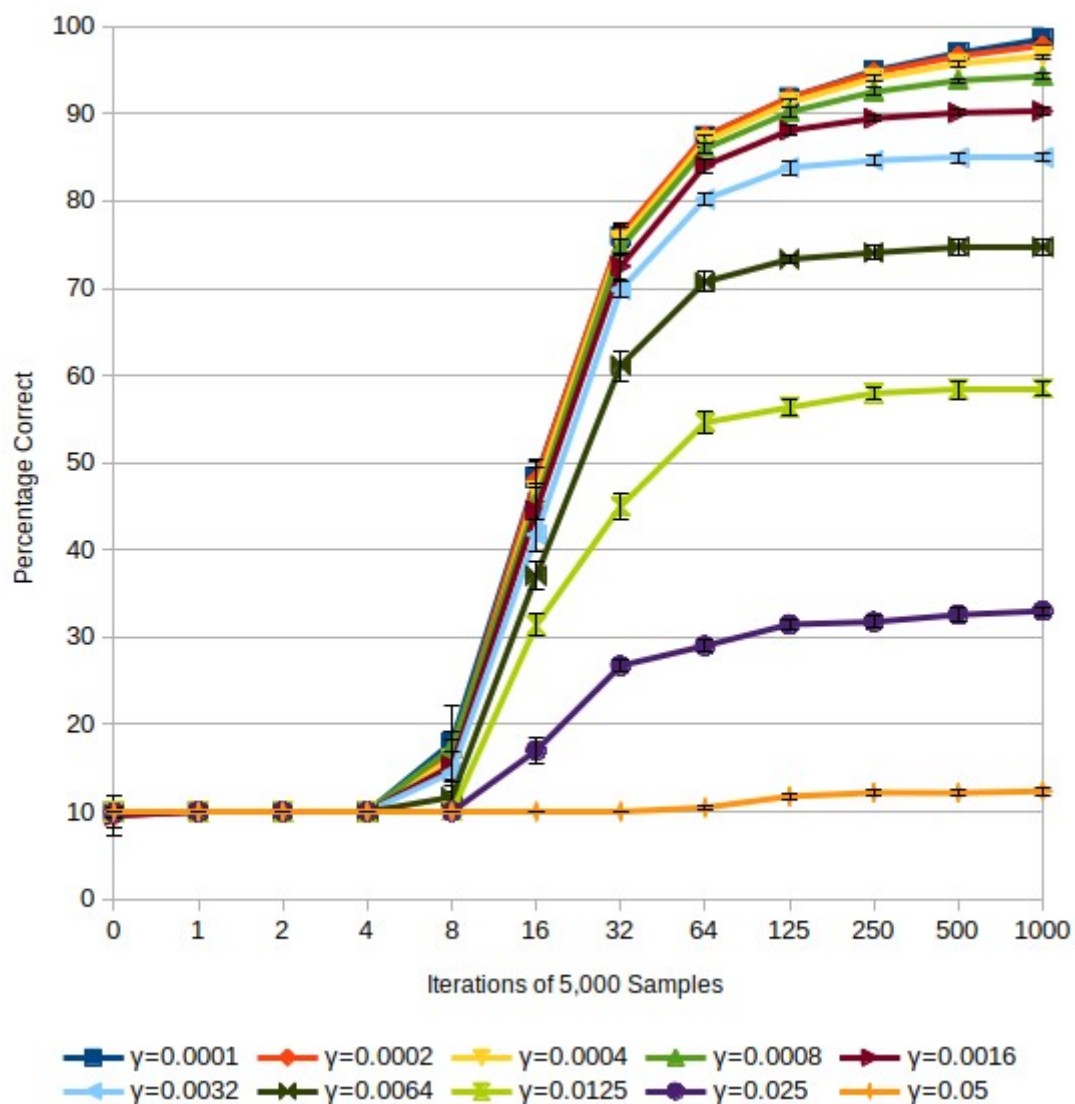


Figure 8.5: Impact of varying the regularisation factor (γ) when using an artificial neural network with 20 hidden neurons and a learning rate of 0.001 to learn to classify ten 20x20 grey-scale hand-written numerals one image at a time

Figure 8.5 shows the impact of using different regularisation factors (γ) on the Flexible Artificial Neural Network's ability to classify hand-written numerals with 20 hidden neurons and a learning rate of 0.001, averaged over 10 simulations. As the regularisation factor is increased, the maximum

percentage of hand-written numerals that can be correctly classified reduces. Regularisation factors above 0.001 significantly reduce the Flexible Artificial Neural Network's ability to classify the training data. Identifying the correct regularisation factor (γ) to use requires testing against separate validation data. However, this was not done, because it was only important to test the functionality of the neural network using iterative learning. Iterative learning is used by the Iterative Learners.

Finally, the impact of varying the learning rate was tested. A regularisation factor of 0.002 was chosen, because this was the recommended regularisation factor used during the initial batch training: 10, divided by the number of samples: 5,000. This is the difference between the γ in equation (5.19) and the gamma in equation (5.20). Similarly, the Flexible Artificial Neural Network was configured with the minimum number of 20 hidden neurons identified.

Figure 8.6 illustrates the impact of varying the learning rate on the Flexible Artificial Neural Network with 20 hidden neurons when using a regularisation factor of 0.002 averaged over 10 simulations. With learning rates of 0.001 and greater, the maximum percentage of numbers that can be correctly classified is significantly less than that achieved with slower learning rates. With learning rates lower than 0.00025, the time required to learn increases rapidly. The best learning rate found was 0.0005, which resulted in learning almost as good as a batch trainer using conjugate gradient descent. A correct classification percentage of 82% was achieved after 125 iterations of 5,000 samples, and 93% after 1,000 iterations.

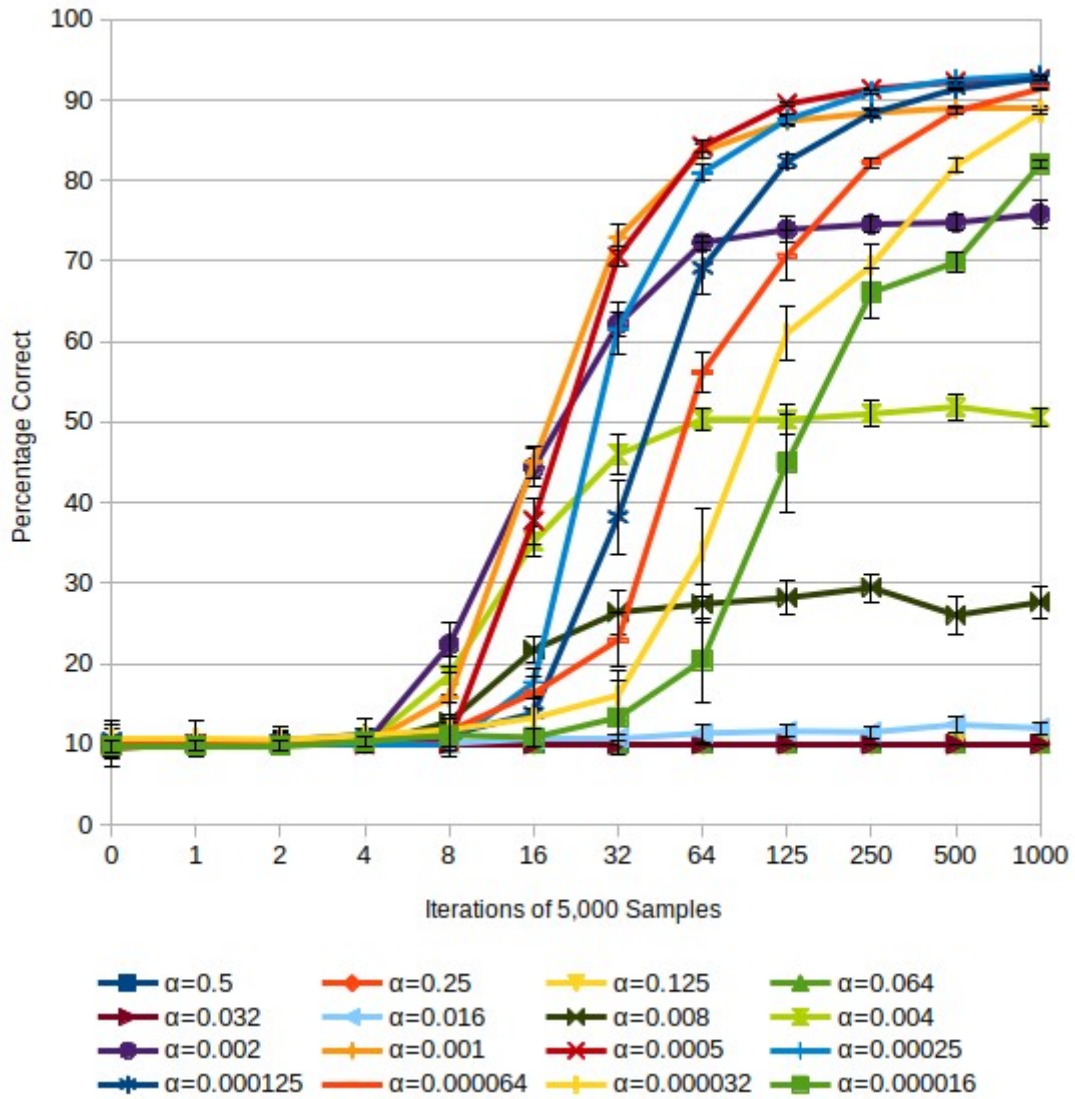


Figure 8.6: Impact of varying the learning rate (α) when using an artificial neural network with 20 hidden neurons and a regularisation factor of 0.002 to learn to classify ten 20x20 grey-scale hand-written numerals one image at a time

Although learning one image at a time takes more time than batch learning using conjugate gradient descent; the number of iterations is similar. When learning iteratively, as long as the length of time between images being presented is greater than the time taken to learn the image, only the number of iterations matters.

8.3 *Testing the Iterative Learners*

The first test used to test the learners was a simple supervised learning task: learning the output of four logic gates: AND, OR, NAND and XOR. Logic gates are the simplest mapping to learn. They have two digital inputs, four potential combinations and one output. The correct output depends on the gate being learnt. However, this conceals the complexity of learning an XOR gate, which is not linearly separable.

The MCMC Learner simply updates its estimated value for each input combination for a steady, consistent decrease in the root mean square (RMS) error. The results are independent of the logic gate being learned. Since simple supervised learning is being tested, there is no temporal sequence of inputs. Therefore, both the MC Network Learner and the TD Network Learner will learn the same way. They do not need to be compared against each other. A Flexible Artificial Neural Network with a single hidden layer with four neurons was used. Four neurons were used, because, with only two neurons in the hidden layer, the ability to learn the correct weights for an XOR gate is dependent on the initial weights.

The results of the learning are presented in Figure 8.7. There were two interesting things to note: First, to learn the simple logic gates, the Flexible Artificial Neural Network based learners required an order of magnitude greater number of iterations. Second, when learning the non-linearly separable XOR logic gate, the neural networks required nearly 500 iterations before any learning is evident. This shows the additional effort required by an artificial neural network to separate non-linearly separable features.

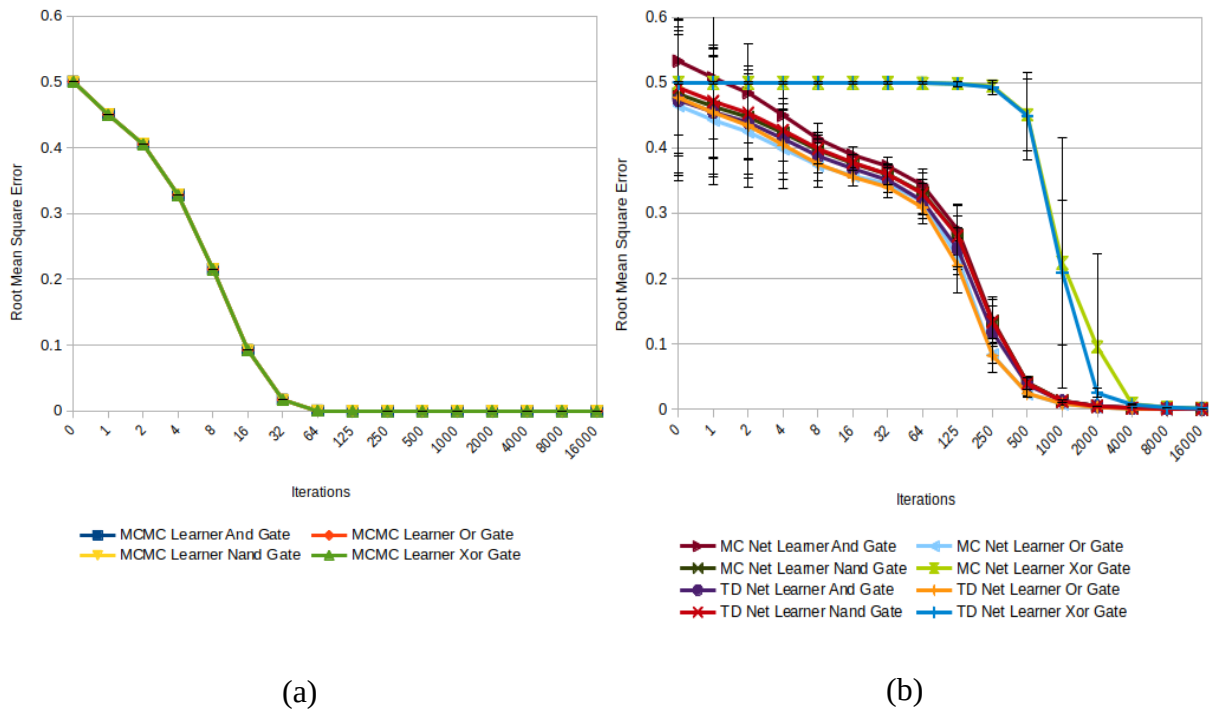
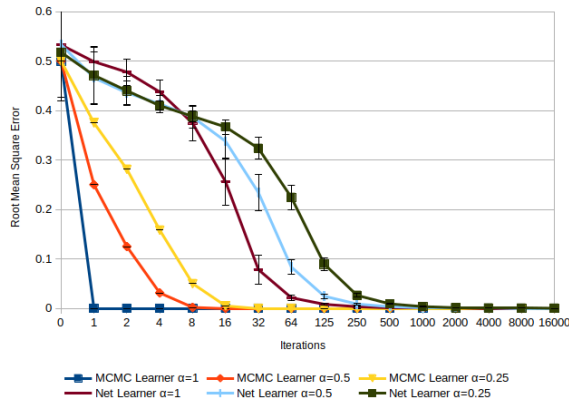
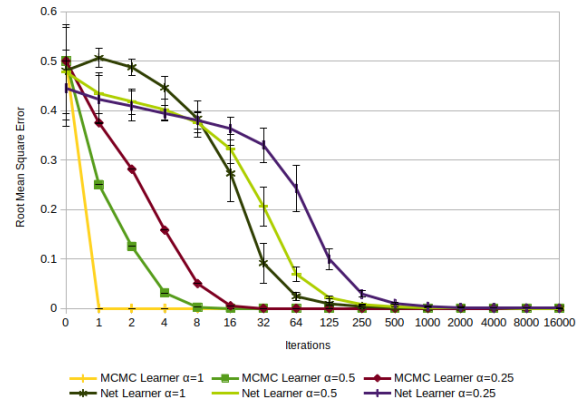


Figure 8.7: Comparing the number of iterations required to learn logic gates averaged over ten simulations using (a) the MCMC Learner and (b) Flexible Artificial Neural Network based learners with one hidden layer with four neurons, and a learning rate of 0.1

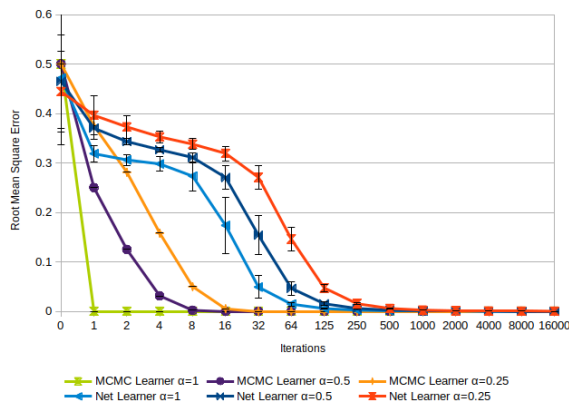
It was found that the learning rate has a significant impact on the learning speed. As shown in Figure 8.8, setting the MCMC Learner's learning rate $\alpha = 1$, i.e. the estimated value is changed by the size of the error, enables the learner to learn instantly. Setting the artificial network learner's learning rate $\alpha = 1$, i.e. changing the weights by the value of the gradient, is also the fastest way for the artificial network based learners to learn these simple functions.



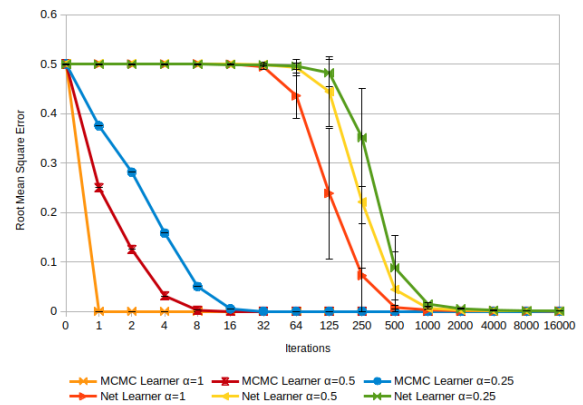
(a)



(b)



(c)



(d)

Figure 8.8: Impact of varying the learning rate (α) on the number of iterations required to learn (a) AND, (b) OR, (c) NAND and (d) XOR logic gates using each of the Iterative Learners

The next learning test was a simple reinforcement learning task: solving a one dimensional random walk. Starting in the middle, the objective is to find the shortest path to the edge. The distance to the edge sets the size of the problem. The complexity of the problem can be easily extended to multiple dimensions. The simulation is achieved by punishing the learner with a negative reward for each step and providing a large final reward for reaching the edge. The reinforcement policy implemented was the simple ϵ -greedy policy: 10% of the time a random action is picked: the learner

moves in a random direction. The other 90% of the time the best action is picked: the learner moves in the direction with the current highest predicted reward.

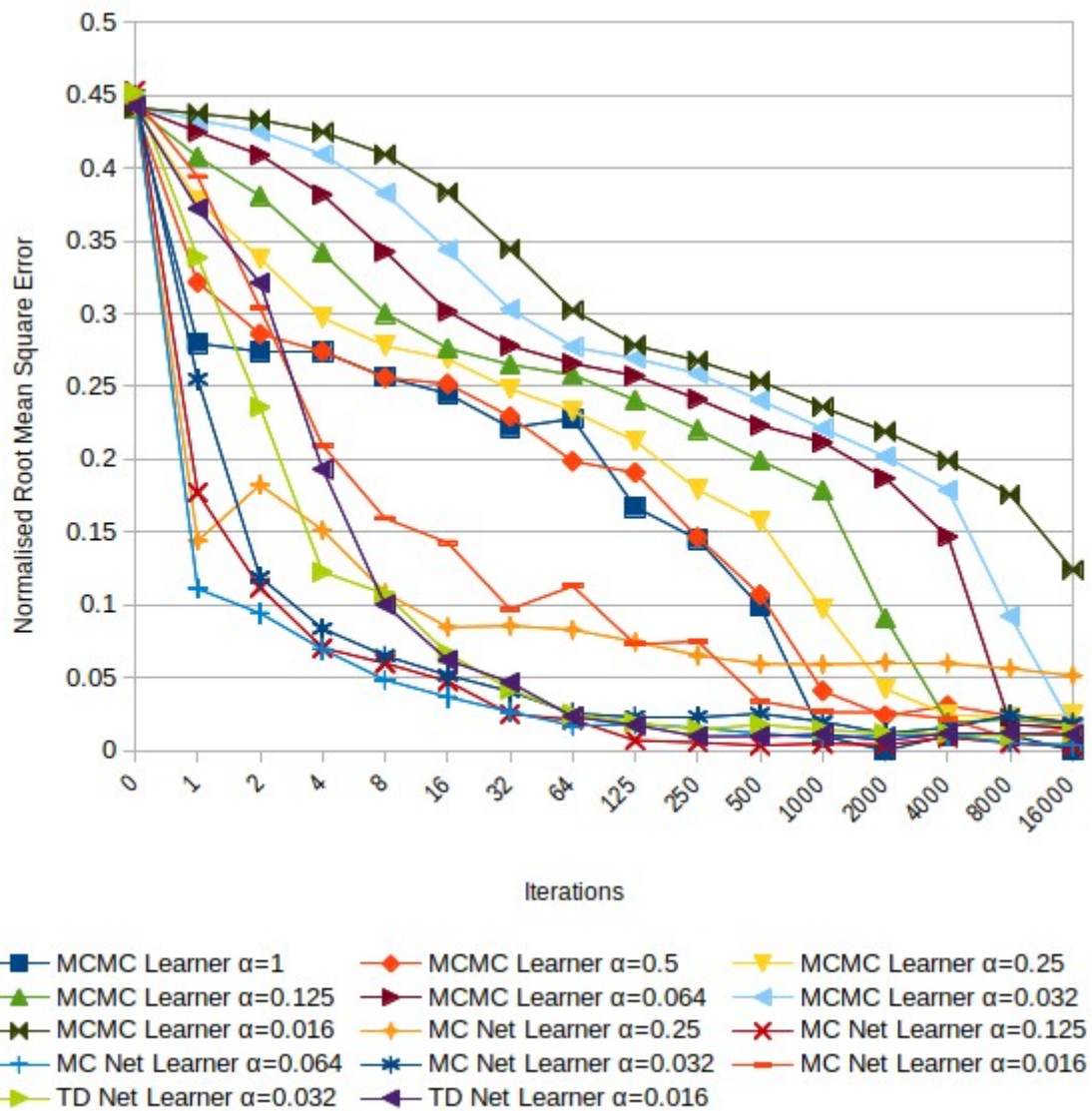


Figure 8.9: Comparing the learning speeds of different learners solving the five position one dimensional random walk problem with different learning rates (α)

Figure 8.9 compares the three learners using different learning rates to solve the one dimensional random walk problem with five positions, averaged over 10 simulations. For clarity, the error bars have not been displayed. The Flexible Artificial Neural Network based learners all have a single hidden layer with twice the number of neurons than the input layer. The main thing to note is that

even with only five possible states, the Flexible Artificial Neural Network based learners learn quicker than the MCMC Learner. The best learner was the MC Network Learner with a learning rate of 0.064. With higher learning rates, the Flexible Artificial Neural Network based learners become unstable; especially the TD Network Learner. They are unable to learn, because they quickly reach a point where the value of all positions is equal. The weight of the bias dominates and the weights of all the hidden neurons have been driven to zero. This is to be expected, because the algorithm is ultimately a multi-dimensional gradient algorithm. The learning rate determines the learning step size. If the step size is too large, instead of the decreasing the error in an effort to find the minimum value, the error increases, which results in the instability observed.

8.4 Creating a Generic Game Interface to Test the Generic, Self-Improving Predictor

The primary function of the Generic, Self-Improving Predictor is to take an input state vector and return a prediction vector. The Generic, Self-Improving Controller will convert the prediction vector using a control policy to determine the desired outcome. However, the Generic, Self-Improving Predictor needs to be tested separately in a controlled, simulated environment.

Games are used to create the controlled, simulated environment because, games provides a precisely quantifiable, discrete digital environment. This can be used to completely define the input state. A game also provides precisely quantifiable, discrete feedback signals. Games are either won, lost or possibly drawn. Turn-based games enable the temporal input sequence to be discrete too. Multi-player games enable multiple instances of the Generic, Self-Improving Predictor to be used simultaneously to simulate each player. This enables the Generic, Self-Improving Predictors to learn from each other. Furthermore, the adaptability of the Generic, Self-Improving Predictor can be demonstrated. As the opposing Generic, Self-Improving Predictor learns, it provides a non-

deterministic, changing environment. A winning move earlier may no longer be a winning move later, because the opposing Generic, Self-Improving Predictor based player has learnt a better response. Finally, games enable the Generic, Self-Improving Predictor's learning to be visualised and experienced.

A generic game interface and generic game player interface were created. The generic, game interface allows any game that can be described using a state and whose players' possible actions can be listed, to be played by a generic game player. The generic game interface requires a game to return three types of basic information: how many players, which player is to play next, and which player has won. This game interface also requires that a game provide a list of possible actions that the next player can make. Intuitively, this should not be a requirement of the game. In normal games, it is the player that should determine the actions that a player can make. Part of good game play is identifying good moves. However, this research is not about playing games, it's about learning. The objective is to learn the best move, not determine the possible moves. Furthermore, there is no inherent requirement for the generic game player to use the actions returned by the game interface. Therefore, this functionality can be disabled at a later date, if required.

The game interface's main function is to receive actions. The actions are passed to the game in the format required by the game. The game then updates itself. The game actions can be generated by a human using either a console or a graphical interface. Alternatively, a game action can be selected from the list provided by the game. This is the approach used by the Generic, Self-Improving Predictor based players.

The generic game player interface has the following requirements: First, given a game, it is required to return an action. Second, given a game, it is required to value it. The value is required by the action selection policy that the Generic, Self-Improving Predictor uses to select its action. The player can use any policy available. The policies available depend on whether the Generic, Self-

Improving Predictor is operating in learning or non-learning mode. Third, given a game, the player is required to return a game state vector. The game state vector is used by the Generic, Self-Improving Predictor's learner as input. This enables different players to define different ways of describing the same game. This is not important for the MCMC Learner, because it doesn't matter how the input state is described as long as it's unique. However, it is important for the Flexible Artificial Neural Network based learners. Although the Flexible Artificial Neural Network should be able to learn the relevant features, learning times are significantly reduced if they don't have to learn the features first. Finally, the generic game player interface requires, given a completed game, return an actual result vector. The actual result vector is used by the Generic, Self-Improving Predictor's learner to improve its predictions.

Since the Generic, Self-Improving Predictor was designed to predict and learn independently, the Generic, Self-Improving Predictor doesn't have to select the actions to learn. It can learn from expert players selecting the actions; by observing human players; or by replaying previously recorded games. Furthermore, since all the players are based on the Generic, Self-Improving Predictor, the underlying Iterative Learner can be changed. Learners can be saved and previous saved learners their learning loaded. Finally, the learning rate and temporal discount rates can be changed at any time.

Once created, the game and the players are added to a game controller. The game controller enables both the Generic, Self-Improving Predictor based players and human players to play the same game. The Generic, Self-Improving Predictor based players can be automated i.e. they provide the next action. The next action selected is based on the policy. The policy is based on whether or not they are configured to learn. If both Generic, Self-Improving Predictor based players are set to automated, and the controller is set to automatically reset the game upon completion, the Generic, Self-Improving Predictor based players will continuously play game after game. If they are also configured to learn, they will learn as quickly as possible from each other.

The game controller can be connected to a game specific visual presentation layer. The visual presentation layer translates the game state into a human recognisable form. The visual presentation layer should also provide a means for a human player to send an action to the game or select an action from the list of available actions. Finally, the visual presentation layer also provides the controls required to slow the rate of play during fully automated learning, so that humans can watch multiple Generic, Self-Improving Predictor based players play and learn.

It is important to remember that the Generic, Self-Improving Predictor does not know that it is playing games. The same Generic, Self-Improving Predictor that will be used with the hardware sensors to control real devices, is used. The Generic, Self-Improving Predictor also does not know that it is playing another version of itself. It is also not even necessarily the case. Playing another version of itself means that the results of an action not specific. In other words, the subsequent game state vector does not only depend on the selected action. The opponent's counter-move is indeterminable. In other words, any specificity learnt will change with time, because the opponent's counter-moves will change as it learns how to improve its chances of winning or drawing.

The complexity of games was progressively increased. The initial plan was to first test the learners on mazes. Mazes require only determining the correct direction to move based on location and the available directions. However, this is simply an extension of the one dimensional random walk problem, and it was not expected to produce interesting results. Therefore, the next test was to apply the Generic Self-Improving Predictor to solving the game of Tic-Tac-Toe⁴⁵. Tic-Tac-Toe has a larger, but still limited, number of possible states and actions, which allows the use of the MCMC Learner.

The second game the Generic, Self-Improving Predictor's was applied to was Backgammon. The number of states in Backgammon is estimated to be 10^{20} and the available actions are not only

⁴⁵ Tic-Tac-Toe is also known as Noughts and Crosses.

dependent on the current state, but also the value of dice thrown. Therefore, the state-space needs to be abstracted, and the MCMC Learner cannot be used.

8.5 Testing the Generic, Self-Improving Predictor by Playing Tic-Tac-Toe

Tic-Tac-Toe provides a useful platform for demonstrating and testing the Generic, Self-Improving Predictor. The temporal nature of the game, where the feedback to earlier predictions is dependent on the success of learning the correct later predictions, provides a significantly more complicated environment to learn than standard fixed input to fixed output supervised learning. Tic-Tac-Toe also has less than 3^9 (19,683) possible states. Too many to specify manually, but small enough to know what the best action in each state is.

The game logic of Tic-Tac-Toe was enclosed in the generic game interface described above. The list of game actions provided to game players is simply a list of indices representing the unoccupied squares. The visual presentation layer shown in Figure 8.10 was created to enable game play to be tested and learning to be visualised.

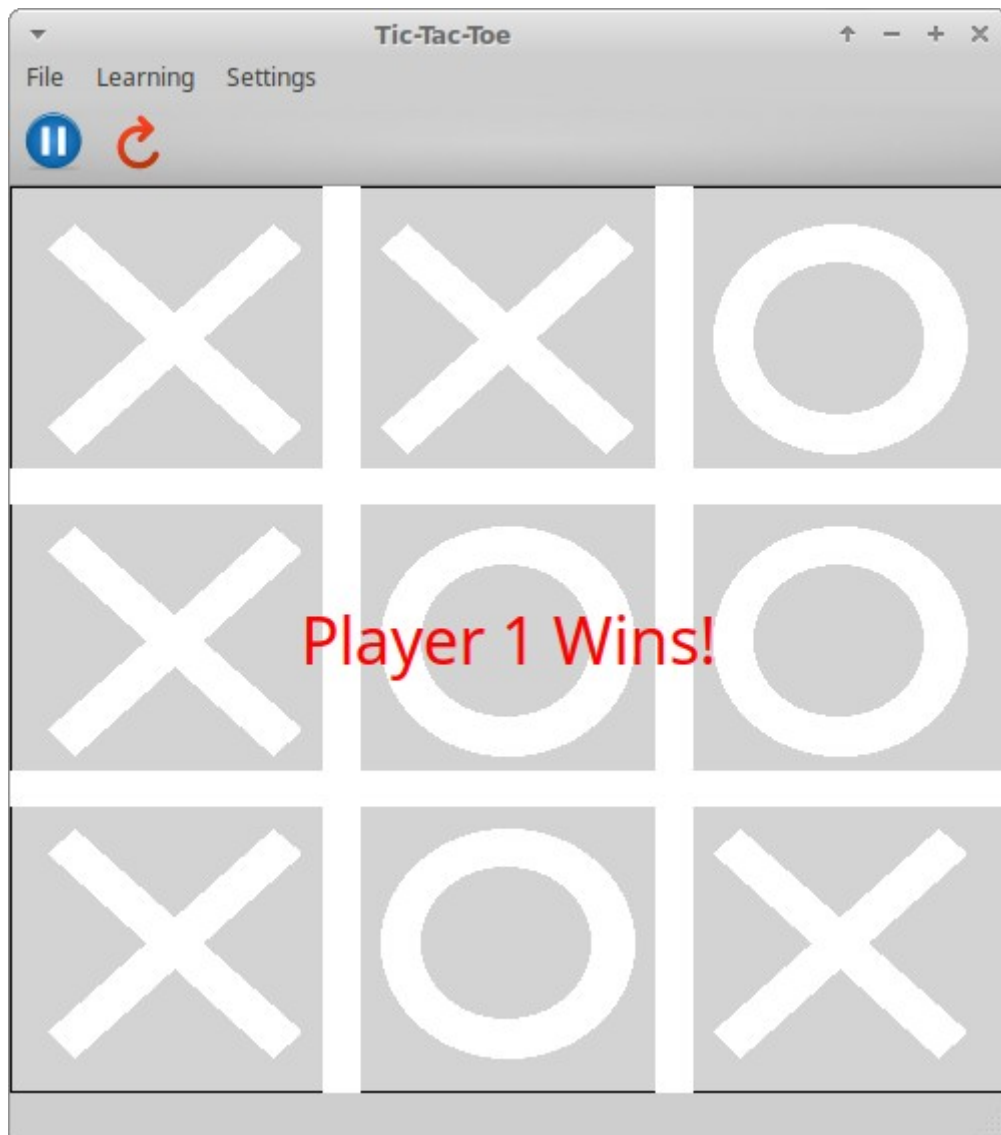


Figure 8.10: Watching the Generic, Self-Improving Predictor learning to play Tic-Tac-Toe

If Tic-Tac-Toe is played well, it will always result in a draw. Therefore the Generic, Self-Improving Predictor is expected to learn that all first moves are expected to end in a draw. However, depending on the first move, the second move is critical. Therefore, to quantify the learning of the Generic, Self-Improving Predictor, the root-mean-square error of the second player's response to an opening move being placed in one of the corner squares is compared. If the first player plays in one of the corner squares, the first player is able to ensure a victory from any response from the second player unless it places a piece in the centre square. The second player should learn to predict a draw only

from the action of placing a piece in the centre square. The second player should learn a loss from any action of placing a piece in any of the other unoccupied squares.

Although only the second player is used for quantifying the learning, both players are configured to use a Generic, Self-Improving Predictor using the same type of learner with the same settings, and they are configured to play each other to learn.

8.5.1 Learning Tic-Tac-Toe Using the MCMC Learner

The Generic, Self-Improving Predictor was first tested using the MCMC Learner. The temporal discount rate was not used and left at its default setting of 1. Figure 8.11 shows the impact of varying the learning rate from 1 to 0.001 averaged over 10 simulations. For clarity, the standard deviation error bars have not been displayed.

With a learning rate of 0.5 the Generic, Self-Improving Predictor, using an MCMC Learner, is able to learn an optimal response after playing just a few thousand games. Although initial learning is quicker with a learning rate of 1, the error rate does not reach the minimum achieved with slower learning rates. Similarly with a learning rate of 0.25, learning is also initially quicker, but, as with all slower learning rates, the effective learning rate is slower and the final root-mean-square error never drops below 0.1. The reason the error rate never drops below 0.1 is due to the learning policy, which ensures that non-optimal actions are continuously tried. The result is that non-optimal results, are feedback to the learner.

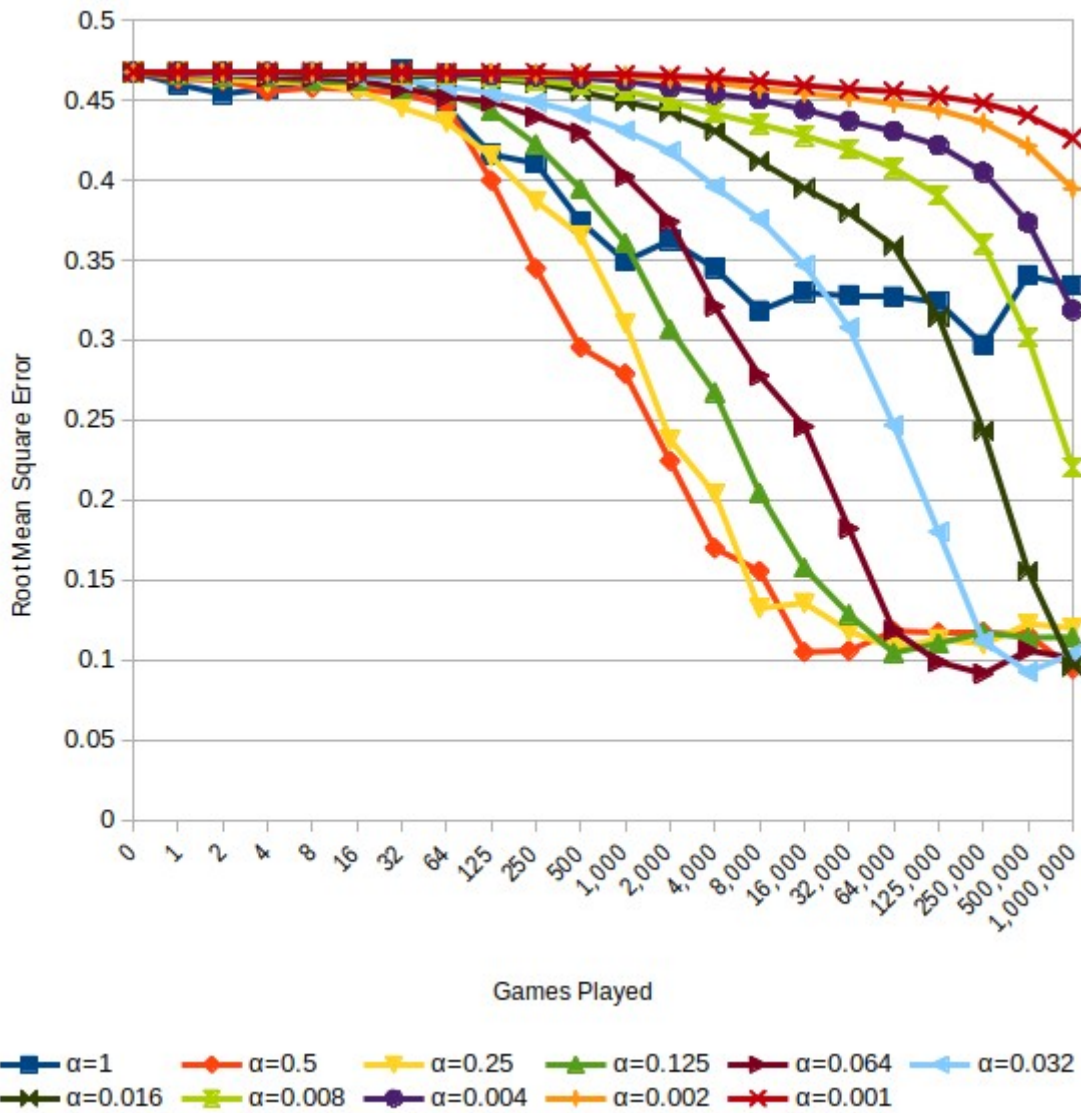


Figure 8.11: Impact of varying the learning rate (α) when using the MCMC Learner to learn to play Tic-Tac-Toe

Note that, for the Generic, Self-Improving Predictor to learn the best response to an opening move in the corner, it, and its opponent, first have to learn the best last action. If it could win by creating a row, or draw by preventing the opponent from creating a row, then it needs to learn to do so first. Only then can the MCMC Learner learn the best second-to-last action, etc. Furthermore, it can only learn the best response to an opening move in the corner when the first player plays an opening move in the corner. Once it learns that its most likely chance of winning is to play its first move in

the corner, it should do so more often. However, since all probabilities for the opening move converge on 0.5; this is not actually the case. It does not determine (at least not significantly so) that it is better to play the first move in the corner, where the only “correct” second move is to play in the centre. In other words, if playing against a random-move player, there is a 7/8 chance of winning, whereas if the first move is played in the centre (or the edge) there is only a 1/2 chance of winning.

Although the learner performs well, it does require that all possible states are explored and the values of all possible actions in each of those states is determined. There are times that game states that have not been explored are encountered, and the algorithm makes an obvious mistake despite having played millions of games.

8.5.2 Learning Tic-Tac-Toe Using the Flexible Artificial Neural Network Based Learners

The Generic, Self-Improving Predictor was next configured to use the Flexible Artificial Neural Network based learners. Figure 8.12 shows the initial results using Flexible Artificial Neural Network based learners with 18 hidden neurons and logistic activation functions. The results shown are again averaged over 10 samples, with the standard deviation error bars removed for clarity. The same regularisation factor used for learning the hand-written numerals was used: 0.002. The temporal discount rate was not used and kept at 1. For the TD network learner, the weight preference (λ) was also kept at 1.

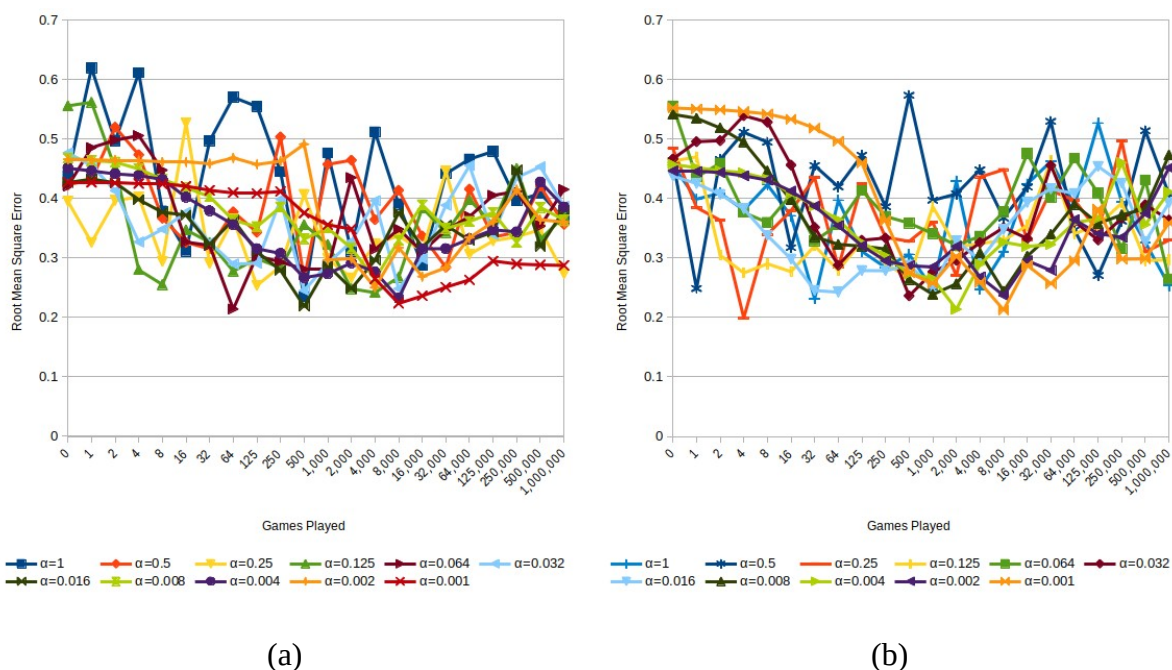


Figure 8.12: Initial results from learning to play Tic-Tac-Toe using the MC Network Learner (a) and the TD Network Learner (b)

Although some learning does take place, and despite varying the learning rate α from 1 to 0.001; the Generic, Self-Improving Predictor never learns the probabilities of winning, drawing or losing; even after playing a million games.

