Uncovering Efficient Learning and Initialisation Algorithms for Neural Networks Using Evolutionary Algorithms and Theoretical Analyses

Ahmet Yilmaz

A thesis submitted for the degree of
Doctor of Philosophy in Computer Science

School of Computer Science and Electronic Engineering
University of Essex

March 2021

Abstract

Artificial Neural Networks (ANNs) are one of the most widely used form of machine learning algorithms. Over the years numerous types of ANN have been developed and applied to many domains. However, there are still important problems to overcome including their slow learning and the inability of certain types of deep ANNs to learn, due to the vanishing gradient problem. This thesis attempted to solve these problems via novel efficient learning and initialisation algorithms.

One of the tools used to do this is Genetic Programming (GP): a form of program evolution. Very little research had been done on the use of GP to induce learning rules for ANNs. This thesis started from where others left and also developed a rigorous methodology for fairly comparing learning rules. GP was able to evolve a learning rule that is fast and general. A qualitative interpretation for the rule and empirical evidence showed it is superior to the standard back-propagation algorithm.

The vanishing gradient problem is a long-standing obstacle to the training of deep ANNs using sigmoid activation functions. The methods proposed in the literature to improve the situation are not very successful. This thesis first used GP to discover an initialisation algorithm that solve the problem. Then, we performed an in-depth analysis of the evolved algorithm and a theoretical analysis of the extent to which the vanishing gradient problem depends on the choice of the mean of the initial weight distribution. Both indicated that initialising the weights with a carefully selected negative mean would give large initial gradients in weight space. Empirical verification finally showed that starting from such a good initial position, the standard back-propagation algorithm is successful and efficient at training deep networks with 10 and 15 hidden layers on a standard set of benchmark problems.

Acknowledgement

I would like to sincerely thank the following people who supported me to finalise this research.

First of all, I would like to express that it is a great honor to study under the supervision of Prof. Riccardo Poli, who has always been understanding to me and supported and guided me in my research. Every meeting and every conversation I had with him was an encouraging experience for me. I would like to say a big thank you to him for all his feedback and help.

I would also like to thank my cousin and inspiration, Prof. Mehmet Yildirim, who has always supported me and offered deep insight into this field.

I would like to express that I am grateful to my parents, my wife and my son. I would not have been able to achieve this research without their patience and support.

Last but not least, I would like to thank the Ministry of National Education of Republic of Turkey and the officers who work there for supporting me to have this PhD degree.

Contents

1 Introduction **12**

1.1 Aims of the Research 14

1.2 Novelty of the Research 16

1.3 Achievement of the Research 17

1.3.1 Comparison Methods for Learning Rules 17

1.3.2 Evolution of Learning Rules for MLPs 17

1.3.3 Initialisation Rules for Deep MLPs 18

1.4 The Structure of the Thesis 19

2 Literature Review **21**

2.1 Neural Networks 21

2.1.1 Biological Neuron 21

2.1.2 Artificial Neuron 21

2.2 Multilayer Perceptron 23

2.2.1 Activation Functions 24

2.2.2 Normal Operations 25

2.2.3 Standard Back-propagation 26

2.3 Improvements of Artificial Neural Networks 27

2.3.1 Back-propagation Learning Rules 28

2.3.2 Weight Initialisation 29

2.3.3 Methods for Vanishing Gradient Problem 32

2.3.3.1 Pre-training 32

2.3.3.2 Use of an Alternative Activation Function 33

2.3.3.3 Residual Network 35

2.3.3.4 Weight Initialisation 35

2.4 Genetic Programming 36

2.4.1 Representation 38

2.4.2 Initialisation of the Population 39

2.4.3 Fitness Function 40

2.4.4 Selection 40

2.4.5 Crossover and Mutation 41
2.4.6 Summary of GP Process 42
2.5 Integrating Neural Networks and EAs 42
2.5.1 Evolving Neural Network Weights 42
2.5.2 Evolving Neural Network Architecture 44
2.5.2.1 Constructive and Pruning Methods 45
2.5.2.2 Use of EAs to Evolve Neural Network Architecture 45
2.5.3 Evolving Neural Network Learning Rules 48
2.6 Conclusion 52

3 Evolving Learning Rules and a Fair Methodology for Comparing Learning Rule Performance 54
3.1 Introduction 54
3.2 Comparing Learning Rules 55
3.2.1 Pairwise Comparisons 55
3.2.2 Success Criteria 56
3.2.3 Optimal Learning Rates 57
3.2.4 Optimal Number of Epochs 57
3.2.4.1 Minimum Computational Effort for Parallel Training 58
3.2.4.2 Minimum Computational Effort for Sequential Training 60
3.3 GP System, Fitness Measure and Datasets 61
3.3.1 Linear GP System 61
3.3.2 Fitness Measure 63
3.3.3 Training and Test Problems 64
3.4 Experimental Results 66
3.4.1 Qualitative Interpretation of Evolved Rule 66
3.4.2 Choice of Performance Criteria 68
3.4.3 Results with Parallel Evaluation 68
3.4.4 Results with Sequential Evaluation 76
3.5 Summary 77

4 Using Unstable Learning Rules to Speed up Learning 79
4.1 Introduction 79
4.2 GP System, Fitness Measure and Datasets 80
4.2.1 Linear GP System 80
4.2.2 Fitness Measure 80
4.2.3 Training and Test Problems 82
4.3 Experimental Results 83
4.3.1 Qualitative Interpretation of Evolved Rules 83

4.3.2 Choice of Performance Criteria 84
4.3.3 Results with Parallel and Sequential Evaluations 84
4.3.3.1 Parallel Evaluation 85
4.3.3.2 Sequential Evaluation 92
4.4 Summary 92

5 Evolving Initialisation Rules to Overcome the Vanishing Gradient Problem 94
5.1 Introduction 94
5.2 GP System, Fitness Measure and Datasets 95
5.2.1 Linear GP System 95
5.2.2 Fitness Measure 96
5.2.3 Training and Test Problems and Their Structures 97
5.3 Initialisation Methods Used for Comparison 99
5.4 Experimental Results 99
5.4.1 Qualitative Interpretation of the Evolved Initialisation Method 100
5.4.2 Results with the Evolved Initialisation Rule 100
5.5 Summary 108

6 A Simple Initialisation Method to Train Deep Neural Networks Using the Logistic Activation Functions 109
6.1 Introduction 109
6.2 Simple Theoretical Model of Initial Gradients 110
6.2.1 Expected Values of Network State and Parameters after First Forward and Backward Propagations 110
6.2.2 Relationship between the Mean of the Distribution of Initial Weights and Network Gradients 112
6.3 Corroboration of the Theory and Resulting Insights 113
6.3.1 Benchmark Problems and Network Structures 113
6.3.2 Empirical Estimation of Gradients in Deep Networks with Identically-sized Hidden Layers 113
6.4 Refining the Initialisation Strategy 117
6.4.1 Analysis of Variance of Initial Net-inputs 117
6.4.2 New Initialisation Method 117
6.5 Training Deep Networks after the Proposed New Initialisation . . . 118
6.5.1 Problems and Network Structures 119
6.5.2 Initialisation Methods Compared 119
6.6 Numerical Optimisation of μ 124
6.6.1 Hill-climbing the Initial Gradient Surface 124
6.6.2 NIM-initialised vs Hill-climber-initialised Gradient Descent 126

6.7 Summary 127

7 Conclusions and Future Work 131

7.1 Contributions of the Research 131

7.1.1 Comparison Methods for Learning Rules 131

7.1.2 Learning Rules Evolved for MLPs 132

7.1.3 Initialisation Rules for Deep MLPs 133

7.2 Limitations in the Research 134

7.2.1 Limitations in the Works that Evolved Learning Rules . . . 134

7.2.2 Limitations in the Works that Developed Initialisation Rules 135

7.3 Future Work 135

7.3.1 Changing the Configurations Used in the Experiment 135

7.3.2 Considering the Limitations of this Thesis 136

Appendix A Additional Results from Chapter 3 137

Appendix B Additional Results from Chapter 4 142

*

List of Figures

2.1 A typical biological neuron 22
2.2 An artificial neuron 22
2.3 An Example MLP 23
2.4 Flowchart of GP 39
2.5 Examples of the representation used in a linear GP (a) and a tree based GP (b) 40
2.6 Examples of a primitive set in linear GP (a) and a program which uses the primitive set (b). 40
2.7 Linear GP crossover 41

3.1 Illustrative comparison of the TSE (top) and weight (bottom) changes over time for an XOR problem with a classical 2-2-1 architecture when training is done by the *SBP* and *NLR*, both starting from the same initial set of weights and biases 69
3.2 Results for all test problems when **parallel** evaluation is used (see text for explanations). 75

4.1 Results for all test problems when **parallel** evaluation is used (see text for explanations). 91

5.1 Results for our six test problems when the SIM, Glorot, Kumar and EIM methods are used to initialise networks of depth L = 10 and depth L=15. 104
5.2 Sum of gradient magnitudes in each layer when the *EIM*, *Kumar*, *SIM* and *Glorot* initialisation methods are applied to the small (L=10) and large (L=15) networks of iris problem. 106

6.1 Mean $|\Delta W|$ for the authorship problem for a network with $L = 15$ layers and different numbers of neurons, n, in the hidden layers, for three representative layers. 115