The initial proposed explanation for the failure to learn was the game state representation used. The way the game is represented is insignificant for the MCMC Learner as long as each game state is unique. The initial Generic, Self-Improving Predictor based player defined the game state vector using nine values: one for each of the squares on the grid. Three discrete values were used: 0 for a blank, -1 for a nought and 1 for a cross. The final result is a single value vector that specifies the value of the final game: 1 for a win, 0 for a loss and 0.5 for a draw.

The concern was that the learners were unable to tell the difference between the different pieces at the same location, because they were both represented using the same state value. Therefore, the way the player represents a game was changed. Instead of the three-state nine dimensional input

state vector, a vector of 18 binary inputs representing whether or not a square has an X or a O was used. As seen in Figure 8.13, changing the input state representation did not help improve the learning. However, despite not helping to improve the learning, the binary representation was kept for the subsequent experiments.

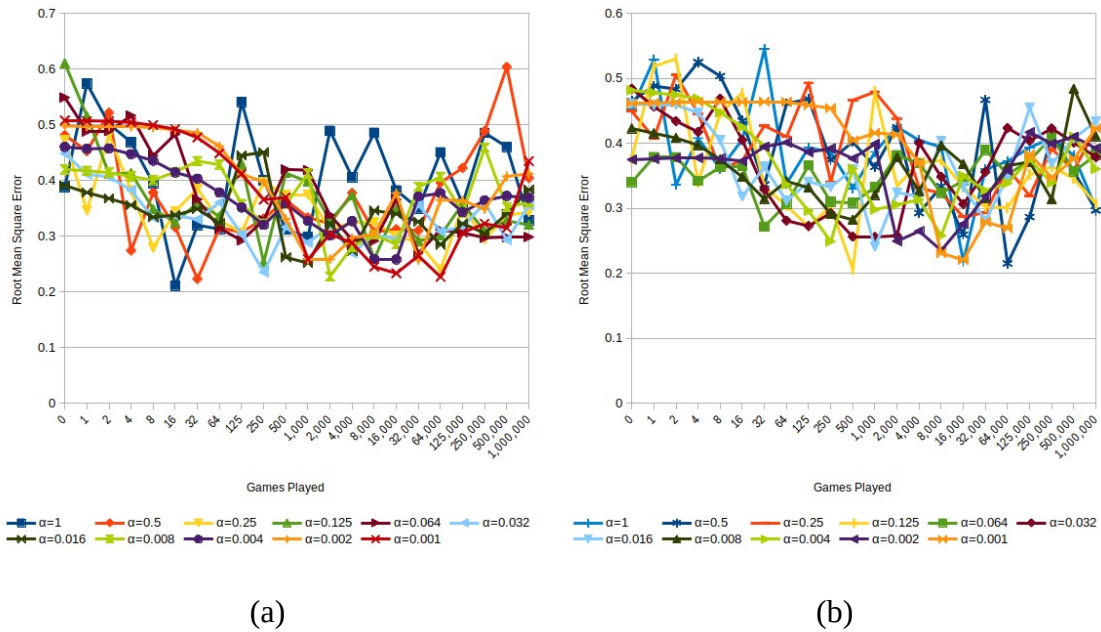


Figure 8.13: Results from using a binary game state representation to learn to play Tic-Tac-Toe using the MC Network Learner (a) and the TD Network Learner (b)

The learning rate is considered the most important parameter affecting learning⁴⁶. This was seen when learning to classify hand-written numerals in Section 8.2. A large regularisation factor was also shown to inhibit learning; especially with higher learning rates. Therefore, the next change made was to measure the impact of the regularisation factor (γ) using the best learning rate identified when classifying the hand-written numerals: $\alpha = 0.0005$. At the same time, to ensure that learning is not simply slow, the total number of games played during each simulation was increased to 8 million. The results, again averaged over 10 simulations, are presented in Figure 8.14.

⁴⁶ (Bengio, 2012b)

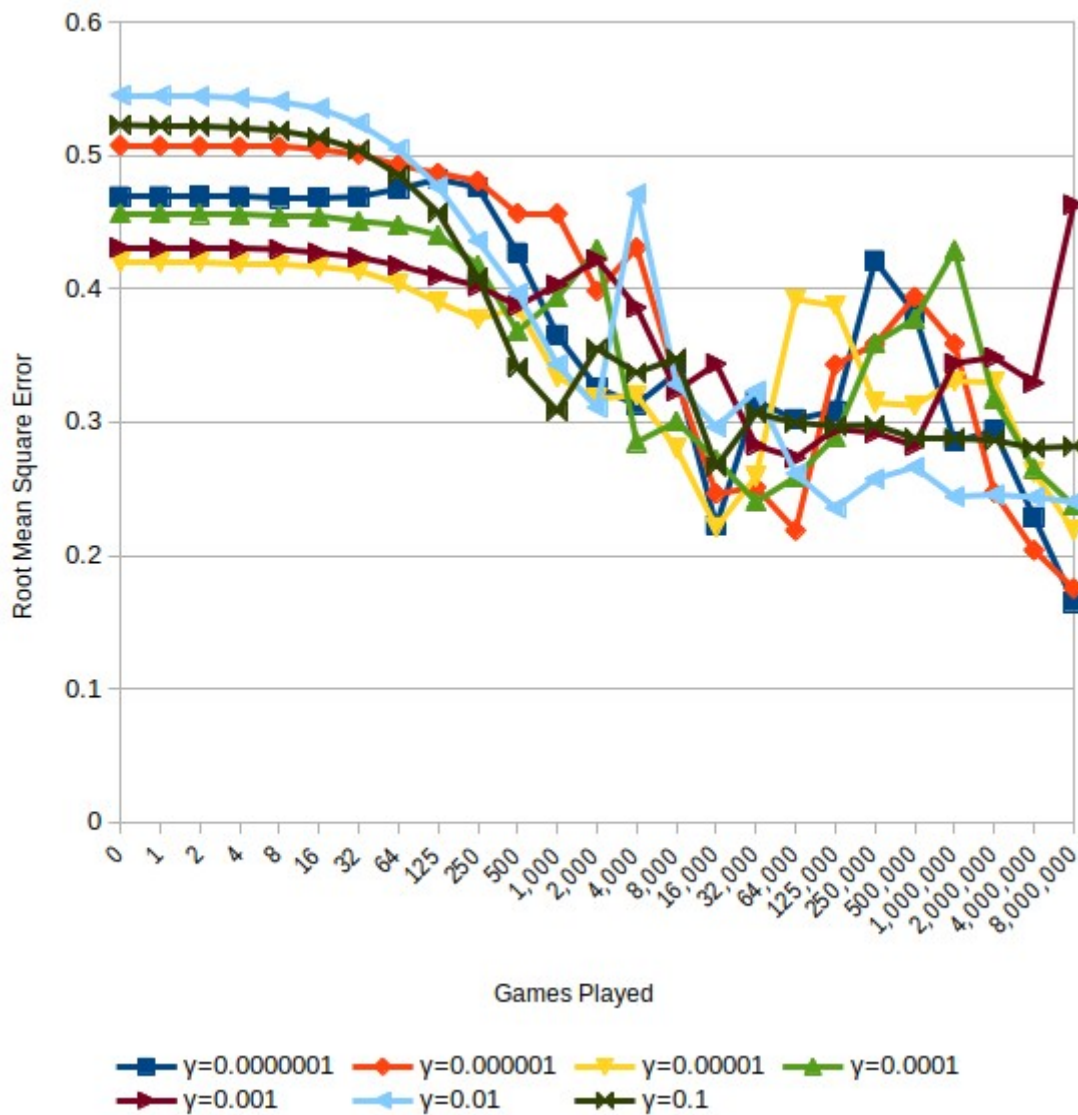
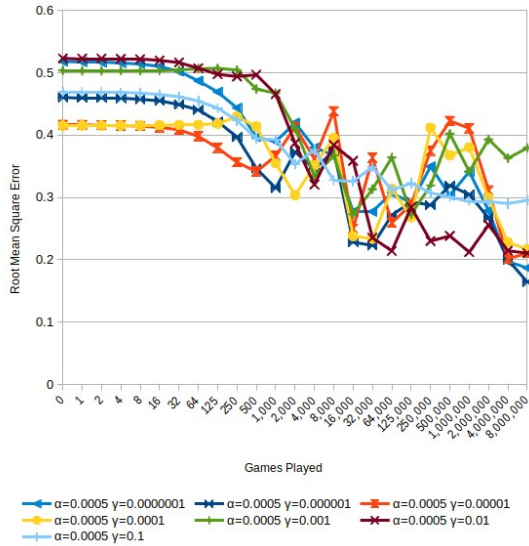
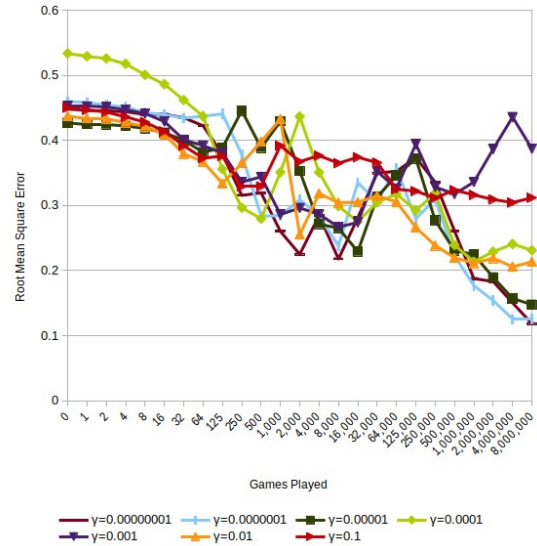


Figure 8.14: Impact of varying the regularisation factor (γ) when learning to play Tic-Tac-Toe using an artificial neural network learner with a learning rate (α) of 0.0005, 18 hidden neurons and a binary representation of the game state

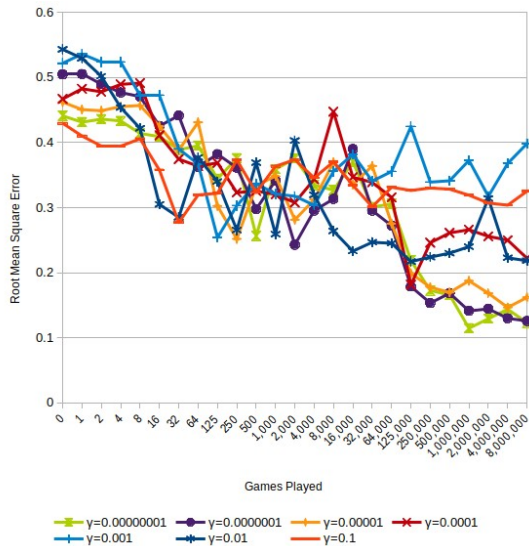
The first thing to note is that, irrespective of the regularisation factor used, more learning is now seen to be taking place. However, it is still not to the level seen with the MCMC Learner. This suggests that a lower learning rate is required. The other thing to note is that with a regularisation factor (γ) of 0.001, which is close to the value calculated from the proposed batch-training value for generalising from the hand-written numerals training data, the initial learning is reversed with more games.



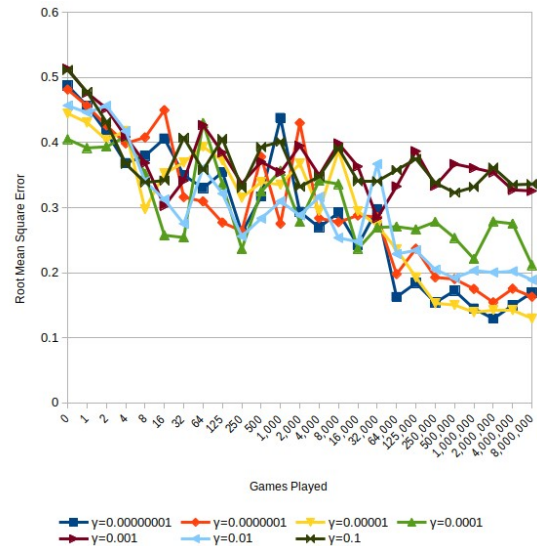
(a)



(b)



(c)



(d)

Figure 8.15: Impact of varying the regularisation factor (γ) when learning to play Tic-Tac-Toe using the MC Network Learner with 15 hidden neurons, a binary representation of the game state, and a learning rate (α) of (a) 0.0005, (b) 0.0032, (c) 0.025, (d) 0.05

The first requirement was to confirm that the better results were due to the lower learning rate.

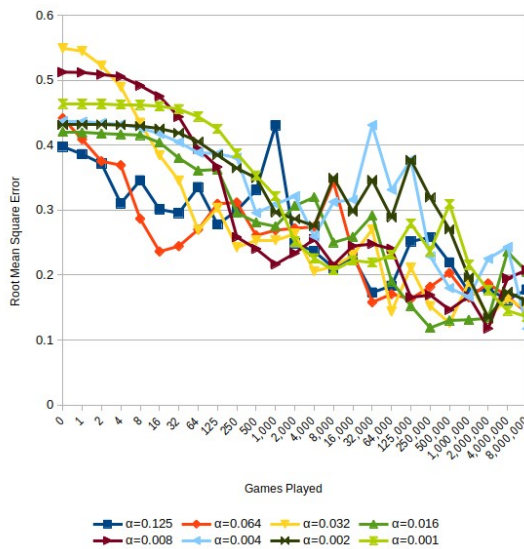
Therefore, the impact of varying the regularisation factor (γ) was tested with higher learning rates.

The results are presented in Figure 8.15.

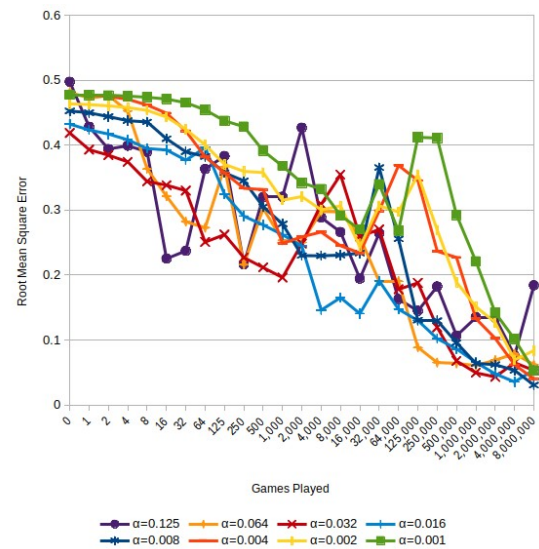
Unexpectedly the learning rate has very little effect on the learning; except that higher learning rates, and not lower learning rates, produce better results. Instead, it's the regularisation factor that is of significance, with regularisation factors of 0.00001 or less required to achieve the learning expected.

Therefore, the results of Figure 8.14 need to be viewed another way: the best results are seen with lower regularisation factors. The regularisation factor is required for generalisation, when the number of samples is less than the number of parameters to avoid over-fitting. With 18 hidden neurons and, even with the binary representation, 18 inputs plus one bias value, the number of parameters in the first layer is only 342. With a single output neuron the total number of parameters is only 361, which is significantly less than the number of samples: millions of games. Therefore, there is no need for a regularisation, and the regularisation factor can be set to zero.

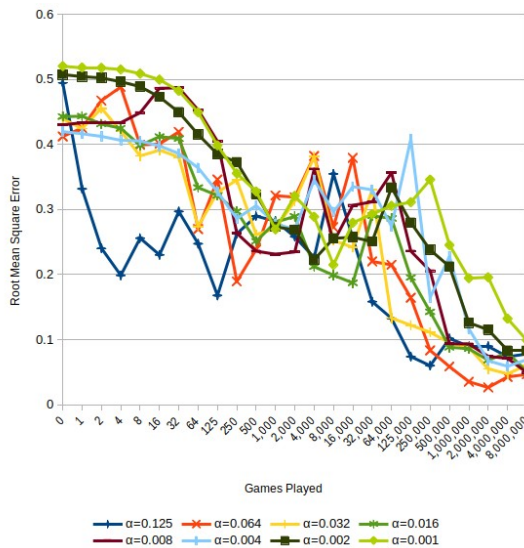
Without regularisation, the MC Network Learner is able to learn. However, this raises the next question: what impact does the number of hidden neurons have on the learning? As shown in Figure 8.16, the results are similar for varying numbers of hidden neurons above 10.



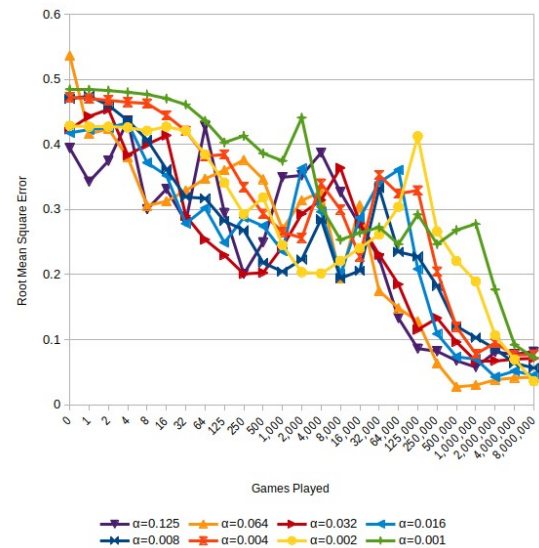
(a)



(b)



(c)



(d)

Figure 8.16: Impact of varying the learning rate (α) when using an MC Network Learner to learn to play Tic-Tac-Toe with (a) 5, (b) 10, (c) 20 and (d) 25 hidden neurons and a binary representation of the game state

Although the MC Network Learner is now learning, the number of iterations is significantly greater than required with the MCMC Learner. The other hyper-parameter available to the MC Network Learner is the temporal discount rate (β). Figure 8.17, shows the impact of varying the temporal

discount rate with different learning rates, while keeping the number of hidden neurons at 15 and the regularisation factor at 0.

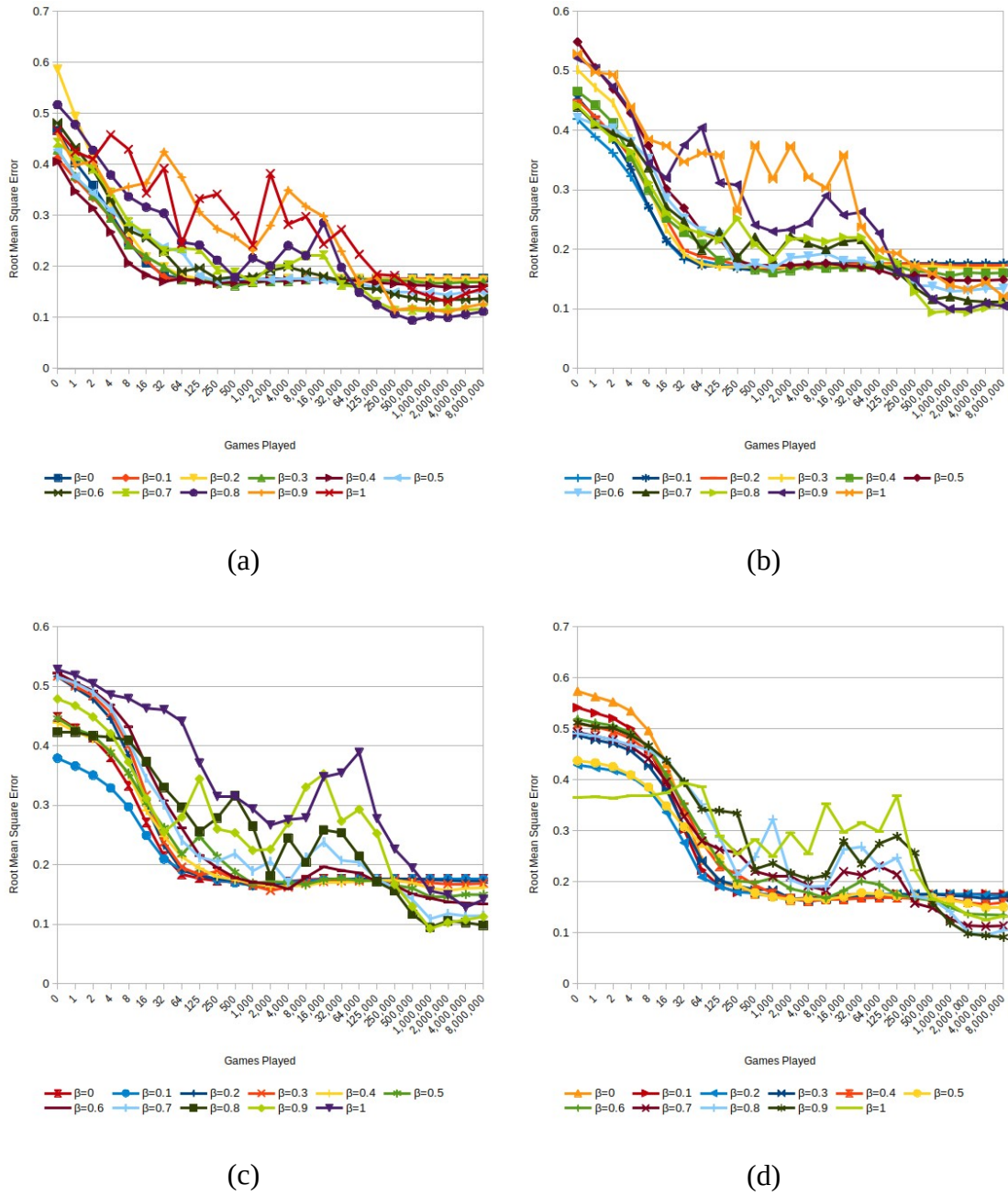


Figure 8.17: Impact of varying the temporal discount rate (β) when using an MC Network Learner and a binary representation of the game to learn to play Tic-Tac-Toe with a learning rate of (a) 0.05, (b) 0.025, (c) 0.0125 and (d) 0.0064

Reducing the discount significantly improves early learning. While lower final root-mean-square errors are seen with higher temporal discount rates. A temporal discount rate of 0.7 provides the best balance between good early learning and low final root-mean-square errors. The temporal discount rate reduces the reward attributed to earlier moves. This appears to effectively reduce the learning rate attributed to early moves providing greater stability. Furthermore, higher learning rates are also associated with better early learning.

Finally, setting the temporal discount rate to 0.7, Figure 8.18, which includes the standard deviation error bars, confirms that the higher learning rate of 0.05 provides the best learning. Although 250,000 games are required to reduce the root-mean-square error to the level expected, most of the learning is achieved with around 1,000 games.

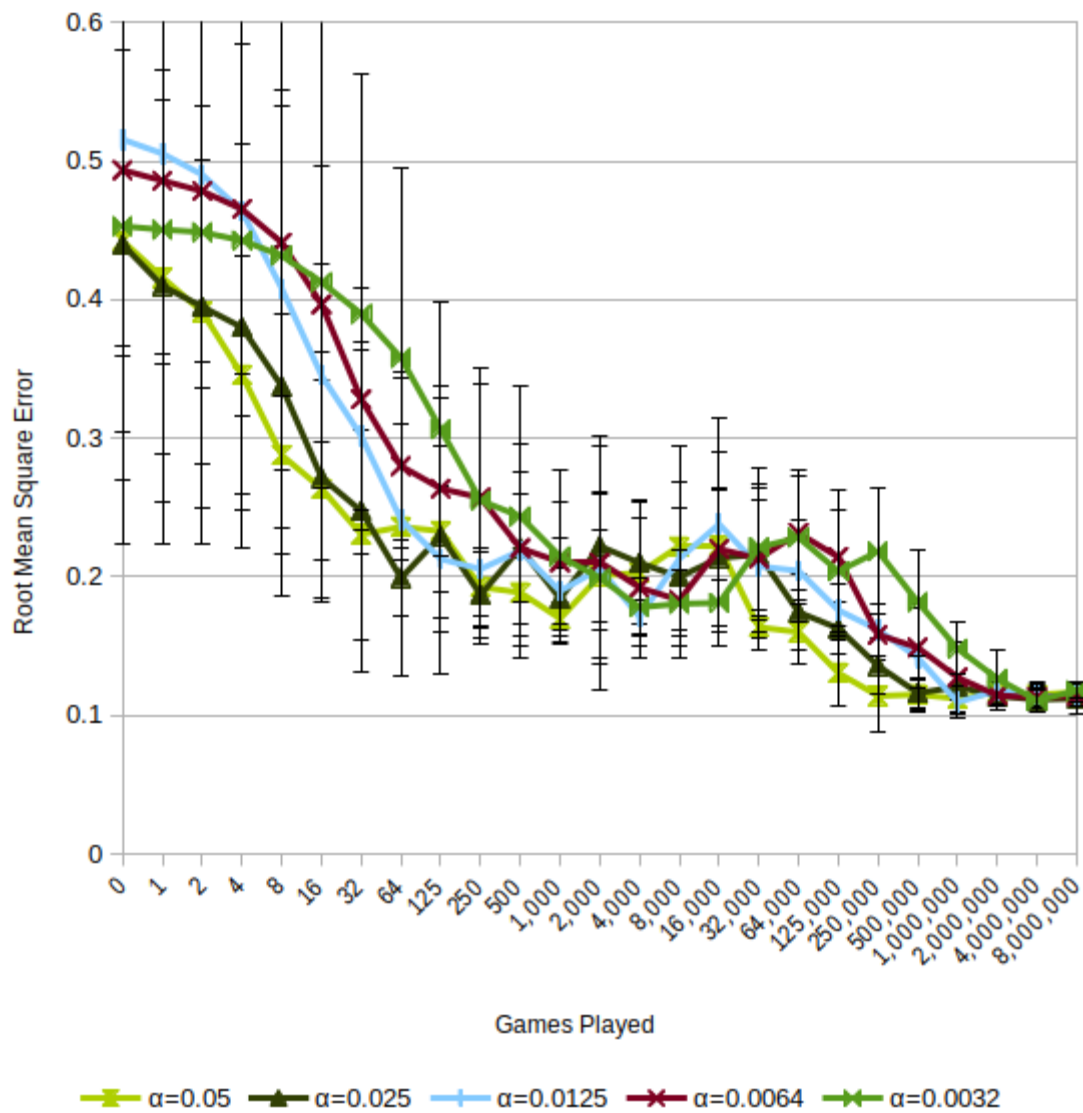


Figure 8.18: Learning to play Tic-Tac-Toe with the MC Network Learner, 15 hidden neurons, a temporal discount rate (β) of 0.7 and various learning rates (α)

8.5.3 Learning Tic-Tac-Toe Using the TD Network Learner

The TD Network Learner should be able to learn faster than the MC Network Learner; however, as seen in Figure 8.19, the results do not reflect this. Unlike the temporal discount rate (β), the weight preference (λ) has an insignificant impact on the learning.

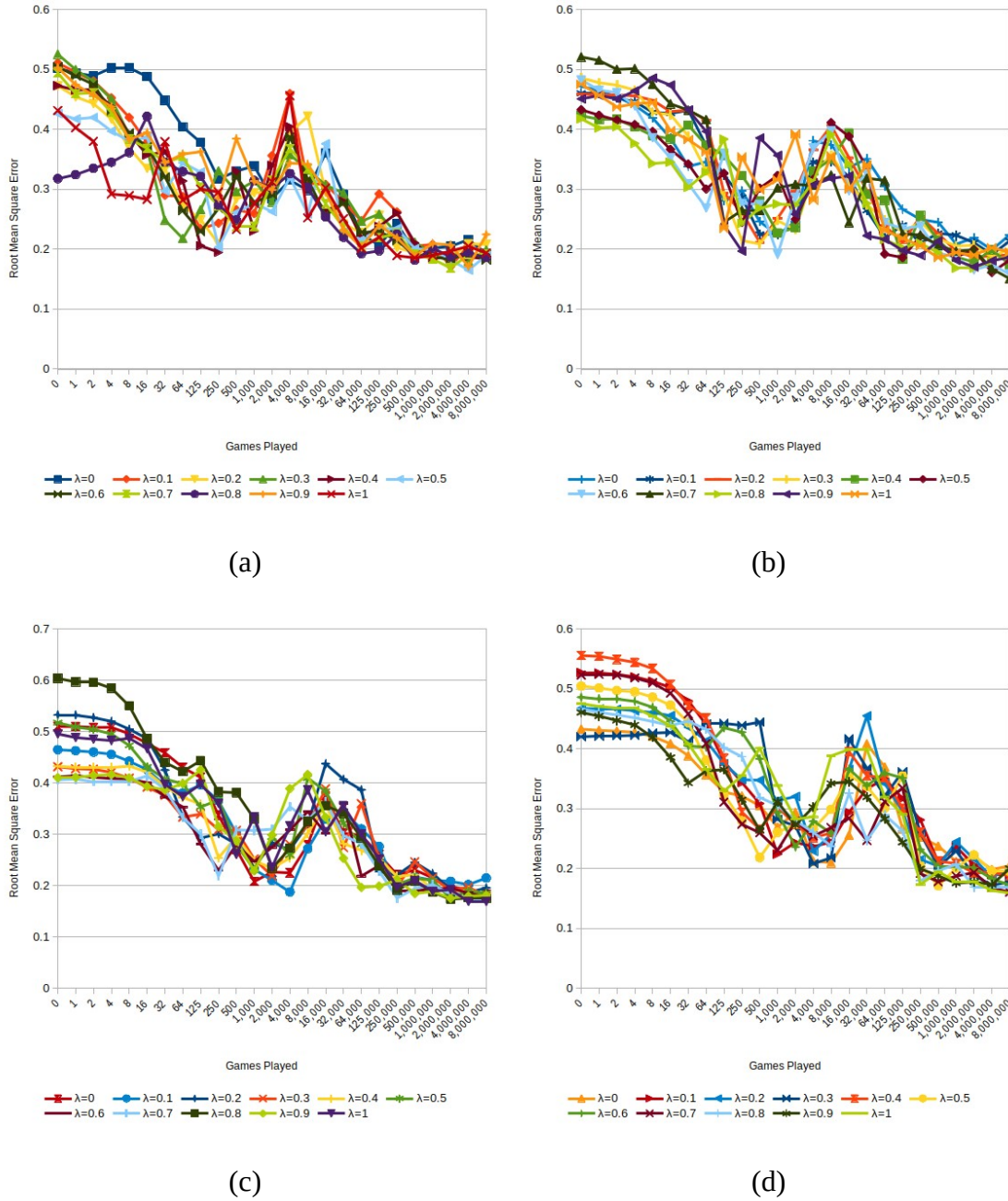


Figure 8.19: Impact of varying the weight preference (λ) using a TD Network Learner with 15 hidden neurons and a learning rate (α) of (a) 0.05, (b) 0.025, (c) 0.0125 and (d) 0.0064

The results presented above demonstrate the ability of the Generic, Self-Improving Predictor that will be used in the Generic, Self-Improving Controller to learn to play Tic-Tac-Toe. With the right

hyper-parameters, the Flexible Artificial Neural Network based learners learn as fast as the MCMC Learner, and they have the advantage of being able to evaluate many more states efficiently.

8.6 Testing the Generic, Self-Improving Predictor by Playing Backgammon

In this section the environment was further complicated by increasing the number of possible states beyond the ability of a reinforcement learner to fully explore. The Generic, Self-Improving Predictor was tasked with learning how to play Backgammon.

Backgammon increases the complexity by combining a large state-space with a temporal sequence of states, delayed feedback, and noisy final reward signals created by the randomisation from the dice. Backgammon has approximately 10^{20} states, many of which will never be visited and those that are visited will only be visited once. Therefore, it is not possible to use a lookup table approach such as used by the MCMC Learner. However, it is still possible to use a Markov chain to store the intermediate states of each game; such as used by the MC Network Learner.

Unlike chess, a draw is impossible and a game played by an untrained network making random moves will eventually terminate. The randomness of the dice rolls results in learners, playing themselves to the best of their ability, to explore a much larger state space without the need for deliberate random exploring, and the expected winner of the game remains uncertain until all the players' pieces have passed each other, and one side has a clear advantage.

An artificial neural network should enable the learner to generalise from its experience. If a new state is encountered, the learner is able to estimate a value for the state based on the results of previous games that contained similar states. The ability of a learner to work with large state sets depends on how appropriately it can generalise from past experience.

The dice generate randomisation in each game, making each move non-deterministic; even when both players are playing optimally. The sequence of board states during a game can be classified as an absorbing Markov process, where the initial state can be considered unique; the subsequent states are random and depend only on the current state; and there is a definite, well-defined terminal state. In other words, this is the ideal environment for maximising the benefits of TD learning.

Although playing Backgammon at expert level is incredibly complex, much of this complexity can be captured in a static evaluation function that does not require deep look-ahead searches⁴⁷. An artificial neural network should be able to learn this static evaluation function. Given the game state, the network learns to predict the probability of a player winning from that game state. A greedy policy is used to select the move that maximises the probability of winning.

As discussed in Chapter 2, in 1992, Tesauro combined the TD learning algorithm with an artificial neural network to learn to play the game of backgammon⁴⁸. The artificial neural network was used to evaluate the game state following various actions. Unfortunately, the derived weights are considered intellectual property. However, others have reported that they have replicated the results. Tesauro's success was a key driver for selecting the TD learning neural-network approach for the Generic, Self-Improving Controller. Therefore, it was important to try to reproduce the results achieved.

The same game interface that was used for Tic-Tac-Toe was used for Backgammon. The list of game actions is a list of board positions that a player can end with given the current board position and a two dice roll. The visual presentation layer shown in Figure 8.20 was created to enable game play to be tested and learning to be visualised.

47 (Berliner, 1979; Tesauro & Sejnowski, 1989)

48 (Tesauro, 1992)

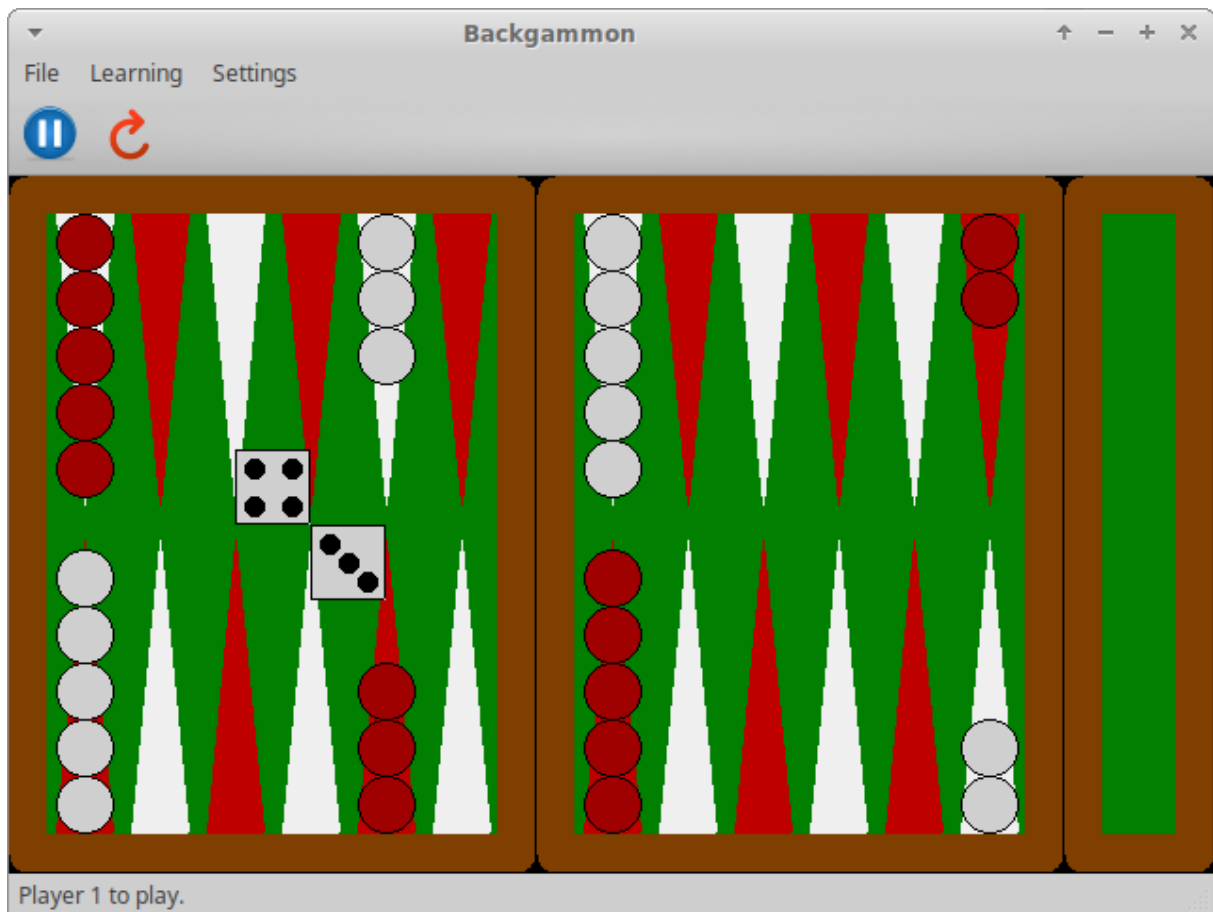


Figure 8.20: Watching the Generic, Self-Improving Predictor learning to play Backgammon

Once again, the Generic, Self-Improving Predictor has no idea it's playing Backgammon. It is given 28 values, which represent the current board state, and how these values change after all possible moves given the throw of the dice. The algorithm is then simply required to pick one. This abstraction requires that the game not only ensure that all the rules are being followed, but it requires that the game provide all the possible legal moves given the throw of the dice. This was done by adding another layer above the game that literally tests every potential move and collects all those that don't break the rules.

Unlike Tic-Tac-Toe, the best move to make given a specific state and a specific dice throw is not known. Therefore, testing the Generic, Self-Improving Predictor's ability to learn the correct move

against known correct moves is not possible. Instead the game plays against another player that doesn't learn and the number of games the learning player wins is then compared.

To provide the learning player with a reasonable opponent, a player that uses a simple heuristic to estimate the value of each state: a pip count, was used. The pip count heuristic is limited in that each possible next state has the same state unless a piece is taken. Therefore, the heuristic will result in the algorithm choosing to take the opponent's piece that is furthest around the board. This is the same strategy used by beginners, but beginners quickly learn to extend this strategy to not take a piece if it leaves them in an even more vulnerable position should the opponent take their pieces on the next turn. This basic algorithm doesn't learn, but it provides a good player to test against.

The results of eight different Generic, Self-Improving Predictors learning to play Backgammon against a basic player are presented in Figure 8.21.

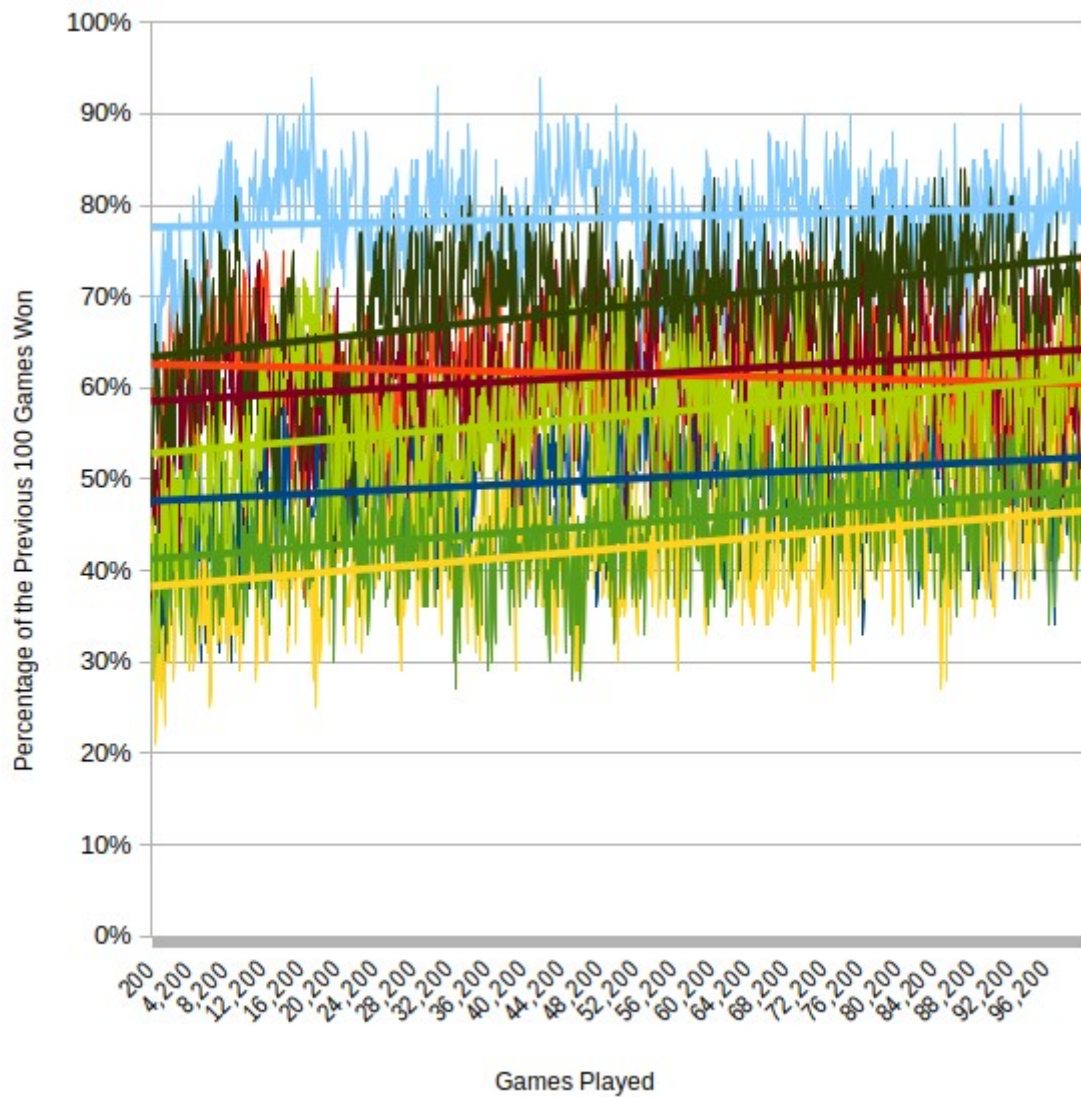


Figure 8.21: Results from different Generic, Self-Improving Predictors learning to play Backgammon by playing 100,000 games

The percentage of the last 100 games won varies considerably, mainly due to the probability of winning, regardless of the skill of the player, being very dependent on the roll of the dice, and the overall success of the Generic, Self-Improving Predictor is very dependent on the initial weights assigned to the underlying Flexible Artificial Neural Network. However, when trend lines are added, it is clear that, even after only a 100,000 games, all the Generic, Self-Improving Predictors, except for one, are performing better over time.