6.2 Heat maps representing the mean $|\Delta W|$ for the authorship problem for a network with $L = 15$ layers and different numbers of neurons, n, in the hidden layers, for three representative layers. 116
6.3 Results for our six test problems when the *SIM*, *Glorot*, *Kumar* and *NIM* methods are used to initialise networks of depth $L = 10$ and depth $L = 15$ 122
6.4 Sum of gradient magnitudes in each layer when for a network of depth $L = 10$ for four different initialisation methods in the iris problem. 124
6.5 Optimal μ values prescribed by NIM in Equation (6.11) and those found by a hill-climber (optimising the mean $|\Delta W|$ in the first hidden layer) the six problems in Table 5.3 for a network with $L = 10$ layers and 10 neurons in each hidden layers. 125
6.6 Same as in Figure 6.5 but for a network with $L = 15$ layers and 100 neurons in each hidden layers. 126
6.7 Mean absolute gradient in the first layer of weights obtained when using the optimal μ values prescribed by NIM in Equation (6.11) and those found by a hill-climber for the six problems in Table 5.3 for a network with $L = 10$ layers and 10 neurons in each hidden layers.127
6.8 Same as in Figure 6.7 but for a network with $L = 15$ layers and 100 neurons in each hidden layers. 128
6.9 Cross-entropy loss as a function of the number of training epochs for NIM-initialised an hill-climber-initialised networks trained with the SBP for the six problems in Table 5.3 for a network with $L = 10$ layers and 10 neurons in each hidden layers. The learning rate was $\eta = 0.25$. Results are averages of 30 independent runs. 129
6.10 Cross-entropy loss as a function of the number of training epochs for NIM-initialised an hill-climber-initialised networks trained with the SBP for the six problems in Table 5.3 for a network with $L = 15$ layers and 100 neurons in each hidden layers. The learning rate was $\eta = 0.25$ for NIM and both $\eta = 0.25$ and $\eta = 0.05$ for hill-climber initialisation. Results are averages of 30 independent runs. 130

A.1 Comparison results of NLR and SBP for test problems when **sequential** evaluation is used. 141

B.1 Comparison results of NLR(1), NLR(2) and SBP for test problems when **sequential** evaluation is used. 146

List of Tables

3.1 Primitive set used in the experiments 62

3.2 Network structure and parameters for the two problems used for evolving learning rules 65

3.3 Information on the data sets used in this study 65

3.4 Network structures adopted for the different test problems used for comparing *NLR* and *SBP* 66

3.5 Different effective learning rates of *NLR* as a function of the activations of the pre-synaptic and post-synaptic neurons associated with a connection weight 67

3.6 Minimal effort and associated optimal learning rate, number of epochs and number of runs for **parallel** evaluation 76

3.7 Minimal effort and associated optimal learning rate, number of epochs and number of runs for **sequential** evaluation 77

4.1 Primitive set used in the experiments 80

4.2 Network structures and parameters for the problems used for evolving learning rules 82

4.3 Minimal effort and associated optimal learning rate, number of epochs and number of runs for **parallel** evaluation 87

4.4 Minimal effort and associated optimal learning rate, number of epochs and number of runs for **sequential** evaluation 93

5.1 Primitive set used in the experiments 96

5.2 Network structure and parameters for the two problems used for evolving the initialisation rule 98

5.3 Network structures adopted for the different test problems used for comparing *NTA* and *SBP* 98

5.4 Testing accuracy of the EIM and Kumar initialisation method on test networks. 107

List of algorithms

1 Pseudo-code illustrating how GP-MLP evolves learning rules 62

Chapter 1

Introduction

In conventional programming methods, the computer does not know how to act. The user explicitly programs the computer so that it produces the desired outputs or, more generally, behaviours. In contrast, machine learning algorithms produce their own solution for a problem by learning the characteristics of the data from a training set of examples [1, 2, 3].

Machine learning is a term which represents numerous techniques that can automatically learn solutions in this manner [4]. Machine learning algorithms are divided into three basic categories: supervised learning, unsupervised learning and reinforcement learning.

In supervised learning [2, 3], the examples in the training set have labels or desired outputs. Both the input features and the desired outputs of the examples are used by the algorithm during the training for it to learn the characteristics of the data. Linear Regression, Logistic Regression, Support Vector Machines, Decision Trees, Naive Bayes and Artificial Neural Networks (ANNs) are the most commonly used supervised learning algorithms. This learning model is usually used for classification and regression problems.

Classification algorithms aim to categorise the pattern shown them-self based on the correlations they learn between incoming data and the labels during the training. The regression algorithms produce continuous values with the aim of predicting proper output for the incoming [2, 3].

In contrast, in unsupervised learning [2, 3], the examples in the training set do not have desired outputs. Only the input features of the examples are provided to the algorithm. The algorithm aims to group the examples depending on their similarities. It does not know whether the groups are correctly categorised. The most commonly used unsupervised learning algorithms are k-means and Principle Component Analysis.

In reinforcement learning [2, 3, 5], the algorithm learns how to act from the outputs produced by itself. The algorithm returns feedback that is called reward or

penalty to the input by evaluating the output whether it is correct or not. It is used as an input for the next step. Reinforcement learning is widely used in video-games, robotics and autonomous cars etc. Most common algorithms in this learning model are Q-learning, Monte Carlo and State-Action-Reward-State-Action which is known as SARSA.

Some well-known machine learning algorithms are listed above based on their usage purpose. ANNs, among those, are one of the most commonly used machine learning algorithms. They are forms of supervised learning, although there are some unsupervised ANNs too. ANNs were developed by taking inspiration from the human nervous system and the brain. In ANNs, neurons are connected to each other to form networks, typically with a layered structure. A neuron in the network receives signals that come from neurons in previous layers, processes them, and outputs a result to neurons in the next layer through the connections. Each connection has a value that is used to weight the signal that comes from the previous neuron. These are called connection weights. The networks can learn the correlations between input signals and the desired outputs in the training data by adjusting the connection weights repeatedly. This system uses a mathematical model of the biological neuron that processes electrical signals coming from outside and transmits the processed signal other neurons. The biological neuron will be explained in detail in the next chapter [34, 35, 6].

The early studies in this area were done in the 1940s. In 1943, McCulloch and Pitts [7] developed a neural model that could compute logical expressions. This model was the inception of the field of ANNs. As a second step, Donald Hebb [8] introduced a first learning rule for neural networks which is now known as 'Hebb learning' in the late 1940s. Afterwards, Rosenblat's perceptron [9] and Widrow's Adaptive Linear Neuron (ADALINE) [10] models were developed. These models have a single-layer of weights between the input and output neurons. Minsky and Papert [11] reported that these models were not able to solve non-linearly separable problems such as the Exclusive OR (XOR). After this report, interest in neural networks decreased. Almost two decades later, researchers developed Multi-layer perceptrons (MLPs), which have one or more layers between input and output layers, and trained them using the gradient descent principle by back-propagation to solve nonlinear problems [12, 13, 14, 15, 16]. This achievement was an important milestone in the neural networks studies.

ANNs started to be widely used after MLPs and the back-propagation [12] algorithm were developed. Another significant improvement in ANNs was deep learning. This term is used for networks which have many hidden layers. In the last decade, studies on shallow (single-hidden-layer) networks have decreased while the studies on the networks with a deep architecture have increased. Since the early

studies on ANNs, researchers have developed many different types of deep and shallow neural networks that can be used for supervised learning, unsupervised learning and the other forms.

MLP with the supervised learning principle is the most widely used form of ANNs. In the next chapter, an example MLP will be shown, and its working principle will be described in detail. MLP can be used with a shallow or a deep architecture for many tasks such as prediction, recognition and classification. Unfortunately, the original algorithm can be trapped by local optima of the error functions, is very slow and can easily overfit. Also, the standard weight initialisation methods (e.g. with zero-mean) cause poor learning performance on the deep networks that use the logistic activation function. Various techniques have been proposed in the literature to improve the neural networks as we will review in the next chapter. However, neural networks still have some drawbacks, such as long training time (particularly with large, deep networks), slow learning, and the difficulty of choosing the correct hyper-parameters. Furthermore, with traditional comparison methods, it is not easy to evaluate which network's hyperparameters (such as the network topology, learning algorithm, and learning rate) provide faster learning than the others. In this thesis, we aim to improve such problems using Genetic Programming (GP) [17] and theoretical analyses.

GP is an evolutionary program-induction technique that can produce problem-solvers (i.e., algorithms) for a specific class of problem. One of the first studies for evolving algorithms was done by Cramer [18] in 1985. The author showed that Genetic Algorithms (GAs) with a tree-based structure could produce simple computer functions. Then, John Koza extended this approach, and in the early 1990s, introduced GP providing a significant number of experimental results that show computer programs produced by GP can solve wide range of problems [17, 19]. The GP components and its working principle will be described in detail in the next chapter.

1.1 Aims of the Research

A variety of deep and shallow ANNs have been widely applied to such data as images, signals and text for a long time [20, 21, 22]. However, there are still some significant questions to be answered:

a. How to choose the network topology

b. How to find a suitable learning rate for training

c. How to choose a suitable activation function

d. How to design a training algorithm

e. How to choose the initial values of the connection weights

For many years, these problems have been research subjects for scientists. However, these studies take a long time because progress often requires knowledge of mathematical principles, engineering knowledge, creativity and sheer luck. In addition, any new algorithm has to be widely tested to analyse its efficiency [23]. Because discovering new training algorithm is a search problem, it has been proposed to use search and optimisation methods, such as GP and Genetic algorithms (GAs), rather than manual trial-and-error methods in searching for new training algorithms [24]. Although algorithms evolved by GA were promising for simple problems [25], GP could evolve efficient algorithms for complex problems since it does not restrict the algorithms with fixed genetic coding [25, 26].

While answering all these questions may contribute to increasing the accuracy of results and speed up running time of the algorithm used, networks are more sensitive to their initial weights and the learning rules used than others. Therefore, this thesis mainly focuses on questions *d* and *e*.

More specifically, this thesis aims to *discover new learning and initialisation algorithms for neural networks using evolutionary algorithms and theoretical analyses*. To achieve this, this thesis tries to answer the following more specific questions that are relevant to questions *d* and *e*:

1. Would it be possible to develop fair comparison methods to compare the algorithms used in neural networks?

2. Are there any learning algorithms that require less computational efforts than the standard back-propagation (SBP) algorithm to solve the problems?

3. Can GP successfully develop new learning algorithms that are fast and general?

4. Can GP find any initialisation algorithms that make deep neural networks (which use the logistic activation function) trainable?

5. Would it be possible to find ways of initialising the weights so that the algorithms converge faster or more reliably?

This thesis mainly attacks problems *d* and *e* (from the list above) by investigating the questions *1, 2 and 3*, and *4 and 5*, respectively.

1.2 Novelty of the Research

Evolving learning rules, which is of particular interest in this thesis, was achieved through GP. Approximately two decades ago, GP enabled researchers [27, 28, 29, 24] to evolve new learning rules (more on this previous work in the next chapter).