Similar results were achieved with different game state vector and game result vector formats.

Further improvements could be made by first training the Generic, Self-Improving Predictor on end games. Additional improvements could be made by improving the move selection algorithm to select moves based on the combination of moves from all the dice; instead of playing one die at a time. However, without being able to design better players or quantify the learning, further experiments would not provide much additional value to this research.

8.7 Summary

The Flexible Artificial Neural Network was first tested using standard, supervised learning, batch training on the MNIST handwritten digit database. The supervised learning approach was then adapted to simulate iterative learning, by providing the samples one at a time instead. Although slower, the final results are comparable.

The three Iterative Learners created: the MCMC Learner, the MC Network Learner and the TD Network Learner were then tested using progressively larger inputs and outputs. The results reveal how the Flexible Artificial Neural Network based learners are significantly slower when the input state size is small, but become comparable as the input state size increases.

To test the Generic, Self-Improving Predictor, games were used. Games provide a discrete state space, an automatic feedback mechanism, and the ability to use two Generic, Self-Improving Predictors playing against each other to provide an opponent that automatically improves simultaneously and adjusts to changes in strategy.

The Tic-Tac-Toe learning experiments produced a number of interesting and unexpected results. First, the way the environment state is represented does not have a significant impact on the Flexible Artificial Neural Network based learners' ability to learn. Even with the larger binary game

state representation, the number of hidden neurons required to learn how to play Tic-Tac-Toe is as low as 10.

Neither the size of the hidden layers, the learning rate nor the weight preference, in the case of the TD Network Learner, have a significant impact on the ability to learn. The key hyper-parameters that impact learning are the regularisation factor (γ) and the temporal discount rate (β). Using a regularisation factor prevented the Flexible Artificial Neural Network based learners from learning. Regularisation is not needed when the number of games played – the number of samples – is significantly greater than the number of parameters – the network weights. Therefore, the regularisation factor was set to zero. Reducing the temporal discount rate – the value of the game attributed to earlier moves – significantly improved the ability of the Flexible Artificial Neural Network based learners to learn the correct earlier moves. However, lowering the temporal discount rate too far, reduced the final ability to minimise the root-mean-square error. The best temporal discount rate found was 0.7.

The main advantage of using the Flexible Artificial Neural Network based learners is their ability to represent many more states. However, there is an additional advantage: speed. As the number of states explored by the MCMC Learner increases the time required to lookup a state increases. This noticeably increases the time required to play and learn. While the time required by the Flexible Artificial Neural Network based learners to play and learn remains fast and constant.

Chapter 9

Testing the Generic, Self-Improving Controller

9.1 *Introduction*

The aim of this research was to investigate a new approach to putting artificial intelligence into wearable human-machine interfaces by a creating a Generic, Self-Improving Controller that fuses data from multiple sensors and uses a self-improving predictor to continually improve the fidelity of the control signals used to drive multiple devices. To achieve this aim, the Generic, Self-Improving Controller needs to be configured as a wearable human-machine interface that fuses data from multiple sensors and is trained to identify gestures that are converted into control signals. To

demonstrate the control, the controller's control signals are used to control a mouse pointer. However, the fidelity of the control has not been measured; so it was used for demonstration purposes only.

Initially, the IMUs and the MMGs described in Chapter 3 were strapped to the users' forearm using a simple elasticated compression sleeve obtained from the local pharmacy. To make it easier to attach and ensure the devices did not move during rapid arm movements, the research group sewed the devices into an elasticated sleeve and added a Velcro loop. The resulting sleeve is shown in Figure 9.1.



Figure 9.1: Strap used to attach an IMU and multiple MMGs to the user's forearm

Since the objective was to detect user-defined gestures, it is important to note that where the IMUs and MMGs are connected to the user was not important. What was important was that the IMUs and MMGs can detect the gestures the user chooses. This is not only important when working with users with different handedness, but particularly important when working with users, who may not

have or be able to control a specific limb. The only limitation is that the user can only use the part of the body that the IMU or IMUs are connected to to perform their chosen gesture.

Two different types of gestures were used. The first type of gesture used were static gestures. Static gestures were selected because they are easier to detect. The testing of the Generic, Self-Improving Controller using user-defined static gestures is detailed in Section 9.2. The second type of gesture used were pre-defined dynamic gestures. Testing the Generic, Self-Improving Controller using pre-defined dynamic gestures is detailed in Section 9.3.

9.2 *Predicting User-Defined Static Gestures*

A static gesture is any gesture that does not involve movement. This eliminates the gyroscope from providing a useful signal, but it does make it easier to capture the gesture from the other sensors. A static gesture will result in a constant signal from the accelerometer, magnetometer and the calculated orientation quaternion. An MMG does not provide a constant signal, because muscle activation signals are not continuous. However, the trigger filters described in Section 3.6.8 were created to overcome this. The trigger filters can be used to covert the repeated muscle activation signal into a constant digital signal. The collection of trigger filtered MMG signals should then indicate which set of muscles are being activated by the gesture. However, most of the user selected gestures did not require continuous muscle activation. Therefore, not only was it more difficult to detect the signal from the MMG, most of the time the MMG signal did not provide a useful signal.

9.2.1 Experimental Set Up

The first experiment involved predicting an increasing number of required mouse pointer desired outcomes. Four users were asked to define five gestures. Four gestures that they wanted to use to move the mouse pointer up, down, left and right, and a fifth gesture that they wanted to use for a

left-click. Users were asked to select gestures using their preferred arm. The sensors were then connected to that arm. As discussed in Section 3.3.1, due to design limitations of the original IMU only a single MMG was connected. To ensure an identifiable signal could be detected, users were asked to select gestures that changed the orientation of the IMU.

Most users automatically used static gestures to move a mouse pointer in a particular direction. The typical gesture was to point in the direction the user wanted to move the mouse on the screen. Most users also automatically selected a dynamic gesture to generate a left-click. This is probably due to the habitual action associated with mouse clicking. However, to keep the benefits of using static gestures for all desired outcomes, the users were asked to select a static gesture for the left-click desired outcome. Most users then chose to point at the screen when a left-click was desired.

The Action Predictor GUI described in Chapter 7 was used to train the Generic, Self-Improving Controller. The user is randomly presented with each of the desired outcomes and required to perform the gesture that they have selected to realise that desired outcome. Each desired outcome is presented for two seconds before the next desired outcome is presented. The learning was divided into tranches of five samples per desired outcome: a total of ten samples per tranche for two desired outcomes, fifteen for three, and up to twenty-five samples per tranche for all five desired outcomes.

To focus on testing the machine learning algorithms developed in this research, the TD Network Learner was used. Since static gestures were specified, the gyroscope and MMG signals did not provide useful signals. In addition, as explained in Chapter 4, the orientation quaternions incorporate the accelerometer and magnetometer data. Therefore, to ensure that the input state vector presented to the underlying Generic, Self-Improving Predictor was information rich, but as short as possible, the IMU Data Collector was configured to only use the orientation quaternions. Furthermore, since a quaternion maximises the information density, a Flexible Artificial Neural

Network with no hidden layers was used. Eliminating the hidden layer ensured that the rate of learning was maximised.

Although users were asked to select gestures for five desired outcomes, the first experiment only used two of the desired outcomes: Move Left and Move Right. The number of desired outcomes was then steadily increased until the users were being asked to perform all their chosen gestures for all five of the desired outcomes.

9.2.2 Experimental Results

The results from the Generic, Self-Improving Controller predicting the desired outcome from two user-defined static gestures from four users are presented in Table 9.1.

Table 9.1: Results from learning to predict two user-defined static gestures

Ave. Iterations per Gesture	0		5		10		15		20		25	
	Left	Right	Left	Right	Left	Right	Left	Right	Left	Right	Left	Right
TD Net Learner 1												
Left	51	49	55	45	54	46	58	42	63	37	65	35
Right	53	47	46	54	38	62	36	64	35	65	38	62
TD Net Learner 2												
Left	54	46	49	51	62	38	64	36	61	39	64	36
Right	53	47	36	64	39	61	40	60	35	65	33	67
TD Net Learner 3												
Left	50	50	47	53	68	32	65	35	71	29	71	29
Right	50	50	30	70	38	62	30	70	42	58	33	67
TD Net Learner 4												
Left	48	52	62	38	65	35	69	31	64	36	61	39
Right	50	50	47	53	46	54	45	55	34	66	33	67
Percentage Correct												
Left	62%	38%	50%	50%	100%	0%	100%	0%	100%	0%	100%	0%
Right	75%	25%	0%	100%	0%	100%	0%	100%	0%	100%	0%	100%
Total Correct												
	Ave.	SEM	Ave.	SEM	Ave.	SEM	Ave.	SEM	Ave.	SEM	Ave.	SEM
	44%	19%	75%	25%	100%	0%	100%	0%	100%	0%	100%	0%

The columns represent each learning tranche of five samples per gesture for each of the two desired outcomes. Column one and two represent the pre-training predictions for the “Move Left” and “Move Right” desired outcomes. Columns three and four the training predictions for the “Move Left” and “Move Right” desired outcomes after five samples per gesture; then ten samples per gesture; and so on.

The rows are grouped into the predictions for each of the tested desired outcomes obtained from each of the four users. They are labelled with the Iterative Learner used and the number assigned to the user. The values are the Generic, Self-Improving Predictor’s outputs with the soft-max function (5.39) applied. Therefore, the first row is the Generic, Self-Improving Predictor’s predictions when the user was asked to perform their chosen gesture for the desired outcome of “Move Left”. The second row contains the predictions when the first user was asked to perform their gesture for the desired outcome “Move Right”.

To help visualise the results and the progress made as more samples per gesture are tested, the cells are coloured from red to green depending on the predicted value. Incorrect predictions are coloured red and correct predictions are coloured green. For cells where the rows and columns contain the same requested and predicted outcome combination, a value below 50% would be an incorrect prediction. For cells where the rows and columns do not contain the same combination, a value above 50% would be incorrect. Incorrect predictions are coloured red. When only two desired outcomes are tested, any value over 50% for the correct desired outcome would be correct. Similarly, for the other outcome, a value below 50% would be correct. However, to reflect the uncertainty in a prediction only slightly over 50% or under 50%, the colour green is only assigned when the prediction for the correct desired outcome reaches an arbitrarily selected 66%; and the incorrect desired outcome reaches 33%.

Following the four groupings for each of the four users, the “Percentage Correct” grouping contains the summary of the predictions. In other words, the first row represents how often the Generic, Self-Improving Predictor predicted “Move Left” versus “Move Right” when the user was asked to perform the gesture for the desired outcome “Move Left”. Similarly, the second row represents how often the predictor predicted “Move Left” versus “Move Right” when the user was asked to perform the gesture for the desired outcome “Move Right”.

Finally, the last row represents the total predictive ability of the Generic, Self-Improving Controller. In other words, how often, on average, across all users, the controller would have sent the correct signal when the user performed their chosen gesture; along with the standard error of the mean (SEM). The results show that the Generic, Self-Improving Controller is able to learn with 100% accuracy two user selected gestures with an average of less than 10 samples per gesture.

Table 9.2 shows the results of learning to predict three user selected gestures for moving a computer mouse in three specified directions using the same Generic, Self-Improving Controller. The table layout is the same as for two gestures; with the addition of the third gesture. The only change made is in the colouring of the cells.

Table 9.2: Results from learning to predict three user-defined static gestures

Ave. Iterations per Gesture	0			5			10			15		
	Left	Right	Up	Left	Right	Up	Left	Right	Up	Left	Right	Up
TD Net Learner 1												
Left	34	34	32	34	29	37	39	25	37	46	26	28
Right	34	34	32	27	40	33	27	35	38	28	44	29
Up	34	34	32	28	33	39	28	27	45	32	31	37
TD Net Learner 2												
Left	32	32	36	41	27	32	42	26	32	42	25	33
Right	34	34	33	31	38	31	29	38	33	27	40	33
Up	32	32	36	35	30	36	31	29	40	29	28	43
TD Net Learner 3												
Left	33	32	35	40	29	31	48	25	27	50	24	26
Right	36	31	33	30	40	30	31	39	31	31	39	30
Up	36	30	34	34	32	34	38	29	34	37	29	34
TD Net Learner 4												
Left	32	32	37	38	31	31	40	28	32	45	25	30
Right	31	32	37	28	42	30	26	45	28	28	40	32
Up	32	32	37	31	36	33	30	34	36	30	29	41
Percentage Correct												
	Left	Right	Up	Left	Right	Up	Left	Right	Up	Left	Right	Up
Left	13%	13%	74%	75%	0%	25%	100%	0%	0%	100%	0%	0%
Right	50%	25%	25%	0%	100%	0%	0%	75%	25%	0%	100%	0%
Up	37%	13%	50%	13%	25%	62%	25%	0%	75%	25%	0%	75%
Total Correct												
	Ave.	SEM		Ave.	SEM		Ave.	SEM		Ave.	SEM	
	29%	11%		79%	11%		83%	8%		92%	8%	
Ave. Iterations per Gesture												
	20			25			30			35		
	Left	Right	Up	Left	Right	Up	Left	Right	Up	Left	Right	Up
TD Net Learner 1												
Left	44	23	32	45	24	31	49	25	26	44	25	32
Right	27	39	34	27	40	33	27	45	28	26	42	33
Up	28	26	47	28	26	46	32	29	39	25	27	48
TD Net Learner 2												
Left	44	25	30	46	25	28	45	24	31	48	24	28
Right	28	42	30	26	45	30	26	43	32	28	46	27
Up	30	28	42	28	28	44	27	26	48	29	28	43
TD Net Learner 3												
Left	49	24	28	45	23	32	47	25	28	50	23	27
Right	29	40	32	25	43	32	25	47	27	30	40	30
Up	33	27	40	27	27	46	28	32	40	31	26	43
TD Net Learner 4												
Left	40	23	37	46	25	29	43	26	30	43	26	31
Right	25	36	39	26	45	29	25	44	31	26	46	28
Up	25	25	50	28	31	41	26	29	45	28	27	45
Percentage Correct												
	Left	Right	Up	Left	Right	Up	Left	Right	Up	Left	Right	Up
Left	100%	0%	0%	100%	0%	0%	100%	0%	0%	100%	0%	0%
Right	0%	75%	25%	0%	100%	0%	0%	100%	0%	0%	100%	0%
Up	0%	0%	100%	0%	0%	100%	0%	0%	100%	0%	0%	100%
Total Correct												
	Ave.	SEM		Ave.	SEM		Ave.	SEM		Ave.	SEM	
	92%	8%		100%	0%		100%	0%		100%	0%	

A correct prediction requires the predicted value for the correct desired outcome to be greater than the other outcomes. To ensure a correct prediction, regardless of the number of other outcomes, requires a value of greater than 50%. However, this is a maximum requirement. The minimum requirement is the inverse of the number of of desired outcomes being predicted. For two gestures, the minimum requirement is also 50%. For three gestures, the minimum requirement is 33%. The actual requirement depends on the predictions for the other outcomes. If the predictions for the other outcomes are equal, then the requirement for the correct, desired outcome is 33%. If the prediction for all but one of the other outcomes is zero, the non-zero incorrect outcome will determine the value required for the correct, desired outcome. The soft-max function (5.39) will ensure that the outputs sum to 1 and the order of predictions will remain consistent. It also tends to squash the values. In other words, the difference between the values is reduced. Therefore, the required predicted value tends to be close to the minimum requirement.

With more than two gestures, for cells where the rows and columns contain the same requested and predicted outcome combination, a value below the minimum requirement are coloured red. Similarly, for cells where the rows and columns do not contain the same combination, a value above the minimum requirement are coloured red. To reflect both the uncertainty and the unknown, variable actual requirement for a correct prediction, an arbitrarily selected value of one third of the minimum requirement was selected to represent the range of the colour spectrum. This is the same relative range selected when only two gestures were being tested.

The Generic, Self-Improving Controller is able to learn with 100% accuracy three user selected gestures with an average of less than 20 samples per gesture.

Tables 9.3 and 9.4 show the results with four user selected gestures, and Tables 9.5, 9.6 and 9.7 show the results of learning to predict all five user selected gestures for all four mouse pointer move directions and a mouse click.

Table 9.3: Results from learning to predict four user-defined static gestures – Part 1

Ave. Iterations per Gesture	0				5				10				15				20				25			
	Left	Right	Up	Down	Left	Right	Up	Down	Left	Right	Up	Down	Left	Right	Up	Down	Left	Right	Up	Down	Left	Right	Up	Down
TD Net Learner 1																								
Left	24	26	24	25	34	22	23	22	36	19	23	22	40	19	21	20	37	19	22	22	38	18	21	22
Right	25	26	25	23	23	32	23	22	22	31	24	23	22	32	23	22	20	35	22	23	21	29	24	25
Up	25	26	25	24	26	28	26	21	27	24	29	21	28	24	28	20	24	26	30	20	26	22	31	21
Down	25	26	25	25	25	29	21	24	25	26	22	27	27	25	21	27	22	27	20	31	22	22	20	36
TD Net Learner 2																								
Left	25	25	24	27	31	21	22	26	34	20	23	23	34	20	23	22	36	19	22	23	34	19	24	23
Right	23	24	26	26	23	27	22	27	22	28	24	25	20	38	21	21	21	32	23	24	20	30	26	24
Up	24	25	25	26	27	27	23	23	27	24	29	21	24	29	28	20	25	25	30	21	22	22	36	20
Down	23	24	26	27	23	23	21	32	24	23	21	32	23	29	21	27	23	23	20	34	22	23	21	34
TD Net Learner 3																								
Left	23	26	25	26	28	21	27	23	32	21	23	24	32	20	26	22	36	19	22	23	37	20	21	22
Right	24	24	26	27	22	23	29	25	22	27	24	27	21	27	29	23	22	30	24	24	20	39	20	21
Up	23	24	27	26	24	21	33	21	26	24	28	22	23	21	37	19	26	24	30	21	25	30	25	19
Down	24	25	25	27	25	23	24	29	23	23	20	34	23	25	22	30	23	24	20	33	22	29	19	31
TD Net Learner 4																								
Left	26	25	24	25	28	22	21	29	29	22	26	24	29	20	25	27	32	19	25	23	32	20	23	25
Right	26	24	26	24	23	25	22	30	22	30	25	23	21	30	24	25	22	30	26	23	20	35	22	23
Up	25	25	25	25	26	26	23	25	23	25	32	20	22	25	31	21	23	22	35	19	23	28	29	21
Down	27	24	25	25	22	22	20	36	23	26	22	30	21	24	20	35	23	26	21	30	21	26	19	34
Percentage Correct																								
Left	24%	38%	0%	38%	75%	0%	0%	25%	100%	0%	0%	0%	100%	0%	0%	0%	100%	0%	0%	0%	100%	0%	0%	0%
Right	13%	25%	25%	37%	0%	38%	24%	38%	0%	87%	0%	13%	0%	75%	25%	0%	0%	100%	0%	0%	0%	100%	0%	0%
Up	7%	31%	31%	31%	25%	50%	25%	0%	0%	0%	100%	0%	13%	25%	62%	0%	0%	100%	0%	0%	25%	75%	0%	0%
Down	25%	25%	0%	50%	0%	25%	0%	75%	0%	0%	0%	100%	13%	25%	0%	62%	0%	0%	100%	0%	0%	0%	0%	100%
Total Correct	Ave. SEM	33%	6%		Ave. SEM	53%	13%		Ave. SEM	97%	3%		Ave. SEM	75%	9%		Ave. SEM	100%	0%		Ave. SEM	94%	6%	

Table 9.4: Results from learning to predict four user-defined static gestures – Part 2

Ave. Iterations per Gesture	30				35				40				45				50			
	Left	Right	Up	Down	Left	Right	Up	Down	Left	Right	Up	Down	Left	Right	Up	Down	Left	Right	Up	Down
TD Net Learner 1																				
Left	38	18	21	23	38	18	22	22	35	18	23	24	40	18	21	22	34	20	23	23
Right	21	27	24	28	21	30	25	24	20	31	23	26	21	31	24	24	20	34	23	23
Up	25	22	33	21	24	21	36	19	22	22	36	20	23	21	36	19	23	22	36	19
Down	21	21	19	39	21	23	20	36	20	23	19	38	22	23	19	36	20	25	20	35
TD Net Learner 2																				
Left	34	19	24	23	33	19	24	24	38	19	21	22	35	19	23	23	40	18	21	21
Right	20	32	24	23	20	32	24	25	20	34	23	23	20	34	23	23	20	37	22	21
Up	22	23	35	20	22	23	35	20	23	24	33	20	23	21	36	19	23	25	33	19
Down	22	23	20	35	22	22	20	35	22	23	20	35	21	24	20	35	24	26	20	30
TD Net Learner 3																				
Left	38	19	21	22	39	18	21	22	34	19	24	23	39	17	22	21	38	18	24	20
Right	22	33	23	23	22	32	22	23	20	33	23	23	22	28	25	26	21	36	22	21
Up	26	24	29	20	27	24	29	20	23	23	35	19	26	20	35	19	23	25	32	19
Down	23	24	20	33	24	23	20	34	22	23	20	35	24	21	20	36	24	26	20	30
TD Net Learner 4																				
Left	37	19	21	24	32	19	27	22	38	18	22	22	36	18	22	23	35	19	23	23
Right	21	33	22	24	21	31	26	22	21	34	22	23	20	35	22	23	21	34	23	23
Up	26	26	28	20	21	22	38	19	25	23	33	20	24	24	32	20	22	25	35	20
Down	22	23	19	35	22	24	21	33	22	24	19	34	21	24	19	36	22	24	19	35
Percentage Correct																				
Left	100%	0%	0%	0%	100%	0%	0%	0%	100%	0%	0%	0%	100%	0%	0%	0%	100%	0%	0%	0%
Right	0%	75%	0%	25%	0%	100%	0%	0%	0%	100%	0%	0%	0%	100%	0%	0%	0%	100%	0%	0%
Up	0%	0%	100%	0%	0%	0%	100%	0%	0%	0%	100%	0%	0%	0%	100%	0%	0%	0%	100%	0%
Down	0%	0%	0%	100%	0%	0%	0%	100%	0%	0%	0%	100%	0%	0%	0%	100%	0%	0%	0%	100%
Total Correct																				
Ave. SEM	94%	6%			100%	0%			100%	0%			100%	0%			100%	0%		
Ave. SEM																				

Table 9.5: Results from learning to predict five user-defined static gestures – Part 1

Ave. Iterations per Gesture	0										5										10										15										20									
	Left	Right Up	Down	Front	Left	Right Up	Down	Front	Left	Right Up	Down	Front	Left	Right Up	Down	Front	Left	Right Up	Down	Front	Left	Right Up	Down	Front	Left	Right Up	Down	Front	Left	Right Up	Down	Front	Left	Right Up	Down	Front	Left	Right Up	Down	Front										
TD Net Learner 1	19	19	20	21	20	19	17	23	21	20	28	17	18	19	17	26	17	21	19	17	28	15	20	18	19	20	19	23	20	19	20	17	27	20	20	17	17	28	15	20	18	19								
Left	19	19	21	21	19	18	19	21	21	20	18	26	18	20	18	17	27	20	20	20	17	22	19	19	23	17	22	19	19	23	17	22	19	19	23	17	22	19	19	23										
Right	20	19	20	21	20	18	18	25	19	20	21	22	21	18	19	19	20	27	17	18	18	18	27	16	20	20	18	27	16	20	18	18	18	18	18	18	18	18	18	18	18									
Up	18	20	22	21	19	18	19	19	24	20	19	22	17	24	18	18	21	17	24	18	18	21	17	24	18	18	21	17	24	18	18	21	17	24	18	18	18	18	18	18	18	18	18							
Down	19	20	21	21	18	18	19	22	21	20	19	24	19	24	19	24	19	24	19	24	19	24	19	24	19	24	19	24	19	24	19	24	19	24	19	24	19	24	19	24	19	24								
Click	19	20	21	21	18	18	19	22	21	20	19	24	19	24	19	24	19	24	19	24	19	24	19	24	19	24	19	24	19	24	19	24	19	24	19	24	19	24	19	24	19	24								
TD Net Learner 2	20	20	20	20	21	24	18	18	22	19	25	17	19	19	20	25	15	22	18	19	31	16	18	17	18	19	20	18	17	18	18	18	18	18	18	18	18	18	18	18	18	18								
Left	19	20	20	20	21	19	24	18	21	19	18	22	19	19	21	17	20	22	18	22	18	24	19	18	20	19	20	18	20	19	20	18	24	19	18	20	19	20	18	20	19	20								
Right	19	20	20	20	21	21	22	19	19	20	20	18	24	17	21	18	16	32	16	19	20	17	27	16	19	20	17	17	16	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18						
Up	20	20	20	20	21	21	22	19	19	20	20	18	24	17	21	18	16	32	16	19	20	17	27	16	19	20	17	17	16	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18					
Down	20	20	20	20	20	19	20	17	25	18	19	20	17	24	20	19	18	18	23	22	21	20	17	23	20	19	20	17	23	20	19	20	17	23	20	19	20	17	23	20	19	20	17	23						
Click	19	20	20	20	20	20	22	18	21	19	19	20	20	19	22	18	18	23	18	22	20	21	20	18	22	20	21	20	18	21	20	18	21	20	18	21	20	18	21	20	18	21	20	18	21					
TD Net Learner 3	20	19	20	21	20	18	17	22	21	21	25	16	19	19	21	22	16	22	22	18	26	16	21	18	19	20	21	18	19	20	21	18	18	18	18	18	18	18	18	18	18	18	18	18	18					
Left	20	19	20	21	20	17	22	20	20	20	21	19	19	20	23	17	19	21	23	21	17	21	20	21	17	21	20	20	21	17	21	20	20	21	17	21	20	20	21	17	21	20	21	17	21					
Right	20	21	19	20	20	17	20	25	18	20	21	17	24	17	21	18	17	29	18	19	18	18	29	17	19	18	18	29	17	19	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18					
Up	20	19	20	22	19	17	20	18	24	20	19	18	17	25	21	17	17	17	30	19	18	18	17	27	20	18	18	17	27	20	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18					
Down	20	20	19	21	19	18	21	20	21	21	20	18	19	19	23	18	18	21	22	21	18	19	20	20	18	19	20	20	20	18	19	20	20	20	18	19	20	20	18	19	20	20	18	19						
Click	20	20	19	21	19	18	21	20	21	21	20	18	19	19	23	18	18	21	22	21	18	19	20	20	18	19	20	20	20	18	19	20	20	20	18	19	20	20	18	19	20	20	18	19						
TD Net Learner 4	20	21	20	18	27	18	19	18	18	18	25	18	19	19	19	27	17	18	18	19	29	16	19	18	19	20	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18					
Left	20	20	21	19	20	20	20	20	20	20	18	21	19	20	21	17	25	18	19	21	18	26	18	18	20	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18					
Right	20	21	19	19	24	19	21	17	19	20	20	19	24	17	20	20	21	22	17	20	19	21	25	16	19	20	18	21	25	16	19	20	18	18	18	18	18	18	18	18	18	18	18	18	18					
Up	20	21	19	19	24	19	21	17	19	20	19	20	19	24	17	20	21	22	17	20	19	21	25	16	19	20	18	21	25	16	19	20	18	18	18	18	18	18	18	18	18	18	18	18	18					
Down	20	20	21	20	19	22	19	18	22	19	19	20	17	24	20	18	21	16	26	19	18	22	17	24	20	18	22	17	24	20	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18					
Click	20	20	22	19	19	22	20	19	19	20	19	21	20	19	21	19	22	18	19	21	19	23	19	21	19	21	19	21	19	21	19	21	19	21	19	21	19	21	19	21	19	21	19	21						
Percentage Correct	Left	0%	13%	13%	50%	24%	50%	0%	50%	0%	0%	0%	0%	0%	0%	0%	84%	0%	8%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%						
Right	0%	6%	44%	44%	6%	5%	54%	18%	18%	5%	0%	62%	0%	38%	0%	0%	0%	13%	24%	0%	0%	62%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%					
Up	0%	37%	13%	25%	25%	25%	25%	50%	0%	0%	0%	25%	75%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%					
Down	0%	6%	57%	31%	6%	13%	0%	0%	87%	8%	0%	0%	0%	100%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%					
Click	0%	6%	44%	44%	6%	25%	34%	25%	8%	8%	0%	38%	0%	62%	0%	0%	0%	25%	25%	0%	0%	38%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%					
Total Correct	Ave. SEM	11%	5%			Ave. SEM	50%	12%			Ave. SEM	80%	9%			Ave. SEM	67%	19%			Ave. SEM	85%	9%																											

Table 9.6: Results from learning to predict five user-defined static gestures – Part 2

Ave. Iterations per Gesture	25					30					35					40					45				
	Left	Right Up	Down Front	Left	Right Up	Down Front	Left	Right Up	Down Front	Left	Right Up	Down Front	Left	Right Up	Down Front	Left	Right Up	Down Front	Left	Right Up	Down Front				
TD Net Learner 1																									
Left	27	15	21	17	19	30	15	18	18	19	29	15	21	18	17	30	16	19	18	17	28	16	20	19	18
Right	17	24	19	18	22	17	22	17	18	26	17	24	20	19	20	17	27	18	19	19	17	26	19	19	20
Up	18	18	29	16	20	19	17	24	16	23	17	17	32	16	18	19	19	26	17	19	17	18	30	16	19
Down	18	19	17	23	23	17	18	16	25	24	17	18	17	29	19	18	19	16	28	19	17	19	16	28	19
Click	18	20	20	18	24	18	18	17	18	28	18	20	21	20	22	19	22	19	19	22	18	21	20	19	22
TD Net Learner 2																									
Left	30	16	18	17	19	33	15	18	17	17	28	16	17	19	20	28	16	17	19	20	29	15	19	18	18
Right	17	26	18	17	22	18	23	19	18	21	17	24	18	19	22	17	25	17	18	23	17	24	18	18	22
Up	19	19	25	16	20	21	17	27	16	19	19	19	24	17	22	19	19	25	16	21	17	18	29	16	20
Down	19	22	16	22	22	20	19	16	25	20	17	19	16	28	20	17	19	16	28	20	17	19	16	29	20
Click	19	22	18	18	23	20	20	19	19	22	18	21	18	19	24	18	20	18	19	25	18	20	19	19	25
TD Net Learner 3																									
Left	27	16	19	18	20	32	15	19	18	17	29	16	18	20	18	30	15	18	18	19	27	16	21	18	17
Right	17	23	19	18	22	18	23	20	18	21	17	23	19	20	21	16	21	18	18	27	16	27	20	18	19
Up	19	18	28	16	20	20	17	28	16	18	19	18	26	17	20	18	17	26	16	23	17	18	32	16	17
Down	19	19	17	24	21	19	19	17	25	20	17	18	16	29	19	17	17	16	26	24	18	21	17	26	19
Click	19	20	19	19	23	20	20	20	19	22	19	20	19	20	22	17	18	18	18	29	18	22	21	19	21
TD Net Learner 4																									
Left	28	16	17	18	21	32	15	17	19	17	27	16	21	19	17	30	16	18	17	18	29	15	20	19	17
Right	17	25	17	18	24	18	24	18	20	20	17	24	20	20	18	17	28	18	17	19	17	25	19	20	19
Up	19	19	23	16	23	23	18	22	17	20	19	17	31	16	17	19	20	27	16	18	17	19	29	16	18
Down	17	20	16	25	22	18	18	16	31	17	17	18	16	31	17	18	21	17	26	18	17	18	16	31	18
Click	19	20	18	18	26	20	20	18	21	21	19	20	21	20	19	19	23	19	19	21	18	21	19	21	21
Percentage Correct																									
Left	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%
Right	0%	100%	0%	0%	0%	0%	0%	0%	0%	25%	0%	100%	0%	0%	0%	0%	75%	0%	0%	25%	0%	100%	0%	0%	0%
Up	0%	0%	87%	0%	13%	25%	0%	75%	0%	0%	0%	0%	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%	100%	0%	0%
Down	0%	8%	0%	71%	21%	0%	0%	0%	100%	0%	0%	0%	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	100%	0%
Click	0%	0%	0%	0%	100%	0%	0%	0%	13%	87%	0%	0%	25%	0%	75%	0%	38%	0%	0%	62%	0%	33%	0%	8%	59%
Total Correct	Ave. SEM	92%	6%			Ave. SEM	87%	6%			Ave. SEM	95%	5%			Ave. SEM	87%	8%			Ave. SEM	92%	8%		

Table 9.7: Results from learning to predict five user-defined static gestures – Part 3

Ave. Iterations per Gesture	50				55				60				65				70							
	Left	Right Up	Down Front	Right Up	Left	Right Up	Down Front	Right Up	Left	Right Up	Down Front	Right Up	Left	Right Up	Down Front	Right Up	Left	Right Up	Down Front	Right Up				
TD Net Learner 1																								
Left	33	15	17	19	16	33	15	17	19	16	33	15	18	18	17	31	16	18	17	32	15	17	20	16
Right	18	23	19	19	20	17	22	18	18	24	17	24	18	18	24	16	26	17	17	24	17	26	18	19
Up	21	17	28	16	18	20	17	28	16	20	20	17	26	16	21	19	18	26	16	21	21	18	26	17
Down	19	18	16	29	18	18	18	16	27	21	18	18	16	26	23	17	19	16	25	22	17	18	16	32
Click	20	20	19	20	22	18	19	19	19	26	18	19	18	18	27	17	20	17	18	28	19	20	18	20
TD Net Learner 2																								
Left	28	16	19	19	18	32	15	18	18	16	27	16	17	17	20	31	15	17	19	17	30	15	20	19
Right	17	27	19	18	19	17	27	18	19	18	17	26	18	18	22	17	25	17	19	22	17	23	20	19
Up	18	17	32	16	18	20	19	28	16	17	20	18	27	16	20	19	18	26	16	20	18	17	32	16
Down	17	19	16	29	18	18	18	16	31	17	19	18	16	27	19	17	18	16	31	19	18	18	17	28
Click	18	21	19	20	21	20	21	19	20	20	19	21	18	18	24	18	20	18	19	24	19	19	20	22
TD Net Learner 3																								
Left	31	15	19	19	17	28	16	19	18	17	30	16	19	18	19	28	15	20	20	17	32	16	18	19
Right	17	28	18	18	19	16	31	17	18	18	17	27	19	17	20	17	23	21	19	20	17	26	19	19
Up	18	19	29	16	18	17	20	29	16	18	17	18	32	15	18	17	16	34	16	18	20	17	29	16
Down	18	20	16	28	18	17	21	16	28	18	18	20	16	26	20	17	18	16	30	19	18	18	16	30
Click	19	22	19	19	21	18	23	18	19	21	18	21	19	18	23	18	19	21	20	23	19	20	20	21
TD Net Learner 4																								
Left	30	15	20	17	18	34	15	17	17	17	31	16	18	18	16	28	16	17	17	21	31	16	18	17
Right	18	22	20	18	22	17	26	18	18	21	18	24	19	18	21	17	24	18	17	24	17	25	19	18
Up	18	17	31	16	19	21	19	25	16	20	19	18	30	15	18	21	17	27	16	20	20	17	30	16
Down	18	18	17	27	21	19	18	16	28	19	20	18	16	26	20	19	18	16	25	22	19	18	17	
Click	18	19	20	19	24	20	20	18	18	19	23	21	19	19	18	19	19	18	18	26	20	20	19	
Percentage Correct																								
Left	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%	100%	0%	0%	0%	100%	0%	0%	0%	0%
Right	0%	87%	0%	0%	13%	0%	75%	0%	0%	25%	0%	13%	0%	0%	13%	0%	87%	0%	0%	13%	0%	100%	0%	0%
Up	0%	0%	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%	100%	0%	0%	0%	100%	0%	0%
Down	0%	0%	0%	100%	0%	0%	0%	0%	100%	0%	0%	0%	100%	0%	0%	0%	0%	0%	100%	0%	0%	0%	100%	0%
Click	0%	38%	0%	0%	62%	0%	50%	0%	0%	50%	0%	0%	0%	0%	50%	0%	0%	0%	0%	100%	0%	0%	0%	100%
Total Correct	Ave. SEM	90%	7%			Ave. SEM	85%	10%			Ave. SEM	97%	3%			Ave. SEM	97%	3%			Ave. SEM	100%	0%	

The results are summarised in Figure 9.2. Each line is associated with a different number of user selected gestures for obtaining desired outcomes. The lines plot the values in the last row of the above tables: the total predictive ability of the Generic, Self-Improving Controller, or how often, on average, across all users, the controller would have sent the correct signal when the user performed their chosen gesture. The error bars represent the standard error of the mean (SEM).

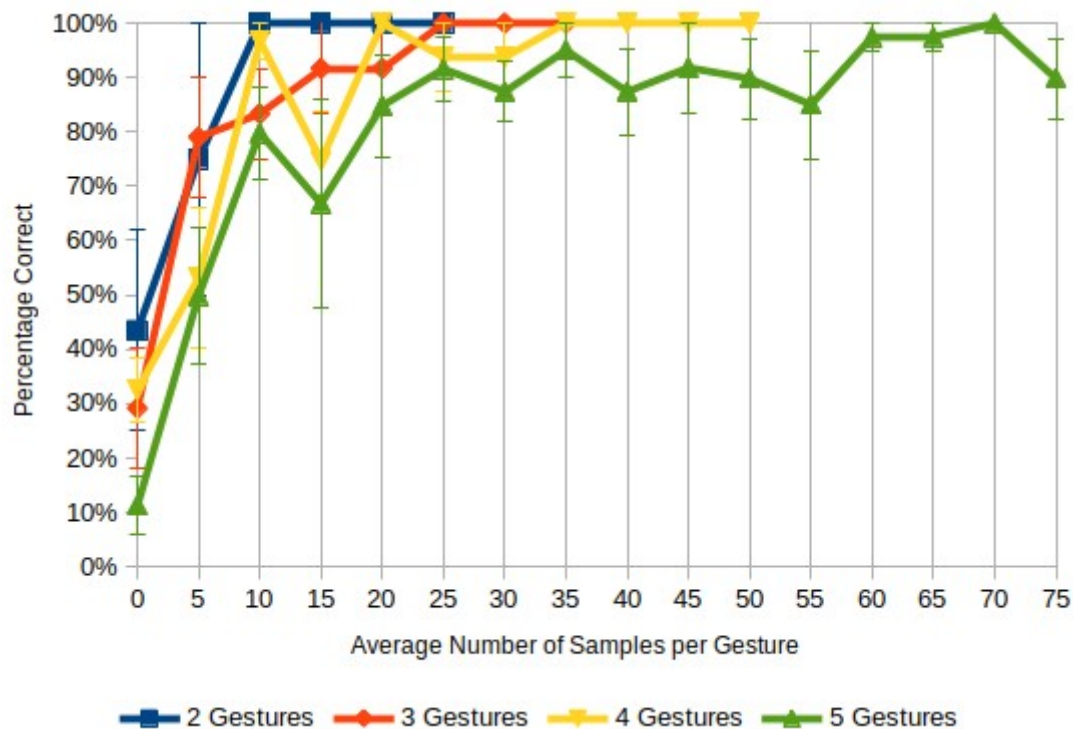


Figure 9.2: Comparing the number of samples per gesture vs. predictive accuracy required to predict increasing numbers of user-defined static gestures

The results show that the Generic, Self-Improving Controller was able to learn two, three and four user-defined static gestures with an accuracy of 100% after an average of less than 10, 20 and 35 samples per gesture respectively. 100% accuracy was achieved with five user-defined static gestures on a number of occasions and at least 80% accuracy was achieved on all occasions. However, the number of samples per gesture has now increased to over 75. Furthermore, the 100% success rate was not maintained once it was reached.

9.2.3 Discussion of the Results

Although the Generic, Self-Improving Controller can learn to accurately predict four user-defined static gestures in under five minutes⁴⁹, a closer look at the data reveals some interesting issues. First, as seen in Figure 9.2, the number of training samples per gesture increases rapidly with each additional gesture. There appears to be doubling of the required samples per gesture for every additional gesture. The total number of gestures required does not increase linearly with the number of gestures, but exponentially.

Extrapolating, this suggests that ten gestures would require around 2,500 samples per gesture or 25,000 samples in total. At two seconds per sample, this would require over twelve hours of training data. As seen in Section 8.2, when training the Iterative Learners on ten hand-written numerals, it took at least 500,000 samples to learn how to classify the training data with 90% accuracy, which suggests even more samples may be required.

To overcome the shortage of data and the effort required to collect it, there is the option to replay the same data with the same settings to the same learner multiple times to simulate longer data gathering. This is the approach used in Section 9.3 when predicting pre-defined dynamic gestures. However, unfortunately, this results in reverting back to the standard supervised learning approach, which this research was trying to avoid.