Despite these early successes, somehow surprisingly, this work went unnoticed by the machine learning community, receiving only relatively few citations mostly by evolutionary computation researchers, and not being followed up by any new research. This thesis aims to fill this knowledge gap in the literature; and picks up from where others left and not only finds new fast, stable and general rules for MLP produced by GP, but also develops a rigorous methodology for comparing learning rules.

Initialisation methods for deep MLPs are the other particular interest of this thesis. The importance of the initialisation of neural networks has been emphasised since the 1990s. There have been a significant number of studies on this subject. Almost all studies presented until the late 2000s were done by considering shallow networks. Because the use of non-sigmoidal activation function is one of the efficient ways to train deep neural networks, most initialisation methods proposed in the last decade have been developed for networks that use non-sigmoidal activation functions. Another knowledge gap in the literature is whether deep MLPs that use the logistic activation function can successfully be trained given the well-known vanishing gradient problem, which hampers the training of such networks.

As mentioned above, training deep networks was achieved by using different activation functions such as Rectified linear Unit (ReLU) and other ReLU based alternatives. However, as we will discuss in the next chapter, such functions have some drawbacks such as dying neuron, bias shift, noise sensitiveness. The use of sigmoidal activation functions can avoid such problems and provide stable learning. In order to achieve this, pre-training and weight initialisation methods have been proposed in the literature. However, the proposed methods for such networks did not perform well when the networks have more than 5 hidden layers (for more, see Section 2.3.3). In a nutshell, training of deep neural networks that use the logistic activation function still suffers from the vanishing gradient problem.

Here we focused on using GP as a mechanism to discover new initialisation strategies, that would potentially reduce the vanishing gradient problem. Later, based on the lessons learnt from GP, we were able to improve on the algorithms GP evolved and find a theoretical explanation for why they work well.

GP is an evolutionary program-induction technique that has produced numerous human-competitive results in the last two decades. However, no one has paid enough attention to the usage of GP in evolving learning and initialisation algorithms for

neural networks. This thesis seeks to fill this gap in the literature by investigating whether GP can evolve learning and initialisation algorithms to train shallow and deep neural networks successfully.

1.3 Achievement of the Research

1.3.1 Comparison Methods for Learning Rules

In Chapter 3, two general rigorous methodologies were developed for fairly comparing the performance of learning rules. These methods answer to question 1 above.

Both methods compute the computational effort, i.e., the total number of training epochs required to obtain at least one successfully trained network within a batch of runs. The first method (with high probability) ensures that if all runs are executed in parallel, at least one run will be successful in a series of runs. The second method assumes that runs are executed one by one sequentially and provides the expected value of the computation required to obtain a successfully trained network.

1.3.2 Evolution of Learning Rules for MLPs

In Chapter 3, GP was applied to the problem of evolving learning rules for MLPs. Preliminary experiments showed that GP tends to find unstable learning rules by increasing the learning rate of the programs in the GP population when the fitness function considers only the error obtained from a single point (the final training epoch). In order to obtain more stable learning rules, the unstable rules in the population were penalised while the rules which have monotonic error descents were rewarded. GP found a form of *SBP* algorithm but with a variable effective learning rate. The rule is fast, stable and general. In Chapter 3, we provided both a qualitative interpretation for the new learning rule and strong evidence of its superiority with respect to the SBP.

In Chapter 4, similarly to Chapter 3, the GP was used to evolve neural network learning rules. In this case, the aim is to speed up learning by allowing the evolution of unstable learning rules that can escape local minima.

To do this, in contrast to the previous chapter, the GP does not consider oscillations on the error during the training. It is designed to optimise either the lowest error or the highest accuracy values obtained throughout the whole training process. The fitness function is the sum of the lowest errors or sum of the highest accuracies of a pre-defined percentage of the training epochs.

Two different learning rules were evolved. The fitness function used the error values for the first one, while it used the accuracy values for the second one. According

to comparison methods proposed; both the leaning rules evolved in this chapter are superior to the *SBP*. However, these rules are slower than the learning rule evolved in Chapter 3.

Although the GP system used different fitness functions and different training (evolution) problems for the experiments done in Chapter 3 and Chapter 4, the system found similar learning rules which adjust the learning rate of the *SBP*. The learning rates of both rules are varied on the basis of the activation values in the pre-synaptic and post-synaptic neurons.

1.3.3 Initialisation Rules for Deep MLPs

The vanishing gradient problem is another interest of this thesis. This is a long-standing obstacle for the training of deep neural networks, where the gradients in the networks become too small or zero. It usually happens in the earlier layer of networks when the networks are relatively deep, and a sigmoidal activation function is used for the neurons. Several different methods have been proposed to address this problem, such as pre-training the networks which is a kind of initialisation method, the use of different activation functions, residual neural networks and initialisation of the networks.

There are several successful methods in the literature that initialise deep networks to speed up learning [30, 31, 32, 33]. However, most of them do not consider the logistic activation functions or do not work successfully with such functions. This thesis is concerned with the initialisation of deep MLPs with neurons using the *logistic* activation function to overcome the vanishing gradient problem because this problem is still a major obstacle for such networks to learn the data successfully..

In Chapter 5, GP was applied to evolve a weight-update rule like the learning rules evolved in the previous chapters. The networks were pre-trained using the weight-update rule evolved. The evolved rule always returns a positive value. Therefore, the mean of the connection weights in a network shifts towards to the negative direction at each update throughout the pre-training process. The amount of shifting in each step depends on the architecture of the network.

The initialisation rule evolved was compared with three state-of-art methods using six classification problems. In order to show that the evolved rule is not sensitive to the network architecture, two different network architectures, one relatively small and one relatively large, were used for each problem. Results showed that the rule was always superior to the other methods used for comparison when the networks were relatively deep and wide. It outperformed two of those comparison methods while it was competitive with the best alternative when the networks were relatively small.

Chapter 6 investigates why initial weights that have a negative-mean can make

a deep MLP, that uses the logistic activation function, trainable. Also, that chapter proposes an initialisation method that does not need to pre-train networks.

The work started from a simple theoretical model that illustrates how mean gradients are affected by the *mean* of the initial weight distribution in such networks. When such a mean is zero, it is well-known that gradients are on average zero. However, the theory leads to a number of somehow surprising predictions: (a) the initial gradients are non-zero when the initial weights have non-zero mean; (b) the initial gradients are maximised when the mean of the initial weights is negative and inversely proportional to the number of neurons in each layer; (c) for networks that are not too deep, there is also a local optimum in the positive region for the mean initial weights; (d) the bigger the number of neurons in a layer, the bigger the initial gradients. These predictions have been verified with a comprehensive set of empirical tests showing that they are correct. The knowledge gained on initial gradients from theory and the empirical tests suggested that a rule that sets the mean initial weights to $-\min(1, 8/\text{number of neurons in layer})$ would maximise such gradients.

To verify to what degree this changed the situation in relation to the vanishing-gradients problems, the SBP method was used, but initial weights were drawn from a Gaussian distribution with such a mean and a standard deviation of 0.1, on a large set of test problems and both shallow and deep networks of up to 15 hidden layers. The results of this method was compared with those obtained with the standard zero-mean initialisation and other initialisation strategies proposed in the literature. In all cases, results showed that, uniquely, the proposed initialisation method prevents the SBP algorithm from suffering from the vanishing gradient problem in both shallow and deep networks.

1.4 The Structure of the Thesis

Chapter 2 begins with describing MLPs, including their components and the training process that uses the SBP. It, then, reviews literature related to the back-propagation learning rules, weight initialisation methods and the methods for the vanishing gradient problem. After that, it describes GP and its components and finally, presents an overview of studies integrating neural network and Evolutionary Algorithms (EAs).

Chapter 3 initially describes two general rigorous comparison methodologies developed to compare learning rules. This chapter then describes the GP system used to evolve learning rules and its configurations such as the GP parameters, the fitness function, the training problems and their parameters. It finally presents and discusses the experimental results.

Chapter 4 aims to speed up learning by evolving unstable learning rules using

the same GP system as that used in Chapter 3 but with different configurations and different fitness functions. It presents these differences, two evolved learning rules and analysis of the experimental results.

In Chapter 5, GP was applied with the aim of evolving an initialisation rule for deep MLPs. This chapter shows that deep MLPs can be successfully trained when the networks are initialised by the evolved method. It also reports the limitations of this method.

Chapter 6 is the extension of Chapter 5. It theoretically and empirically investigates why the initial weights drawn with a negative-mean make deep networks trainable; and presents an initialisation rule that does not require any pre-training process.

The final chapter contains a summary of the studies done in this thesis and their results, presents some conclusion and describes possible future work.

Chapter 2

Literature Review

2.1 Neural Networks

A human brain has billions of neurons and many biological neural networks. A neural network consists of a group of neurons that are connected to each other. We learn the environment thanks to biological neural networks that interpret signals received via our sense organs such as categorising objects according to their similarities using our eyes and distinguishing a dog voice from a cat voice using our ears [34]. ANNs are the mathematical model of biological neural systems and used with the aim of achieving such learning with machines.

2.1.1 Biological Neuron

Figure 2.1 shows a typical biological neuron. A biological neuron consists of four main components: soma (cell body), dendrite, axon and synapse. The soma receives electrical signals (messages) from the previous neurons through dendrites. When the signals accumulated in the soma exceeds the threshold, the soma fires and transfers the signals to synapses through the axon. Synapses are a biological node for signals to be transmitted to the dendrites of the next neurons. Numerous neurons that are connected to each other construct a biological neural system [34, 35].

2.1.2 Artificial Neuron

The artificial neuron is a mathematical model of the biological neuron. An artificial neuron is shown in Figure 2.2. It consists of input signals, connection weights, a neuron, an output signal. Input signals are weighted by the corresponding connection weights. The resulting values are summed in the neuron. The sum of those values is processed in the neuron by applying an activation function. The activation value is the output of the neuron and transferred to the other neurons through the corresponding connection weights.

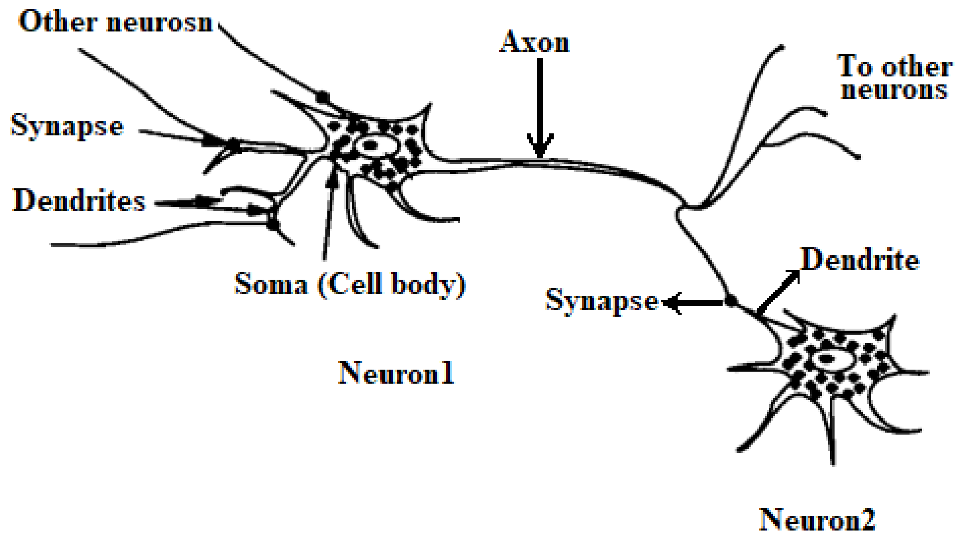


Figure 2.1: A typical biological neuron
(from [35])

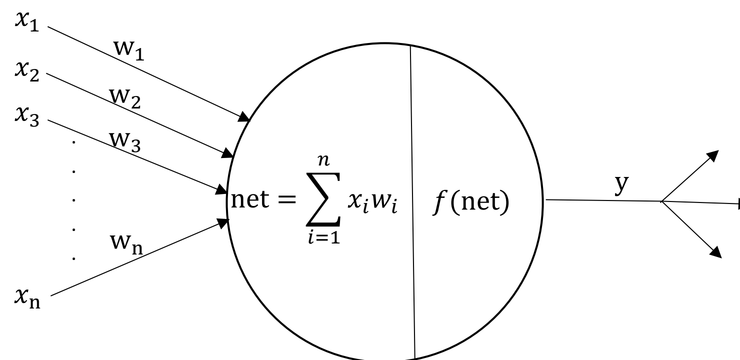


Figure 2.2: An artificial neuron

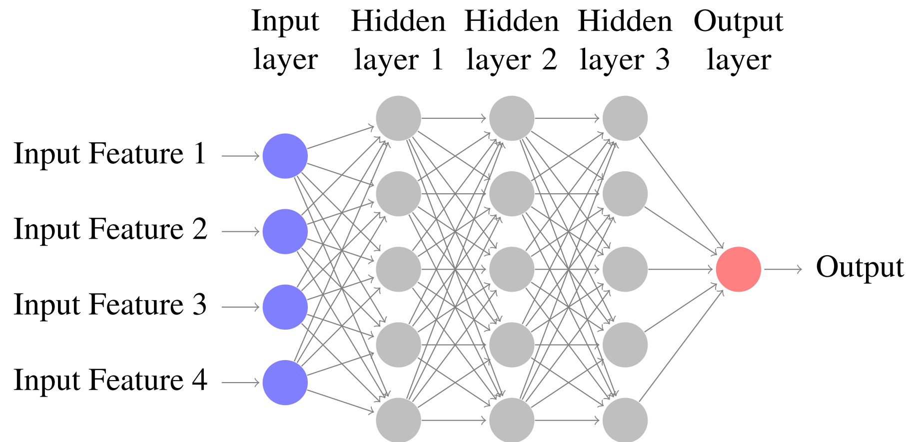


Figure 2.3: An Example MLP

2.2 Multilayer Perceptron

As stated in the previous chapter, networks introduced before MLP could not solve non-linearly separable problems. The development of MLP was an important milestone in the neural network history since it could solve the non-linearly separable problems. Since then, interest in neural networks has increased, and many different ANN types of ANNs have been introduced for different purposes.

MLP is a widespread form of feed-forward ANNs. It generally uses supervised learning and is commonly used for the classification and regression problems in many fields such as signal processing, image processing and pattern recognition.

In MLP, neurons, that are also called nodes, are connected to each other to form networks, typically with a layered structure. Each neuron has connection weights between itself and the neurons in the previous and/or next layer(s). An MLP has an input layer, one or more hidden layers and an output layer. A single hidden-layer MLP is called ‘shallow’ while an MLP which has many hidden layers is called ‘deep’. Figure 2.3 shows an example MLP that has four input features, three hidden layers, five neurons in each hidden layer and one output neuron.

Input layer: Every single neuron in the input layer represents one of the features of the problems. No activation function is applied to the neurons in the input layer. Incoming signals in the input layers are transferred to the neurons in the next (hidden) layer by being weighted with the corresponding connection weights. The sum of the weighted values in each neuron is used for net input by that neuron.

Hidden layer(s): The signals that come from the neurons of the previous (input or hidden) layer to a neuron of the current layer is called the net input. An activation function chosen by the designer is applied to the net input values to determine the output value of each neuron. Then, the resulting values are transferred to the neurons in the next layer by being weighted with the corresponding connection weights. Neurons in the next layer use the sum of weighted values that come to themself as

net input. This process is repeated until the activations of the neurons in the output layer are updated. This recursion process and the relevant equations are presented in Section 2.2.2.

Output layer: The process is similar to the operations in hidden layers. An activation function is applied to the net inputs of each neuron in the output layer. The resulting values represent the network outputs.

Each layer except the input layer in a network is also connected to a bias node which is usually set to 1. These connections have a weight value which is added to the net input of the corresponding neuron. These weights are also optimised via back-propagation. Its mathematical expression is provided in Section 2.2.2.

2.2.1 Activation Functions

Activation functions are mathematical functions that compute the outputs of neurons given their net input values [36]. There are many activation functions in the literature [37, 38]. In order to solve complex problems, the network needs non-linear activation functions. The most commonly used activation functions are the logistic, hyperbolic tangent (tanh) and rectified linear unit (ReLU) [39] functions.

Logistic function bounds the output value between 0 and 1. Its mathematical expression is

$$f(x) = \frac{1}{1+e^{-x}}, \tag{2.1}$$

and its derivative is

$$f(x)' = f(x)[1-f(x)]. \tag{2.2}$$

The **tanh** function limits the output value between -1 and 1. It is defined as

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \tag{2.3}$$

and its derivative is

$$f(x)' = [1+f(x)][1-f(x)]. \tag{2.4}$$

ReLU keeps the output value in the range of $[0,\infty)$. It is

$$f(x) = \begin{cases} x & \text{if } x > 0, \\ 0 & \text{if } x \leq 0, \end{cases} \tag{2.5}$$

and its derivative is

$$f(x)' = \begin{cases} 1 & \text{if } x > 0, \\ 0 & \text{if } x \leq 0. \end{cases} \tag{2.6}$$

In Equations (2.1) - (2.6), x is the net input for neurons. The formula used to

compute net inputs is provided in Equation (2.9) in Section 2.2.2.

Another commonly used activation function is the **softmax** function which transforms the net inputs in the output layer into probabilities that the pattern shown to the network belongs to each class. It is only applied to the output layer when the neural network is used as a classifier, and, typically, there are more than 2 classes. Its mathematical expression is

$$p_i = \frac{e^{x_i}}{\sum_{k=1}^{n} e^{x_k}}, \tag{2.7}$$

and its derivative is

$$\frac{\partial p_i}{\partial x_j} = \begin{cases} p_i \left(1 - p_j\right) & \text{if } i = j, \\ -p_i \cdot p_j & \text{if } i \neq j. \end{cases} \tag{2.8}$$

Here x is the net input of the neuron, i is the index number of the neuron in the output layer (it represents the class number of the input pattern); and n is the total number of classes in the data.

Although the logistic and tanh activation functions are widely used in practice, they suffer from the vanishing gradient problem. Therefore, different functions have been proposed such as ReLU [39], Leaky Rectified Linear Unit, (LReLU) [40] and Parametric Rectified Linear Unit (PReLU) [33]. Some well-known literature on the vanishing gradient problem is described in Section 2.3.3.

2.2.2 Normal Operations

The structure of an MLP consists of multiple layers of neurons connected by corresponding weights.

Let a^l be a vector representing the activation of the neurons in layer l. If l is not the output layer, this can also be considered as the input vector for the neurons in layer $l+1$. Let W^l be weights matrix between layer l and layer $l+1$. Following the standard convention we include in this matrix also the bias vector W_0^l which includes the biases of the neurons in layer $l+1$. Let net^l be net input vector of layer l, which has elements

$$net_j^l = W_{0j}^{l-1} + \sum_{i=1}^{n} a_i^{l-1} W_{ij}^{l-1}. \tag{2.9}$$

Then an activation function is applied to the corresponding neuron, so the elements of a^l are given by

$$a_j^l = \begin{cases} f(net_j^l) & \text{if } 1 < l \leq L, \\ x_j & \text{if } l = 1, \end{cases} \tag{2.10}$$

where L is the number of layers in the network and x_j is the j-th element of input pattern x.

2.2.3 Standard Back-propagation

The aim of back-propagation is to decrease the error of the network calculated at the output layer and optimise the connection weights. When the input data is applied to the neurons in the first layer of the network, various operations are performed on that data until it reaches to the output layer such as weighting the signals, computing the net input and output values in the neurons. The error is the difference between actual activation values in the output layer and the desired values over the training set. The network error is computed after the error is computed for every single example in the data. The computed error signals are scaled and back-propagated through the weights in the previous layer using gradient descent. Thus, each of the neurons in the hidden layers is attributed a part of the total error. This process is repeated until reaching the input layer. Once the errors are for all the neurons are known, the weights can be adjusted. This arrangement allows the network to produce actual outputs that consistently match the desired outputs after the training process is completed.

In vector form, the error produced when training pattern x^k is presented in input to the network is

$$\varepsilon^k = a^L(x^k) - y^k, \tag{2.11}$$

where $a^L(x^k)$ is the activation vector of the last layer in the presence of input pattern x^k and y^k is the corresponding desired output vector.