The next issue requires looking more closely at the output data presented in Tables 9.1 and 9.2. Although the confidence levels for two gestures increases rapidly, with three gestures, despite the correct desired outcome being easily predicted, the confidence levels for the correct desired outcome remain much lower: barely exceeding 50%, and sometimes remaining as low as 40%. This issue becomes even more significant when looking at Tables 9.3 and 9.4. With four gestures, confidence levels for the correct desired outcome do not exceed 40%, and confidence levels are

⁴⁹ Four gestures $\times$ thirty-five samples per gesture $\times$ two seconds per gesture = 280 seconds.

sometimes as low as 30%. This is still sufficient for achieving a correct prediction with only four user-defined static gestures. However, when looking at Tables 9.5, 9.6 and 9.7, it reveals the reason for the failure to achieve 100% accuracy every time with five gestures. The confidence level is sometimes too low to confidently predict the correct desired outcome reliably. Two of the correct desired outcomes are being predicted with a confidence level of only 21%. Since the failure at five gestures consistently occurs between the same two desired outcomes: “Move Right” and “Left Click”, this can be partially explained by suggesting that the difference between the selected gestures for the desired outcomes is not significant enough. Although these are user-defined gestures, most users selected the same gestures for each desired outcome. The gesture selected remains important when using particular sensors to detect gestures.

Another factor to consider is the network used. If insufficient hidden units are used, the network may not be able to support the complexity of the different input signals. To facilitate fast learning, the number of weights that needed to be learnt was minimised in a two ways. First, only the quaternions were used; resulting in an equivalent input state vector length of only four values. This may explain some of the increasing difficulty in classifying more than four options. Second, the Flexible Artificial Neural Network used had no hidden layer. Therefore, it would not be able to separate non-linearly separable inputs. Finally, the learning rate used was kept artificially high at 0.1, which, as seen in Section 8.5.2, when training the Flexible Artificial Neural Network based learners on Tic-Tac-Toe, will result in volatility in the final output.

9.3 Predicting Pre-Defined Dynamic Gestures

Unlike static gestures, dynamic gestures involve movement. The gyroscope will provide a useful signal. All signals will vary with time. Detecting a signal requires a temporal sequence of sufficient length. Therefore, in addition to the trigger filters, the historical data feature of the IMU Data Collector will be required.

9.3.1 Experimental and Simulation Set Up

This research reused data captured by other members of the research group for their experiments. Unfortunately, instead of allowing users to pick their gestures for desired outcomes, the traditional approach of providing a list of gestures to perform was used. Seven experimental subjects were asked to perform a varying number of gestures, in a varying number of tranches with ten gestures samples per tranche. Unfortunately, when examining the data, it appears as if the IMU and MMG data was also captured at different rates. Table 9.8 provides a summary of the actual per subject metadata.

Table 9.8: Per subject dynamic gesture metadata

Subject	Gestures	Tranches	Samples per Tranche	Data Rate
A	18	2	10	500 Hz
C	12	10	10	500 Hz
D	4	10	10	500 Hz
M	12	10	10	500 Hz
N	12	10	10	500 Hz
S	38	10	10	1000 Hz

Since there are only 100 samples per gesture, and in one instance, only twenty samples per gesture, additional data was simulated. To create additional simulated samples, the data was reused. As covered in Chapter 2, training data can be increased by randomly varying or removing random data

elements to create additional samples. However, this was not done with this data for two reasons. First, the examples used in Chapter 2 are all based on image data, where the surrounding data is similar. With the IMU and MMG data the surrounding data is generally not similar. Second, even when the surrounding data is similar, filters are used to eliminate the noise. Therefore, generating new noise would be counter productive. However, reusing data shouldn't be a problem, because the regularisation factor added to the TD learning algorithm in Section 5.3 should eliminate the issue of over fitting created by reusing sample data.

All sensor data was filtered using the Median Filter described in Section 3.6.2. The filter duration was set to 3 ms to eliminate any data spikes. The MMG data was additionally band-pass filtered using the Simple Low-Pass Filter described in Section 3.6.4 with a 10 Hz cut-off frequency and the Simple High-Pass Filter described in Section 3.6.5 with a 100 Hz cut-off frequency. Finally, the Binary Trigger described in Section 3.6.8 with a 10 ms duration was applied to the MMG data to convert detected signals into a temporal digital input.

The Generic, Self-Improving Controller was configured with a TD Network Learner that used a Flexible Artificial Neural Network with a single hidden layer. The learning rate was initially set to 0.001. The regularisation factor was set to 0.001. And the temporal discount rate was set to 0.7.

Although the data input data rate was either 500 or 1,000 Hz, the learner's input data sample rate was set to 10 Hz. This ensured that each iterative learning step could be completed regardless of the length of historic data used. Increasing the length of the historic data increases the length of the input state vector. The length of the hidden layer required to ensure features could be identified in the historic data was correspondingly increased to twice the length of the input state vector. As long as the historic data length is greater than 50 ms, no data will go unused.

During each learning tranche, 9 out of the 10 samples were used for training, the final sample was used for testing. As described in Section 7.4, the samples within a tranche are selected at random

until all the samples have been used. When all the samples in a tranche have been used, the next tranche is used. When all the tranches have been used, the data is reused until the required number of samples per gesture have been presented to the learner.

9.3.2 Simulation Results

Figure 8.12 presents the initial results from learning two pre-defined gestures and simulating 1,800 samples per gesture. The x -axis represents the number of simulated samples per gesture that were presented to the underlying Generic, Self-Improving Predictor. Each learning tranche consisted of ten samples per gesture. Nine samples were used for training and one was used for testing. For clarity, only every tenth test result has been plotted. The y -axis represents the value predicted for a gesture, when that gesture was presented during testing. The output value of the underlying Generic, Self-Improving Predictor is converted into a prediction confidence by the Generic, Self-Improving Controller. Depending on the random values of the initial weights assigned to each neuron in the underlying Flexible Artificial Neural Network, the initial predicted values could be anything. Therefore, no value should be placed in the difference between the initial predicted values and an average random predicted value.

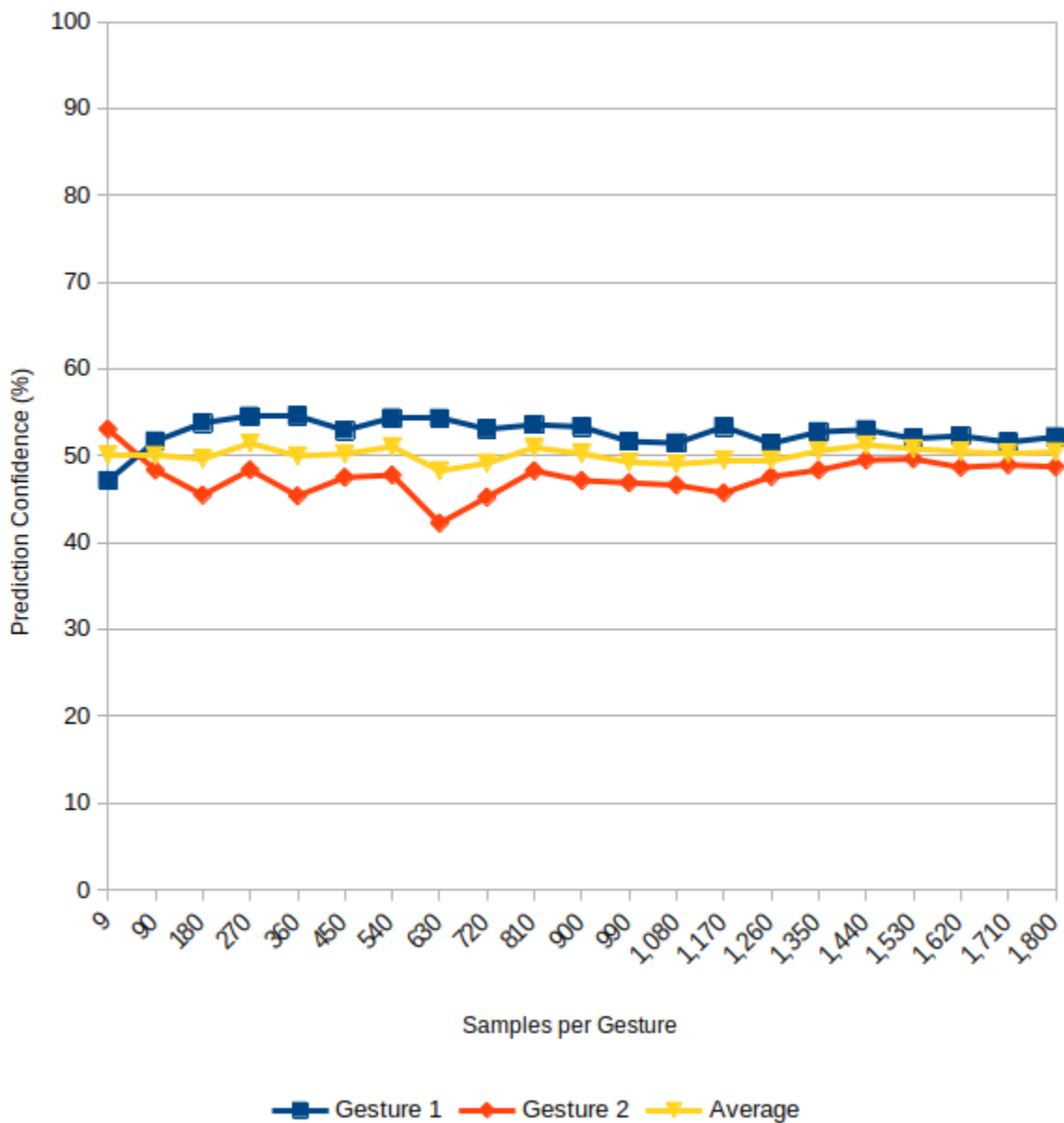


Figure 9.3: Initial results from learning to predict two pre-defined dynamic gestures

Unfortunately, the Generic, Self-Improving Controller does not appear to be learning the difference between the two gestures.

No historical data was used. Since the underlying Generic, Self-Improving Predictor is only sampling the input data at 10 Hz, the input data is under sampled. Therefore, for the next simulation, the length of the historic data used was set to 200 ms. 200 ms was used, because, as discussed in Chapter 2, the length of a muscle activation signal is expected to be around 200 ms.

Since it is known that the pre-defined gestures did not involve moving the forearm, the accelerometer, gyroscope, magnetometer and quaternion data is not expected to change over time. Therefore, to keep the length of the Flexible Artificial Neural Network's input state vector to a minimum, only the historical MMG data was used. The results from learning two pre-defined gestures and simulating 1,800 using 200 ms of historic MMG data are presented in Figure 9.4a.

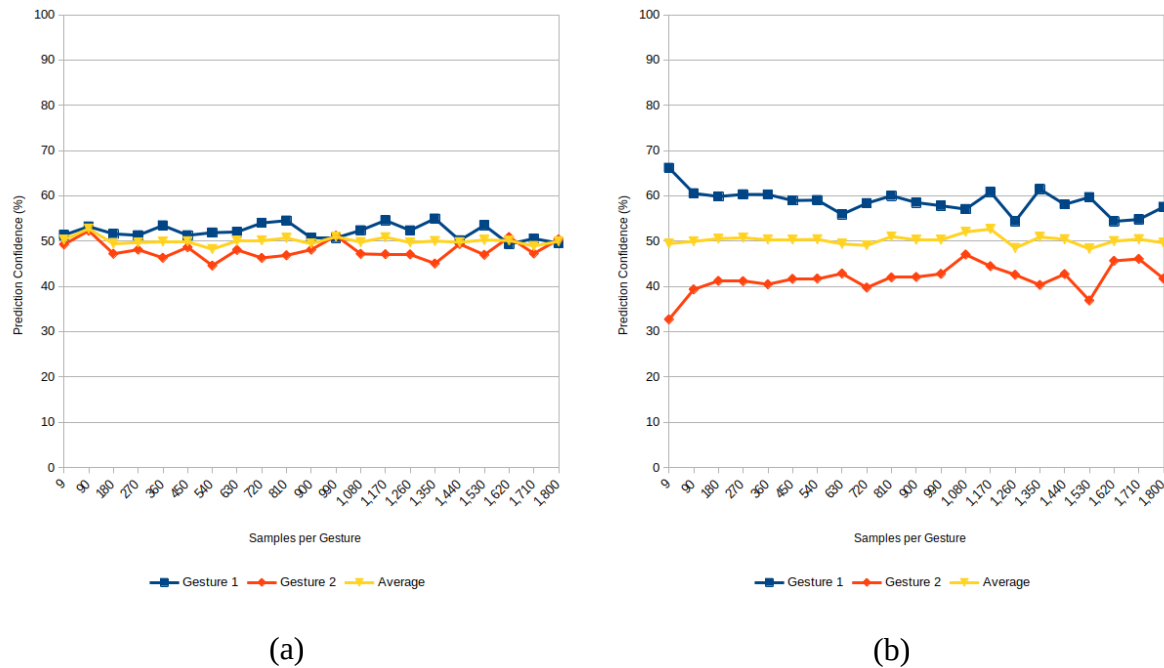


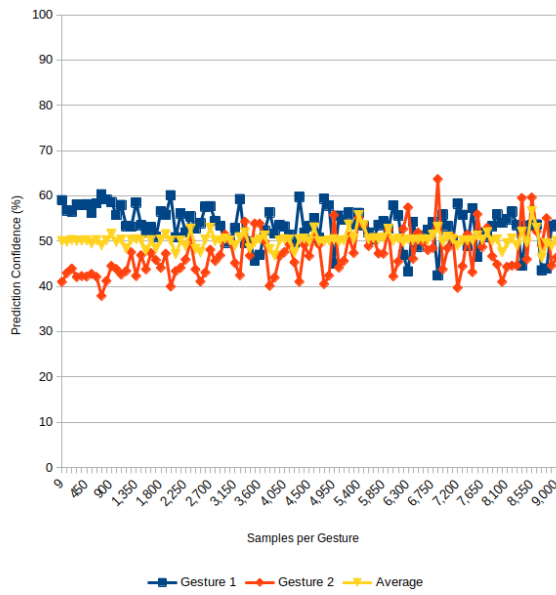
Figure 9.4: Learning to predict two pre-defined gestures using historical MMG data with (a) 200 ms of historical data and (b) 100 ms of historical data and an extended artificial neural network hidden layer

Although 200 ms of historical MMG data was used, the hidden layer of the neural network was not increased from its default length. For the next simulation the length of the hidden layer was increased too. However, the history length was reduced to 100 ms for two reasons. First, although the length of a muscle signal is expected to be around 200 ms, the TD learner should be able to use the historic data from the previous input. Second, increasing the length of both the input layer and the hidden layer exponentially increases the number of neurons in the network. To minimise the amount of learning required, the number of neurons needs to be kept to a minimum. Figure 9.4b

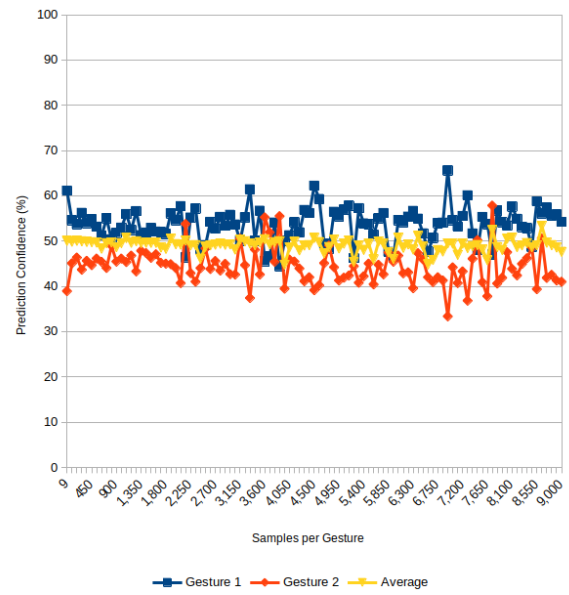
shows the results using only 100 ms of historical MMG data, but with a hidden layer of twice the input data vector length.

To ensure sufficient learning could take place, the number of times the recorded data was presented to the underlying Generic, Self-Improving Predictor was increased. To simulate 5 hours of active data and ignoring the data between samples, the recorded data was presented 100 times for a total of 9,000 training samples per gesture. In addition, to avoid unknown problems with the quality of the recorded data, the simulations were repeated four times. The data from Subjects C, D, M and N in Table 9.8 were used. The data from these subjects should be comparable, because they all have the same metadata and were asked to perform the same gestures. The data includes 10 tranches of 10 samples each for each gesture i.e. 100 samples per gesture were available. Furthermore, the data from all these subjects appears to have been recorded using the same IMU data rate of 500 Hz. Although the data from Subject D only includes four gestures, the initial simulations only used two gestures. Therefore, Subject D's dataset was also used.

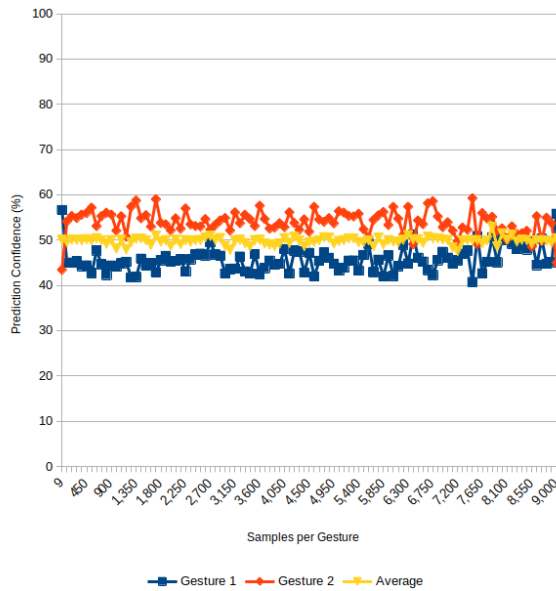
First, 100 ms of historical data was used. Next, 200 ms and then 500 ms of historical data were used. Each time, the length of the hidden layer was increased to twice the input data vector length. The number of neurons in the Flexible Artificial Neural Network that needed to be updated on each iteration was now too many to be completed in the 10 Hz input data sample rate. However, since playback can be set to be done as fast as possible, and as fast as possible does not necessarily mean faster. The playback setting now provided the ability to achieve this in the simulated, recorded data environment.



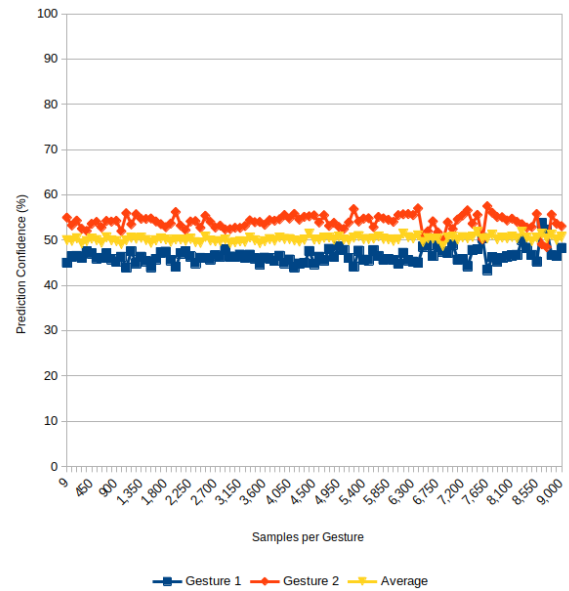
Subject C



Subject D



Subject M



Subject N

Figure 9.5: Learning two pre-defined dynamic gestures using 100 ms of historical data and 9,000 simulated samples per gesture

Figure 9.5 presents the results from learning two pre-defined dynamic gestures using 100 ms of historical data and 9,000 simulated samples per gesture. Similarly, Figure 9.6 presents the results of learning two pre-defined gestures using 200 ms of historical data and 9,000 simulated samples per

gesture, and Figure 9.7 presents the results of learning two pre-defined gestures using 500 ms of historical data and 9,000 simulated samples per gesture.

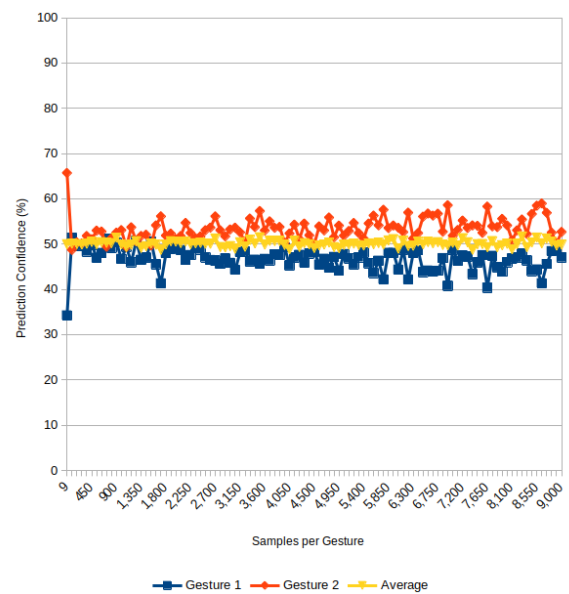
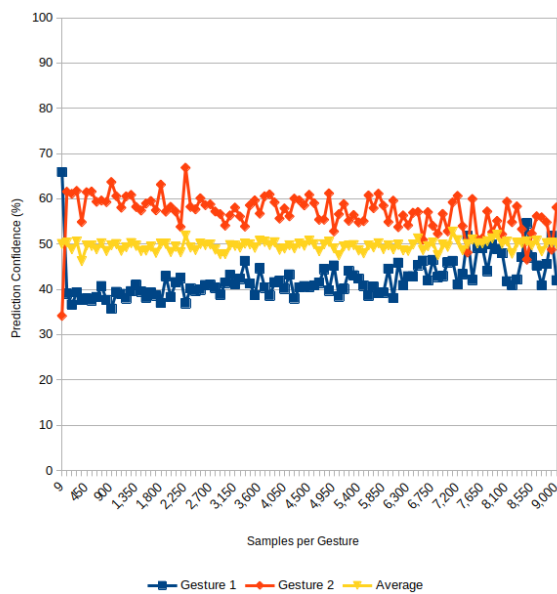
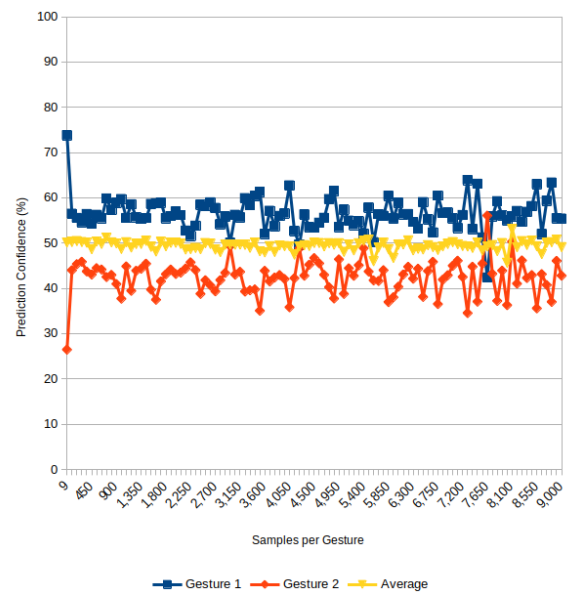
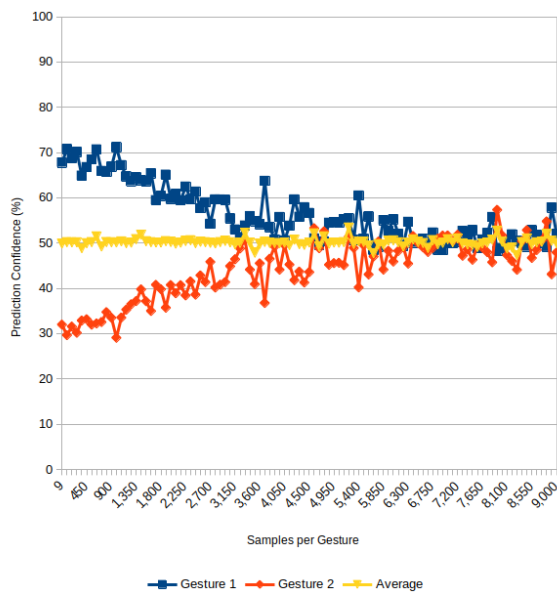
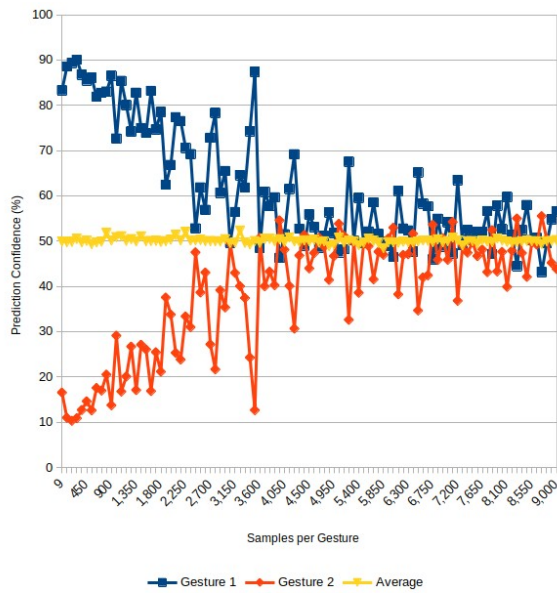
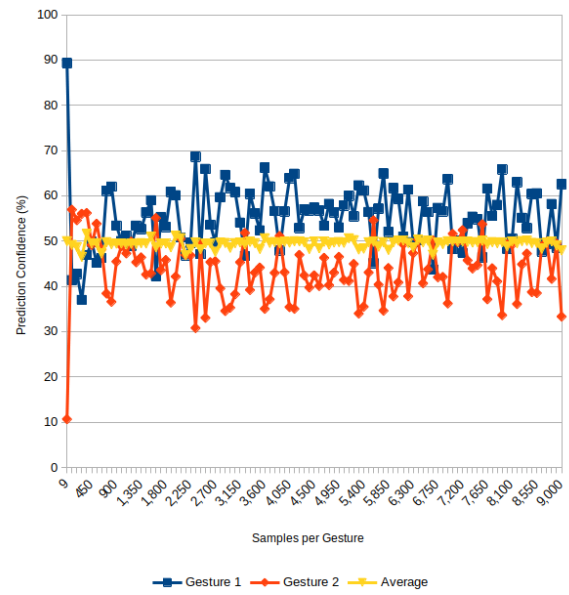


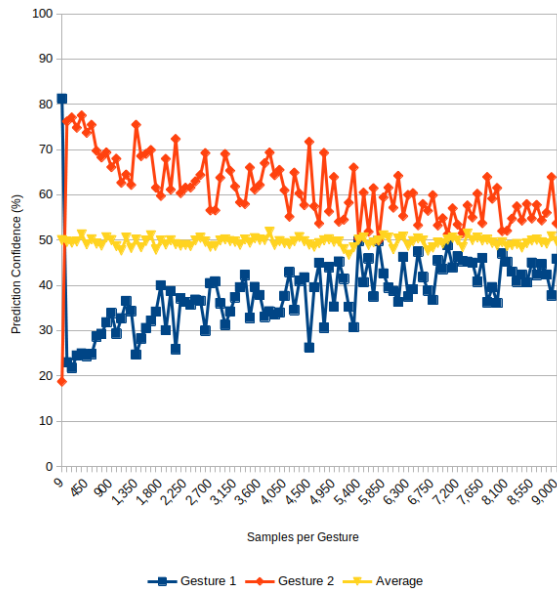
Figure 9.6: Learning two pre-defined dynamic gestures using 200 ms of historical data and 9,000 simulated samples per gesture



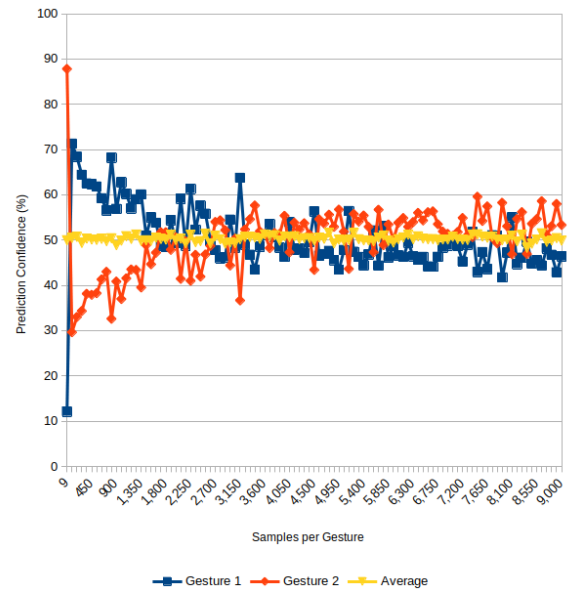
Subject C



Subject D



Subject M

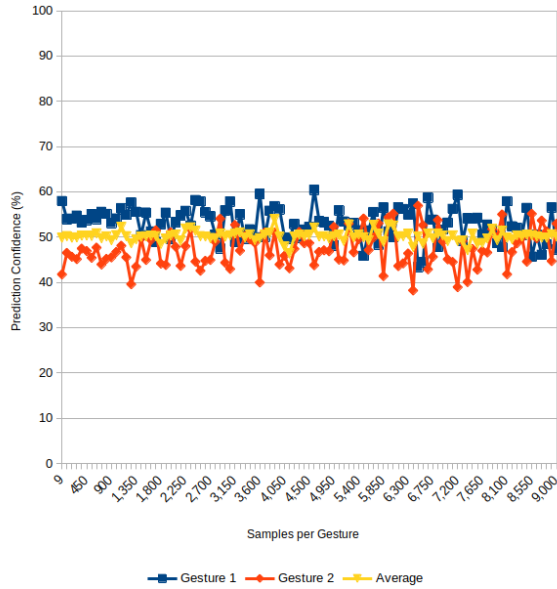


Subject N

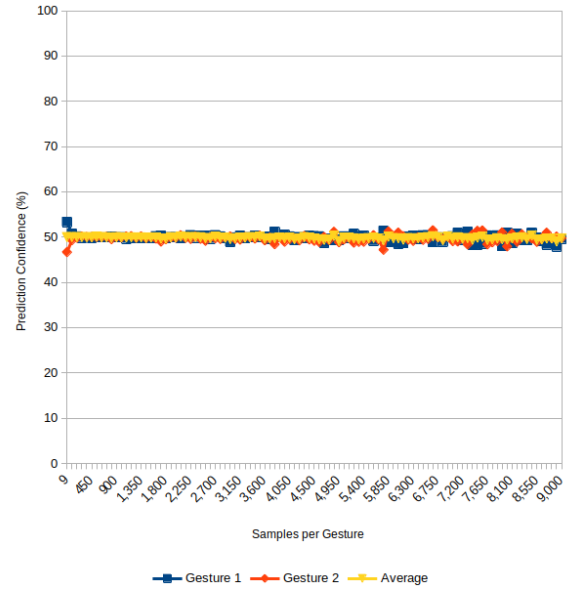
Figure 9.7: Learning two pre-defined dynamic gestures using 500 ms of historical data and 9,000 simulated samples per gesture

Unfortunately, the Generic, Self-Improving Controller still does not appear to be learning the difference between the two gestures.

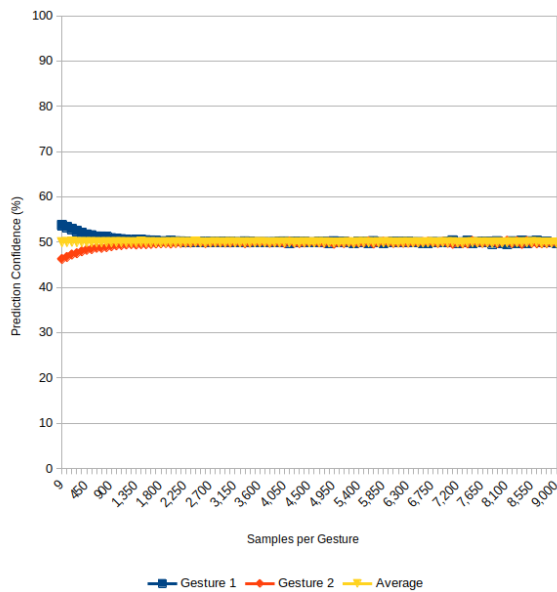
The prediction data presented in Figures 9.5, 9.6 and 9.7 is surprisingly volatile given the low learning rate of 0.001. This was particularly noticeable when using 500 ms of historical data and hence a large Flexible Artificial Neural Network.



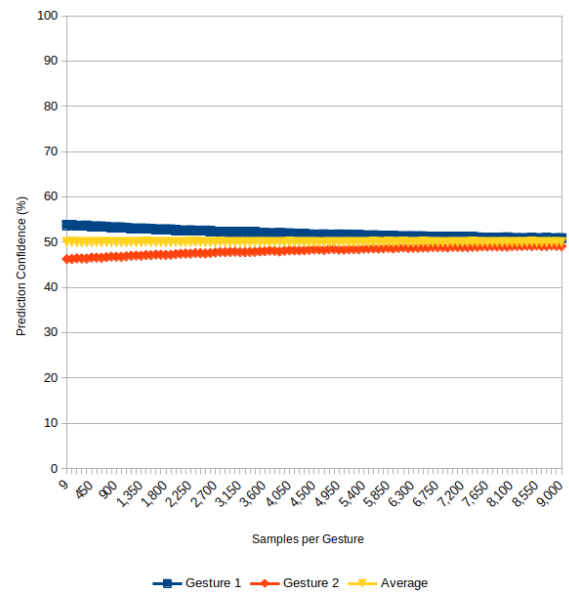
(a)



(b)



(c)



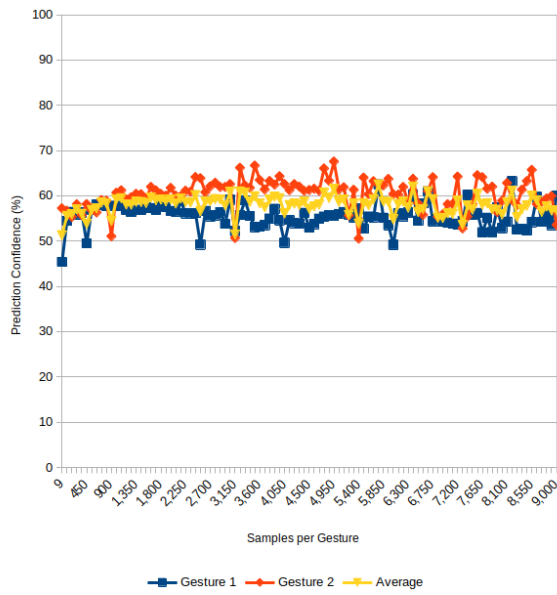
(d)

Figure 9.8: Learning two pre-defined dynamic gestures using 500 ms of historical data and 9,000 simulated samples per gesture with a learning rate of (a) 0.001, (b) 0.0001, (c) 1×10^{-5} , and (d) 1×10^{-6}

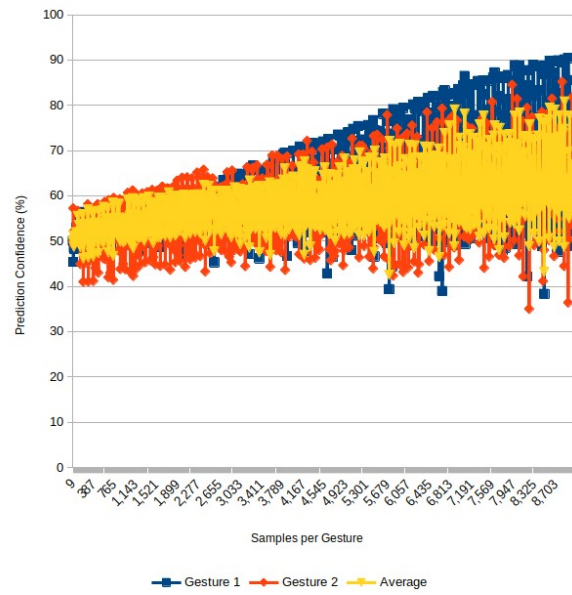
Therefore, to ensure that the learning rate wasn't too high, the next set of simulations steadily reduced the learning rate. Figure 9.8 presents the results using different learning rates while keeping the historical data at 500 ms and the size of the hidden layer at twice the size of the input state vector.

Reducing the learning rate removed the volatility in the results, but did not result in the Generic, Self-Improving Controller learning the difference between the two gestures. The only apparent learning is the removal of the original bias associated with the original random weights in the Flexible Artificial Neural Network.

One interesting result was captured. The data from Subject S had been mostly ignored, because it was captured using a different input data rate. Using no historical data, the Generic, Self-Improving Controller had shown some learning. The results of learning two pre-defined dynamic gestures using data captured from an IMU and 8 MMGs at 1000 Hz and 9,000 simulated samples per gesture are presented in Figure 9.9. Figure 9.9a plots the test results after every 10th tranche. Figure 9.9b plots the same test results after every tranche. To confirm these results were not just an anomaly, the simulations were re-run twice. The results of these simulations are presented in Figure 9.10.

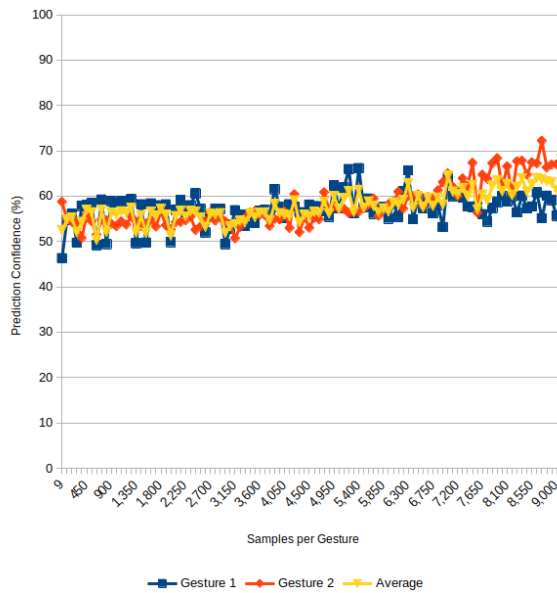


(a)

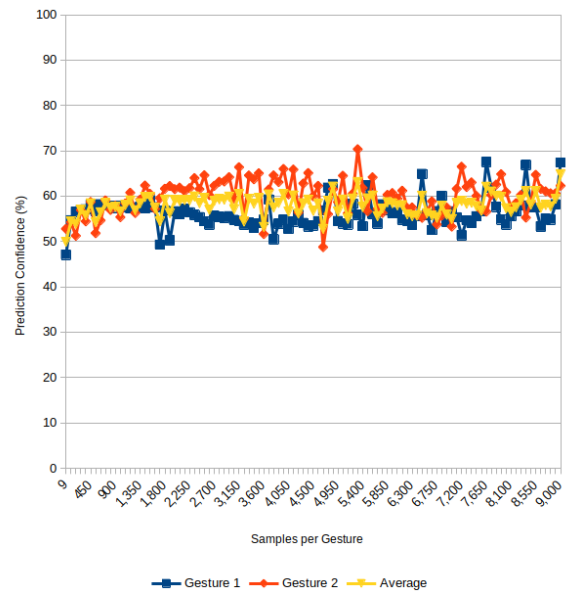


(b)

Figure 9.9: Learning two pre-defined dynamic gestures using data captured from an IMU and 8 MMGs at 1000 Hz and 9,000 simulated samples per gesture plotting the predictions after (a) every 10th tranche and (b) every tranche

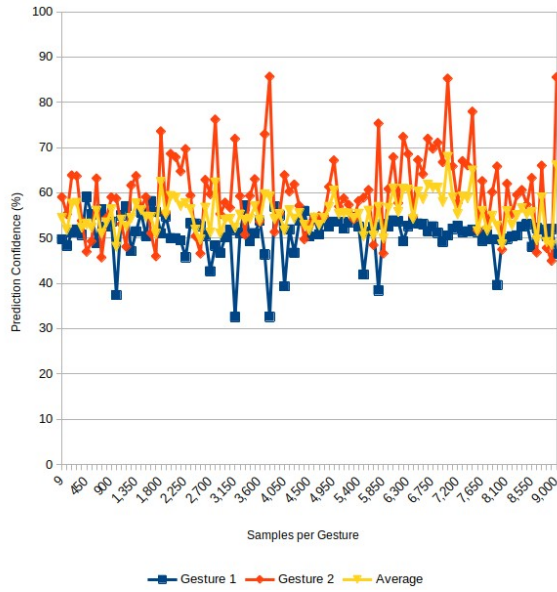


Simulation 2

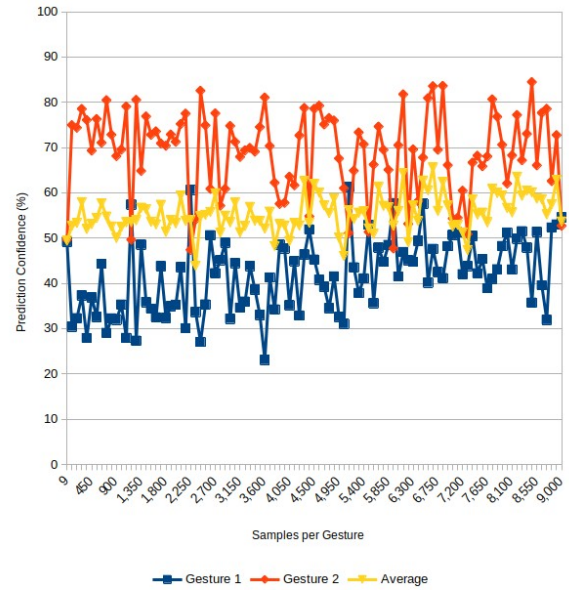


Simulation 3

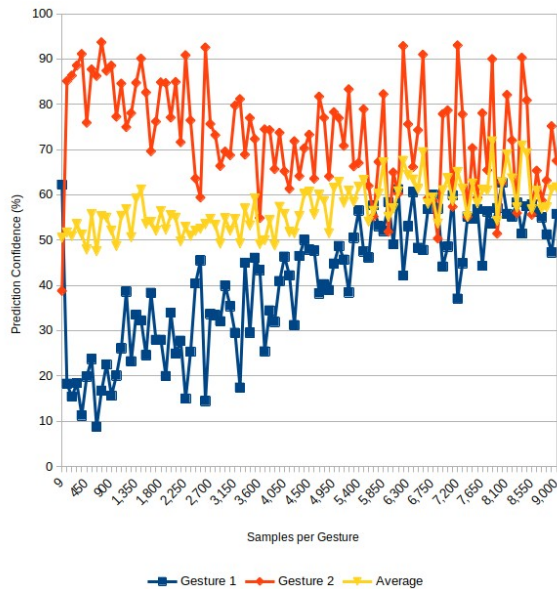
Figure 9.10: Consistently learning two pre-defined dynamic gestures using data captured from an IMU and 8 MMGs at 1000 Hz



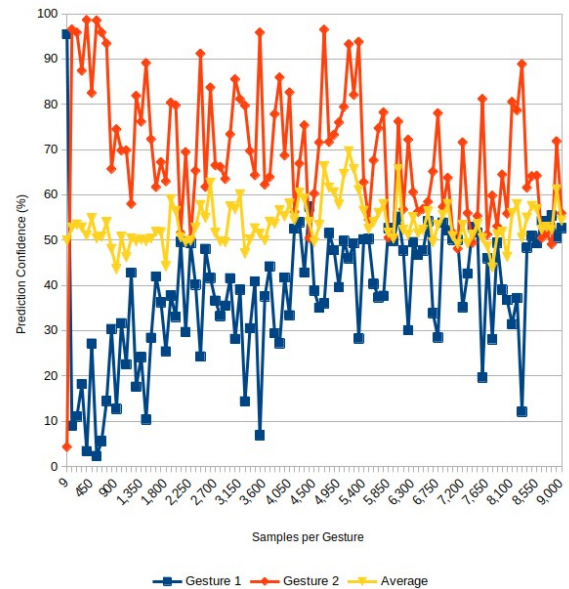
(a)



(b)



(c)



(d)

Figure 9.11: Learning two pre-defined dynamic gestures using data captured from an IMU and 8 MMGs at 1000 Hz using (a) 200 ms of historic data without extending the length of the hidden layer, (b) 100 ms, (c) 200 ms and (d) 500 ms of historic data and extending the hidden layer to twice the length of the input state vector

As the data is under-sampled, it was expected that the results could be improved by including historical data. The results of including historical data are presented in Figure 9.11. Using historical data did not improve the results.