There are many methods to measure network performance in the literature. In this thesis, for boolean induction problems, *the total sum of squared error* (TSE) and for multi-classes problems, *average cross-entropy loss* are used as these are the most commonly used ones in the literature for such problem types. Average cross-entropy is used only when the output layer of a network is softmax. The TSE is

$$TSE = \frac{1}{2}\sum_{k=1}^{m}\left(\varepsilon^k \cdot \varepsilon^k\right), \tag{2.12}$$

and average cross entropy loss is

$$L = -\frac{1}{m}\sum_{k=1}^{m} y^k \cdot \log a^L(x^k), \tag{2.13}$$

where m is the number of examples in the data set, $\cdot$ is the scalar product and, so, $(\varepsilon^k \cdot \varepsilon^k)$ is the squares of the components of ε^k.

In the SBP learning rule, at each iteration, t, each weight is updated with a fraction of the derivative of the error with respect to the weight, i.e.,

$$\Delta W^l_{ij}(t) = \eta \delta^{l+1}_j a^l_i, \tag{2.14}$$

where η is the learning rate and

$$\delta_j^l = \begin{cases} f'(net_j^l)\varepsilon_j^L & \text{if } l = L \text{ and activation funtion is sigmoid,} \\ \varepsilon_j^L & \text{if } l = L \text{ and activation funtion is softmax,} \\ f'(net_j^l)\sum_i W_{ji}^l \delta_i^{l+1} & \text{otherwise,} \end{cases} \quad (2.15)$$

where ε_j^L in the error in output neuron j and f' is the derivative of the activation function f.

After all ΔW in the network are computed, the connection weights in each layer are updated by the formula of

$$W_{i,j}^l(t+1) = W_{i,j}^l(t) - \Delta W_{i,j}^l(t). \quad (2.16)$$

The frequency of updating of network weights varies. Some well-known models are:

Online learning: In this model, the process described above is applied to every single training example in the data. The weights are updated immediately after the ΔW values are computed.

Batch learning: In this model, the ΔWs are computed using the same connection weights for all examples in the training set. The weights are updated using the accumulated ΔWs [41].

Mini batch learning: In this model, the data is divided into small batches. The ΔWs are computed using the same weights for all examples in a batch. The weights are updated using the accumulated ΔWs for the batch [42]. The examples in the next batch are processed using the updated weights.

An iteration is defined as the completion of each pair of the forward- and back-propagation. This means that an iteration is completed after each weight update. An epoch is defined as the weight update phases are completed for all examples in the data.

2.3 Improvements of Artificial Neural Networks

Some researchers reported that multi-hidden-layer networks outperformed to single-hidden-layer networks [43, 44, 45]. However, deeper networks were not considered much in the literature because they could not be easily trained using the back-propagation algorithm until Hinton [46, 47] showed that deep neural networks could successfully be trained. Since then a variety of deep and shallow ANNs have been widely applied to such data as images, signals and text for a long time [20, 21, 22]. However, the original SBP algorithm can be trapped by local optima of the error

functions. It is very slow and can easily overfit.

Various techniques have been proposed in the literature to improve the situation. Some common themes in the literature are the following: setting the initial weights [48, 49, 50, 51, 52, 30, 53, 54, 55, 56], choosing optimal network architectures [57, 58, 59, 60, 61, 62, 63], overcoming the vanishing gradient problem [64, 39, 65, 66, 67], avoiding overfitting [68, 69] and other speed-up techniques such as learning rate optimisation and optimal learning rule [70, 71, 72, 73, 74, 75, 76, 77, 78, 79].

This thesis will focus on back-propagation learning rules, weight initialisation methods and the vanishing gradient problem of deep neural networks.

2.3.1 Back-propagation Learning Rules

A significant number of gradient descent optimisation algorithms have been introduced [80]. Some of these algorithms are widely used to train neural networks. The use of a momentum term [12] is one of the first and best-known methods to accelerate the convergence of the back-propagation algorithm. It controls the step size of the current weight change by adding a proportion of the previous weight change. As long as the weight change is in the same direction, the step size grows since the momentum is accumulated. This accelerates learning by reducing the oscillations during the training.

Silva and Almedia [70] proposed applying an independent learning rate to each weight. The proposed algorithm has two different coefficients for the learning rate, which are d and u. Their ranges are given by $0 < d < 1 < u$. The algorithm increases the learning rate exponentially if the previous weight change and current weight change in a particular connection have the same sign (the same direction). If they have different signs, the algorithm decreases the learning rate exponentially. The current learning rate is computed by multiplying the previous learning rate by the appropriate coefficient u and d, respectively. The algorithm is applied to all connection weights separately. Resilient back-propagation (Rprop) [71] uses a similar approach. It has two different learning rates which are η^- and η^+. These are constant and chosen from the range of $\eta^- < 1 < \eta^+$. The algorithm independently decides which learning rate will be used for every single weight by considering signs of the current and previous weight changes. If the sign of the current and previous weight changes are the same, that connection uses η^+ as the learning rate. If those signs are different, the connection uses η^- as the learning rate. These are the early studies on which use adaptive learning rate instead of a constant learning rate. The authors reported that this approach accelerates learning.

In the last decade, several adaptive methods have been introduced to optimise learning rule parameters and have become quite common. The Adagrad [81] algorithm improves the performance of networks by adaptively scaling the learning rate

set at the beginning of training. However, learning slows down or stops over time since the algorithm reduces the learning rate by taking into account all past squared gradients in every single epoch. The Adadelta [75] and RMSprop [76] algorithms were proposed to overcome the vanishing learning rate problem of Adagrad algorithm. Adadelta limits the size of accumulated past gradients. This prevents the learning from slowing down and allows the network to continue learning even after many epochs. RMSprop scales the learning rate for each weight by the root mean square of the past gradients. This is approximated via an exponential moving average filter. Adam [77] is another popular learning algorithm that adaptively scales the learning rate. This algorithm computes learning rates for each weight depending on two parameters, namely first and second moments of the gradients of the weights.

Researchers still propose alternative adaptive gradient descent algorithms to improve over these methods, such as Nadam and NadaMax [82]. However, Wilson *et al.*[83] compared the gradient descent and stochastic gradient descent (SGD) algorithms with the adaptive methods including Adagrad, RMSprop and Adam on some of the benchmark problems used in deep learning and reported that gradient descent and SGD provided higher generalisation than adaptive methods.

2.3.2 Weight Initialisation

The performance of neural networks is very sensitive to initial conditions. If the same initial value is used for all weights of a network, it leads the network to produce the same outputs in the neurons, thereby making learning difficult [84]. This is known as the symmetry problem. Preventing the network from this situation by initialising it at random is called as symmetry breaking [54]. Most networks are initialised randomly. Fahlman [85] proposed to draw weights uniformly from the intervals $[-1,\ 1]$ and $[-2,\ 2]$. He reported that small changes in the ranges do not make the results very different. Another random initialisation method was proposed by Nguyen and Widrow [48] (INITNW) where weights are initialised using uniformly at random in the interval $[-0.7h^{1/n},\ 0.7h^{1/n}]$ where h is the number of hidden neurons and n is the number of input neurons.

Boers and Kuiper [86] argued that the net input of neurons should be prevented from driving the activation of a neuron in areas where the slope of the activation function is small. Because they used the logistic function, they determined that net inputs should be in the interval $[-3,\ 3]$. They proposed to initialise the weights in the interval $[-3/\sqrt{n},\ 3/\sqrt{n}]$ since this can always guarantee the inputs of neurons to be in the large slope of the activation function when all input values are assumed to be in the range of $[0,\ 1]$. This interval is the same as the that proposed by Wessels and Barnard [49], who also proposed a more refined method to avoid false local minima which were defined as being far from optimal classification performance.

The proposed approach separately initialises the set of weights of different the layers.

Drago and Ridella [87] (SCAWI) argue that (1) a neuron is saturated when its output magnitude is greater than a threshold (0.9) and; (2) a neuron is paralysed when it is saturated, and the magnitude of the error associated with its output is greater than a threshold (0.9). They determined the initialisation range by considering the percentage of paralysed neurons. According to statistical analysis, they proposed to initialise the weights between the input and hidden layers using the uniformly distributed interval $[-1.3/\sqrt{1+n_{input} \cdot v^2},\ 1.3/\sqrt{1+n_{input} \cdot v^2}]$ where v is mean of the normalised input values and initialise the weights between hidden and the output layers using the range $[-1.3/\sqrt{1+n_{hidden}},\ 1.3/\sqrt{1+n_{hidden}}\ ]$. This method and the INITNW were developed when scaling the input data to the interval $[-1,\ 1]$ and using the tanh activation function.

Thimm and Fiesler [84] analysed the results of various experiments with three different activation functions (linear, tanh, and shifted/scaled tanh) on multilayer and high-order perceptrons. High-order perceptron used in this work is a single layer perceptron that can solve non-linearly separable problems. It has an additional interlayer. The nodes in this layer multiply the input signals and transfer the resulting values to the output layer. They used eight benchmark problems to test the method and reported that the optimal initialisation method is a random initialisation within the very small interval $[-0.017,\ 0.017]$ for high order perceptrons with that use logistic activation function. When the tanh is used as an activation function, the interval $[-\alpha/\sqrt{n},\ \alpha/\sqrt{n}]$ provides the best results where α is the optimal weight variance which is approximately 0.8 for the tanh.

Similarly to the method of Boers and Kuiper mentioned before, Yam and Chow [88], Adam *et al.* [54], and Bhatia *et al.* [56] emphasise that the outputs of neurons should be in the active region of the activation function to avoid premature saturation by maximising the derivative of neurons. Yam and Chow determined the active region in a larger interval than the one proposed by Boers and Kuiper. It is $[-4.36,\ 4.36]$ for the logistic activation function. As a geometric result of this region, they determined the following formulas to compute the minimum and maximum weights — $-W_{max}$ and W_{max} — of each layer layer:

$$W_{max}^{hidden} = \frac{15.1}{h},$$

$$W_{max}^{input} = \frac{8.72}{D^{input}} \sqrt{\frac{3}{n}}.$$

Here D^{input} is the maximum distance in the input space and is computed by

$$D^{input} = \sqrt{\sum_{i=1}^{n}[max(x_i) - min(x_i)]^2}$$

where x_i is i^{th}, feature of the input data and, once again, h is the number of hidden neurons.

The active region of the sigmoid activation function determined by Adam *et al.* [54] is where the derivative of the function is greater than 0.04 or 0.05. Initially, the input variables normalised in the range of $[0,\ 1]$ or $[-1,\ 1]$ are re-scaled using some statistical values (e.g., min, max and q-quartile). Then, the network is initialised. They proposed different formulas or intervals for the biases and weights between (1) the input and hidden layers; (2) the hidden layers; (3) the hidden and output layers. The interval for the weights between the hidden and output layers ($[-3A/\sqrt{n},\ 3A/\sqrt{n}]$) is the same as that proposed by Boers and Kuiper, and Wessels and Barnard when A is equal to 1. In this study, A may be greater than 1 depending on the number of neurons in the (last) hidden layer and the activation function in order to prevent the corresponding weights from being in a very small range.

Bhatia *et al.* [56] also proposed the same interval that was proposed by Boers and Kuiper, and Wessels and Barnard for weight initialisation which is $[-3/\sqrt{n},\ 3/\sqrt{n}]$. The main difference of this work is that the biases are set to appropriate value to make the net input of neurons zero. This ensures that the derivatives in the neurons are maximum, that is that the outputs of the neurons are in the most active region of the activation function. This method was compared only against randomly initialised networks using ten prediction problems.

Of course, there are many other weight initialisation methods for neural networks in the literature [89, 90, 91, 92, 93, 94]. However, in this section, we reviewed only some famous early works and some recent works in detail. The majority of these works were tested on one specific and/or limited number of problems. Another common shortcoming of these methods is that they were tested on shallow networks.

Training deep networks was very difficult because of the vanishing gradient problem. This problem has been a research subject for a long time. In the second half of the 2000s, deep networks were started to be trained using various methods such as pre-training, use of different activation function, proper weight initialisation. These methods, including weight initialisation methods, will be reviewed in the next section.

2.3.3 Methods for Vanishing Gradient Problem

Hinton *et al.* [46] showed that a deep belief network (DBN) could be trained using a greedy layer-wise pre-training method. After this achievement, deep learning became a popular machine learning technique [2, 95]. Nonetheless, deep neural networks suffer from the vanishing gradient problem [40, 96, 97], which is a strong obstacle to training the networks.

This problem usually happens when a sigmoidal activation function, such as the logistic and tanh functions, is used in a deep network. The vanishing gradient problem is that the gradients in the weights (usually in the earlier layers of a deep network) is very small or zero. As previously described, the network error is proportioned and back-propagated to the weights in the earlier layers using gradient descent. Each weight is updated with a fraction of the derivative of the error with respect to the weight. If the network is relatively deep and the activation function is sigmoidal, the gradients in the earlier layers are more likely to be smaller than those in the later layers. They may sometimes be very small depending on the hyper-parameters of the network, such as the architecture, the activation function used and the initial weights. In such cases, the network does not learn the training set because the weights are updated with tiny changes preventing those layers from being trained.

There are a small number of techniques to cope with this problem in the literature. The use of ReLU based activation functions is an effective way to avoid this problem. However, they suffer from other problems such as dying neurons, bias shift and noise sensitiveness. Therefore, if the vanishing gradient problem of sigmoidal activation functions would be overcome, networks with such activation function could be more noise-robust and provide more stable learning than those which use other activation functions.

The techniques proposed to overcome the vanishing gradient problem will be reviewed below.

2.3.3.1 Pre-training

Bengio *et al.* [52] proposed unsupervised greedy layer-wise pre-training. This method initially trains the first hidden layer and then adds a new hidden layer. The outputs of the pre-trained layer are used as inputs for the added hidden layer. The process of adding and pre-training layers continues until all hidden layers in the architecture have been processed. At the end of pre-training, the algorithm adds an output layer and trains the whole network. They reported that this method improves not only the network performance but also generalisation. Sagheer and Kotb [98] proposed a similar method for time-series prediction. In this work, a Long Short

Term Memory (LSTM) based auto-encoder (AE) is used to initialise the weights. Neither of these works directly addresses the vanishing gradient problem, but they both can successfully train deep (up to three-hidden-layer) networks. The drawback of these methods is that it is computationally expensive as the algorithms train the weights in each layer separately.

2.3.3.2 Use of an Alternative Activation Function

Another way for deep networks to be trained without the vanishing gradient problem is the use of non-sigmoidal activation functions such as ReLU [39] and its derivations. ReLU function directly transfers the net input of a neuron to the output when the net input is positive; otherwise, it transfers 0 to the output. Because its derivative is 1 when the input is positive, the error can be propagated to the weights in the earlier layers through the active neurons without getting smaller. Therefore, ReLU does not suffer from the vanishing gradient problem as much as sigmoidal activation functions. However, it has some other drawbacks.

If there is a large gradient for a connection weight, this may cause the corresponding neuron to die when the weight is updated. A neuron with ReLU activation is dead if the net input of the neuron is always negative as this implies that the activation is zero and so is its derivative. A dying neuron can never be active again. If the number of dying neurons increases, the network may suffer from the vanishing gradient problem since inactive neurons do not have gradients and cannot transfer the information in either forward or backward directions.

Another drawback of ReLU is that non-zero-mean activation functions lead to the bias shift problem [96, 97, 66]. When the activation function of a network is ReLU, the mean of the outputs in the layers is larger than zero. Therefore, layer by layer, the outputs grow bigger and bigger. This causes oscillation in the learning. This problem is called the bias shift or mean shift problem in the literature.

To overcome such ReLU disadvantages, other ReLU versions have been introduced, such as the LReLU [40] and the PReLU [33]. The LReLU and PReLU provide non-zero activations in the negative interval by replacing the constant part of ReLU with a linear function, but with a very small slope. The slope of the line (α) for the LReLU is a predefined constant value. Its default value is 0.01. For the PReLU, the α is a parameter which is learnt by the network. These functions do not have the dying neuron problem. However, Clevert *et al.* [96] reported that these activation functions are not robust to noise.

The Exponential Linear Unit (ELU) [96] and Hyperbolic Linear Unit (HLU) [97] were proposed to overcome this sensitivity to noise and the bias shift problem. The ELU and HLU use the same function as the other ReLU family functions in the positive interval. Their difference from the other ReLU family function is that they

both have a non-linear function in the negative interval. Authors reported that these functions not only allow neurons to have negative activation but also are more robust to noise than the others. Also, they reported that these achievements allow networks to learn faster than others.

Rectified Linear Hyperbolic Tangent (ReLtanh) [66] is another activation function recently introduced that addresses the vanishing gradient problem and some of the drawbacks of the ReLU family functions. This function uses the non-linear tanh between positive (λ^+) and negative (λ^-) thresholds. It has straight lines on both sides. These lines are defined between the corresponding $\lambda^\pm$ and $\pm\infty$. The designer determines the slopes of the straight lines and their start points.

The more formal expression of these activation functions are follows:

PReLU is

$$f(x) = \begin{cases} x & \text{if } x > 0, \\ \alpha x & \text{if } x \leq 0, \end{cases} \tag{2.17}$$

LReLU is

$$f(x) = \begin{cases} x & \text{if } x > 0, \\ \alpha x & \text{if } x \leq 0, \end{cases} \tag{2.18}$$

ELU is

$$f(x) = \begin{cases} x & \text{if } x > 0, \\ \alpha(e^{-x} - 1) & \text{if } x \leq 0, \end{cases} \tag{2.19}$$

HLU is

$$f(x) = \begin{cases} x & \text{if } x > 0, \\ \alpha x/(1-x) & \text{if } x \leq 0, \end{cases} \tag{2.20}$$

ReLtanh is

$$f(x) = \begin{cases} \tanh'(\lambda^+)(x - \lambda^+) & \text{if } x \geq \lambda^+, \\ \tanh(x) & \text{if } \lambda^- < x < \lambda^+, \\ \tanh'(\lambda^-)(x - \lambda^-) & \text{if } x \leq \lambda^-. \end{cases} \tag{2.21}$$

In Equation (2.17) - (2.21), x is net input of the neuron. α is a negative slope parameter which is learnt by the network in Equation (2.17), and it is a predefined negative slope coefficient in Equation (2.18) - (2.20).

The common disadvantage of the ReLU based activation functions is that none of them can be used for the output layer as a classifier. In other words, they can only be used for hidden layers.

ReLU function has some alternatives such as PReLU, LRelu and ELU functions. Although these functions overcome the dying neuron and bias shift problems of

ReLU function, ReLU is still the most popular activation function in recent years.

2.3.3.3 Residual Network

He *et al.* [95] addressed the difficulty of training deep networks and proposed Residual Network (ResNet) to cope with the vanishing gradient problem. The proposed model is based on shortcut connections that jump over one or more layers. For example, the output values in layer l are added to the output values of layer $l+k$. Here, k is a small number. There are many shortcut connections on the networks from the input layer to the output one. According to experimental results, the ResNets provide higher accuracy and lower error than plain networks. However, this method was tested using the ReLU activation function. Because this method has not been tested using the sigmoid or tanh activation function, it is hard to say if the ResNet is a general solution to the vanishing gradient problem.

2.3.3.4 Weight Initialisation

Weight initialisation is another technique that can overcome the vanishing gradient problem. As described in Section 2.3.2, studies on this subject began approximately three decades ago. They usually aimed to accelerate the training of networks. Methods were tested on shallow networks. After the interest in deep networks increased, weight initialisation approaches have also been used for deep networks with the aim of reducing the impact of the vanishing gradient problem.

Glorot and Bengio [30] studied the reasons for the difficulty of training deep neural networks. They analysed and compared the logistic, tanh and soft-sign [99], $x/(1+|x|)$, activation functions. Moreover, they introduced a new weight initialisation interval for deep MLPs. The argument was that the error gradients push the weights to zero because the outputs of the hidden units are pushed to zero. This is good for the tanh and soft-sign because around zero activation values can be propagated backwards with these activation functions. This is useless for the logistic function because the outputs in the saturation region do not allow the errors to be back-propagated.