The success of using 1000 Hz data and no historical data was then extended. First the number of simulated samples was increased to 90,000 samples. The results are presented in Figure 9.12. For clarity, only every 100th test result has been plotted.

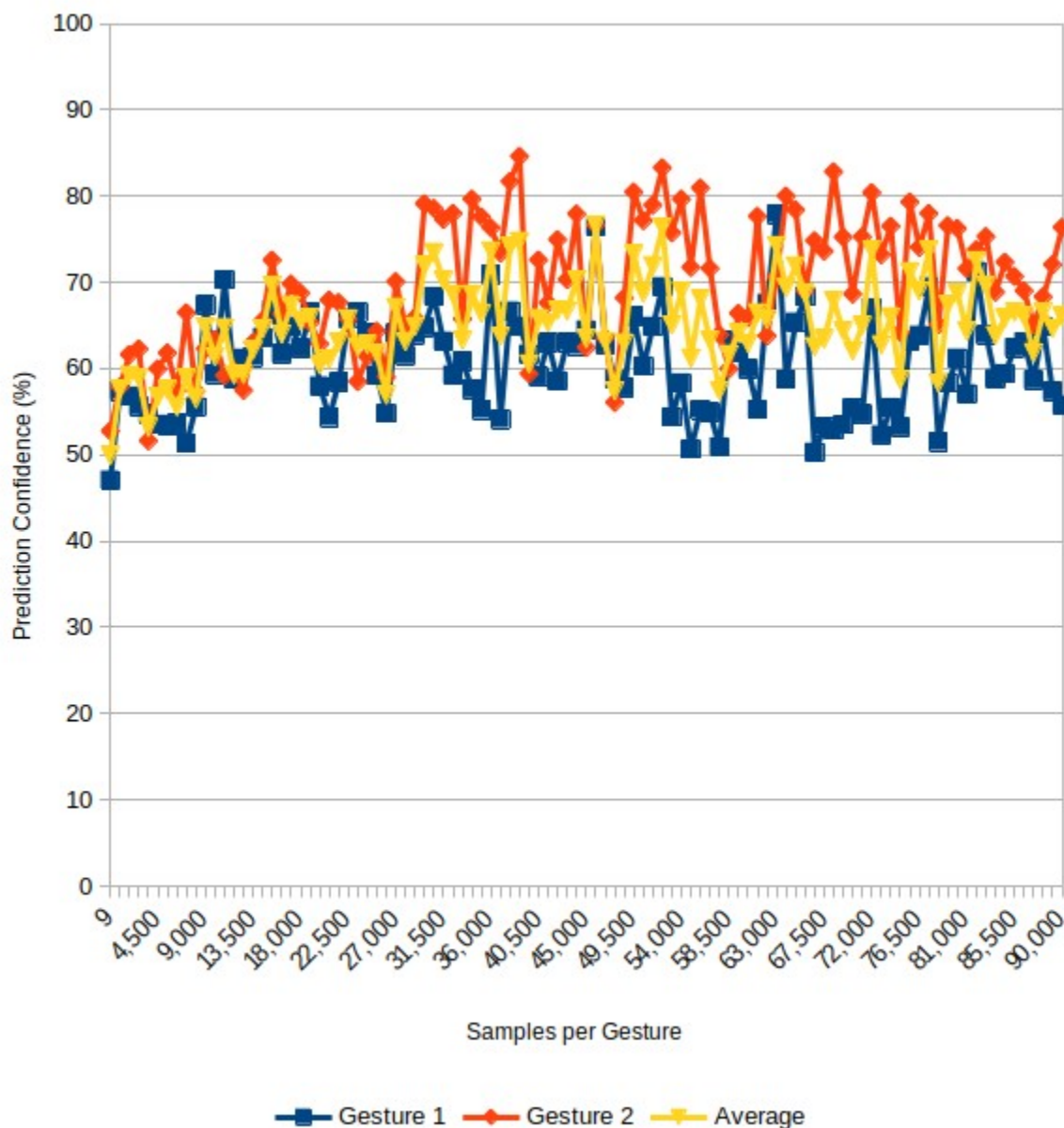
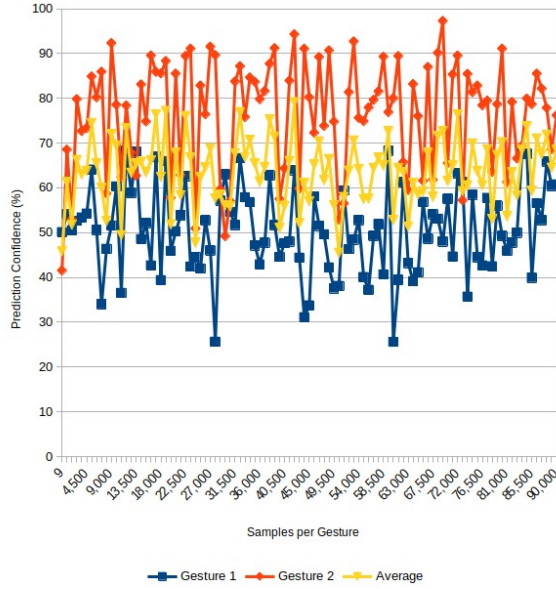
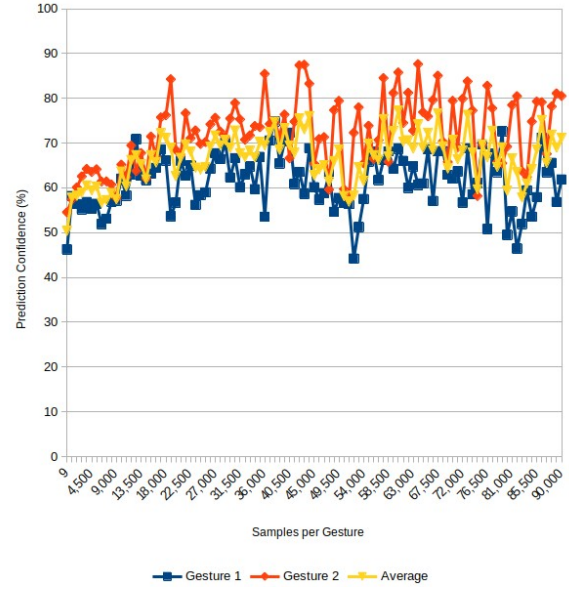


Figure 9.12: Learning two pre-defined dynamic gestures using data captured from an IMU and 8 MMGs at 1000 Hz and 90,000 simulated samples per gesture

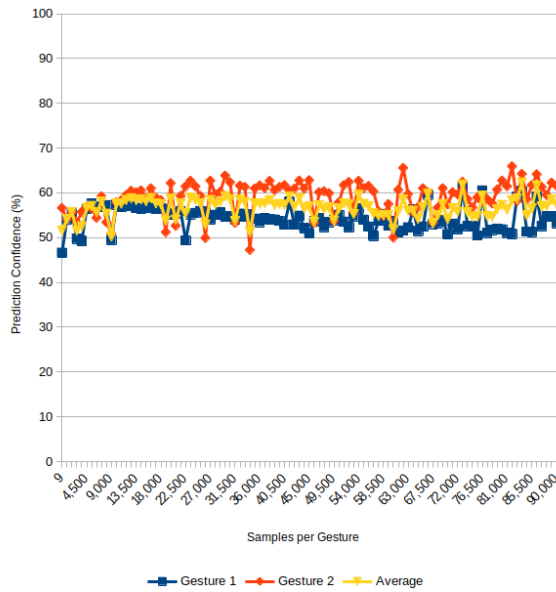
After 90,000 simulated samples, learning appears to plateau around a confidence level of between 60% and 70%. However, the volatility in the predictions results in predictions outside of this range and occasionally predictions below 50% i.e. incorrect predictions.



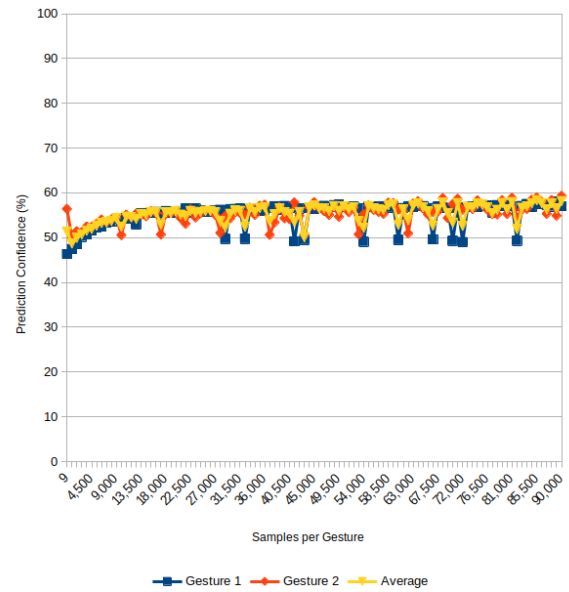
(a)



(b)



(c)



(d)

Figure 9.13: Learning two pre-defined dynamic gestures using data captured from an IMU and 8 MMGs at 1000 Hz and 90,000 simulated samples per gesture using a learning rate of (a) 1×10^{-2} , (b) 1×10^{-3} , (c) 1×10^{-4} and (d) 1×10^{-5}

In an attempt to remove the volatility, the extended simulation was rerun four times with varying learning rates. The results are presented in Figure 9.13. Although volatility decreased with slower learning rates, it was not eliminated. As with previous experiments, a learning rate of 0.001 was deemed to provide the best balance between learning rate and volatility.

It was now time to increase the number of gestures. To ensure learning was visible, the successful learning rate of 0.001 was used and the number of gestures was increased. Figure 9.14 shows the results of learning three pre-defined dynamic gestures using data captured from an IMU and 8 MMGs at 1000 Hz and 90,000 simulated samples per gesture.

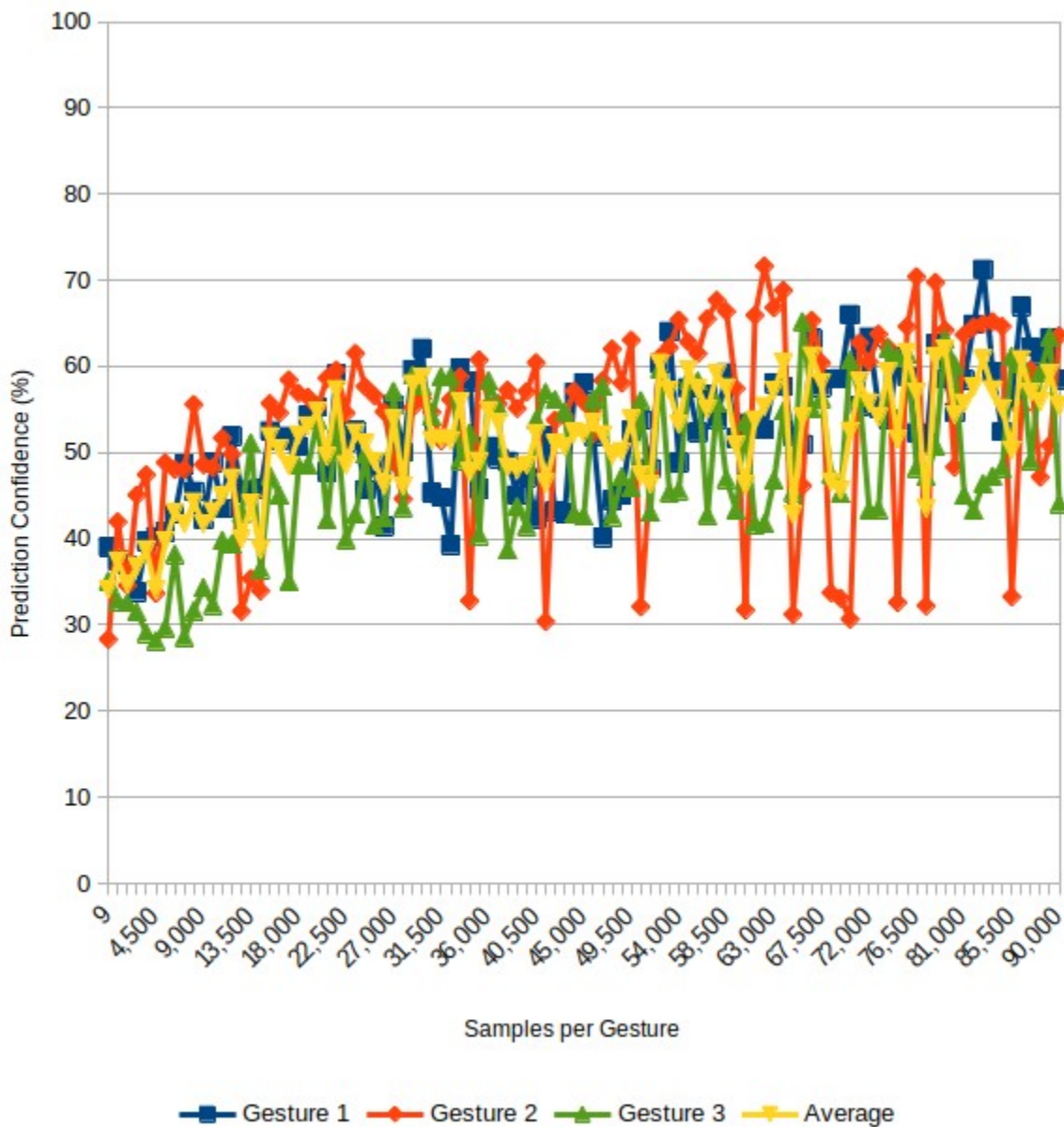


Figure 9.14: Learning three pre-defined dynamic gestures using data captured from an IMU and 8 MMGs at 1000 Hz and 90,000 simulated samples per gesture

With three gestures, the random expected predicted confidence would have an average of 33%.

After 90,000 simulated samples per gesture, the average predicted confidence is over 50%.

Figures 9.15, 9.16 and 9.17 show the results of learning four, five and six pre-defined dynamic gestures respectively. All simulations use the same data; so they are comparable.

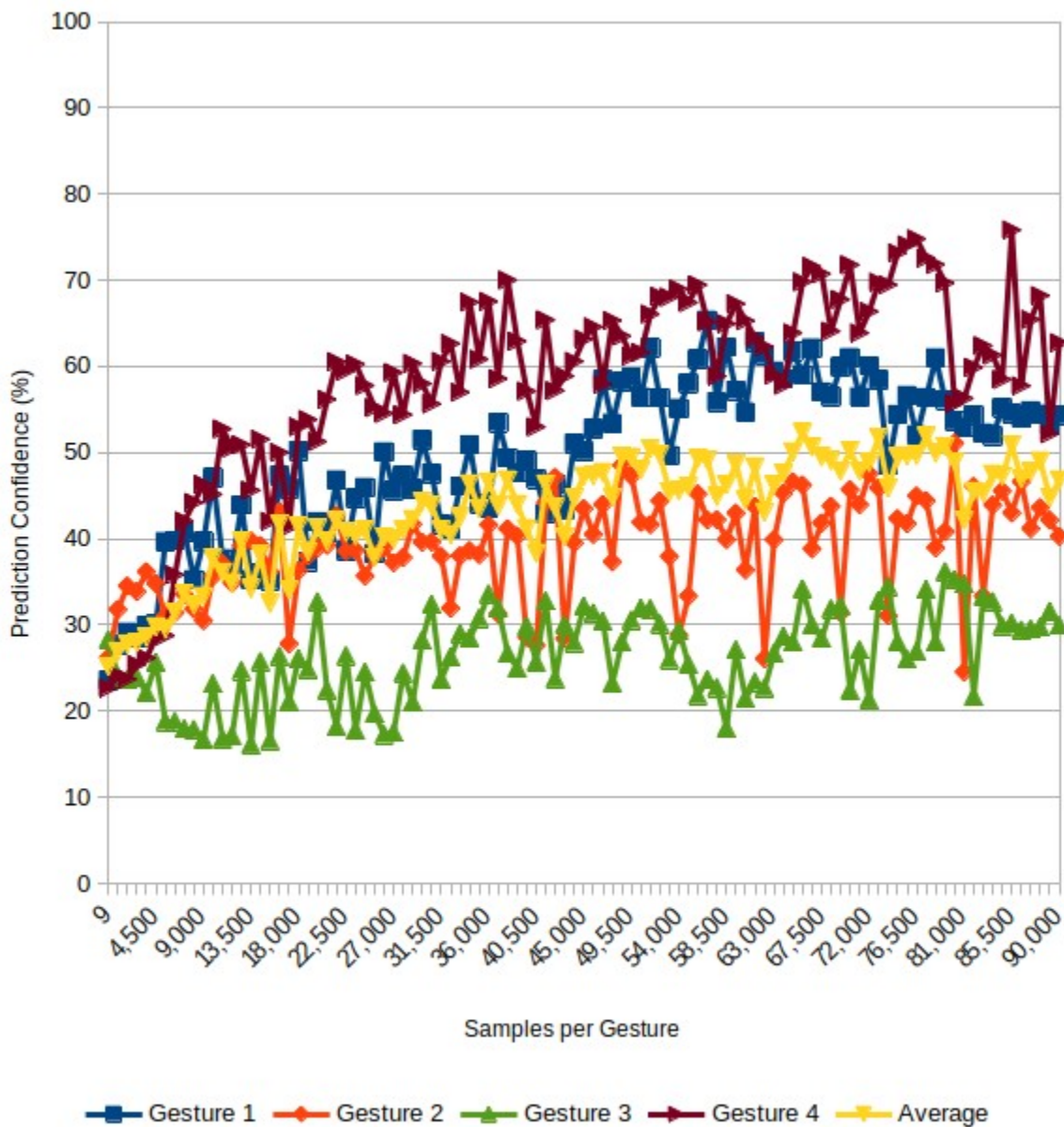


Figure 9.15: Learning four pre-defined dynamic gestures using data captured from an IMU and 8 MMGs at 1000 Hz and 90,000 simulated samples per gesture

With four gestures, the random expected predicted confidence would have an average of 25%. After 90,000 simulated samples per gesture, the average predicted confidence is just under 50%.

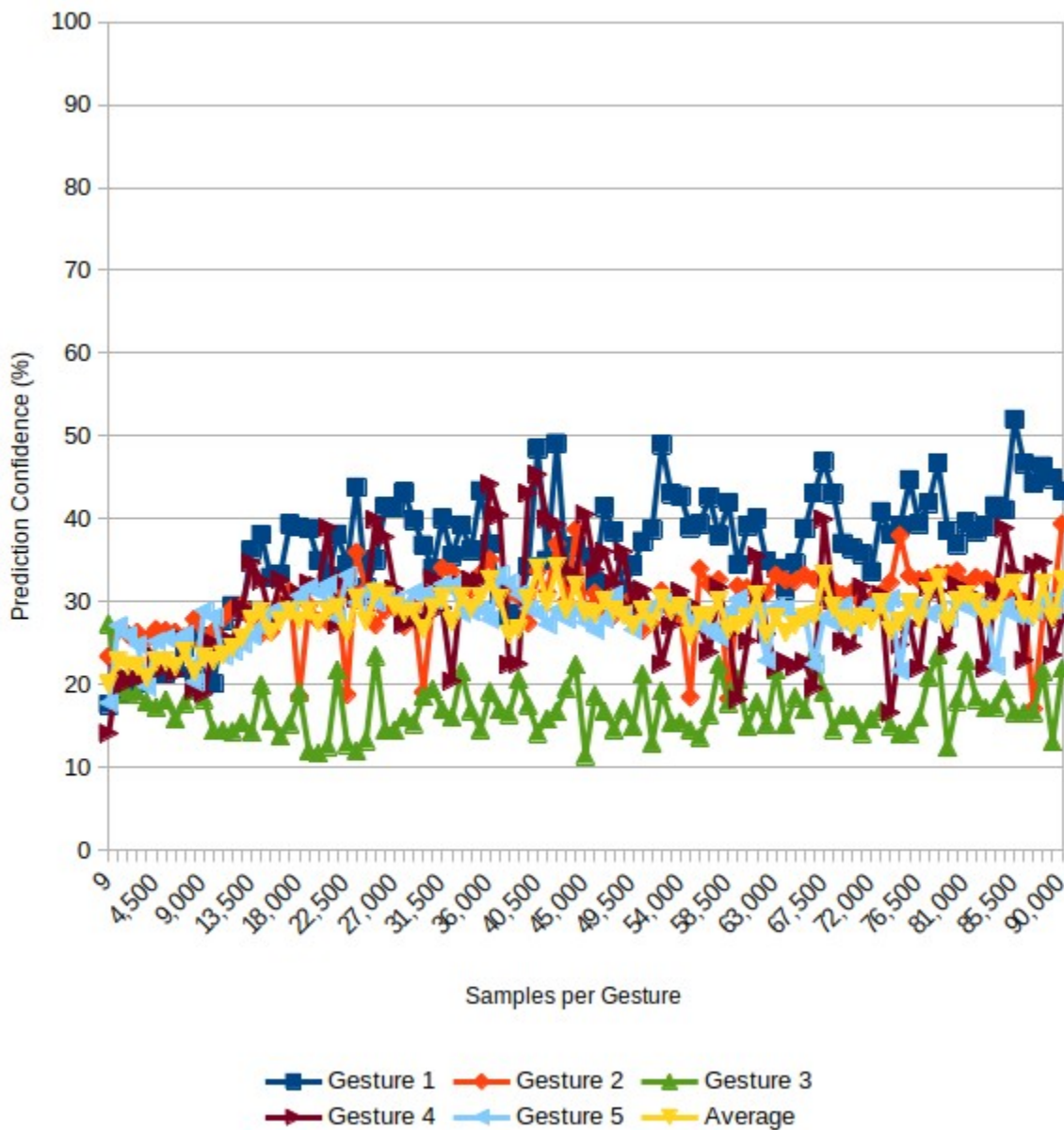


Figure 9.16: Learning five pre-defined dynamic gestures using data captured from an IMU and 8 MMGs at 1000 Hz and 90,000 simulated samples per gesture

With five gestures, the random expected predicted confidence would have an average of 20%. After 90,000 simulated samples per gesture, the average predicted confidence is around 30%.

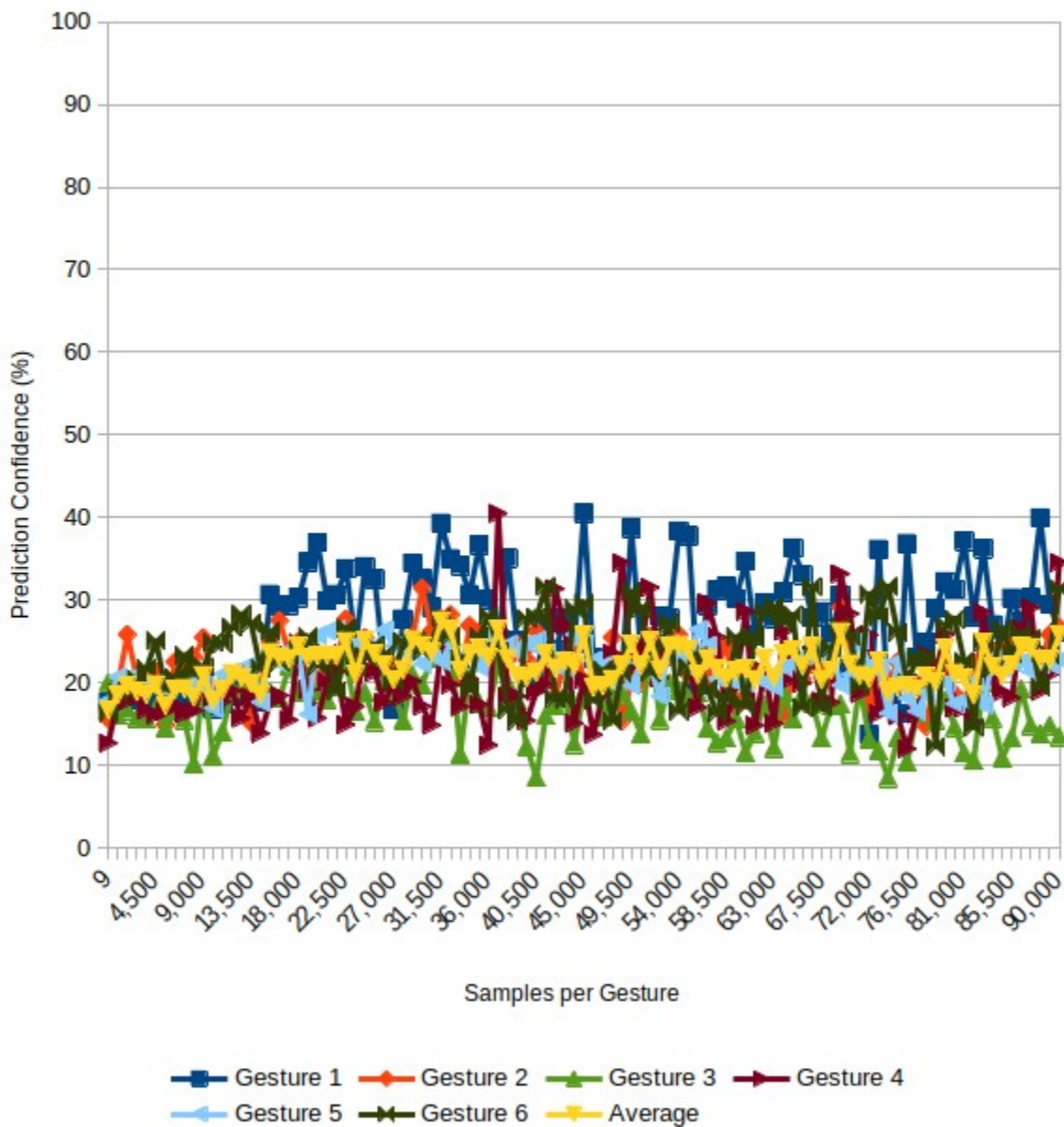


Figure 9.17: Learning six pre-defined dynamic gestures using data captured from an IMU and 8 MMGs at 1000 Hz and 90,000 simulated samples per gesture

With six gestures, the random expected predicted confidence would have an average of 17%. After 90,000 simulated samples per gesture, the average predicted confidence is around 25%.

For the Generic, Self-Improving Controller to be successful, all the desired outcomes need to be correctly predicted reliably. This requires all the gestures to be predicted correctly. Figure 9.18 compares the time required to correctly predict increasing numbers of gestures. The percentage of

correct predictions during a testing round are plotted. However, instead of only plotting every 100th test result, the average of the previous 100 test results is plotted.

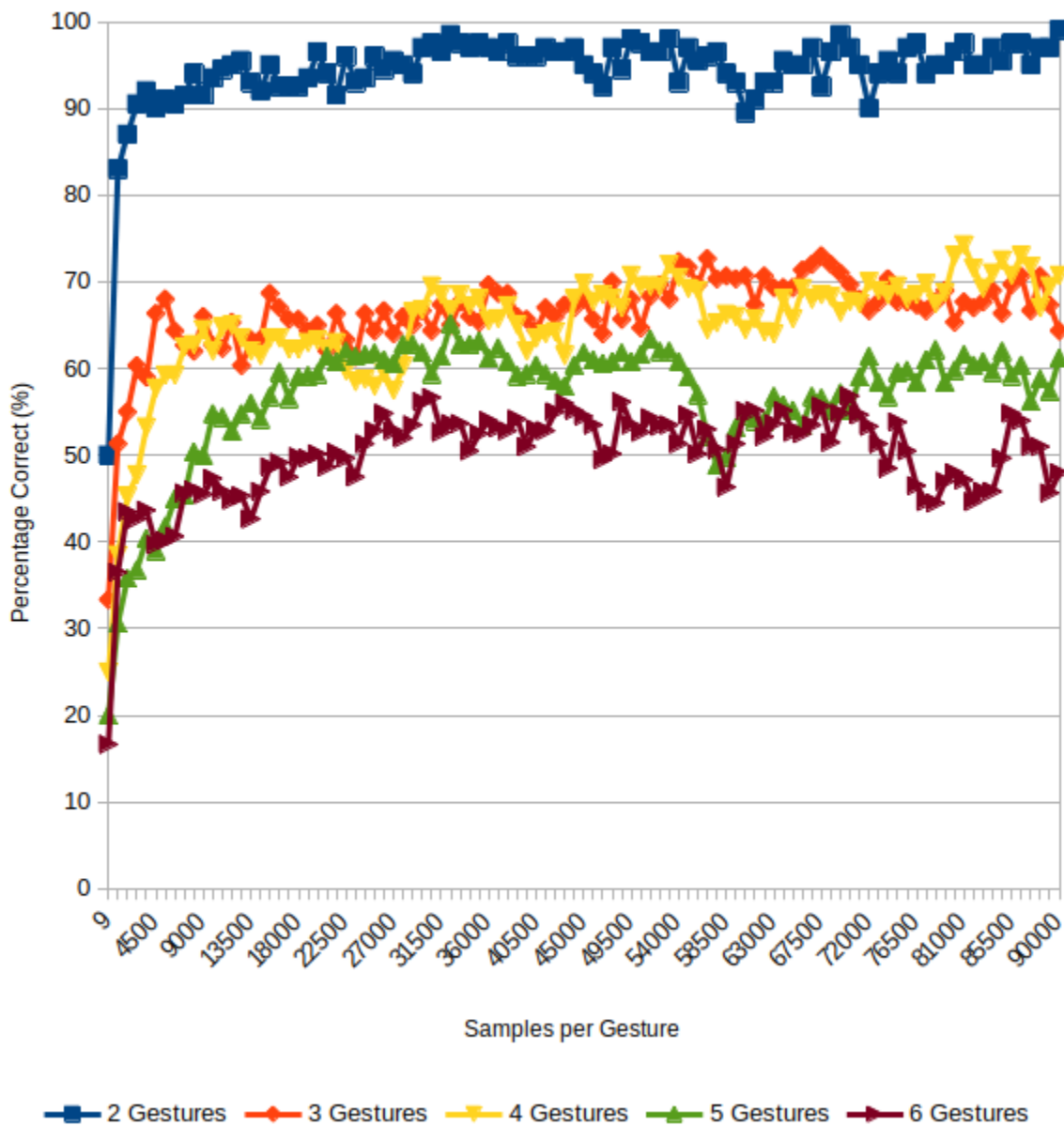
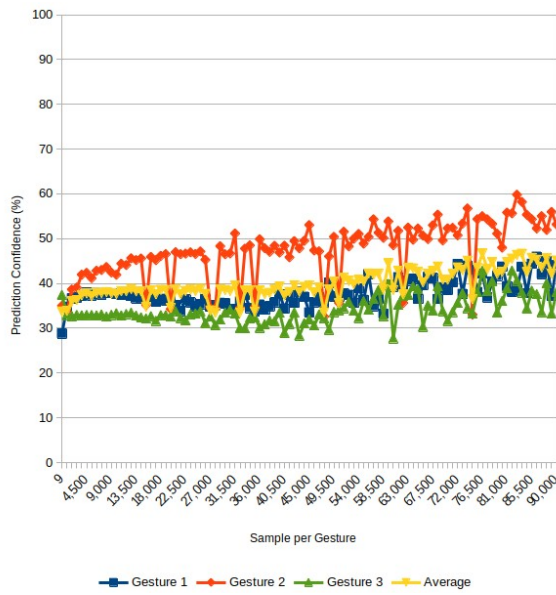
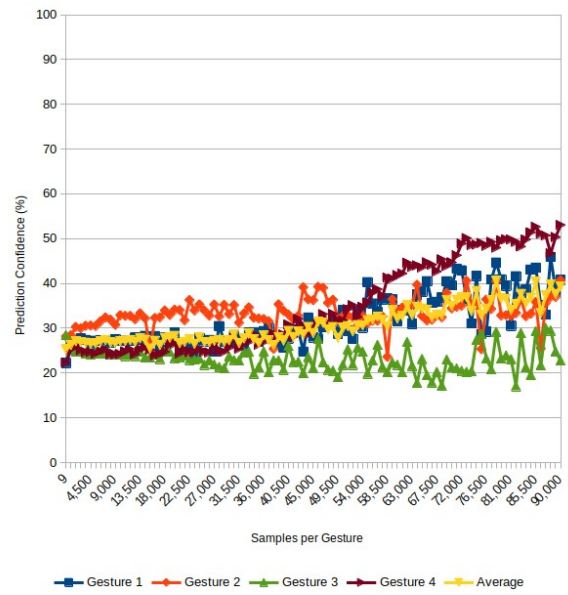


Figure 9.18: Comparing the number of samples per gesture vs. predictive accuracy required to predict increasing numbers of pre-defined dynamic gestures with a learning rate of 0.001

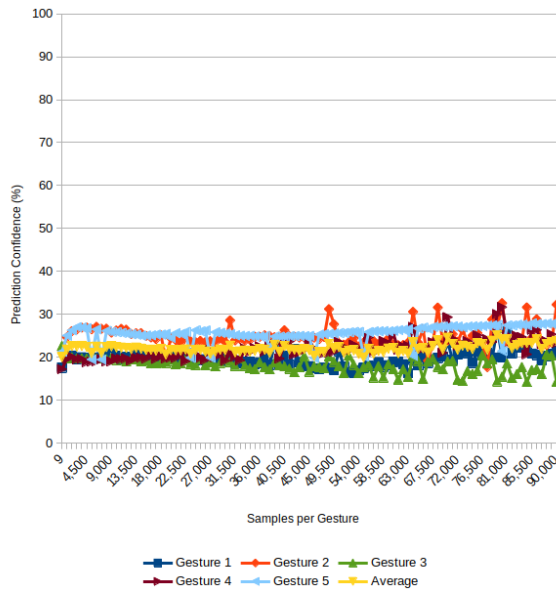
Finally, in an attempt to remove the volatility, and hopefully improve the predictive ability of the results, the simulations with increasing numbers of gestures were rerun with a reduced learning rate of 0.0001. The learning results are presented in Figure 9.19.



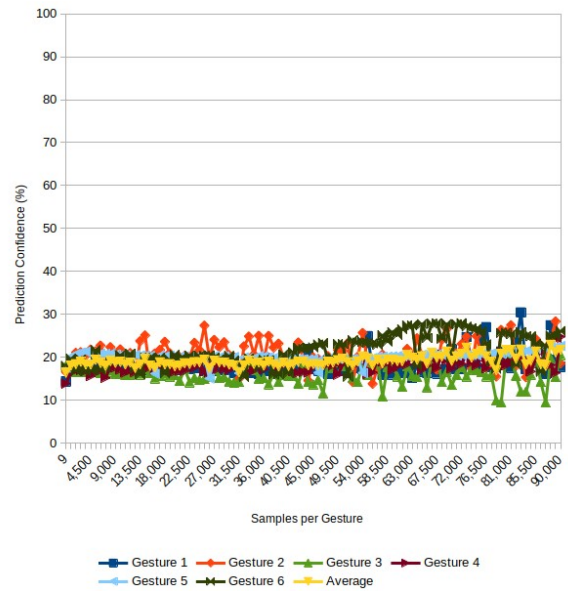
(a)



(b)



(c)



(d)

Figure 9.19: Learning (a) three, (b) four, (c) five and (d) six pre-defined dynamic gestures using data captured from an IMU and 8 MMGs at 1000 Hz and 90,000 simulated samples per gesture and a learning rate of 0.0001

The percentage of correct predictions against number of samples with the learning rate set to 0.0001 for increasing numbers of gestures is plotted in Figure 9.20. Once again the average of the previous 100 test results is plotted.

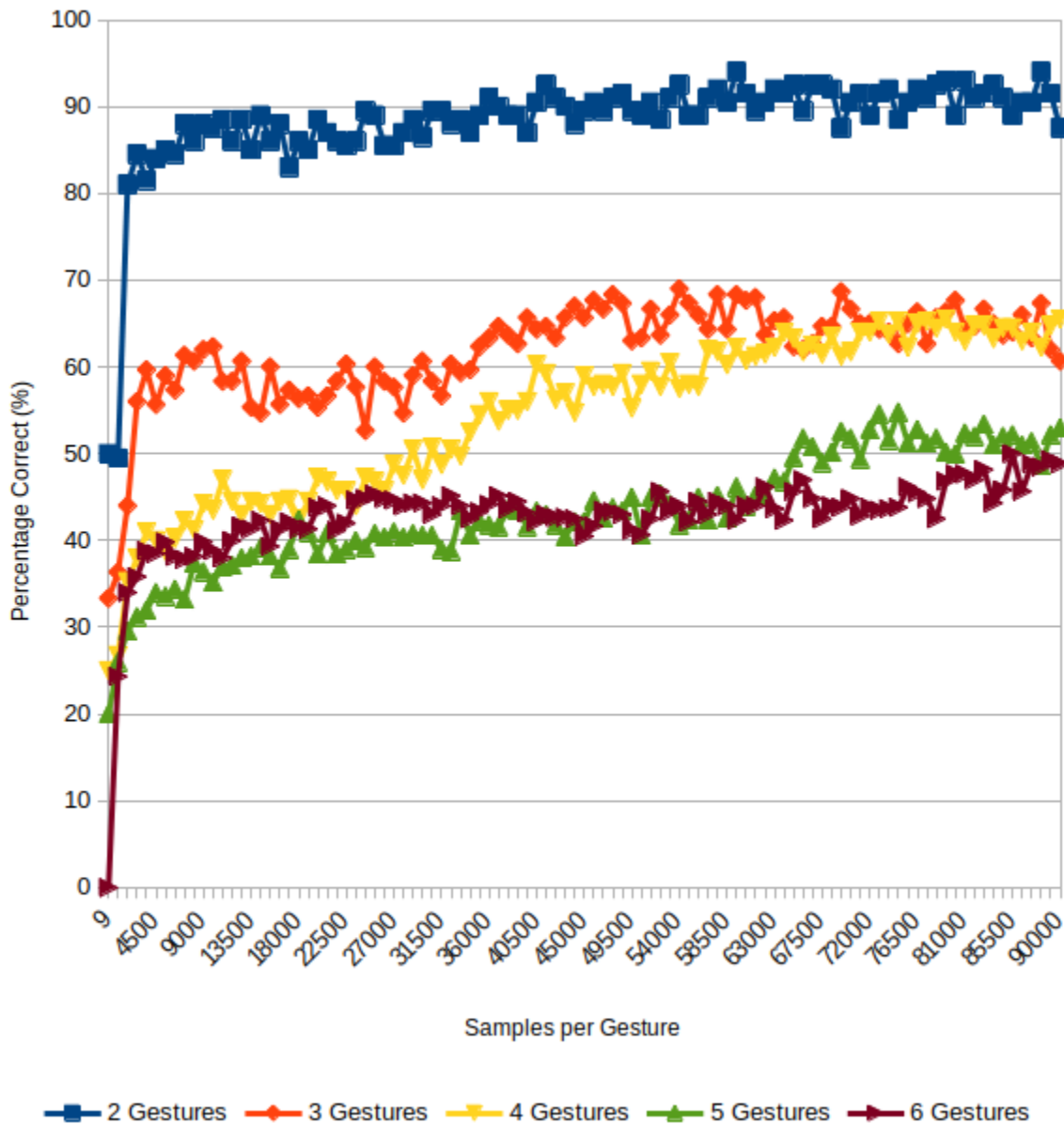


Figure 9.20: Comparing the number of samples per gesture vs. predictive accuracy required to predict increasing numbers of pre-defined dynamic gestures with a learning rate of 0.0001

Although volatility decreased with a learning rate of 0.0001, it did not improve the results. In fact overall learning and predictive ability is generally less than with a learning rate of 0.001.

9.3.3 Discussion of the Results

Learning pre-defined dynamic gestures was achieved using data captured from an IMU and 8 MMGs at 1000 Hz using no historic data. This is a surprising result, for two reasons:

First, the results could not be replicated using data captured from other subjects when the IMU data rate was only 500 Hz. The IMU data rate should not impact the results, because the underlying Generic, Self-Improving Predictor is only sampling the IMU input data at 10 Hz. Since no historic data is used, the effective IMU data rate is 10 Hz. Unfortunately, attempts to increase the underlying Generic, Self-Improving Predictor input data sample rate above 10 Hz resulted in even slower actual sample rates, because the trigger interrupts began interrupting each other. The period of the underlying Generic, Self-Improving Predictor input data sample rate needs to be greater than the time required to update all the weights in the Flexible Artificial Neural Network. The larger the Flexible Artificial Neural Network, the longer the period between samples required. Any computer, especially those not designed for training an artificial neural network in real-time, will have this limitation. Since the IMU data rate should not impact the results, this suggests that there may be an issue with the quality of the data captured from the other subjects.