The proposed initialisation method in [30] is $W^l \sim \mathcal{N}(0,\ \sigma^2)$. Here, σ is $\sqrt{2/(n^l+n^{l+1})}$ where n^l is the number of neurons in layer l. There is another version of this method which uses a uniform distribution, in the interval $[-\sqrt{6}/\sqrt{n^l+n^{l+1}},\ \sqrt{6}/\sqrt{n^l+n^{l+1}}]$. According to the experimental results of [30], 5-hidden-layer networks which use the logistic activation function can never learn the training sets while 4-hidden-layer ones tend to learn the training sets. However, 4-hidden-layer networks with the logistic activation perform worse than the other networks which use tanh or soft-sign activation function.

Kumar [31] studied the problem of how one should initialise the weights in a network in such a way that the variances of the outputs of neurons in all layers are identical. Naturally the answer depends on the activation function used. He studied three activation functions logistic, tanh and ReLU. The analysis showed that σ=3.6/ $\sqrt{n^l}$ produces the desired effect when the activation function is the logistic.

Mishkin and Matas [32] proposed a method and tested it against four different methods using the logistic, tanh, ReLU and LReLU activation functions. The method performs a pre-training stage followed by the normalisation of the variance of the outputs in each layer. Performance of those methods varied depending on the activation function used. However, no method was able to train deep networks when the activation function was the logistic.

He *et al.* [33] introduced a new initialisation method for deep networks using ReLU and PReLU activation functions. The method is an extension of the method proposed by Glorot and Bengio [30]. It initialises the weights using $W^l \sim \mathcal{N}(0,\ \sigma^2)$. Here, σ is $\sqrt{2/(n^l)}$ and, once again, n^l is the number of neurons in layer l. This initialisation method is one of the most commonly used for deep networks. However, this method does not work well when the networks use sigmoid activation functions.

Since ReLU and its other alternatives made deep networks trainable, this achievement increased interest in deep learning. As a result of this, most neural network research, including studies on the weight initialisation, consider deep networks that use such activation functions. Because this thesis is concerned with the use of the logistic activation function, other initialisation techniques which use non-sigmoid activation functions will not be considered here.

2.4 Genetic Programming

EAs include several different population-based optimisation techniques such as Evolutionary Programming (EP) [100], Evolution Strategies (ESs) [101, 102], GAs [103] and GP [17]. These methods use similar principles and tools that take inspiration from nature such as natural selection, creating offspring and undergoing mutation.

Natural selection is that being prefered the individuals that are well-adapted to the environmental conditions for reproduction. This means that the individuals which are fitter than the others to the environment condition have more chance to create offsprings and others die without having offsprings over time [104]. This is known as the Darwinian principle of survival of the fittest [105, 104]. All EAs, including those listed above, use this principle. Although the original algorithms are different from one another, it can sometimes be difficult to classify newer ones since an algorithm can also be a hybrid of multiple others [106]. In this thesis, GP is used to evolve learning and initialising rules for MLPs.

One of the first studies for evolving algorithms was done by Cramer [18] in 1985. The author showed that GAs with a tree-based structure could produce simple computer functions. However, GA, in that study, needed to predefine the size of computer programs. This is a severe obstacle for the algorithm to search the space freely. John Koza [17, 19] extended this approach, and in the early 1990s, introduced GP that allows the size and shape of computer programs to be varying dynamically. GP is an evolutionary program-induction technique that has produced numerous human-competitive results in the last two decades. A particular feature of GP is that it can not only solve problems but also discover *problem solvers*, i.e., algorithms that can solve a specific class of problem [107, 108, 109].

Because discovering/developing new and efficient algorithms by manual trial-and-error methods often requires mathematical principles, engineering knowledge, and creativity, GP may achieve this faster than manual trial-and-error methods. Therefore, in this thesis GP is used to discover learning and initialising rules for MLPs.

A GP has a population of computer programs. These programs are also called individuals that is the general term of the possible solutions in EAs. GP has two sets of primitive elements called the 'terminal' and 'function' sets. These sets are used to form and mutate the individuals in the population. The terminal set consists of variables and constants that are used as an input such as x, y, 1 and 2 while the function set consists of functions such as arithmetic, logical, mathematical and conditional such as $+$, *AND*, *sin* and if. Elements in the terminal and functions sets are chosen depending on the problem. The population is randomly initialised by choosing operands and operators from the terminal and function sets. Then, GP, repeatedly, selects one or more individual(s) that are fitter than others among a group of individuals and applies the crossover and/or mutation operators to them to evolve the population [110]. These operators, crossover and mutation, are used to generate new individuals from selected parents.

A well-known form of GP is the tree-based one. However, over time different versions of GP have been introduced such as grammar-based GP, linear GP and Cartesian GP (CGP) [111, 112, 113]. The cycle of the evolutionary process is the same in all GP types. However, different GP system differ depending on the representation used for individuals.

Figure 2.4 shows the cycle of an evolutionary process. The process starts by generating the initial populations at random in step 1. Then, in step 2, the individuals are measured how close they are to the problem's solution. Step 3 sets generation number to 1. Step 4 checks if the algorithm meets any stopping criteria. This can be, for example, by checking whether any programs (individuals) provide an acceptable solution for the problem or whether the generation number reaches the maximum

number of generations predefined. If it does, the algorithm returns the best individual; otherwise, the algorithm passes the next step to generate new individuals. Step 5 selects one or more individuals as parents based on a selection method. Step 6 creates offsprings by applying the crossover and/or mutation operators to the selected parents based on a predefined probability criterion of applying the genetic operators. Step 7 measures the fitness of offsprings. Then, step 8 replaces the offsprings with lower-fitness-individuals among a group of selected individuals from the population. The next step updates the generation number. Then, the algorithm passes the step 4 as a final step of each generation and check whether the stopping criteria are met.

Simple examples for the representation of an individual in a linear GP [114, 115] and a tree-based GP [17] are shown in Figure 2.5. Different representations require different operators. For example, the crossover and mutation operators swap/change selected instructions of the parents in linear GP while they swap/change the selected nodes (sub-trees or leaves) of the parents in a tree-based one. Since a register-based linear GP is used in this thesis, only this type will be described, and the GP term will refer to the register-based linear GP hereafter.

2.4.1 Representation

Individuals (programs) used in a linear GP are represented by sequences of instructions, as shown in Figure 2.5. Instructions, in a register-based linear GP, read their inputs from one or more constants(c) and/or registers(r) and assign the resulting values to a register such as $r_i = r_i + c_k$, $r_j = r_i * r_j$, and $r_k = r_i * c_j$ where i, j and k are indexes of the register and constant [111, 114]. The output of an individual is read from a predefined register after all instructions in the individual have been sequentially executed.

A register based linear GP has a set of instructions. It is predefined by the user. The table in Figure 2.6 shows an example set of instructions (from [116]). In the example, four registers are used: `r0` and `r1` are used for mathematical calculations; `rs` is used for swapping the registers; `ri` is the input register. The numbers in the first column of the table are constants, and `NOP` means no-operations. The number of input registers differs depending on how many inputs need to be used for the problem. The initial values of registers are defined by the user (but are typically zero).

Figure 2.6-(b) shows an example GP program (individual) which uses the set of instructions in Figure 2.6-(a). If the output of the program is read from the register `r0`, the result will be `2ri`; if the output of the program is read from the register `r1`, the result will be `ri`.

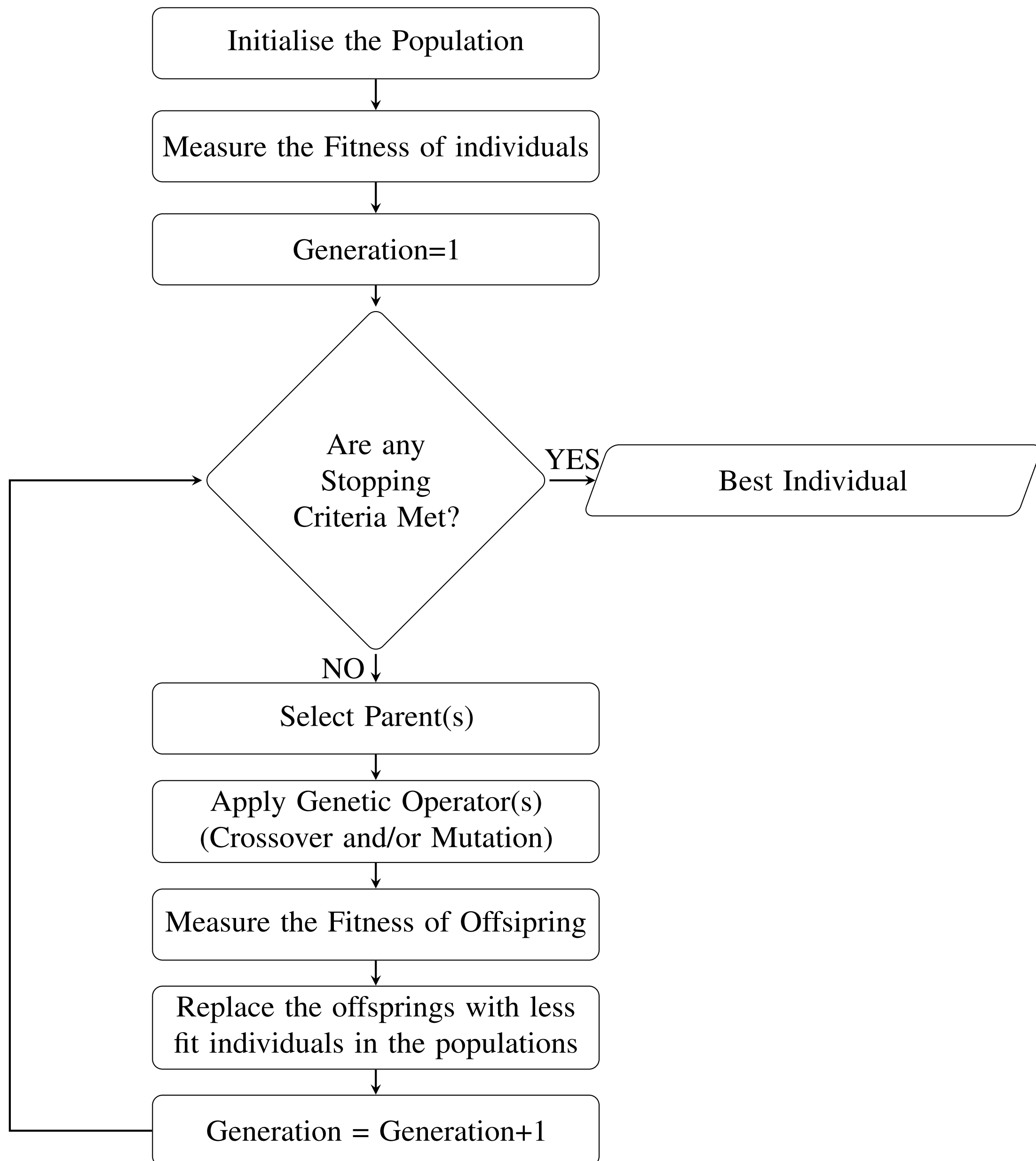
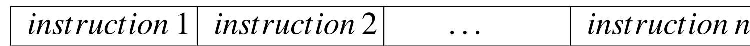