Second, when historic data is used, the learning ability is either lost or too slow to be detected. The capture and presentation of historic data to the underlying Generic, Self-Improving Predictor was meant to overcome the processing limitations of the computer used. When the historic data is presented in parallel instead of sequentially, the benefits of the TD learning are lost. However, depending on the configuration of the Flexible Artificial Neural Network, the learning results should be comparable. During the simulations, larger Flexible Artificial Neural Networks could be used.

Figures 9.12, 9.14, 9.15, 9.16 and 9.17 show that, as expected, with more gestures the rate of learning drops off significantly. However, Figure 9.12 also suggests that even with only two

gestures, learning is limited and confidence levels remain low. As discussed in Section 9.2.3, with two gestures, a correct outcome is when all predictions are above 50%. Figure 9.18 shows that after around 6,000 simulated samples the Generic, Self-Improving Controller will predict the correct gesture over 90% of the time. When combined with the Blended Control Policy created and described in Section 5.7, the Generic, Self-Improving Controller will provide usable control signals.

Figure 9.18 indicates that when learning three gestures, it can reliably only correctly predict two of the gestures. The dips in Figure 9.14 suggest that this is Gesture 2, which was actually more reliably predicted when only two gestures needed to be differentiated. However, a closer examination of the test data reveals that these are temporary dips. It is Gesture 3 that is the gesture that is more often not correctly predicted. This is supported by the results presented in Figure 9.15: Learning four predefined dynamic gestures. Once again, it is Gesture 3 that the Generic, Self-Improving Controller is struggling to learn. Figure 9.18 supports this. Adding a fourth gesture did not reduce the long-term, average correct prediction percentage. As discussed in Section 9.2.3 the type of gesture used, and the sensors used to detect it, remains significant.

Adding a fifth gesture reduces the long-term average correct prediction percentage to 60%. In other words it is only successfully predicting three out of the five gestures. Once again it is Gesture 3 that is not being correctly predicted, but it's not obvious which of the other gestures it's failing to predict. When trying to differentiate between five gestures, Figure 9.16 indicates that it's not only about being able to differentiate between gestures, but also there is a bias towards Gesture 1.

However, as seen in Figure 9.17, when trying to differentiate between six gestures, the bias towards Gesture 1 is gone, and it's now simply struggling to differentiate between gestures. As discussed in Section 9.2.2, the output of the underlying Generic, Self-Improving Predictor is a value between 0 and 1. To convert this output into a Prediction Confidence, the Generic, Self-Improving Controller scales the values so that they sum to 1 or 100%. With only two gestures, a correct prediction is one with a confidence level above 50%, because the other gesture will be by definition below 50%.

When more than two gestures are being learnt, 50% is not the minimum confidence level required to be a correct prediction. Neither will a value that is one over the number of gestures converted to a percentage be sufficient. The value required will be between these two values, and it will depend on the values of the other predictions. A correct prediction is when the prediction of the correct gesture is higher than the predictions for the other gestures. It is possible that the correct gesture and one of the other incorrect gestures both have values close to 50% and the other gestures have values close to 0%. The correct gesture would require 50% to be correct. On the other hand, if all gestures are being predicted as equally likely, a correct gesture may be one that is just over 1 over the number of gestures converted to a percentage. Gestures that are being correctly predicted with a higher confidence level than other gestures will actually tend to reduce the overall percentage of correct gestures. Gestures that are being correctly predicted with a higher confidence level will also tend to have higher values when incorrect, which increase the required confidence level of the correct gesture for a correct prediction. This bias was most noticeable when the learning rates were higher. With higher learning rates, the bias also tends towards the last gesture learnt. As shown in Figure 9.13, reducing the learning rate not only reduces the volatility, but also reduces the bias of one gesture over the other.

Finally, it is worth noting that Figure 9.18 shows that, most of the learning takes place during the first 9,000 samples per gesture. This is probably mostly due to the reuse of the data in the simulations. There is nothing new to learn. As shown in Figure 9.13, reducing the learning rate eliminates most of the volatility in the results, but it does not result in an improvement in the overall long-term learning.

9.4 Summary

The Generic, Self-Improving Controller was tested using user-defined static gestures to predict the desired outcomes required to move a mouse pointer and send a mouse click. The Generic, Self-

Improving Controller was also tested using pre-defined dynamic gestures. The static and dynamic gestures were captured using IMUs and MMGs. The Action Predictor GUI was used to train the underlying Generic, Self-Improving Predictor.

The Generic, Self-Improving Controller was shown to learn two static gestures with an accuracy of 100% in less than an average of 10 samples per gesture. It was also able to learn three static gestures after an average of less than 20 samples per gesture, and four static gestures with an average of less than 35 samples per gesture. Although 100% accuracy was achieved with five gestures on a number of occasions, only 80% accuracy was achieved on all occasions. Furthermore, the number of samples per gesture required to learn five static gestures was over 75 samples per gesture.

Due to the number of samples per gesture required, the input and learning signals were recorded by the Action Predictor GUI and replayed multiple times to create simulated gesture data. The Generic, Self-Improving Controller was shown to learn two pre-defined dynamic gestures with an accuracy of 90% in less than 6,000 samples per gesture. This was achieved using no historical data.

Using the same data, the Generic, Self-Improving Controller was trained on three pre-defined dynamic gestures. Although an accuracy of 66% was achieved within 6,000 samples per gesture, this indicates it failed to learn one dynamic gesture. The accuracy did not significantly improve even after presenting 90,000 samples per gesture. However, when presented with four pre-defined dynamic gestures, the Generic, Self-Improving Controller was also able to achieve a better average accuracy of around 70%. The dynamic gesture it struggled to differentiate when differentiating four pre-defined dynamic gestures is the same gesture that the Generic, Self-Improving Controller struggled to differentiate when only three pre-defined dynamic gestures were presented. This suggests the Generic, Self-Improving Controller's difficulty in differentiating gestures may be down to the specific gestures and how the data is captured. This is further supported, by the Generic, Self-

Improving Controller's difficulty in differentiating any gestures when different data was used.

However, this may have more to do with the quality of the data than the gestures being used.

Finally, the Generic, Self-Improving Controller was also presented with six pre-defined pre-defined dynamic gestures. A maximum accuracy of around 50% was achieved after 50,000 samples per gesture. Further increasing the number of samples per gesture did not improve the learning. This is probably due to the fact that additional samples are simply the same data being repeatedly presented.

One of the key design components of the underlying Generic, Self-Improving Predictor, was the ability to also learn as fast as possible. When reprocessing the same data using different learning settings the same results could be achieved whether the data was presented in real-time or with a timestamp. The original intention of this design was to save time when rerunning experiments.

However, an additional benefit was that Iterative Learners with larger Flexible Artificial Neural Networks could also be used. Although the record trigger to sample the input data is processed as fast possible, as fast as possible may not mean faster than real-time. The processing time limitation described above is no longer a limitation when running simulations. However, with large samples and large artificial neural networks, individual simulations can take several days to run.

Chapter 10

Conclusion and Future Work

10.1 Thesis Summary

Learning human gestures is hard. The most common approach is to identify the important gestures; ask users to perform those gestures and record multiple instances of each gesture; pass the recorded data through a supervised learning algorithm; and use the learnt mapping to create feed-forward controller. However, the reason prosthetic users give for not using the advanced prosthetics available is that the control system is not intuitive, not responsive, and doesn't give feedback.

The goal of this research was to further existing research aimed at putting artificial intelligence into a wearable human-machine interfaces by a creating a Generic, Self-Improving Controller. A controller that could continually adapt to changing conditions. A controller that was intuitive to use,

because the user decided what gestures to use. A controller that was generic enough to use different input sensors and control different devices depending on need. The Generic, Self-Improving Controller created can use data captured from different types of IMUs and multiple MMGs to control the movement of a mouse pointer and mouse click using user-defined static gestures.

This research showed how IMU and MMG data can be captured processed, fused and stored in such a way that it can be presented to a Self-Improving Predictor that is indifferent to the source of the data. This research also showed how a Generic, Self-Improving Predictor can be used to iteratively learn input to output mappings irrespective of the meaning of the input and output data through simple feedback. In addition, this research showed how a new Blended Control Policy can use the output of the Generic, Self-Improving Predictor to provide a useable control signal even with minimal training. Finally, this research showed how these three components are combined into a Generic, Self-Improving Controller.

The Generic, Self-Improving Predictor components were tested using standard machine learning testing methods. The Flexible Artificial Neural Network was tested using the standard Modified National Institute of Standards and Technology (MNIST) handwritten digit database. The Iterative Learners were tested using logic gates and the one dimensional random walk problem. Finally the Generic, Self-Improving Predictor was tested using Tic-Tac-Toe and Backgammon.

Experiments were performed to demonstrate the functionality and test the limits of the Generic, Self-Improving Controller. The Generic, Self-Improving Controller was able to learn two user-defined static gestures with an accuracy of 100% in less than 10 samples per gesture. It was also able to learn three user-defined static gestures after less than 20 samples per gesture; and four user-defined static gestures with an average of less than 35 samples per gesture. Although 100% accuracy was achieved with five user-defined static gestures on a number of occasions, only 80%

accuracy was achieved on all occasions. Furthermore, the number of samples per gesture required to learn five static gestures was over 75 samples per gesture.

To increase the number of samples per gesture, the input and learning signals were recorded and played back multiple times to create simulated gesture data. Using 100 samples per gesture of recorded data played back multiple times, the Generic, Self-Improving Controller was shown to learn two pre-defined dynamic gestures with an accuracy of 90% in less than 6,000 samples per gesture. This was achieved using no historical data. The Generic, Self-Improving Controller learned three pre-defined dynamic gestures with an accuracy of 66% within 6,000 samples per gesture, and the accuracy did not significantly improve even after presenting 90,000 samples per gesture.

However, when presented with four pre-defined dynamic gestures, the Generic, Self-Improving Controller was also able to achieve a better average accuracy of around 70% after 50,000 samples per gesture. The dynamic gesture it struggled to differentiate when differentiating four pre-defined dynamic gestures is the same gesture that the Generic, Self-Improving Controller struggled to differentiate when only three pre-defined dynamic gestures were presented. The conclusion drawn was that the specific gesture and how the data is captured remains a limiting factor. This is further supported, by the Generic, Self-Improving Controller's difficulty to differentiate gestures when different data was used. Finally, the Generic, Self-Improving Controller was also presented with six pre-defined dynamic gestures. A maximum accuracy of around 50% was achieved after 50,000 samples per gesture. Increasing the number of samples per gesture did not improve the learning. However, this is probably due to the fact that additional samples are simply the same data being repeatedly presented.

This research has achieved another step towards putting artificial intelligence into wearable human machine interfaces. It tested a new approach that required creating a controller that would continually adapt to changing conditions; be intuitive to use; and be able to use different input sensors and control different devices depending on need. The Generic, Self-Improving Controller

created uses a new regularised TD learning equation applied to the artificial neural network back-propagation algorithm. Finally it identified three key limitations to the approach. First, capturing human gestures is dependent on the gesture, the sensors used to capture the gesture and the quality of the data. Second, the amount of data required to train the controller; especially when the human gestures are dynamic. Third, it becomes exponentially more difficult to differentiate between gestures as the number of gestures that need to be differentiated increases.

10.2 Achievements

This research combined signals from multiple different hardware sensors to create a standard input signal. The hardware sensors included multiple different types of IMU and MMGs. The IMU and MMG sensors were attached to users' forearms to measure forearm movement and orientation, and hand and finger muscle activation.

Artificial intelligence was used to continually improve the conversion of a standard input signal into the desired output signal. The output signal was used to reliably control a computer mouse pointer. Users were allowed to define their own static gestures used to perform the desired mouse pointer control outcomes.

During this research a number of new, improved and publicly formulated algorithms were created including:

- An improved AHRS orientation quaternion calculation algorithm (Chapter 4)
- Public formulation of the regularised TD learning algorithm (Section 5.3)
- Public formulation of the TD learning algorithm applied to the artificial neural network back-propagation algorithm (Section 5.4)

- A new regularised TD learning algorithm applied to the artificial neural network back-propagation algorithm (Section 5.5)
- A new prioritised reinforcement learning policy (Section 5.6)
- A new blended control policy (Section 5.7)

Where possible, this work has been published or submitted for publication.

Finally, the software tools and frameworks created during this research have been made available via on Bitbucket: <https://bitbucket.org/madmiraal/> via the GNU General Public Licence version 2.

10.3 Future Work

This research has provided an interesting first step towards a Generic, Self-Improving Controller. However, Chapter 9 highlighted that to realistically develop this approach further requires overcoming the need for an exponentially increasing number of samples for each gesture as the number of gestures increases. Many of the recent successes in using deep learning are due to the availability of enormous labelled datasets. In the case of reinforcement learning, there is a need for feedback. Games, with their built-in goal mechanism, provide an automatic feedback mechanism. However, correctly identifying human gestures currently still requires manual feedback or data labelling. Therefore, training needs to be done deliberately. Batch training can be used to train the controller offline and continuous improvement can then be made online, but the continuous improvement learning still requires a feedback mechanism. Intermediate goals can be used to overcome the need for continuous feedback, but even intermediate goals or tasks require feedback.

Furthermore, requiring users to learn pre-defined gestures is what makes using complicated machines hard and time consuming. This is the reason we require a licence to drive a car or fly a plane, and why operating even more complicated machines remains impractical. To reduce the

human learning burden, human-machine interfaces need to be intuitive. This is the main driver behind allowing the user to define the gestures used. Since users choose different gestures for the same desired outcome, it is not possible to create a labelled human-machine interface dataset. When the correct answers or labelled data are not available, supervised learning cannot be used. Without feedback, reinforcement learning cannot be used. Human-machine interfaces currently lack an automated reward signal, which prevents fully automated learning. One potential way to address these issues is to use the unsupervised learning approach used to train the first deep neural networks. This would enable most of the network's learning to be done without labelled data or feedback.

Even with automated feedback, a key challenge faced when using learning algorithms, especially artificial neural network based learning algorithms, is determining the best values for the large number of hyper-parameters. Their impact on the ability to learn and the speed of learning was seen in Chapters 8 and 9. One potential option for addressing the challenge of selecting the best values for the hyper-parameters is to use Gaussian processes⁵⁰. Gaussian processes can explore large multi-dimensional state-spaces to identify the locations where the greatest improvement can be made. The hyper-parameter state space may be best explored using Gaussian processes.

Artificial neural networks were an obvious way to address the curse of dimensionality in reinforcement learning, but artificial neural networks are not guaranteed to converge and they frequently diverge, even when trying to approximate simple linear functions like the logic gates in Section 8.2. Many of the problems with training artificial neural networks are due to the non-local impact of weight changes. Furthermore, a lot of the information in the analogue signals generated by the IMUs and MMGs is in the temporal patterns and not in the instantaneous data. Including a sufficient amount of historical data makes the input space orders of magnitude larger. The filters and triggers created can reduce the input space; however, this is akin to using hand-crafted features,

⁵⁰ (Rasmussen, 2006a)

which this research was attempting to avoid. A lot of these issues have been overcome in recent deep learning results. This research could be extended by applying these and other approaches obtained from new research.

In addition, all the learning algorithms assume that the vector representing the current state has the Markov property⁵¹. However, the vector representing the current state provided by the sensor data doesn't have the Markov property. In particular, identifying muscle activity, as measured using MMG data, is dependent on historical data. This research attempted to overcome this issue by storing historical data and using filters and triggers on the real-time sensor data. This significantly increases the size of the input state vector, which further impacts learning speeds. Furthermore, the amount of historical data required by each sensor not only creates additional hyper-parameters that need to be determined, they are also not constant. The historical data required varies between both the different types of gesture and between instances of the same gesture. A potential solution to this problem is to use LSTM or other new recurrent neural network techniques.

Learning is also severely impacted by two delays. The first delay is the delay between the user being instructed to perform a gesture and actually performing the gesture. The first delay is further aggravated if the user is still performing the previous gesture. Second, the delay after the user has performed a gesture. Static gestures, such as pointing, work with quaternions, because they are performed continuously. However, most dynamic gestures are only performed once, which results in an additional delay after the gesture has been performed. All the sensor data captured during these delays should either not be used or assigned to a different desired outcome. However, it is currently being used to incorrectly train the learner. Furthermore, when testing predictions, the output value at specific point in time is measured. This point in time may not be the point in time

51 A state has the Markov property if the probabilities of future states depend only on the present state (Bremaud, 1999).

the gesture was made. These issues may be overcome by using threshold triggers to indicate the start and finish of the user gesture.

When differentiating between more than two gestures, 50% is not the minimum confidence level required to be a correct prediction, but it is the maximum confidence level required. This fact is useful when considering a “do nothing” desired outcome. In most HMI situations, the “do nothing” desired outcome is actually the most likely desired outcome. The Action Predictor GUI already has the option to use the data between instructions to perform a gesture to train a “Do Nothing” desired outcome. However, this option has not yet been used.

Finally, the controller’s predictions can be used to both interpret a user's immediate intention and to predict the long-term goal or task the user is attempting to perform. By combining the two, the short-term differences can be used to update long-term predictions. By predicting a user’s long-term goal or task, the cognitive input of the user could be reduced, because the device does what it is expected to do. The controller could then be used to remotely control a robotic arm. The feedback signal would be the comparison of the next input signal with the predicted signal. The controller could then be extended to control multiple devices simultaneously. By predicting the user’s intention, remotely controlled robotic devices that suffer from feedback delay can be improved. This could be significant when attempting to control space based robotic arms from earth. Further applications include: automatically learning and interpreting human marshalling signals for automatic plane taxiing; predicting sport movements such as boxing and tennis to assist with training; and controlling multiple devices simultaneously such as swarms with natural gestures.

10.4 Final Thoughts

Exponential growth in processing power means that by 2030 a home computer is expected to have the processing power of the human brain. By 2033 a home computer will have the processing power

of the entire human population. Automation is fast making many jobs redundant. Technology used to create new jobs, but these jobs are becoming harder and harder to learn, and require more and more time to master. New jobs are already being automated faster than people can learn how to do them. Humans are being engineered out, because they're unable to learn how to use new technologies.

However, people shouldn't be learning how to use new technologies. New technologies should be learning what it is people want them to do. New technologies should be tools for humans not replacements for humans. To achieve this requires improving the integration with technology; of improving the human machine interface. Most people cannot imagine being that integrated with their technology, but even with today's technology, most people will be cyborgs by the time they die. Even if it is just a hip or a knee transplant.

Already people are volunteering to have knee transplants before they are necessary, simply because an artificial knee is better than their current worn out ones. In Germany, a man opted to have his severely damaged, but still partially functional hand amputated so that he could take advantage of a prosthetic one. Amputees are being denied access to the able bodied Olympics, because their artificial limbs are said to give them an unfair advantage. So the real question is, how long will it be before someone chooses to replace a perfectly good limb with an artificial one; simply because it's better?

If recent machine learning successes can be combined with the brain's ability to adapt and learn, prosthetic devices could become tools as simple to attach as slipping on a glove, and no more difficult to learn to use than learning how to drive a car or ride a bicycle. Furthermore, prosthetic devices are not limited to fully controlled hands, they include brain prosthetics. Imagine an attachable maths coprocessor for the brain allowing one to solve complex differential equations in their head.

It's all about designing tools. Whether it's a prosthetic to replace a lost limb, or a glove to provide the extra strength, dexterity or stability that a task demands. An interface to control a robotic arm on Mars or intuitively manipulate a third arm when needed. Tools that become second nature to use, in the same way that connecting to the Internet is today and making a phone call was last century. Tools that were unimaginable before.

References

- Abe, S. & Lan, M.-S. (1995) A method for fuzzy rules extraction directly from numerical data and its application to pattern classification. *IEEE Transactions on Fuzzy Systems*. [Online] 3 (1), 18–28. Available from: doi:10.1109/91.366565.
- Admiraal, M., Wilson, S. & Vaidyanathan, R. (2017) Improved formulation of the IMU and MARG orientation gradient descent algorithm for motion tracking in human-machine interfaces. In: *Multisensor Fusion and Integration for Intelligent Systems (MFI), 2017 IEEE International Conference on*. 2017 IEEE. pp. 403–410.
- Alcaide-Aguirre, R.E., Morgenroth, D.C. & Ferris, D.P. (2013) Motor control and learning with lower-limb myoelectric control in amputees. *The Journal of Rehabilitation Research and Development*. [Online] 50 (5), 687. Available from: doi:10.1682/JRRD.2012.06.0115.
- Anderson, C.W. (1986) *Learning and problem solving with multilayer connectionist systems*. Doctoral Dissertation. University of Massachusetts.
- Anderson, C.W. (1989) Learning to control an inverted pendulum using neural networks. *IEEE Control Systems Magazine*. [Online] 9 (3), 31–37. Available from: doi:10.1109/37.24809.
- Angeles, P., Mace, M., Admiraal, M., Burdet, E., et al. (2016) A Wearable Automated System to Quantify Parkinsonian Symptoms Enabling Closed Loop Deep Brain Stimulation. In: *Conference Towards Autonomous Robotic Systems*. 2016 Springer. pp. 8–19.
- Armijo, L. (1966) Minimization of functions having Lipschitz continuous first partial derivatives. *Pacific Journal of Mathematics*. 16 (1), 1–3.
- Asghari Oskoei, M. & Hu, H. (2007) Myoelectric control systems - A survey. *Biomedical Signal Processing and Control*. [Online] 2 (4), 275–294. Available from: doi:10.1016/j.bspc.2007.07.009.
- Assad, C., Wolf, M., Stoica, A., Theodoridis, T., et al. (2013) BioSleeve: A natural EMG-based interface for HRI. In: *2013 8th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*. [Online]. 2013 pp. 69–70. Available from: doi:10.1109/HRI.2013.6483505.
- Åström, K.J. & Wittenmark, B. (1995) *Adaptive Control*. Second Edition. Lund Institute of Technology, Addison-Wesley.

- Babiyak, M.A. (2004) What you see may not be what you get: a brief, nontechnical introduction to overfitting in regression-type models. *Psychosomatic medicine*. 66 (3), 411–421.
- Bellman, R. (2013) Google-Books-ID: CG7CAgAAQBAJ. *Dynamic Programming*. Courier Corporation.
- Bengio, Y. (2012a) Deep learning of representations for unsupervised and transfer learning. In: *Proceedings of ICML Workshop on Unsupervised and Transfer Learning*. 2012 pp. 17–36.
- Bengio, Y. (2009) Learning deep architectures for AI. *Foundations and trends in Machine Learning*. 2 (1), 1–127.
- Bengio, Y. (2012b) Practical recommendations for gradient-based training of deep architectures. *arXiv:1206.5533 [cs]*. [Online] Available from: <http://arxiv.org/abs/1206.5533> [Accessed: 21 June 2017].
- Bengio, Y., Lamblin, P., Popovici, D., Larochelle, H., et al. (2007) Greedy layer-wise training of deep networks. *Advances in neural information processing systems*. 19, 153.
- Berliner, H. (1979) On the Construction of Evaluation Functions for Large Domains. In: *Proceedings of the 6th International Joint Conference on Artificial Intelligence - Volume 1*. IJCAI'79. [Online]. 1979 San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. pp. 53–55. Available from: <http://dl.acm.org/citation.cfm?id=1624861.1624875> [Accessed: 8 August 2017].
- Bishop, C.M. (2006) *Pattern Recognition and Machine Learning*. Springer.
- Bitzer, S. & van der Smagt, P. (2006) Learning EMG control of a robotic hand: towards active prostheses. In: *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006*. [Online]. May 2006 pp. 2819–2823. Available from: doi:10.1109/ROBOT.2006.1642128.
- Blinn, J.F. (1997) Floating-point tricks. *IEEE Computer Graphics and Applications*. [Online] 17 (4), 80–84. Available from: doi:10.1109/38.595279.
- Boyan, J.A. (1992) *Modular neural networks for learning context-dependent game strategies*. University of Cambridge. Computer Laboratory.
- Boyan, J.A. & Moore, A.W. (1995) Generalization in reinforcement learning: Safely approximating the value function. *Advances in neural information processing systems*. 369–376.
- Bremaud, P. (1999) *Markov Chains: Gibbs Fields, Monte Carlo Simulation, and Queues*. Texts in Applied Mathematics. [Online]. New York, Springer-Verlag. Available from: [//www.springer.com/gp/book/9780387985091](http://www.springer.com/gp/book/9780387985091) [Accessed: 9 July 2018].

- Burr, C. (2017) *The Corner App*. [Online]. 2017. Available from: <https://thecornerapp.com/>.
- Castellini, C. & van der Smagt, P. (2008) Surface EMG in advanced hand prosthetics. *Biological Cybernetics*. [Online] 100 (1), 35–47. Available from: doi:10.1007/s00422-008-0278-1.
- Chan, F.H.Y., Yang, Y.-S., Lam, F.K., Zhang, Y.-T., et al. (2000) Fuzzy EMG classification for prosthesis control. *IEEE Transactions on Rehabilitation Engineering*. [Online] 8 (3), 305–311. Available from: doi:10.1109/86.867872.
- Chilimbi, T.M., Suzue, Y., Apacible, J. & Kalyanaraman, K. (2014) Project Adam: Building an Efficient and Scalable Deep Learning Training System. In: *OSDI*. [Online]. 2014 pp. 571–582. Available from: <https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-chilimbi.pdf> [Accessed: 29 August 2017].
- Cireřan, D., Meier, U. & Schmidhuber, J. (2012) Multi-column deep neural networks for image classification. In: *2012 IEEE Conference on Computer Vision and Pattern Recognition*. [Online]. June 2012 pp. 3642–3649. Available from: doi:10.1109/CVPR.2012.6248110.
- Clancy, E.A., Morin, E.L. & Merletti, R. (2002) Sampling, noise-reduction and amplitude estimation issues in surface electromyography. *Journal of Electromyography and Kinesiology*. [Online] 12 (1), 1–16. Available from: doi:10.1016/S1050-6411(01)00033-5.
- Collobert, R., Weston, J., Bottou, L., Karlen, M., et al. (2011) Natural Language Processing (almost) from Scratch. *arXiv:1103.0398 [cs]*. [Online] Available from: <http://arxiv.org/abs/1103.0398> [Accessed: 24 August 2017].
- Corbett, E.A., K rding, K.P. & Perreault, E.J. (2013) Real-Time Evaluation of a Noninvasive Neuroprosthetic Interface for Control of Reach. *IEEE Transactions on Neural Systems and Rehabilitation Engineering*. [Online] 21 (4), 674–683. Available from: doi:10.1109/TNSRE.2013.2251664.
- Cormen, T.H., Leiserson, C.E., Rivest, R.L. & Stein, C. (2009) *Introduction to algorithms*. 3. ed. Cambridge, Mass., MIT Press.
- Diebel, J. (2006) Representing attitude: Euler angles, unit quaternions, and rotation vectors. *Matrix*. 58 (15–16), 1–35.
- Dumas, B., Lalanne, D. & Oviatt, S. (2009) Multimodal interfaces: A survey of principles, models and frameworks. *Human machine interaction*. 3–26.

- Englehart, K. & Hudgins, B. (2003) A robust, real-time control scheme for multifunction myoelectric control. *IEEE Transactions on Biomedical Engineering*. [Online] 50 (7), 848–854. Available from: doi:10.1109/TBME.2003.813539.
- Farina, D. & Merletti, R. (2000) Comparison of algorithms for estimation of EMG variables during voluntary isometric contractions. *Journal of Electromyography and Kinesiology*. [Online] 10 (5), 337–349. Available from: doi:10.1016/S1050-6411(00)00025-0.
- Farrell, T.R. & Weir, R.F. (2007) The Optimal Controller Delay for Myoelectric Prostheses. *IEEE Transactions on Neural Systems and Rehabilitation Engineering*. [Online] 15 (1), 111–118. Available from: doi:10.1109/TNSRE.2007.891391.
- Fletcher, R. & Reeves, C.M. (1964) Function minimization by conjugate gradients. *The Computer Journal*. [Online] 7 (2), 149–154. Available from: doi:10.1093/comjnl/7.2.149.
- Frisoli, A., Loconsole, C., Leonardis, D., Banno, F., et al. (2012) A New Gaze-BCI-Driven Control of an Upper Limb Exoskeleton for Rehabilitation in Real-World Tasks. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*. [Online] 42 (6), 1169–1179. Available from: doi:10.1109/TSMCC.2012.2226444.
- Fukuda, O., Tsuji, T., Kaneko, M. & Otsuka, A. (2003) A human-assisting manipulator teleoperated by EMG signals and arm motions. *IEEE Transactions on Robotics and Automation*. [Online] 19 (2), 210–222. Available from: doi:10.1109/TRA.2003.808873.
- Gallego, J.Á., Ibanez, J., Dideriksen, J.L., Serrano, J.I., et al. (2012) A Multimodal Human-Robot Interface to Drive a Neuroprosthesis for Tremor Management. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*. [Online] 42 (6), 1159–1168. Available from: doi:10.1109/TSMCC.2012.2200101.
- Geman, S., Bienenstock, E. & Doursat, R. (1992) Neural Networks and the Bias/Variance Dilemma. *Neural Computation*. [Online] 4 (1), 1–58. Available from: doi:10.1162/neco.1992.4.1.1.
- Glorot, X. & Bengio, Y. (2010) Understanding the difficulty of training deep feedforward neural networks. In: *Aistats*. [Online]. 2010 pp. 249–256. Available from: http://www.jmlr.org/proceedings/papers/v9/glorot10a/glorot10a.pdf?hc_location=ufi [Accessed: 21 June 2017].
- Goldstein, A.A. (1962) Cauchy's method of minimization. *Numerische Mathematik*. 4 (1), 146–150.
- Goodfellow, I., Bengio, Y. & Courville, A. (2016) Book in preparation for MIT Press. *Deep Learning*. [Online]. Unpublished. Available from: <http://www.deeplearningbook.org>.

- Gopura, R.A.R.C., Bandara, D.S.V., Gunasekara, J.M.P. & Jayawardane, T.S.S. (2013) Recent Trends in EMG-Based Control Methods for Assistive Robots. In: Hande Turker (ed.). *Electrodiagnosis in New Frontiers of Clinical Research*. InTech. pp. 237–264.
- Graf, U. (2012) Google-Books-ID: wCIGCAAQBAJ. *Applied Laplace Transforms and z-Transforms for Scientists and Engineers: A Computational Approach using a Mathematica Package*. Birkhäuser.
- Gupta, L., Chung, B., Srinath, M.D., Molfese, D.L., et al. (2005) Multichannel fusion models for the parametric classification of differential brain activity. *IEEE Transactions on Biomedical Engineering*. [Online] 52 (11), 1869–1881. Available from: doi:10.1109/TBME.2005.856272.
- Hall, D.L. & Llinas, J. (1997) An introduction to multisensor data fusion. *Proceedings of the IEEE*. [Online] 85 (1), 6–23. Available from: doi:10.1109/5.554205.
- Hall, D.L. & McMullen, S.A.H. (2004) Google-Books-ID: HxjDMcJWLYwC. *Mathematical Techniques in Multisensor Data Fusion*. Artech House.
- Hall, D.L. & Steinberg, A.N. (2001) Dirty Secrets in Multisensor Data Fusion. In: *Multisensor Data Fusion*. CRC Press. p.
- Hargrove, L.J., Scheme, E.J., Englehart, K.B. & Hudgins, B.S. (2010) Multiple Binary Classifications via Linear Discriminant Analysis for Improved Controllability of a Powered Prosthesis. *IEEE Transactions on Neural Systems and Rehabilitation Engineering*. [Online] 18 (1), 49–57. Available from: doi:10.1109/TNSRE.2009.2039590.
- Hargrove, L.J., Simon, A.M., Young, A.J., Lipschutz, R.D., et al. (2013) Robotic Leg Control with EMG Decoding in an Amputee with Nerve Transfers. *New England Journal of Medicine*. [Online] 369 (13), 1237–1242. Available from: doi:10.1056/NEJMoa1300126.
- Harrison, H.R. (1999) Quaternions and Rotation Sequences: a Primer with Applications to Orbits, Aerospace and Virtual Reality, J. B. Kuipers, Princeton University Press, 41 William Street, Princeton, NJ 08540, USA. 1999. 372pp. Illustrated. £35.00. ISBN 0-691-05872-5. *The Aeronautical Journal*. [Online] 103 (1021), 175–175. Available from: doi:10.1017/S0001924000065039.
- He, K., Zhang, X., Ren, S. & Sun, J. (2015) Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *arXiv:1502.01852 [cs]*. [Online] Available from: <http://arxiv.org/abs/1502.01852> [Accessed: 12 February 2015].
- Hestenes, M.R. & Stiefel, E. (1952) Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*. 49 (6), 409–436.

- Hinton, G., Osindero, S. & Teh, Y.-W. (2006) A fast learning algorithm for deep belief nets. *Neural computation*. 18 (7), 1527–1554.
- Hinton, G.E. & Salakhutdinov, R.R. (2006) Reducing the Dimensionality of Data with Neural Networks. *Science*. [Online] 313 (5786), 504–507. Available from: doi:10.1126/science.1127647.
- Hochreiter, S. & Schmidhuber, J. (1997) Long Short-Term Memory. *Neural Computation*. [Online] 9 (8), 1735–1780. Available from: doi:10.1162/neco.1997.9.8.1735.
- Hoffman, M.W., Lazaric, A., Ghavamzadeh, M. & Munos, R. (2011) Regularized least squares temporal difference learning with nested ℓ_2 and ℓ_1 penalization. In: *European Workshop on Reinforcement Learning*. [Online]. 2011 Springer. pp. 102–114. Available from: https://link.springer.com/chapter/10.1007/978-3-642-29946-9_13 [Accessed: 25 July 2017].
- Huang, H., Zhang, F., Hargrove, L.J., Dou, Z., et al. (2011) Continuous Locomotion-Mode Identification for Prosthetic Legs Based on Neuromuscular Mechanical Fusion. *IEEE Transactions on Biomedical Engineering*. [Online] 58 (10), 2867–2875. Available from: doi:10.1109/TBME.2011.2161671.
- Hudgins, B., Parker, P. & Scott, R.N. (1993) A new strategy for multifunction myoelectric control. *IEEE Transactions on Biomedical Engineering*. [Online] 40 (1), 82–94. Available from: doi:10.1109/10.204774.
- Jaimes, A. & Sebe, N. (2007) Multimodal human–computer interaction: A survey. *Computer Vision and Image Understanding*. [Online] 108 (1), 116–134. Available from: doi:10.1016/j.cviu.2006.10.019.
- Kaelbling, L.P., Littman, M.L. & Moore, A.W. (1996) Reinforcement learning: A survey. *Journal of artificial intelligence research*. 4, 237–285.
- Kendall, A. & Gal, Y. (2017) What uncertainties do we need in bayesian deep learning for computer vision? In: *Advances in neural information processing systems*. 2017 pp. 5574–5584.
- Kingma, H. (2005) Thresholds for perception of direction of linear acceleration as a possible evaluation of the otolith function. *BMC Ear, Nose and Throat Disorders*. [Online] 5, 5. Available from: doi:10.1186/1472-6815-5-5.
- Kotsiantis, S.B., Zaharakis, I. & Pintelas, P. (2007) Supervised machine learning: A review of classification techniques. *Informatica*. (31), 249–268.
- Krizhevsky, A. & Hinton, G.E. (2011) Using very deep autoencoders for content-based image retrieval. In: *ESANN*. 2011 Citeseer. p.

- Krizhevsky, A., Sutskever, I. & Hinton, G.E. (2012) Imagenet classification with deep convolutional neural networks. In: *Advances in neural information processing systems*. [Online]. 2012 pp. 1097–1105. Available from: <http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks> [Accessed: 29 August 2017].
- LeCun, Y., Cortes, C. & Burges, C. (n.d.) *MNIST handwritten digit database*. [Online]. Available from: <http://yann.lecun.com/exdb/mnist/>.
- Lemaréchal, C. (2012) Cauchy and the Gradient Method. *Documenta Mathematica*. Extra Volume ISMP, 251–254.
- Lenz, I., Lee, H. & Saxena, A. (2015) Deep learning for detecting robotic grasps. *The International Journal of Robotics Research*. [Online] 34 (4–5), 705–724. Available from: doi:10.1177/0278364914549607.
- Liggins, M., II, Hall, D. & Llinas, J. (2017) *Handbook of Multisensor Data Fusion: Theory and Practice, Second Edition*. CRC Press.
- Lillicrap, T.P., Hunt, J.J., Pritzel, A., Heess, N., et al. (2015) Continuous control with deep reinforcement learning. *arXiv:1509.02971 [cs, stat]*. [Online] Available from: <http://arxiv.org/abs/1509.02971> [Accessed: 30 August 2017].
- Lipton, Z.C., Berkowitz, J. & Elkan, C. (2015) A Critical Review of Recurrent Neural Networks for Sequence Learning. *arXiv:1506.00019 [cs]*. [Online] Available from: <http://arxiv.org/abs/1506.00019> [Accessed: 4 January 2016].
- Liu, B., Mahadevan, S. & Liu, J. (2012) Regularized off-policy TD-learning. In: *Advances in Neural Information Processing Systems*. [Online]. 2012 pp. 836–844. Available from: <http://papers.nips.cc/paper/4685-regularized-off-policy-td-learning> [Accessed: 25 July 2017].
- Lorusso, A., Eggert, D.W., Eggert, A.L.D. & Fisher, R.B. (1995) *A Comparison of Four Algorithms for Estimating 3-D Rigid Transformations*.
- Lovell, J. & Kluger, J. (2006) Google-Books-ID: LDJ43xYxK5YC. *Apollo 13*. Houghton Mifflin Harcourt.
- Mace, M., Abdullah-al-Mamun, K., Naeem, A.A., Gupta, L., et al. (2013) A heterogeneous framework for real-time decoding of bioacoustic signals: Applications to assistive interfaces and prosthesis control. *Expert Systems with Applications*. [Online] 40 (13), 5049–5060. Available from: doi:10.1016/j.eswa.2013.03.028.
- Madgwick, S.O., Harrison, A.J.L. & Vaidyanathan, R. (2011) Estimation of IMU and MARG orientation using a gradient descent algorithm. In: *Rehabilitation Robotics (ICORR), 2011 IEEE International*

Conference on. [Online]. 2011 IEEE. pp. 1–7. Available from:
<http://ieeexplore.ieee.org/abstract/document/5975346/> [Accessed: 2 February 2017].

Maier, S. & van der Smagt, P. (2008) Surface EMG suffices to classify the motion of each finger independently. In: *Proceedings of MOVIC 2008, 9th International Conference on Motion and Vibration Control*. 2008 pp. 15–18.

Mamun, K.A., Mace, M., Gupta, L., Verschuur, C.A., et al. (2012) Robust real-time identification of tongue movement commands from interferences. *Neurocomputing*. [Online] 80, 83–92. Available from: doi:10.1016/j.neucom.2011.09.018.

McClelland, J.L. (2015) *Explorations in parallel distributed processing: A handbook of models, programs, and exercises*. Second Edition-Draft.

McCulloch, W.S. & Pitts, W. (1943) A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*. 5 (4), 115–133.

Minsky, M.L. & Papert, S.A. (1969) *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge.

Mitchell, H.B. (2007) Google-Books-ID: 2hwcFSxQ1CAC. *Multi-Sensor Data Fusion: An Introduction*. Springer Science & Business Media.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., et al. (2013) Playing Atari with Deep Reinforcement Learning. *arXiv:1312.5602 [cs]*. [Online] Available from: <http://arxiv.org/abs/1312.5602> [Accessed: 25 February 2015].

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., et al. (2015) Human-level control through deep reinforcement learning. *Nature*. [Online] 518 (7540), 529–533. Available from: doi:10.1038/nature14236.

Nielsen, M.A. (2015) *Neural Networks and Deep Learning*. [Online]. Available from: <http://neuralnetworksanddeeplearning.com> [Accessed: 24 May 2016].

Novak, D. & Riener, R. (2015) A survey of sensor fusion methods in wearable robotics. *Robotics and Autonomous Systems*. [Online] 73, 155–170. Available from: doi:10.1016/j.robot.2014.08.012.

Orizio, C. (1993) Muscle Sound: Bases for the Introduction of a Mechanomyographic Signal in Muscle Studies. *Critical Reviews in Biomedical Engineering*. 21 (3), 201–243.

- Oviatt, S. (2003) Advances in Robust Multimodal Interface Design. *IEEE Comput. Graph. Appl.* [Online] 23 (5), 62–68. Available from: doi:10.1109/MCG.2003.1231179.
- Padgett, W. & Anderson, D. (2009) Google-Books-ID: EclcAQAAQBAJ. *Fixed-Point Signal Processing*. Morgan & Claypool Publishers.
- Peerdeman, B., Boere, D., Witteveen, H., Huis in 't Veld, R., et al. (2011) Myoelectric forearm prostheses: State of the art from a user-centered perspective. *The Journal of Rehabilitation Research and Development*. [Online] 48 (6), 719–738. Available from: doi:10.1682/JRRD.2010.08.0161.
- Pilarski, P.M., Dawson, M.R., Degris, T., Carey, J.P., et al. (2013) Adaptive artificial limbs: a real-time approach to prediction and anticipation. *IEEE Robotics Automation Magazine*. 20 (1) pp.53–64.
- Pilarski, P.M., Dawson, M.R., Degris, T., Fahimi, F., et al. (2011) Online human training of a myoelectric prosthesis controller via actor-critic reinforcement learning. In: *2011 IEEE International Conference on Rehabilitation Robotics (ICORR)*. [Online]. 29 June 2011 ETH Zurich Science City, Switzerland. pp. 1–7. Available from: doi:10.1109/ICORR.2011.5975338.
- Pistohl, T., Cipriani, C., Jackson, A. & Nazarpour, K. (2013) Abstract and Proportional Myoelectric Control for Multi-Fingered Hand Prostheses. *Annals of Biomedical Engineering*. [Online] Available from: doi:10.1007/s10439-013-0876-5.
- Polak, E. & Ribière, G. (1969) Note sur la convergence de méthodes de directions conjuguées. *Revue française d'informatique et de recherche opérationnelle, série rouge*. 3 (1), 35–43.
- Pollack, J.B. & Blair, A.D. (1997) Why did TD-Gammon Work? *Advances in Neural Information Processing Systems*. 10–16.
- Posatskiy, A.O. & Chau, T. (2012) Design and evaluation of a novel microphone-based mechanomyography sensor with cylindrical and conical acoustic chambers. *Medical Engineering & Physics*. [Online] 34 (8), 1184–1190. Available from: doi:10.1016/j.medengphy.2011.12.007.
- Powell, M.J.D. (1976) Some global convergence properties of a variable metric algorithm for minimization without exact line searches. *SIAM-AMS Proceedings*. 9, 53–72.
- Press, W.H. (2007) *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press.
- Proakis, J.G. & Manolakis, D.G. (1996) *Digital Signal Processing*. Prentice Hall.

- Rabiner, L.R. & Sambur, M.R. (1974) Algorithm for determining the endpoints of isolated utterances. *The Journal of the Acoustical Society of America*. [Online] 56 (S1), S31–S31. Available from: doi:10.1121/1.1914118.
- Raina, R., Battle, A., Lee, H., Packer, B., et al. (2007) Self-taught learning: transfer learning from unlabeled data. In: *Proceedings of the 24th international conference on Machine learning*. [Online]. 2007 ACM. pp. 759–766. Available from: <http://dl.acm.org/citation.cfm?id=1273592> [Accessed: 11 August 2014].
- Ranzato, M.A., Poultney, C., Sumit, C. & Yann, L. (2006) Efficient learning of sparse representations with an energy-based model. In: *Advances in neural information processing systems*. 2006 pp. 1137–1144.
- Rasmussen, C.E. (2006a) *Gaussian processes in machine learning*. In: 2006 MIT Press. pp. 67–75.
- Rasmussen, C.E. (2006b) *Minimize*. [Online]. 6 April 2006. Available from: <http://learning.eng.cam.ac.uk/carl/code/minimize/>.
- Reiter, R. (1948) Eine neue elektrokunsthand. *Grenzgebiete der Medizin*. (4), 133–135.
- Riedmiller, M. (2005) Neural fitted Q iteration—first experiences with a data efficient neural reinforcement learning method. In: *Machine Learning: ECML 2005*. [Online]. Springer. pp. 317–328. Available from: http://link.springer.com/chapter/10.1007/11564096_32 [Accessed: 25 February 2015].
- Rosenblatt, F. (1958) The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*. 65 (6), 386.
- Rosetta Code (n.d.) *Fast Fourier transform*. [Online]. Available from: https://rosettacode.org/wiki/Fast_Fourier_transform#C.2B.2B [Accessed: 14 August 2017].
- RSLSteeper (2014) *Bionic hand with 14 grip patterns that makes common tasks easy*. [Online]. 18 July 2014. The world's most advanced Prosthetic Hand - bebionic. Available from: http://bebionic.com/the_hand/grip_patterns [Accessed: 18 July 2014].
- Rumelhart, D.E., Hinton, G.E. & Williams, R.J. (1985) *Learning internal representations by error propagation*.
- Russell, S.J. & Norvig, P. (2010) *Artificial Intelligence: A Modern Approach*. Prentice Hall.
- Samuel, A.L. (1959) Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*. 3 (3).

- Saunders, I. & Vijayakumar, S. (2011) The role of feed-forward and feedback processes for closed-loop prosthesis control. *Journal of neuroengineering and rehabilitation*. 8 (60), 60–72.
- Scheme, E. & Englehart, K. (2011) Electromyogram pattern recognition for control of powered upper-limb prostheses: State of the art and challenges for clinical use. *The Journal of Rehabilitation Research and Development*. [Online] 48 (6), 643–660. Available from: doi:10.1682/JRRD.2010.09.0177.
- Schulman, J., Abbeel, P. & Chen, X. (2017) Equivalence Between Policy Gradients and Soft Q-Learning. *arXiv:1704.06440 [cs]*. [Online] Available from: <http://arxiv.org/abs/1704.06440> [Accessed: 8 May 2017].
- Seide, F., Li, G. & Yu, D. (2011) Conversational speech transcription using context-dependent deep neural networks. In: *Twelfth Annual Conference of the International Speech Communication Association*. [Online]. 2011 p. Available from: <https://pdfs.semanticscholar.org/3fa7/2c96b7548a3207a69f1e55e039fc5a314696.pdf> [Accessed: 29 August 2017].
- Sensinger, J.W., Lock, B.A. & Kuiken, T.A. (2009) Adaptive Pattern Recognition of Myoelectric Signals: Exploration of Conceptual Framework and Practical Algorithms. *IEEE Transactions on Neural Systems and Rehabilitation Engineering*. [Online] 17 (3), 270–278. Available from: doi:10.1109/TNSRE.2009.2023282.
- Shepperd, S.W. (1978) Quaternion from rotation matrix. [four-parameter representation of coordinate transformation matrix]. *Journal of Guidance and Control*. [Online] Available from: <https://ntrs.nasa.gov/search.jsp?R=19780048191> [Accessed: 2 February 2017].
- Silver, D., Huang, A., Maddison, C.J., Guez, A., et al. (2016) Mastering the game of Go with deep neural networks and tree search. *Nature*. [Online] 529 (7587), 484–489. Available from: doi:10.1038/nature16961.
- Simpson, D.C. (1974) The choice of control system for the multimovement prosthesis: Extended physiological proprioception (EPP). In: P. Herberts (ed.). *The control of upper-extremity prostheses and orthoses*. Springfield (IL). pp. 146–150.
- Smith, J.O. (2007) *Mathematics of the discrete Fourier transform (DFT): with audio applications*. Julius Smith.
- Smith, L.H., Hargrove, L.J., Lock, B.A. & Kuiken, T.A. (2011) Determining the Optimal Window Length for Pattern Recognition-Based Myoelectric Control: Balancing the Competing Effects of Classification

Error and Controller Delay. *IEEE Transactions on Neural Systems and Rehabilitation Engineering*. [Online] 19 (2), 186–192. Available from: doi:10.1109/TNSRE.2010.2100828.

ST (2013) *LSM9DS0 Datasheet*.p.74.

Stokes, M.J. & Dalton, P.A. (1991) Acoustic myography for investigating human skeletal muscle fatigue. *Journal of Applied Physiology*. 71 (4), 1422–1426.

Sutton, R.S. (1988) Learning to predict by the methods of temporal differences. *Machine Learning*. [Online] 3 (1), 9–44. Available from: doi:10.1007/BF00115009.

Sutton, R.S. & Barto, A.G. (1998) *Reinforcement Learning: An Introduction*. MIT Press.

Tesauro, G. (1998) Comments on “Co-Evolution in the Successful Learning of Backgammon Strategy”. *Machine Learning*. [Online] 32 (3), 241–243. Available from: doi:10.1023/A:1007469231743.

Tesauro, G. (1992) Practical Issues in Temporal Difference Learning. In: Richard S. Sutton (ed.). *Reinforcement Learning*. The Springer International Series in Engineering and Computer Science. [Online]. Springer US. pp. 33–53. Available from: doi:10.1007/978-1-4615-3618-5_3 [Accessed: 16 February 2016].

Tesauro, G. (1995) Temporal Difference Learning and TD-Gammon. *Commun. ACM*. [Online] 38 (3), 58–68. Available from: doi:10.1145/203330.203343.

Tesauro, G. & Sejnowski, T.J. (1989) A parallel network that learns to play backgammon. *Artificial intelligence*. 39 (3), 357–390.

Tsitsiklis, J.N. & Van Roy, B. (1997) An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*. [Online] 42 (5), 674–690. Available from: doi:10.1109/9.580874.

Valenti, R.G., Dryanovski, I. & Xiao, J. (2016) A Linear Kalman Filter for MARG Orientation Estimation Using the Algebraic Quaternion Algorithm. *IEEE Transactions on Instrumentation and Measurement*. [Online] 65 (2), 467–481. Available from: doi:10.1109/TIM.2015.2498998.

Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., et al. (2010) Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *The Journal of Machine Learning Research*. 11, 3371–3408.

- Vinyals, O., Ewalds, T., Bartunov, S., Georgiev, P., et al. (2017) StarCraft II: A New Challenge for Reinforcement Learning. *arXiv:1708.04782 [cs]*. [Online] Available from: <http://arxiv.org/abs/1708.04782> [Accessed: 26 August 2017].
- Weir, R.F. (2003) The great divide - the human-machine interface. Issues in the control of prostheses, manipulators, and other human machine systems. In: *Proceedings of Bioengineering Conference, 2003 IEEE 29th Annual*. [Online]. 2003 pp. 275–276. Available from: doi:10.1109/NEBC.2003.1216101.
- Werbos, P. (1974) *Beyond regression: New tools for prediction and analysis in the behavioral sciences*. Harvard University.
- Widrow, B. & Hoff, M.E. (1960) *Adaptive switching circuits*.
- Williams, T.W., III (2011) Progress on stabilizing and controlling powered upper-limb prostheses. *Journal of rehabilitation research and development*. [Online] 48 (6), ix–xix. Available from: doi:10.1682/JRRD.2011.04.0078.
- Winder, S. (2002) Google-Books-ID: NuRvHVMuzI4C. *Analog and Digital Filter Design*. Elsevier.
- Wojtczak, P., Amaral, T.G., Dias, O.P., Wolczowski, A., et al. (2009) Hand movement recognition based on biosignal analysis. *Engineering Applications of Artificial Intelligence*. [Online] 22 (4–5), 608–615. Available from: doi:10.1016/j.engappai.2008.12.004.
- Wolfe, P. (1969) Convergence conditions for ascent methods. *SIAM review*. 11 (2), 226–235.
- Wolfe, P. (1971) Convergence conditions for ascent methods. II: Some corrections. *SIAM review*. 13 (2), 185–188.
- Woodward, R., Shefelbine, S. & Vaidyanathan, R. (2014) Pervasive Motion Tracking and Muscle Activity Monitor. In: *2014 IEEE 27th International Symposium on Computer-Based Medical Systems (CBMS)*. [Online]. May 2014 pp. 421–426. Available from: doi:10.1109/CBMS.2014.43.
- Wright, S.J. & Nocedal, J. (1999) *Numerical optimization*. Springer New York.
- Yin, Y.H., Fan, Y.J. & Xu, L.D. (2012) EMG and EPP-Integrated Human-Machine Interface Between the Paralyzed and Rehabilitation Exoskeleton. *IEEE Transactions on Information Technology in Biomedicine*. [Online] 16 (4), 542–549. Available from: doi:10.1109/TITB.2011.2178034.
- Yosinski, J., Clune, J., Bengio, Y. & Lipson, H. (2014) How transferable are features in deep neural networks? In: Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, et al. (eds.). *Advances in*

Neural Information Processing Systems 27. [Online]. Curran Associates, Inc. pp. 3320–3328. Available from: <http://papers.nips.cc/paper/5347-how-transferable-are-features-in-deep-neural-networks.pdf> [Accessed: 29 August 2017].

Yuan, Y. (2008) Step-sizes for the gradient method. *AMS IP Studies in Advanced Mathematics*. 42 (2), 785.

Zecca, M., Micera, S., Carrozza, M.C. & Dario, P. (2002) Control of Multifunctional Prosthetic Hands by Processing the Electromyographic Signal. *Critical Reviews in Biomedical Engineering*. [Online] 30 (4–6), 459–485. Available from: doi:10.1615/CritRevBiomedEng.v30.i456.80.

Zhao, J., Xie, Z., Jiang, L., Cai, H., et al. (2006) EMG Control for a Five-fingered Underactuated Prosthetic Hand Based on Wavelet Transform and Sample Entropy. In: *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*. [Online]. October 2006 pp. 3215–3220. Available from: doi:10.1109/IROS.2006.282425.

Appendix A Software Created

Appendix A.1 Software Selection Decisions

The majority of the software created during this project was implemented in C++. The main reason for this choice was driven by the cross-platform support available – especially for cross-platform windows framework support – before Microsoft made C# open source at the end of 2014.

Furthermore, the performance expectations of a compiled language are higher than interpretive languages such as C# and Java; although these performance differences are considered smaller now.

The lack of strong-typing makes Python an easy language to learn, but personal experience with creating more complex programmes in Python has resulted in difficult to troubleshoot run-time crashes that would have been detected at compile time by a strong-typed language compiler such as GCC for C++.

Matlab was not chosen, because it has been found to be too slow for running real-time applications. Furthermore, although the optimisations in Matlab make matrix calculations incredibly fast, the multi-threaded matrix calculations created in C++ during this research are as fast.

To create cross-platform graphical user interfaces (GUIs), the wxWidgets library⁵² was chosen. In addition to being open source, its main advantage over other libraries is its use of native software development kits (SDKs), which makes applications look native to the operating system they run on.

At the start of this research, there was an artificial neural network open source library available in C++ called FAAN⁵³. This library was not chosen, because this research required updating not only the interface to support iterative learning and learning while being used, but also the way the back-propagation algorithm is implemented to support TD learning and this research's reformulation that includes regularisation.

There are now many more artificial neural network libraries^{54, 55}, including many popular implementations such as Theano in Python⁵⁶, Torch in Lua⁵⁷, Lush in Lisp⁵⁸ and TensorFlow⁵⁹. Even Matlab has extended their statistics toolbox to support machine learning⁶⁰. All these options were not available at the start of this research, and it has not yet been investigated how easy it would be to implement the changes required by this research.

All the software code created during this project can be found on Bitbucket:

<https://bitbucket.org/madmiraal/> and is made available through the GNU General Public Licence

52 <https://www.wxwidgets.org/>

53 <http://leenissen.dk/fann/wp/>

54 <http://caffe.berkeleyvision.org/>

55 <http://www.opennn.net/>

56 <http://deeplearning.net/software/theano/>

57 <http://torch.ch/>

58 <http://lush.sourceforge.net/>

59 <https://www.tensorflow.org/>

60 <https://uk.mathworks.com/products/statistics.html>

version 2. The full GNU General Public License version 2 is available at

<https://www.gnu.org/licenses/old-licenses/gpl-2.0.html>.

Appendix A.2 Library and Application Dependencies

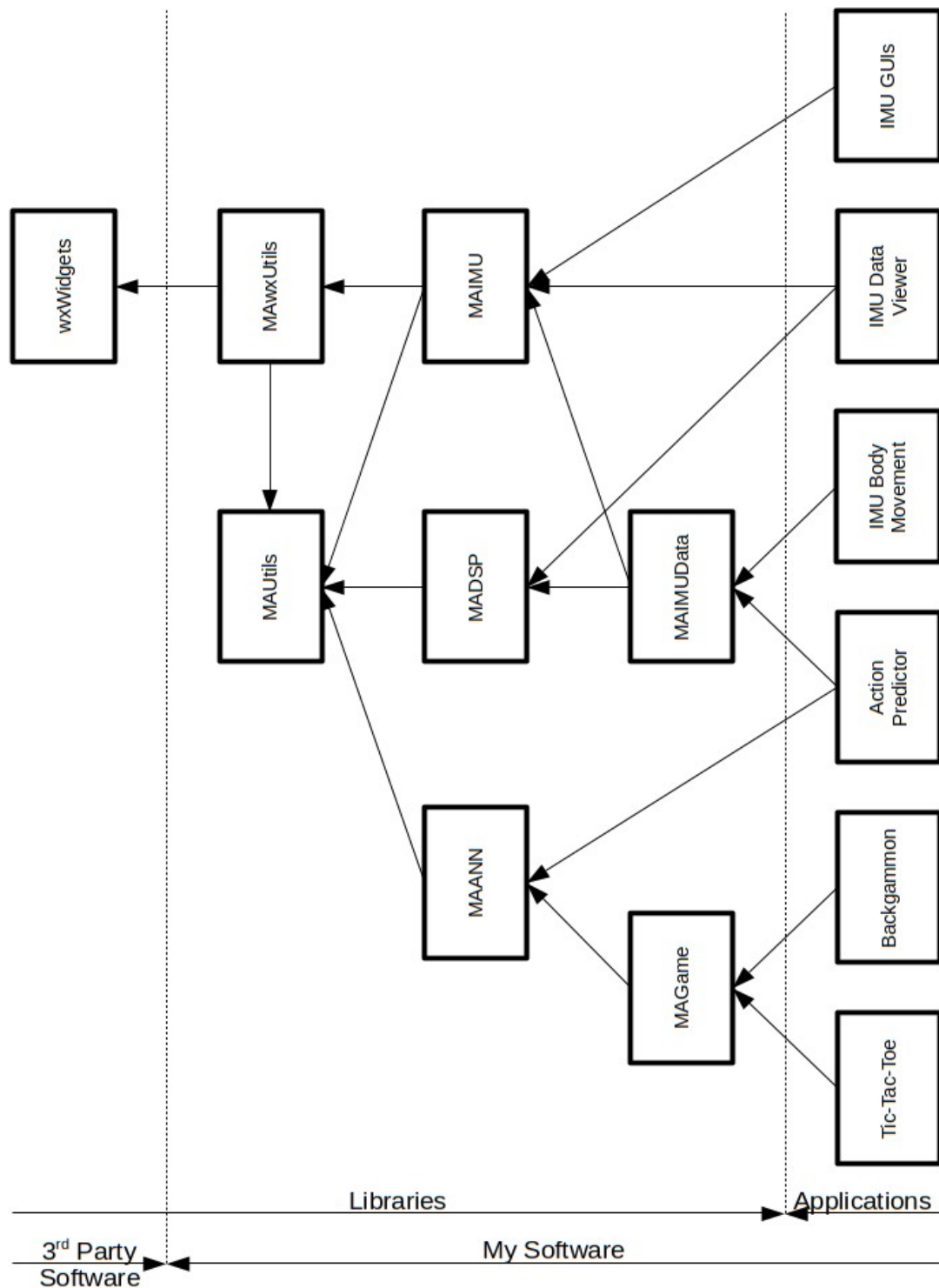


Figure A.1: Software library and application dependencies.

Appendix A.3 *MAUtils Library*

A cross-platform C++ library of utilities:

Appendix A.3.1 String Utilities

ma::Str: A class for manipulating character arrays as strings.

ma::StrStream: A class for streaming to and from the Str class strings.

Appendix A.3.2 Maths Utilities

ma::Complex: A class for creating and manipulating complex numbers.

ma::Vector: A class for creating and manipulating vectors.

ma::Vector3D: A class for creating and manipulating three-dimensional vectors.

ma::VectorComplex: A class for creating and manipulating vectors of complex numbers.

ma::Quaternion: A class for creating and manipulating quaternions.

ma::Matrix: A class for creating and manipulating matrices.

Appendix A.3.3 Communication Utilities

ma::Serial: A cross-platform class for connecting to, reading from and writing to serial ports.

ma::Device: Enables working with input devices used by a Linux system.

ma::Pointer: A cross-platform class for programmably controlling the pointer.

Appendix A.3.4 Collection Utilities

ma::LinkedList: A template class for creating a node in a linked list.

ma::List: A template class that can be used as a variable length array.

ma::Stack: A template class for creating a stack of objects.

ma::Collection: A template class for bundling objects of a particular class together.

ma::Map: A template class for storing key-value pairs.

ma::Iterator: A template class for iterating though objects in a ma::Collection or a linked list of ma::LinkedList.

Appendix A.3.5 Optimisation Utilities

ma::Learner: A base abstract class for creating temporal learning algorithms.

ma::MCMCLearner: A Markov Chain, Monte-Carlo temporal learning algorithm.

Appendix A.3.6 Conjugate Gradient Descent Algorithm

ma::minimise: Returns the minimum of a continuous differentiable multivariate function. It uses the Polack-Ribiere flavour of conjugate gradients to compute search directions. It performs a line search using quadratic and cubic polynomial approximations. The Wolfe-Powell stopping criteria are used together with the slope ratio method for guessing initial step sizes. It checks to ensure that exploration is taking place and that extrapolation will not be boundlessly large.

Appendix A.3.7 Logging Utilities

ma::Logger: Logs data to a log file along with a time stamp.

ma::LogPlayback: Replays files created by ma::Logger.

ma::TemporalData: Keeps a record of data values over a period of historical time.

Appendix A.3.8 Other Utilities

ma::approxEqual: Enables comparison of doubles with rounding errors.

ma::displayPacket: Enables viewing an array of bytes in various output formats.

ma::intSqrt: A quick integer approximation of the square root of an integer without using division or a squareroot function.

ma::invSqrt: A quick approximation of the inverse square root of a float without using division or a squareroot function.

ma::renameFile: Renames a filename by adding or incrementing the version number.

Appendix A.3.9 Algorithms

ma::AHRS: Attitude and heading reference system quaternion calculations and corrections from gyroscope, accelerometer and magnetometer readings.

ma::fft: Cooley-Tukey Fast Fourier Transform, in-place, breadth-first, decimation-in-frequency.

ma::mergeSort: Sorts an array of items that have the < operator defined using the merge sort algorithm.

ma::quickSort: Sorts an array of items that have the < operator defined using the quick sort algorithm.

ma::quickSelect: Returns the i^{th} smallest element using the quick select algorithm.

ma::inversions: Returns the number of inversions in an array.

Appendix A.3.10 Compile-Time Series Calculations

struct sin: Compile time evaluation of the sine time series.

struct cos: Compile time evaluation of the cosine time series.

Appendix A.4 *MawxUtils Library*

A cross-platform C++ library for utilities used with wxWidgets:

Appendix A.4.1 Control Utilities

ma::Control: A generic control class that runs inside a `ma::ControlFrame` and includes a continuously looping thread that can be used as the primary thread, and any number of regularly triggered timers.

ma::Logger: A generic logger that logs any data with a timestamp relative to the start of the log file.

ma::LogPlayback: A generic playback class that replays a logfile created with the logger with the same time gaps.

Appendix A.4.2 Display Utilities

ma::ControlFrame: A generic control frame class. Enables the `ma::Control` class and includes a menu bar, a tool bar and a status bar that can be updated outside the frame's thread.

ma::SericalPanel: A `wxPanel` for connecting to the MAUtils `ma::Serial` ports.

ma::OscilloscopePanel: A wxPanel for displaying multiple data streams in real-time.

ma::FitTextPanel: A wxPanel that automatically resizes text to fit inside the panel.

ma::CuboidPanel: A wxPanel for displaying a rotatable 3-dimensional cube.

ma::Display3DPanel: A wxPanel for displaying a 3-dimensional vector.

Appendix A.4.3 Factory Utilities

ma::rgbaToWxBitmap: Converts a RGBA character array to a wxBitmap.

ma::getColours: Creates an array of the required number of different colours.

Appendix A.5 MAANN Library

A cross-platform C++ library for creating the Flexible Artificial Neural Network.

ma::NeuralNetwork: A generic neural network class for creating a neural network of as many layers and layer sizes as required.

ma::Predictor: Uses the MAUtils `ma::Learner` derived learners for making predictions while continually learning from new temporal data. Currently supports the `ma::MCMCLearner`, the `ma::NetLearner` and the `ma::TDNetLearner`.

Appendix A.5.1 Network Training

ma::batchTrain: Trains an artificial neural network using supervised batch learning. Weights are modified to minimise the error between a set of target outputs and a set of calculated outputs from a set of inputs using conjugate gradient descent.

ma::NetLearner: Extends the MAUtils `ma::Learner` to use temporal sequence of inputs with a single output to sequentially train a neural network iteratively.

ma::TDNetLearner: Extends the MAUtils `ma::Learner` to use TD learning to train a neural network iteratively.

Appendix A.6 MADSP Library

A cross-platform C++ library for real-time digital signal processing. Filters can be created directly using the `ma::FilterFactory`, or graphically using a `ma::MultifilterPanel` or a `ma::MultifilterDialog`.

Appendix A.6.1 Filters

ma::Filter: The base abstract class that all filters are derived from.

ma::MultiFilter: Combines multiple filters into a single filter.

ma::SMAFilter: A simple moving average filter that averages the values for a period.

ma::BinaryTrigger: Sets the output to 1 if the input is above a threshold for a period.

ma::FrameTrigger: Sets the output to 1 if the input is above a threshold minimum threshold below a maximum threshold, for a minimum period, but not longer than a maximum period.

ma::LowPassFilter: A simple low-pass filter.

ma::HighPassFilter: A simple high-pass filter.

ma::EnvelopeFilter: An envelope filter using the magnitude of the analytic signal to obtain the instantaneous amplitude.

ma::DiffFilter: Direct Form II implementation of the standard difference equation: Includes constructors for creating any order low, high, band-pass or band-stop Butterworth filters.

Appendix A.7 MAIMU Library

A cross-platform C++ library for connecting to IMUs. Currently two types of IMU are supported: the x-IMU and the nuIMU.

Appendix A.8 MGame Library

A cross-platform C++ library for creating automated multi-player games.

ma::Game: Base class for creating automated multi-player games.

ma::GameAction: Base class for creating game actions that can be processed by Game.

ma::GameController: Game controller.

ma::GameFrame: Game control frame.

ma::GamePlayer: Game player.

ma::GamePolicy: Base class for creating game policies.

Appendix A.8.1 Reinforcement Learning Policies

ma::GreedyGamePolicy: Greedy policy.

ma::EGreedyGamePolicy: e-Greedy policy.

ma::ProbabilityGamePolicy: Probability policy.

Appendix A.9 MAIMUData Library

A cross-platform C++ library for working with data from multiple IMUs of different types.

Currently two types of IMU are supported: the x-IMU and the nuIMU.

Appendix A.9.1 IMU Data Collection

ma::IMUDataCollector: Automatically stores the required data received from the IMUs, which can be extracted when required.

ma::IMUConfig: A wxDialog that facilitates configuring the IMU Data Collector.

ma::IMULogger: When added to the IMU Data Collector, stores all data received from the IMUs, irrespective of the settings specifying what data is stored.

ma::IMUPlayback: Replays a logfile created with IMULogger into an IMU Data Collector to enable different data to be stored and processed.

Appendix A.10 Tic-Tac-Toe Application

A cross-platform reinforcement learning Tic-Tac-Toe game.

Appendix A.11 Backgammon Application

A cross-platform reinforcement learning Backgammon game.

Appendix A.12 nuIMU Data Viewer Application

A cross-platform C++ application for viewing nuIMU saved data files.

Appendix A.13 x-IMU GUI Application

A cross-platform C++ graphical application for managing the x-IMU.

Appendix A.14 nuIMU GUI Application

A cross-platform C++ graphical application for managing the nuIMU.

Appendix A.15 IMU Body Movement Application

A cross-platform C++ graphical application for visualising the tracking of body motion using IMUs.

Appendix A.16 Action Predictor GUI Application

A cross-platform C++ graphical application that takes input from multiple sensors and any normalised analogue input and learns a list of actions iteratively while being used.

Appendix B Overview of Machine Learning and Artificial Neural Networks

Appendix B.1 Introduction

There are three basic types of machine learning (Russell & Norvig, 2010):

- **Supervised Learning:** Supervised learning algorithms learn a mapping between a set of training inputs and their corresponding outputs. Supervised learning includes the regression of continuous output data and the classification of discrete output data used in decision making.

- **Unsupervised Learning:** Unsupervised learning algorithms find structure in data. Unsupervised learning includes finding data clusters and reducing data dimensionality.
- **Reinforcement Learning:** Reinforcement learning algorithms interact with a dynamic environment. The algorithms learn the best actions required to achieve a goal or maximise a reward through repeated experience.

Using supervised learning to learn linear and non-linear functions in regression and classification is covered in Sections B.2 and B.3 respectively, while reinforcement learning is covered in Section B.10. Although the extraction of features by artificial neural networks is a form of unsupervised learning (Yosinski *et al.*, 2014), finding structure in the data is not directly relevant to this research; therefore it is not covered.

Using complex models, such as an artificial neural network, with small datasets, as available in this research, will result in over-fitting (Babiyak, 2004). The use of regularisation to address this problem is covered in Section B.4, while the use of artificial neural networks in supervised learning and how they are trained is covered in Section B.5.

The traditional approach to training artificial neural networks using batch training, minimisation and the gradient descent algorithm are covered in Section B.6, while the TD learning approach and its benefits when presented with a sequential, temporal sequence of data – as will be the case in this research – is covered in Section B.7.

The need for adaptive control and the reason for the rise of the reinforcement learning approach, is covered in Section B.9; before the section on reinforcement learning itself: B.10.

Appendix B.2 Linear Regression and Classification

Supervised machine learning of continuous outputs, also known as regression, involves minimising a cost function. For a linear function:

$$f_w(\mathbf{x}) = w_0 x_0 + w_1 x_1 + \dots + w_n x_n \quad (\text{B.1})$$

with an n dimensional input $\mathbf{x}$ and weights $\mathbf{w}$, the cost function with respect to the weights is defined as the average of the root-mean-square errors:

$$J(\mathbf{w}) = \frac{1}{2m} \sum_{i=1}^m (f_w(\mathbf{x}^{(i)}) - y^{(i)})^2 = \frac{1}{2m} \sum_{i=1}^m (\mathbf{w}^T \mathbf{x}^{(i)} - y^{(i)})^2 \quad (\text{B.2})$$

where $y^{(i)}$ are the supervised target outputs for each of them inputs: $\mathbf{x}^{(i)}$ (Bishop, 2006).

The minimum is found when the gradient of the cost function with respect to the weights is zero:

$$\frac{\partial J(\mathbf{w})}{\partial w_j} = \frac{1}{m} \sum_{i=1}^m (\mathbf{w}^T \mathbf{x}^{(i)} - y^{(i)}) x_j^{(i)} = 0 \quad (\text{B.3})$$

If the m inputs $\mathbf{x}$ are stored in a matrix $\mathbf{X}$, and, if $\mathbf{X}^T \mathbf{X}$ is not singular (i.e. $n \leq m$ and the features are linearly independent) the minimum can be solved analytically:

$$\mathbf{w} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y} \quad (\text{B.4})$$

However, the inversion of matrix $\mathbf{X}$ is an $O(n^3)$ calculation; so for large n ($n > 10,000$) an iterative minimisation algorithm is used instead (Bishop, 2006).

Using linear regression and a threshold classifier on a classification problem is not a good idea, because outliers, which don't add new information, can significantly shift the threshold. Linear regression also returns values much greater and smaller than the actual options (0 or 1). Therefore, the output is modified using a logistic function:

$$g(f(x)) = \frac{1}{1+e^{-f(x)}} \quad (\text{B.5})$$

which has a simple derivative:

$$g'(f(x)) = g(f(x))(1-g(f(x))) \quad (\text{B.6})^{61}$$

Binary classification can then be done using a threshold of 0.5.

With the logistic function (and complex decision boundaries) the squared error cost can result in non-convex cost functions; so gradient descent won't work. Therefore, the error function is modified to the cross-entropy error (McClelland, 2015):

$$\delta(g(f(x)), y) = \begin{cases} -\log(g(f(x))) & \text{if } y=1 \\ -\log(1-g(f(x))) & \text{if } y=0 \end{cases} \quad (\text{B.7})^{62}$$

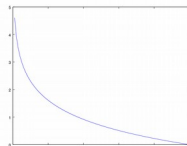
which can be represented using:

$$\delta(g(f(x)), y) = -y \log(g(f(x))) - (1-y) \log(1-g(f(x))) \quad (\text{B.8})$$

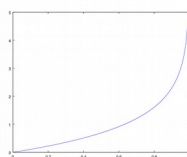
to give a revised cost function:

$$\begin{aligned} 61 \quad z'(x) &= \frac{e^{-x}}{(1+e^{-x})^2} \\ &= \frac{1+e^{-x}-1}{(1+e^{-x})^2} \\ &= \frac{1+e^{-x}}{(1+e^{-x})^2} - \frac{1}{(1+e^{-x})^2} \\ &= \frac{1}{1+e^{-x}} - \frac{1}{(1+e^{-x})^2} \\ &= \frac{1}{1+e^{-x}} \left(1 - \frac{1}{1+e^{-x}} \right) \\ &= z(x)(1-z(x)) \end{aligned}$$

62 $-\log(f(x))$:



$-\log(1-f(x))$:



$$J(\mathbf{w}) = -\frac{1}{m} \sum_{i=1}^m y^{(i)} \log(g(f(\mathbf{x}^{(i)}))) + (1 - y^{(i)}) \log(1 - g(f(\mathbf{x}^{(i)}))) \quad (\text{B.9})$$

with derivative:

$$\frac{\partial J(\mathbf{w})}{\partial w_j} = \frac{1}{m} \sum_{i=1}^m (g(f(\mathbf{x}^{(i)})) - y^{(i)}) x_j^{(i)} \quad (\text{B.10})^{63}$$

which is exactly the same as the gradient of the cost function for the linear regression function; so the minimum can be found the same way (Bishop, 2006).

To classify more than two options, multiple one-vs-all (a.k.a. one-vs-rest) comparisons are used:

$$f_w^{(k)}(\mathbf{x}) = P(k = y | \mathbf{x}; \mathbf{w}) \quad (\text{B.11})$$

where k is each of the options, and the maximum probability: $f_w^{(k)}(\mathbf{x})$ is picked (Bishop, 2006).

Appendix B.3 Non-Linear Regression and Classification

A similar approach can be used for a non-linear function $f_w(\mathbf{x})$. The root-mean-square cost function with respect to the weights is:

$$J(\mathbf{w}) = \frac{1}{2m} \sum_{i=1}^m (f_w(\mathbf{x}^{(i)}) - y^{(i)})^2 \quad (\text{B.12})$$

and its gradient with respect to the weights is:

$$\nabla J(\mathbf{w}) = \frac{1}{m} \sum_{i=1}^m (f_w(\mathbf{x}^{(i)}) - y^{(i)}) \nabla f_w(\mathbf{x}^{(i)}) \quad (\text{B.13})$$

⁶³
$$\begin{aligned} \frac{\partial J(\mathbf{w})}{\partial w_j} &= -\frac{1}{m} \sum_{i=1}^m y^{(i)} \frac{g(f(\mathbf{x}^{(i)}))(1 - g(f(\mathbf{x}^{(i)}))) x_j^{(i)}}{g(f(\mathbf{x}^{(i)}))} + (1 - y^{(i)}) \frac{-g(f(\mathbf{x}^{(i)}))(1 - g(f(\mathbf{x}^{(i)}))) x_j^{(i)}}{1 - g(f(\mathbf{x}^{(i)}))} \\ &= -\frac{1}{m} \sum_{i=1}^m y^{(i)} (1 - g(f(\mathbf{x}^{(i)}))) x_j^{(i)} + (1 - y^{(i)}) (-g(f(\mathbf{x}^{(i)}))) x_j^{(i)} \\ &= -\frac{1}{m} \sum_{i=1}^m (y^{(i)} - g(f(\mathbf{x}^{(i)}))) x_j^{(i)} \\ &= \frac{1}{m} \sum_{i=1}^m (g(f(\mathbf{x}^{(i)})) - y^{(i)}) x_j^{(i)} \end{aligned}$$

Therefore, as long as the gradient of the function at each input $\mathbf{x}$ is known, a local minimum can be found in the same iterative way (Bishop, 2006).

Appendix B.4 Regularisation

If a function has many features (relative to the data), the learned hypothesis may fit the training data too well, but fail to generalise to new examples – it over-fits the data. There are three ways to address over-fitting: increase the number of data samples; reduce the number of features; or use regularisation. Regularisation reduces the magnitude of parameters, because smaller parameter values result in a “simpler” hypothesis, which is less prone to over-fitting (Bishop, 2006).

Regularisation penalises larger parameters by adding a term to the cost function:

$$J(\mathbf{w}) = \frac{1}{2m} \left(\sum_{i=1}^m (f_{\mathbf{w}}(\mathbf{x}^{(i)}) - y^{(i)})^2 + \gamma \sum_{j=1}^n w_j^2 \right) \quad (\text{B.14})$$

where γ is the regularisation factor. Its partial derivatives with respect to the weights are:

$$\frac{\partial J(\mathbf{w})}{\partial w_j} = \frac{1}{m} \sum_{i=1}^m (f_{\mathbf{w}}(\mathbf{x}^{(i)}) - y^{(i)}) \frac{\partial}{\partial w_j} f_{\mathbf{w}}(\mathbf{x}^{(i)}) + \frac{\gamma}{m} w_j \quad (\text{B.15})$$

Typically w_0 is not penalised, but this is only by convention and has very little effect (Bishop, 2006).

Regularisation requires specifying γ . If γ is too large, minimisation will drive $w_j \rightarrow 0$, which results in under-fitting (Bishop, 2006).

Appendix B.5 Artificial Neural Networks

Artificial Neural Networks (ANN) were first described as long ago as 1943 (McCulloch & Pitts, 1943), but it was Rosenblatt's perceptron in 1957 that was first used for pattern recognition (Rosenblatt, 1958).

The perceptron algorithm learns a linear function for binary classification:

$$p(x) = \begin{cases} 1 & \text{if } \mathbf{w}^T \mathbf{x} + b > 0 \\ 0 & \text{otherwise} \end{cases} \quad (\text{B.16})$$

where $\mathbf{w}$ is a weight vector and b is the bias. The weights express the importance of the respective inputs (Nielsen, 2015). The learning rule involves repeatedly applying all the training inputs. For each incorrect result the following updates are made: If the correct result is 1, the input $\mathbf{x}^{(i)}$ vector is added to $\mathbf{w}$ and the bias is incremented. If the correct result is 0, the input $\mathbf{x}$ is subtracted from $\mathbf{w}$ and the bias is decremented (Bishop, 2006).

The power of perceptrons is revealed by noticing that a perceptron with two inputs can act as a NAND gate: set each weight to -2 , and the bias to 3 . Since a NAND gate is universal for computation, networks of perceptrons can be used to compute any logical function (Nielsen, 2015).

A multi-layer perceptron takes the output of multiple perceptrons as the input to the next layer of multiple perceptrons:

$$\mathbf{p}^{(l)} = \mathbf{W}^{(l)} \cdot \mathbf{p}^{(l-1)} \quad (\text{B.17})$$

where $\mathbf{W}^{(l)}$ is a matrix of the l layer's input weights for each perceptron, and $\mathbf{p}^{(0)} = \mathbf{x}$, and each layer is calculated iteratively. The layers between the input and final output are referred to as hidden layers, because they don't interact with the environment at the input or the output (McClelland, 2015). Multi-layer perceptrons also allow lower layers to be reused to calculate multiple outputs for classifying more than two options (Bishop, 2006).

However, in 1969, Marvin Minsky showed that perceptrons were only capable of learning linearly separable patterns (Minsky & Papert, 1969). Furthermore, linear algebra can be used to prove that multi-layer perceptrons with linear activation functions can be reduced to a single layer perceptron. In addition, perceptron learning requires a small change in a weight or bias to cause only a small change in the network output; so the weights and biases can be modified to alter the network to provide incremental improvements in the predicted output. However, due to the binary nature of perceptrons, a small change in the weights or bias of any single perceptron in the network can sometimes cause the output of that perceptron to completely flip, from 0 to 1, or visa versa (Nielsen, 2015).

Both of these problems are resolved by using non-linear activation functions. Multi-layer perceptrons with non-linear activation functions, can be used to model any non-linear function, and hence can classify data that is non-linearly separable. Two common non-linear activation functions are the logistic function:

$$g(p_j) = \frac{1}{1 + e^{-p_j}} \quad (\text{B.18})$$

which converts each perceptron's output to a sigmoid function between 0 and 1, and the hyperbolic tangent function:

$$g(p_j) = \tanh(p_j) = \frac{e^{2p_j} - 1}{e^{2p_j} + 1} \quad (\text{B.19})^{64}$$

⁶⁴ $\tanh(x) = \frac{\sinh(x)}{\cosh(x)}$
 $= \frac{\frac{1}{2}(e^x - e^{-x})}{\frac{1}{2}(e^x + e^{-x})}$
 $= \frac{e^x(e^x - e^{-x})}{e^x(e^x + e^{-x})}$
 $= \frac{e^{2x} - 1}{e^{2x} + 1}$

which converts each output to a sigmoid function between -1 and 1 (Bishop, 2006), while remaining approximately linear for small values close to zero (Tesauro, 1998).

These activation functions have simple derivatives:

$$g'(p_j) = g(p_j)(1 - g(p_j)) \quad (\text{B.20})^{65}$$

and

$$g'(p_j) = 1 - (g(p_j))^2 \quad (\text{B.21})^{66}$$

respectively. These smooth functions also eliminate the binary flip problem associated with the original perceptrons. Multi-layer perceptrons with non-linear activation functions now form the basis of various artificial neural networks, with deep neural networks being artificial neural networks with many hidden layers (Nielsen, 2015). Although, Glorot and Bengio have shown that using the sigmoid activation function as the top hidden layer of a deep supervised network without unsupervised pre-training can result in severe optimisation errors (Glorot & Bengio, 2010).

Perceptron learning worked by ensuring that every time the weights change, they get closer to every feasible set of weights. With non-linear functions this doesn't work, because averaging two good solutions may be a bad solution. Instead, learning requires that the weights are slowly varied in a way that the predicted output values get closer to the target values. This is done by minimising a cost function with respect to the weights using a minimisation method (Bishop, 2006).

⁶⁵ Derived above in footnote 61.

⁶⁶
$$\begin{aligned} g'(x) &= \frac{d}{dx} \tanh(x) \\ &= \frac{d}{dx} \frac{\sinh(x)}{\cosh(x)} \\ &= \frac{\cosh(x) \frac{d}{dx} \sinh(x) - \sinh(x) \frac{d}{dx} \cosh(x)}{\cosh^2(x)} \\ &= \frac{\cosh(x) \cosh(x) - \sinh(x) \sinh(x)}{\cosh^2(x)} \\ &= 1 - \frac{\sinh^2(x)}{\cosh^2(x)} \\ &= 1 - \tanh^2(x) \end{aligned}$$

Minimisation methods require the gradient of the cost function with respect to the weights. For a multi-layer network, the partial derivatives of the cost function with respect to the weights in the output layer are calculated as above. The derivatives of the cost function with respect to the weights in the lower layers was initially done by perturbing each weight and measuring the change in the error. In 1974, Paul Werbos developed the back-propagation of errors algorithm to calculate the errors of the lower layers from the errors of the layer above (Werbos, 1974). From the errors, the derivatives can be calculated directly. However, it wasn't until Rumelhart, Hinton and Williams' 1985 paper (Rumelhart, Hinton & Williams, 1985), that the importance of the back-propagation algorithm in simplifying the calculation of the cost function derivatives was highlighted. The back-propagation algorithm reduces the processing requirements from $O(n \times m)$ to $O(m)$, where n is the number of weights and m is the number of samples (Nielsen, 2015).

The error back-propagation algorithm defines a layer's errors $\delta^{(l)}$ in terms of the error in the layer above, $\delta^{(l+1)}$. The error $\delta_j^{(l)}$ of neuron j in layer l is defined as:

$$\delta_j^{(l)} \equiv \frac{\partial J}{\partial p_j^{(l)}} \quad (\text{B.22})$$

i.e. the partial derivative of the cost function J with respect to the output of the neuron (before the activation function). Using the chain rule this is expanded to:

$$\delta_j^{(l)} = \frac{\partial J}{\partial p_j^{(l)}} = \sum_{k=1}^{s^{(l+1)}} \frac{\partial J}{\partial p_k^{(l+1)}} \frac{\partial p_k^{(l+1)}}{\partial p_j^{(l)}} \quad (\text{B.23})$$

which is just the sum of the errors in the layer above multiplied by the derivative of the output of the layer above with respect to the output of this layer. With

$$p_k^{(l+1)} = \sum_{j=1}^{s^{(l)}} w_{kj}^{(l+1)} g_j^{(l)}(p_j^{(l)}) + b_k^{(l+1)} \quad (\text{B.24})$$

the derivative with respect to each output in this layer is:

$$\frac{\partial p_k^{(l+1)}}{\partial p_j^{(l)}} = w_{kj}^{(l+1)} \frac{\partial g_j^{(l)}}{\partial p_j^{(l)}} \quad (\text{B.25})^{67}$$

making

$$\delta_j^{(l)} = \sum_{k=1}^{s^{(l+1)}} \delta_k^{(l+1)} w_{kj}^{(l+1)} \frac{\partial g_j^{(l)}}{\partial p_j^{(l)}} \quad (\text{B.26})$$

Therefore, beginning with the output error:

$$\delta^{(L)} = \frac{\partial J}{\partial \mathbf{p}^{(L)}} \quad (\text{B.27})$$

calculated from the cost function, the error propagated down is:

$$\delta^{(l)} = \sum_{k=1}^{s^{(l+1)}} \left[\delta_k^{(l+1)} w_{kj}^{(l+1)} \right] \frac{\partial g_j^{(l)}}{\partial p_j^{(l)}} \quad (\text{B.28})$$

with the input error defined as zero: $\delta^{(0)} = \mathbf{0}$ (Nielsen, 2015).

The partial derivatives of the cost function with respect to each of the weights in each layer $\frac{\partial J}{\partial w_{kj}^{(l)}}$

is expanded using the chain rule to:

$$\frac{\partial J}{\partial w_{kj}^{(l)}} = \frac{\partial J}{\partial p_k^{(l)}} \frac{\partial p_k^{(l)}}{\partial w_{kj}^{(l)}} \quad (\text{B.29})$$

With

$$p_k^{(l)} = \sum_{j=1}^{s^{(l-1)}} w_{kj}^{(l)} g_j^{(l-1)} + b_k^{(l)} \quad (\text{B.30})$$

the derivative with respect to the weights is:

⁶⁷ $\frac{\partial g}{\partial p}$ is the derivative of the neuron's activation function and depends on the activation function used.

$$\frac{\partial p_k^{(l)}}{\partial w_{kj}^{(l)}} = g_j^{(l-1)} \quad (\text{B.31})$$

i.e. simply the output of the lower layer; and $\frac{\partial J}{\partial p_k^{(l)}}$ is defined above as the error $\delta_k^{(l)}$; so this is

simplified to:

$$\frac{\partial J}{\partial w_{kj}^{(l)}} = \delta_k^{(l)} g_j^{(l-1)} \quad (\text{B.32})$$

Similarly the partial derivatives of the cost function with respect each of the bias weights in each layer is:

$$\frac{\partial J}{\partial b_k^{(l)}} = \delta_k^{(l)} \quad (\text{B.33})$$

Finally, using the modified classification error function from above, the neural network regularised cost function is:

$$J(\mathbf{w}) = -\frac{1}{m} \sum_{i=1}^m \sum_{k=1}^{s^{(L)}} \left(y_k^{(i)} \log(f_w(\mathbf{x}^{(i)})_k) + (1 - y_k^{(i)}) \log(1 - (f_w(\mathbf{x}^{(i)})_k)) \right) + \frac{\gamma}{2m} \sum_{l=1}^{L-1} \sum_{j=1}^{s^{(l)}} \sum_{k=2}^{s^{(l+1)}} (w_{kj}^{(l)})^2 \quad (\text{B.34})$$

where $f_w(\mathbf{x}^{(i)})_k$ is the k^{th} output of the network for input $\mathbf{x}^{(i)}$, $y_k^{(i)}$ is the target output, m is the number of samples, L is the number of layers, and $s^{(l)}$ is the number of units in layer l . In the first half the subscript represents the output unit, and the superscript represents the sample. In the second half the subscripts represent the node connected to and from respectively, and the superscript represents the layer. The regularisation term is the squared sum of all the weights in the network excluding the weights from the bias.

The partial derivatives of the regularised cost function with respect to the neuron outputs is:

$$\frac{\partial J}{\partial w_{kj}^{(L)}} = \frac{1}{m} \sum_{i=1}^m \left(f_w(\mathbf{x}^{(i)})_k - y_k^{(i)} \right) \frac{\partial p_k}{\partial w_{kj}} + \frac{\gamma}{m} w_{kj}^{(L)} \quad (\text{B.35})$$

which defines

$$\delta_k^{(L)} = f_w(\mathbf{x}^{(i)})_k - y_k^{(i)} \quad (\text{B.36})$$

Using the error back-propagation, the partial derivatives of the regularised cost function with respect to the weights for lower layers is calculated iteratively:

$$\frac{\partial J}{\partial w_{kj}^{(l)}} = \frac{1}{m} \sum_{i=1}^m \left(\sum_{h=1}^{s_{l+1}} [\delta_h^{(l+1)} w_{hk}] \frac{\partial g_k}{\partial p_k} \right) \frac{\partial p_k}{\partial w_{kj}^{(l)}} + \frac{\gamma}{m} w_{kj}^{(l)} \quad (\text{B.37})$$

These can be combined and simplified using the error term and the output of the lower layer:

$$\frac{\partial J}{\partial w_{kj}^{(l)}} = \frac{1}{m} \sum_{i=1}^m \delta_k^{(l)} g_j^{(l-1)} + \frac{\gamma}{m} w_{kj}^{(l)} \quad (\text{B.38})$$

(Nielsen, 2015) (Bishop, 2006)

Some important points to note with artificial neural networks include: When the output from the

activation from the neuron in the lower layer i.e. $g^{(l-1)}$ is small, the gradient term $\frac{\partial J}{\partial w}$ will also tend to be small, and will not change much during gradient descent – weights output from low-activation neurons learn slowly (Nielsen, 2015). $J(\mathbf{w})$ may not be convex – it may have many local minima – so gradient descent may not find the global minimum. However, in networks with many hidden units, local minima appear rare (McClelland, 2015). Furthermore, it may have many saddle points surrounded by very flat regions, which makes optimisation difficult (Goodfellow, Bengio & Courville, 2016).

By the 1980s, it was expected that back-propagation of multi-layered neural networks would be a very effective mechanism for pattern recognition, provided that computers were fast enough and data sets were big enough (to avoid over-fitting). However, there is a problem with the back-propagation algorithm in neural networks with many hidden layers: When calculating the output,

the non-linear sigmoid output functions at each layer prevent the outputs from each layer from increasing exponentially, but when back propagating the error the functions remain linear. If the weights are small, the gradients shrink exponentially. If the weights are big, the gradients grow exponentially. This can be avoided by choosing the initial weights carefully, but choosing the right initial weights remained an unsolved problem for decades (Bengio, 2009).

Except for special convolutional neural networks (Bengio, 2009), deep multi-layer neural networks were not successfully trained until 2006 (Glorot & Bengio, 2010). In 2006, Hinton and Salakhutdinov showed how a stack of restricted Boltzmann machines (RBMs) each having only one layer of feature detectors can be used to obtain good initial weights. Each layer is trained independently, and then the learned feature activations of one RBM are used as the input for training the next RBM in the stack. Finally, the RBMs are connected together to create a multi-layer neural network, which is then fine-tuned using back-propagation (Hinton & Salakhutdinov, 2006). This work was extended by Ranzato et al. and Bengio et al. to develop the auto-encoders used to train many deep neural networks (Ranzato *et al.*, 2006) (Bengio *et al.*, 2007) for classification, regression, dimensionality reduction, modelling textures, modelling motion, object segmentation, information retrieval, robotics, natural language processing, and collaborative filtering (Bengio, 2009).

Deep neural networks are important for a number of reasons. First, they allow feature hierarchies that are learnt, where features from higher levels of the hierarchy being a combination of the features from lower levels (Glorot & Bengio, 2010), and the lower features can then be reused in other classification tasks (Raina *et al.*, 2007). Second, a deep neural network actually reduces the number of weights and neurons required, with a neural network with too few layers for the task potentially requiring exponentially more (Bengio, 2009).

Appendix B.6 Minimisation

Iterative minimisation using the gradient method was developed by Cauchy in 1847 (Lemaréchal, 2012). Cauchy suggested iteratively improving an initial estimate by either taking a step in the direction of the negative gradient (the gradient method) or actually finding the minimum in the direction of the negative gradient (steepest descent). By 1962, both these methods had been shown to converge under certain conditions, and the gradient method was considered the more effective (Goldstein, 1962). However, it must be remembered that gradient-descent methods can only find local minima, which may not be the global minimum (Tesauro, 1992).

The gradient method uses a fixed step-size α of the negative gradient at the current estimate of the root-mean-square error function to update the estimate and converge to a minimum (Widrow & Hoff, 1960). Therefore, the cost function can be minimised with respect to the weights using the iterative algorithm:

$$\mathbf{w} := \mathbf{w} - \alpha \nabla J(\mathbf{w}) \quad (\text{B.39})$$

However, the gradient method often converges very slowly, because, not only is the right step-size α dependent on the function to be minimised, the negative gradient direction is not necessarily towards the minimum (Yuan, 2008).

Line search is the iterative approach for finding a step size for each iteration of the gradient method that satisfies convergence criteria. There are a number of ways to perform a line search. The simplest is the backtracking line search. It begins by taking an initial step size that is sufficiently large i.e. passes the minimum (increasing it iteratively if necessary), and then iteratively decreasing (backtracking) it until it reduces the function sufficiently (Armijo, 1966).

The gradient does not provide the direction that maximises the improvement in the error, because the curvature – the rate of change of the gradient – is normally not constant i.e. it's not a quadratic

error surface. Maximum error reduction depends on the ratio of the gradient to the curvature. A good direction to move in is one with a high ratio of gradient to curvature, even if the gradient itself is small. A curvature matrix specifies how the gradient changes in each direction i.e. how the change in a weight is affected by the changes in the other weights. The error surface curvature is made constant by multiplying the gradient vector by the inverse of the curvature matrix. Not only is the gradient then in the right direction, but the minimum can be found in a single line searched step. However, inverting a high-dimensional (curvature) matrix is generally infeasible (Wright & Nocedal, 1999).

Instead of inverting the curvature matrix (and then minimising the error in one step) Hestenes and Stiefel independently developed the conjugate gradient descent method in 1952 (Hestenes & Stiefel, 1952). It provides a simple iterative solution; solving n linear equations in n steps, and works by finding successive, quadratic minima in each conjugate direction.

The first non-linear conjugate gradient method was introduced by Fletcher and Reeves in 1964 (Fletcher & Reeves, 1964). The non-linear conjugate gradient method generalises the conjugate gradient method to find a local minimum of non-linear functions using their gradient alone. It works by assuming the function is approximately quadratic at the minimum. A line search is done to find the approximate minimum in the direction been taken, and modifications are required to find the subsequent conjugate direction. However, as the minimum is only an approximate, the step length along the line search needs to satisfy the strong Wolfe conditions to ensure that subsequent conjugate directions are descent directions (Wright & Nocedal, 1999).

The Wolfe-Powell conditions are the conditions currently used for ensuring convergence. The original Wolfe conditions were developed in 1969 (Wolfe, 1969) (Wolfe, 1971). They ensure that each iteration both decreases the function and reduces the curvature sufficiently. Ensuring the function decreases sufficiently is achieved using Armijo's rule. Ensuring the curvature is reduced

sufficiently is required to ensure convergence when the search direction $\mathbf{s}$ is not in a descent direction:

$$1. f(\mathbf{x}_k + \alpha_k \mathbf{s}_k) \leq f(\mathbf{x}_k) + c_1 \alpha_k \mathbf{s}_k^T \nabla f(\mathbf{x}_k) \quad (\text{B.40})$$

$$2. \mathbf{s}_k^T \nabla f(\mathbf{x}_k + \alpha_k \mathbf{s}_k) \geq c_2 \mathbf{s}_k^T \nabla f(\mathbf{x}_k) \quad (\text{B.41})$$

where, for each iteration k , $f(x)$ is the function that needs to be minimised, α_k is the step length that needs to satisfy the inequality, c_1 is the constant that defines the reduction in the function required, and c_2 is the constant that defines the reduction in the curvature required.

In 1969, Polak and Ribière proposed a modified method for selecting subsequent conjugate directions (Polak & Ribière, 1969). The Polak and Ribière method tends to be more robust and efficient than other methods for selecting conjugate gradients. However, the strong Wolfe conditions are not sufficient to ensure a negative descent direction. To overcome this, the Wolfe conditions were adapted to detect this. The method was modified to reset and deliberately take the steepest descent when the following condition fails:

$$|\mathbf{s}_k^T \nabla f(\mathbf{x}_k + \alpha_k \mathbf{s}_k)| \leq c_2 |\mathbf{s}_k^T \nabla f(\mathbf{x}_k)| \quad (\text{B.42})$$

This modification also ensures that the step is as close to a the line search minimum (Wright & Nocedal, 1999).

In 1975, Powell developed a bracketing method for calculating α , which requires $0 < c_1 < c_2 < 1$, and by setting $c_1 < \frac{1}{2}$ ensures a sufficient rate of convergence (Powell, 1976).

There is a second type of multi-dimensional minimisation algorithms: the quasi-Newton or variable metric methods. They are typified by the Davidon-Fletcher-Powell (or simply Fletcher-Powell) and Broyden-Fletcher-Goldfarb-Shanno algorithms. However, whereas the conjugate gradient methods require linear (in the dimensions) storage, the quasi-Newton methods require quadratic storage (Press, 2007).

The main advantage of gradient descent algorithms is that convergence does not depend on the size of the training set, only on the number of updates and the quality of the training data (Bengio, 2012b). However, all minimisation methods depend on the function's gradients being available.

In practice, to compute the gradient $\nabla J(\mathbf{w})$ the gradients $\nabla J(\mathbf{w})_x$ need to be computed separately for each training input, $\mathbf{x}$, and then averaged:

$$\nabla J(\mathbf{w}) = \frac{1}{n} \sum_x \nabla J(\mathbf{w})_x \quad (\text{B.43})$$

which, for large training sets, would be too computationally intensive. Therefore, *stochastic gradient descent* can be used to speed up learning. The gradient $\nabla J(\mathbf{w})$ is estimated by computing $\nabla J(\mathbf{w})_x$ for a small sample of randomly chosen training inputs and averaging over them. The estimate won't be perfect, but it doesn't need to be perfect: as long as the computed gradient will help decrease $J(\mathbf{w})$. An extreme version of stochastic gradient descent is to use a sample size of just 1, which is also known as online learning (Nielsen, 2015).

It is important to remember that even the true gradient direction averaged over the whole training set is only locally the steepest descent direction. The right direction for larger steps is generally different. Therefore, there is no value in accurately computing the direction of steepest descent precisely (Bengio, 2012b).

Doing more updates more frequently helps to explore more and faster; especially with large learning rates. From a statistical point of view, samples should be independent and identically distributed. However, during iterative learning they come from an unknown stochastic process that has serial correlation. With large datasets the most efficient approach is to update the parameters of the model as each sample arrives. Although the approach itself may not be optimal, it can be shown that Iterative Learners that use non-repetitive training data also minimise the generalisation error (Bengio, 2012b).

Bengio uses mini-batch updates, where the weight updates are made only after a batch of samples. The updates are based on the average of the gradients. Using the average results in a smaller change after each small batch of samples, but only has a small effect on the optimal learning rate. Using the average also reduces the variance of the gradient. Therefore, larger batches can allow for a small increase in the learning rate (Bengio, 2012b).

Appendix B.7 TD Learning

TD learning is a technique for learning to predict an expected value of a variable occurring at the end of a sequence of states. The earliest known use of a TD method is in Samuel's checker-playing program (Samuel, 1959). However, they remained poorly understood until 1988, when Sutton proved their convergence and optimality for special cases and related them to supervised-learning methods (Sutton, 1988).

Conventional supervised-learning approaches disregard the temporal structure of the input data (Sutton, 1988). TD methods can be used for any pattern recognition problem in which data is gathered over time. It combines Monte-Carlo ideas with dynamic programming ideas, but with a number of advantages: Unlike Monte-Carlo methods that require complete episodes, it can be updated iteratively, which is crucial for continuous tasks. It can learn from each action, unlike Monte-Carlo methods, where some episodes must be ignored or discounted when experimental actions are taken. Finally, unlike dynamic programming, it does not require a model of the environment (Sutton & Barto, 1998).

It is ideal for learning predictions. Where conventional prediction-learning methods use the difference between predicted and actual outcomes, TD learning uses the difference between temporally successive predictions. Furthermore, it uses the training examples from the temporal sequence of ordinary sensory input – no separate supervisor is needed (Sutton, 1988).

TD methods have two advantages over conventional prediction-learning methods: First, they are incremental, which not only allows each update to be computed with each step (which makes it effective for online learning) it also requires a lot less memory. Conventional methods wait until the actual final outcome is received, which requires each step to be saved, and then all the steps' updates need to be computed, which can limit their ability to be used online. Second, they tend to make more efficient use of their experience: they converge faster and produce better predictions. To be effective, TD methods only require that the system predicted be a dynamic one: it has a state which can be observed evolving over time (Sutton, 1988).

Most supervised learning problems are actually prediction problems, and many problems that are treated as single-step prediction problems can be better treated as multi-step problems. Therefore, TD learning should be considered a simpler way for efficiently learning to predict arbitrary events, not just goal-related ones (Sutton, 1988).

Using a sequence of observational inputs $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \mathbf{x}^{(3)} \dots, \mathbf{x}^{(m)}$, and the final outcome y , where each $\mathbf{x}_t$ is a vector describing the observation at time t , the gradient method learning update procedure for minimising the cost function with respect to the weights $\mathbf{w}$ is:

$$\mathbf{w} := \mathbf{w} - \alpha \nabla J(\mathbf{w}) \quad (\text{B.44})$$

where

$$\nabla J(\mathbf{w}) = \frac{1}{m} \sum_{t=1}^m (f_{\mathbf{w}}(\mathbf{x}^{(t)}) - y^{(t)}) \nabla f_{\mathbf{w}}(\mathbf{x}^{(t)}) \quad (\text{B.45})$$

Defining $p^{(1)}, p^{(2)}, p^{(3)} \dots, p^{(m)}$ as the sequence of predictions $p^{(t)} = f_{\mathbf{w}}(\mathbf{x}^{(t)})$ ⁶⁸ and $\nabla_{\mathbf{w}} p^{(t)} = \nabla f_{\mathbf{w}}(\mathbf{x}^{(t)})$ at

each time t , and use the fact that $p^{(t)} - y = \sum_{h=t}^m (p^{(h)} - p^{(h+1)})$ where $p^{(m+1)} \equiv y$ then:

⁶⁸ Each $p^{(t)}$ can be a function of all preceding observation vectors.

$$\nabla J(\mathbf{w}) = \frac{1}{m} \sum_{t=1}^m \sum_{h=t}^m (p^{(h)} - p^{(h+1)}) \nabla_{\mathbf{w}} p^{(t)} \quad (\text{B.46})$$

$$= \frac{1}{m} \sum_{h=1}^m \sum_{t=1}^h (p^{(h)} - p^{(h+1)}) \nabla_{\mathbf{w}} p^{(t)} \quad (\text{B.47})^{69}$$

$$= \frac{1}{m} \sum_{t=1}^m (p^{(t)} - p^{(t+1)}) \sum_{h=1}^t \nabla_{\mathbf{w}} p^{(h)} \quad (\text{B.48})$$

Now the update procedure becomes:

$$\mathbf{w} := \mathbf{w} - \frac{\alpha}{m} \sum_{t=1}^m (p^{(t)} - p^{(t+1)}) \sum_{h=1}^t \nabla_{\mathbf{w}} p^{(h)} \quad (\text{B.49})$$

which can be processed iteratively at each time step:

$$\mathbf{w} := \mathbf{w} - \alpha (p^{(t)} - p^{(t+1)}) \sum_{h=1}^t \nabla_{\mathbf{w}} p^{(h)} \quad (\text{B.50})^{70}$$

(Sutton, 1988)

Two successive predictions are needed to form the error term. $\mathbf{x}^{(t+1)}$ is required for prediction $p^{(t+1)}$, before the update for prediction $p^{(t)}$ from $\mathbf{x}^{(t)}$ can be made. The error at time $t+1$ also provides weight changes for updating all of the previous inputs' predictions. This is done simultaneously by

69 Equation (B.47) is not obviously equal to equation (B.46), but their equality is verified by expanding the sums and simplifying:

$$\begin{aligned} \frac{1}{m} \sum_{t=1}^m \sum_{h=t}^m (p^{(h)} - p^{(h+1)}) \nabla_{\mathbf{w}} p^{(t)} &= \frac{1}{m} \left(\sum_{h=1}^m (p^{(h)} - p^{(h+1)}) \nabla_{\mathbf{w}} p^{(1)} + \sum_{h=2}^m (p^{(h)} - p^{(h+1)}) \nabla_{\mathbf{w}} p^{(2)} + \dots + \sum_{h=3}^m (p^{(h)} - p^{(h+1)}) \nabla_{\mathbf{w}} p^{(m)} \right) \\ &= \frac{1}{m} \left((p^{(1)} - p^{(2)}) \nabla_{\mathbf{w}} p^{(1)} + (p^{(2)} - p^{(3)}) \nabla_{\mathbf{w}} p^{(1)} + \dots + (p^{(m)} - p^{(m+1)}) \nabla_{\mathbf{w}} p^{(1)} + \right. \\ &\quad \left. (p^{(2)} - p^{(3)}) \nabla_{\mathbf{w}} p^{(2)} + \dots + (p^{(m)} - p^{(m+1)}) \nabla_{\mathbf{w}} p^{(2)} + \dots + (p^{(m)} - p^{(m+1)}) \nabla_{\mathbf{w}} p^{(m)} \right) \\ \frac{1}{m} \sum_{h=1}^m \sum_{t=1}^h (p^{(h)} - p^{(h+1)}) \nabla_{\mathbf{w}} p^{(t)} &= \frac{1}{m} \left(\sum_{t=1}^1 (p^{(1)} - p^{(2)}) \nabla_{\mathbf{w}} p^{(t)} + \sum_{t=1}^2 (p^{(2)} - p^{(3)}) \nabla_{\mathbf{w}} p^{(t)} + \dots + \sum_{t=1}^m (p^{(m)} - p^{(m+1)}) \nabla_{\mathbf{w}} p^{(t)} \right) \\ &= \frac{1}{m} \left((p^{(1)} - p^{(2)}) \nabla_{\mathbf{w}} p^{(1)} + (p^{(2)} - p^{(3)}) \nabla_{\mathbf{w}} p^{(1)} + (p^{(2)} - p^{(3)}) \nabla_{\mathbf{w}} p^{(2)} + \dots + \right. \\ &\quad \left. (p^{(m)} - p^{(m+1)}) \nabla_{\mathbf{w}} p^{(1)} + (p^{(m)} - p^{(m+1)}) \nabla_{\mathbf{w}} p^{(2)} + \dots + (p^{(m)} - p^{(m+1)}) \nabla_{\mathbf{w}} p^{(m)} \right) \\ &= \frac{1}{m} \left((p^{(1)} - p^{(2)}) \nabla_{\mathbf{w}} p^{(1)} + (p^{(2)} - p^{(3)}) \nabla_{\mathbf{w}} p^{(1)} + \dots + (p^{(m)} - p^{(m+1)}) \nabla_{\mathbf{w}} p^{(1)} + \right. \\ &\quad \left. (p^{(2)} - p^{(3)}) \nabla_{\mathbf{w}} p^{(2)} + \dots + (p^{(m)} - p^{(m+1)}) \nabla_{\mathbf{w}} p^{(2)} + \dots + (p^{(m)} - p^{(m+1)}) \nabla_{\mathbf{w}} p^{(m)} \right) \end{aligned}$$

70 The α in equation (B.50) is the α in equation (B.49) divided by m . However, since m may be unknown and α is a parameter that needs to be determined anyway, for readability, the same variable name is used.

maintaining the running sum of the gradient $\nabla_w p^{(h)}$. All preceding predictions are updated to make each closer to the prediction for the current state $p^{(t+1)}$ (McClelland, 2015).

The TD learning algorithm will only improve the current prediction if the next prediction is a better estimate of the target value than the current prediction. If there is high volatility and the next input is not well correlated with the current input, the next prediction will be a poor estimate of the target value and may be worse than the current prediction (Tesauro, 1992).

TD procedures can also be altered by increasing the weight of more recent predictions. Exponential weighting is typically used, where alterations to the predictions of observation vectors occurring k steps in the past are discounted using λ^k for $0 < \lambda < 1$:

$$\mathbf{w} := \mathbf{w} - \alpha (p^{(t)} - p^{(t+1)}) \sum_{h=1}^t \lambda^{t-h} \nabla_w p^{(h)} \quad (\text{B.51})$$

The original TD learning algorithm has $\lambda = 1$, and is known as TD(1). With $\lambda = 0$:

$$\mathbf{w} := \mathbf{w} - \alpha (p^{(t)} - p^{(t+1)}) \nabla_w p^{(t)} \quad (\text{B.52})$$

i.e. the weight increment is only affected by the prediction associated with the most recent observation. The exponential form has the added advantage that the sum can be computed iteratively:

$$\sum_{h=1}^{t+1} \lambda^{t+1-h} \nabla_w p^{(h)} = \nabla_w p^{(t+1)} + \lambda \sum_{h=1}^t \lambda^{t-h} \nabla_w p^{(h)} \quad (\text{B.53})$$

(Sutton, 1988)

Defining the *eligibility trace* at time t as:

$$\mathbf{e}^{(t)} = \sum_{h=1}^t \lambda^{t-h} \nabla_w p^{(h)} \quad (\text{B.54})$$

With the *error* at time t is defined as:

$$\delta^{(t)} = p^{(t)} - p^{(t+1)} \quad (\text{B.55})$$

the definition for $\Delta \mathbf{w}^{(t)}$ is simplified to:

$$\Delta \mathbf{w}^{(t)} = -\alpha \delta^{(t)} \mathbf{e}^{(t)} \quad (\text{B.56})$$

Using TD(1) learning over an entire sequence of inputs produces the same weight changes as the Widrow-Hoff procedure. However, the updates only depend on successive predictions and the sum of all previous values for $\nabla_{\mathbf{w}} p^{(t)}$ (Sutton, 1988).

Although both supervised and TD methods should converge to the same evaluations with infinite experience, TD methods outperform supervised-learning procedures, because they eliminate a random statistical fluctuation source of noise. By taking advantage of the information provided by the temporal sequence of observations, they can make better evaluations of new observational states using intermediate observational states that already have good estimates. Although it is possible for this information to be misleading, particularly early in the learning, it is generally helpful (Sutton, 1988).

The Widrow-Hoff procedure is optimal: it minimizes the root-mean-square error between its predictions and the actual outcomes in a training set. However, the performance of the TD method on the bounded random walk problem improves rapidly as λ is reduced below 1 (the Widrow-Hoff procedure); with the best performance occurring at $\lambda = 0$. In fact, all the TD methods with $\lambda < 1$ perform better both in absolute terms and over a wider range of α values. This performance improvement is due to the fact that the Widrow-Hoff procedure minimises the error in the training data. Whereas TD(0) converges to the maximum-likelihood estimate of the underlying Markov process. Since the goal is for the prediction to match the expected value of the subsequent outcome,

not the actual outcome occurring in the training set; TD methods perform better than supervised learning methods (Sutton, 1988).

Sutton proved that $TD(0)$ ⁷¹, and Dayan that $TD(\lambda)$, converges to the ideal predictions for data sequences generated by absorbing Markov processes (Tesauro, 1992). It is further conjectured that, if α is reduced according to an appropriate schedule, e.g. $\alpha = 1/n$, then the variance converges to zero as well. Basically, TD methods can be viewed as gradient-descent procedures. However, linear $TD(0)$ converges to the optimal predictions, while supervised-learning procedures converge to sub-optimal predictions. This also helps explain why TD methods show empirically faster learning rates: they are moving toward a better final result (Sutton, 1988).

TD methods are also not limited to predicting the final outcome of a sequence. They can also be used to predict a quantity that accumulates over a sequence. Furthermore, they can be used with infinite sequences, where the only actuals are the incremental rewards at each time step. In these scenarios, each observation is used to predict the total future cumulative reward rather than a single target value, and the update procedure is updated to:

$$\mathbf{w} := \mathbf{w} - \alpha (p^{(t)} - p^{(t+1)} + r^{(t+1)}) \mathbf{e}^{(t)} \quad (\text{B.57})$$

where $r^{(t+1)}$ is the actual reward received between t and $t+1$ (Sutton, 1988).

Appendix B.8 TD Learning in Artificial Neural Networks

TD learning addresses the temporal error assignment over a sequence of predictions. A simple implementation of TD learning uses a lookup table. The predicted output for each input is stored in a table, and the predictions are adjusted with training. This method works for tasks with an enumerable input space. However, many tasks have input spaces that are too large or even continuous to explore effectively or simply store practically.

⁷¹ However, the convergence theorem requires every possible state to be visited infinitely many times (Tesauro, 1992).

Artificial neural networks enable learning to be generalised across similar inputs, and back-propagation solves the problem of structural credit assignment – how to adjust the weights in the network to minimize the error. Combining TD learning and artificial neural networks with back propagation would result in an algorithm that can flexibly learn to maximize cumulative rewards over multiple time steps while learning structural similarities in input patterns and generalise predictions over new inputs. However, there is no easy way to apply the artificial neural networks' back-propagation algorithm to the TD algorithm (McClelland, 2015).

In 1989, Anderson successfully combined the TD learning algorithm with the artificial neural network's back propagation algorithm (Anderson, 1989), and in 1995, Tesauro used TD learning and an artificial neural network with 40 hidden units to create a Backgammon learner that not only surpassed all previous Backgammon computer programs, it also plays at a strong master level that is extremely close (within a few hundredths of a point) to equalling the world's best human players. It even learnt genuinely novel strategies that are now used by top human players (Tesauro, 1995).

The key difference between regular back propagation and TD back propagation is that the weights for the input at time t are adjusted at time $t+1$ (McClelland, 2015).

Appendix B.9 Adaptive Control

When the physics of a mechanical process is known, it is possible to determine controller parameters for different operating conditions. However, most processes are complex and not well understood, and an adaptive controller can provide a good alternative. Similarly, in cases where only some of the dynamics are well understood.

Feedback attempts to reduce the effects of disturbances and uncertainty, whereas an adaptive controller responds to changes in the dynamics of the process and the character of the disturbances. Unlike a constant-gain feedback system, an adaptive controller has adjustable parameters and a

means for adjusting those parameters. To achieve this, it needs a normal feedback loop and a parameter adjustment feedback loop.

Adaptive control requires understanding the desired behaviour; determining the structure of the controller and its adjustable parameters; finding a way to adjust the parameters; and implementing the controller.

Interest in adaptive control waned in the 1950s, because the problem was considered too hard to solve. After state space and stability theory, and stochastic control theory were introduced in the 1960s, Bellman introduced dynamic programming, and Tsytkin described a common framework for learning and adaptive control. In the late 1970s and early 1980s, the first proofs for the stability of adaptive systems appeared. This allowed adaptive techniques to be applied to single-loop controllers and provide automatic tuning of controllers.

Finding a controller that minimises the loss function is difficult. However, by making sufficient assumptions, dynamic programming can be used to find a solution. The primary assumption – the Bellman equation – is that the current state describes the entire problem. A conditional probability distribution of the current state is obtained from the measurements. The controller is then a non-linear function that maps the current state onto the control variables. To find the right balance between good control and errors in the estimation of the state, an optimal controller attempts to drive the output to its desired value, while adding perturbations when the parameters are uncertain.

An adaptive controller cannot replace good process knowledge. If the process dynamics are constant, it is best to use a controller with constant parameters. If the process dynamics are varying, a controller with varying parameters should be used, but it is better to use gain scheduling if the process varies predictably (Åström & Wittenmark, 1995).

However, adaptive control focuses on dynamic systems, where system dynamics are smooth – they are linear or can be approximated as linear around a desired trajectory. Cost functions in adaptive control are typically quadratic errors. Although the dynamic model of the system is not known and must be estimated from the data, the structure of the model is fixed. Therefore, model estimation is simply a parameter estimation problem (Kaelbling, Littman & Moore, 1996).

Appendix B.10 Reinforcement Learning

A stochastic process has a Markov property if the probabilities of future states depend only on the present state. A Markov process is a random process that evolves over time, where the next value depends only on the current value. A Markov chain is a type of Markov process that describes the possible states and the probabilities of moving from one state to another. A Markov Decision Process (MDP) is an extension of the Markov chain. First, the probabilities of moving to another state are dependent on the current state $\mathbf{x}$ as well as the action taken: a . Second, the transition from one state to another can result in a reward $r(\mathbf{x}, a)$. The dynamic decision problem requires finding the action policy π that maximises a state's total discounted reward:

$$V(\mathbf{x}_0) = \max_{\pi(a_{t=0}^{\infty})} \sum_{t=0}^{\infty} \beta^t r(\mathbf{x}_t, a_t) \quad (\text{B.58})$$

where $\mathbf{x}_t$ is the state at time t , and β is the temporal discount rate (Bremaud, 1999).

Complex problems can be simplified by recursively breaking them into sub-problems. If the sub-problems are used multiple times, their solutions can be stored, which reduces computation time. Dynamic programming compares combinations of solutions to sub-problems to find the optimal solution to a given problem. Bellman used this approach to recursively solve the dynamic decision problem:

$$V(\mathbf{x}) = \max_{\pi(\mathbf{x}, a)} (r(\mathbf{x}, a) + \beta \cdot V(\mathbf{x}')) \quad (\text{B.59})$$

where $\mathbf{x}'$ is the next state (Bellman, 2013).

In a stochastic process, the next state $\mathbf{x}'$ is a function that depends on both the probabilities of moving from one state to another state given a particular action, and the policy: the probability of selecting a particular action $\pi(\mathbf{x}, a)$. The Bellman equation describes the value of a particular state in terms of the policy:

$$V^\pi(\mathbf{x}) = \sum_a \pi(\mathbf{x}, a) \sum_{\mathbf{x}'} P_{\mathbf{x} \rightarrow \mathbf{x}'}^a (r_{\mathbf{x} \rightarrow \mathbf{x}'}^a + \beta \cdot V^\pi(\mathbf{x}')) \quad (\text{B.60})$$

The value of a state equals the policy weighted sum of the action taken multiplied by the probability weighted sum of the discounted value of the expected next state plus the expected reward (Sutton & Barto, 1998).

By defining an action value as the expected return from taking action a in state $\mathbf{x}$ with policy π as:

$$Q^\pi(\mathbf{x}, a) = \sum_{\mathbf{x}'} P_{\mathbf{x} \rightarrow \mathbf{x}'}^a (r_{\mathbf{x} \rightarrow \mathbf{x}'}^a + \beta \cdot V^\pi(\mathbf{x}')) \quad (\text{B.61})$$

The policy π can be improved by identifying when $Q^\pi(\mathbf{x}, a) > V^\pi(\mathbf{x})$: if taking a particular action from a state (and then following the current policy) is better than following the current policy, then there is a better policy (Sutton & Barto, 1998).

If the immediate effect of an action on a state is known, even if the state will subsequently change randomly, the value of the state after the action can be used instead. This is the approach used in games where the state of the environment after an action can be specified and its value estimated before the actions of opposition players modify the state (Sutton & Barto, 1998).

Dynamic programming was widely considered the only feasible way to solve general stochastic optimal control problems. However, despite being more efficient and more widely applicable than

any other method, dynamic programming suffers from the curse of dimensionality: its computational requirements grow exponentially with the number of state variables (Sutton & Barto, 1998), and to solve problems in complex real-world environments requires high dimensional sensor data (Mnih *et al.*, 2015).

Monte Carlo methods use random sampling to obtain approximate results. Markov chain, Monte-Carlo (MCMC) methods use random sampling to estimate the probabilities of a Markov chain. MCMC does not assume complete knowledge of the environment it only requires experience (Sutton & Barto, 1998).

Reinforcement learning problems are very closely related to optimal control problems; especially those formulated as an MDP. Therefore, optimal control solutions, such as dynamic programming, are a type of reinforcement learning. At a minimum, a reinforcement learner consists of two components: a policy and a value function. However, it can be improved by also including a model of the environment. The policy defines what action is taken given a particular state. The value function maps state-action pairs to the estimated total, temporally discounted reward.

Reinforcement learning defines how the policy is updated as a consequence of its experience (Sutton & Barto, 1998) to maximise its total, average or time limited, temporally discounted reward (Kaelbling, Littman & Moore, 1996).

With supervised-learning, the feedback from the environment directly indicates what the correct action should have been, and is independent of the action selected. Even if the feedback is noisy, it is still more informative than evaluative feedback. A single sample can be used to correctly change the action selection rule. However, a supervised learner cannot control or influence its environment because it follows the instructive information it receives. Instead of trying to make its environment behave in a certain way, it tries to make itself behave as instructed by its environment (Sutton & Barto, 1998).

It is also expected that the environment is non-deterministic – the same action in the same state on different occasions may result in different next states and different rewards. Although reinforcement learning is effective in slowly-varying non-stationary environments, the algorithms do assume that the environment is stationary – the state transition probabilities and rewards do not change over time (Kaelbling, Littman & Moore, 1996).

Unlike supervised learning, where learning relies on knowing the correct action given a specific state, reinforcement learning relies on discovering which actions to take by interacting with an environment, through trial and error, to achieve a goal or maximise a reward signal. In addition, actions may not only affect the rewards received, but also the environment, and hence all subsequent rewards too (Sutton & Barto, 1998). In real-world environments, learning requires the learner to derive a representation of the environment efficiently and then generalise from past experience to work effectively in new situations (Mnih *et al.*, 2015).

Another advantage is that reinforcement learning can be performed on-line. The challenge is finding the right balance between exploiting the best solution found thus far, and exploring other, possibly better solutions (Kaelbling, Littman & Moore, 1996).

Reinforcement learning only receives evaluative feedback. Although the reward received after each action gives some indication about how good the action was, it does not indicate whether or not the action was the correct or best action. An action's value can only be determined by trying all actions and comparing the rewards. The evaluative feedback must be compared with that of other actions before any inferences can be made about action selection. Therefore, the problem requires explicit searching of action trees (Sutton & Barto, 1998).

A reinforcement learner can be viewed as an extension of a supervised learner, because a supervised learner's training data can be used to train a reinforcement learner. Instead of a temporal sequence of states and policy driven actions followed by a final reward and optional interim rewards, each

state is given its known reward. Finally, since the goal of a supervised learner is to minimise a cost function, the goal of a reinforcement learner can be configured to maximise the negative cost.

There are problems that a supervised learner cannot solve. The simplest example is a problem consisting of only two possible rewards: success or failure, and only two possible actions. If one action has a high probability of success and the other action has a high probability of failure, a supervised learner will quickly find the best action. However, if both actions have a high probability of success the solution would get randomly stuck on one action. Conversely, if both actions have a high probability of failure and the learner assumes that a failure indicates that the other action would be successful, then the learner would oscillate between the two (Sutton & Barto, 1998).

The boundary of a reinforcement learner should not be the limit of its physical body, but the limit of its control. This implies that determining the reward must be outside of the learner. The reinforcement learner maximises its reward. Therefore, rewards must reflect what should be accomplished, not how to accomplish it. For example, when learning to play backgammon, the learner should only be rewarded for winning, not for achieving sub-goals like taking its opponent's pieces or for the distribution of pieces on the board (Sutton & Barto, 1998).

Reinforcement learning can also address important problems in neural network learning by providing a learning algorithm for multi-layer networks (Sutton & Barto, 1998). Mnih et al. successfully combined reinforcement learning with a deep artificial neural network to approximate an optimal action-value function (Mnih *et al.*, 2015).

Finally, accurate representations of the environment are not necessary. Some early studies of learning to solve the pole-balancing task used a very coarse state signal which divided cart positions into three regions: right, left and middle; and similar rough quantizations of the other three intrinsic state variables. This was sufficient to allow the task to be solved easily by reinforcement learning methods. In fact, this coarse representation may have facilitated rapid learning by forcing the

learning agent to ignore fine distinctions that would not have been useful in solving the task (Sutton & Barto, 1998).

Appendix B.11 Hyper-Parameters

To successfully use machine learning and artificial neural networks requires correctly selecting a large number of hyper-parameters (Bengio, 2012b). There are two types of hyper-parameters:

Hyper-parameters associated with learning optimisation and hyper-parameters associated with the size and structure of the neural network – the model. Learning hyper-parameters include:

- The initial learning rate;
- The rate at which the learning rate is changed;
- The size of the mini-batch⁷²;
- The number of training iterations; and
- The gradient momentum⁷³.

Model hyper-parameters include:

- The number of hidden units;
- The weight decay regularisation coefficient;
- The sparsity of activation regularisation coefficient;
- The neuron activation function used;
- The initial weight initialisation scaling coefficient; and

⁷² The mini-batch size should only impact training time and not the learning performance (Bengio, 2012b).

⁷³ Used to slow the rate at which the gradient is changed.

- The randomisation seed.

TD learning also has the temporal discount hyper-parameter (Tesauro, 1992). In addition to the above hyper-parameters, data pre-processing is also used to improve learning (Bengio, 2012b).