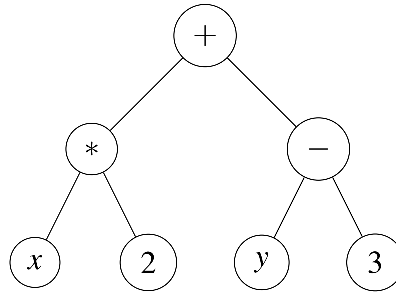
Figure 2.4: Flowchart of GP

2.4.2 Initialisation of the Population

Individuals are usually randomly initialised in population-based optimisation algorithms. Over time, different initialisation methods have been introduced. These methods vary depending on the GP type used. There are various initialisation forms for linear GP including free, variable-length, constant-length and maximum initialisation methods [115]. It is hard to say which method is best as their performance may vary depending on the comparison conditions. In general, according to Brameier and Banzhaf [115], too-large initial lengths may cause the programs to be less flexible while too-small ones may lead most programs in the population to be identical.



(a) Linear GP (from [111])



(b) Tree-based GP

Figure 2.5: Examples of the representation used in a linear GP (a) and a tree based GP (b)

NOP	r0 <- -1	r1 <- r0 + r1
r0 <- 0	r1 <- 1	r0 <- r0 * r1
r1 <- 0	r0 <- -r0	r1 <- r0 * r1
r0 <- 0.5	r1 <- -r1	r0 <- r0 * r0
r1 <- -0.5	r0 <- r0 + ri	r1 <- r1 * r1
r0 <- -0.1	r1 <- r1 + ri	rs <-> r0
r1 <- 0.1	r0 <- r0 + r1	rs <-> r1

(a) An example for set of instructions (from [116])

r0 <- 0	r1 <- 0	r0 <- r0 + ri	NOP	r1 <- r1 + ri	r0 <- r1 + ri

(b) An example program

Figure 2.6: Examples of a primitive set in linear GP (a) and a program which uses the primitive set (b).

2.4.3 Fitness Function

The primitive set defines the search space of GP. In order for GP to search the solution of the problem in good regions of the space, it needs to know which regions are good or bad [111]. The fitness function measures how close individuals in the population are to the solution.

The fitness function is one of the crucial elements for GP to solve problems. Many notions of performance can be used for the design of the fitness function such as the error (the difference between the desired output and actual output), time cost, and the complexity of the program. It can be a simple function that uses a single criterion or a sophisticated function that uses multiple criteria.

2.4.4 Selection

EAs need a selection operator to choose individuals that will undergo mutation and/or be crossed-over. There are many different selection operators in the literature, such as random pairing, roulette wheel and tournament selection, to name a few [117].

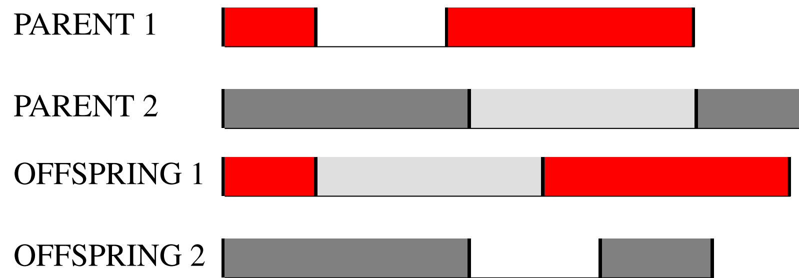


Figure 2.7: Linear GP crossover (from [111])

In GP, tournament selection is the most commonly employed operator due to its scalability and ease of use [111].

In tournament selection a group of individuals are randomly chosen from the population. The one which has the best fitness is selected as a parent. When the genetic operator is the mutation, it is applied to the chosen parent. When the operator is the crossover, the selection process is repeated one more time as crossover requires two parents [111].

2.4.5 Crossover and Mutation

The crossover operator is used for generating offspring from the parents in EAs. Since the parents are selected based on the fitness, it is expected for crossover to generate fitter offspring, which will be used as individuals for the next generation. The main task of mutation is to keep diversity in the population by randomly changing elements in selected individuals.

Various crossover and mutation methods are available in the literature [118]. Figure 2.7 illustrates a typical linear GP crossover. In each parent, two instructions are randomly selected. The instructions between the selected ones are excised from parents. The parents exchange the code blocks excised as shown in the figure.

The standard mutation operator, in linear GP, chooses an instruction from the individual being mutated and then replaces a component of it (an operator, a register or a constant) into another from the appropriate set [119]. This is called micro mutation. The other mutation type is macro mutation. In this type, an instruction is either deleted from the individual or inserted into the individual [115].

The population size over the generation is usually kept the same. This means that offsprings are not directly inserted into the population. Instead, they are replaced with selected individuals from the population. In other words, an individual is selected from the population for each offspring, and the selected individual is permanently discarded from the population; then, the offspring is inserted into the population. In this process, usually, worse fit individuals are selected to be discarded while fitter ones are kept in the population. This is known as elitism in the EAs [104].

2.4.6 Summary of GP Process

GP is one of the most commonly used population-based EAs. GP is a program-induction technique. It can not only solve problems, but it also produces problem-solvers for a specific class of problems like computer programs and algorithms. A GP has a population that includes possible solutions to the problem. These are initially created at random using appropriate elements such as variables, constants, arithmetic operators, logical operators, conditions for the problem. Fitness of programs to the problem solution is measured by a fitness function. Based on this fitnesses, the population is evolved by generating new programs (offsprings) from fitter programs in the population over the generations. GP, like the other EAs, cannot guarantee the optimal solution but it provides the optimal or near-optimal solution to the problem at the end of the evolution process.

In this thesis, GP is used to evolve learning and initialisation rules for neural networks. In the next section, the evolution of neural networks will be reviewed.

2.5 Integrating Neural Networks and EAs

As explained in Section 2.2.3, a supervised neural network aims to produce correct outputs for the corresponding inputs shown during training [120]. To achieve this, the network weights are usually optimised by error back-propagation using the gradient descent principle. However, there are some drawbacks of this algorithm such as trapping by local minimal, slow learning and over-fitting. In order to overcome these problems, many methods have been proposed. In addition, the evolution of neural networks is a frequently used approach to reduce the impact of such problems and improve neural networks.

EAs can be applied to the neural networks for many purposes. They are mostly used with the intent of weight optimisation, architecture optimisation and learning rule evolution. We will look at each of these areas in the following sections.

2.5.1 Evolving Neural Network Weights

All methods that optimise neural network weights, including both back-propagation and EAs, aim to minimise the network error. The error is usually computed by considering the difference between the actual output and the desired output across a training set. Connection weights optimisation is a search problem. Therefore, EAs may be a good way to find optimal or near-optimal weights for the network [121].

In this approach, mostly two types of representations, bit-string [122, 123, 124] and real-valued [125, 126, 127, 128, 129, 130, 131, 132], are used for the weights. In both models, all connection weights and biases of a network in a vector form

constitute a chromosome. In the bit-string representation, each weight is represented by a certain predefined number of bits; and it is easy for the algorithm to apply the standard crossover and mutation operators for binary strings to the individuals. However, the standard crossover operator may be inappropriate for evolving weights or architectures of MLPs because it easily disrupts the structures or connection distribution [133].

In the real-valued representation, each weight is represented by its real value (within the limits of the floating-point representations) in a chromosome. The binary crossover and mutation operators cannot be applied to real-valued chromosomes. Therefore, different forms of those operators have been developed [134, 135, 136, 117]. Most efforts to evolve connection weights have used mutation-based EAs (i.e., EP and ESs). Mutation-based methods avoid the competing-conventions problem which is also known as the permutation problem [137] encountered in neuro-evolution [133]. This problem can be defined as genotypically different networks being phenotypically the same. The algorithm suffers from this problem when the number of representations (genotypes) that lead to the same neural networks (phenotype) increase [137].

The weight evolution methods mentioned above are also known as gradient-free methods since the weights are optimised by an EA without using the knowledge of the gradient for a network. The weights evolved by an EA are more likely to be more optimal than those optimised by back-propagation since EAs search the space more globally [133, 138]. However, using the direct encoding described above, it is challenging for EAs to be applied to large-size networks such as *deep neural networks*, since the evolution of many connections requires a long computational time [133, 139]. There are few studies done to evolve the weights of such networks [140]. We will review them below.

David and Greental [141] evolved a fixed-sized deep autoencoder using a real-valued direct encoding. The network had four hidden layers. So, it had five sets of weights. The GA optimises the weights layer-by-layer. It firstly evolves the weights between the input layer and the first hidden layer. After fixing the optimised weights, it starts to evolve the next set of weights. This process is repeated until the last set of weights. The back-propagation is applied to the network to fine-tune the weights after the evolution of all weights in the network has been completed. This approach is relatively cheap in terms of computational time because it avoids long chromosome size thanks to layer-by-layer evolution.

In order to reduce the computational time of the evolution of deep networks, some researchers have divided the data into parts. Each part is called a mini-batch. Lander and Shang [142] proposed to evaluate each chromosome on a mini-batch of training data. Morse and Stanley [143] proposed to evaluate all chromosomes using

a mini-batch of training samples in each generation instead of using whole training data.

One of the most important advantages of connection weight evolution is that EAs search globally optimal weights of networks since they do not need an error gradient [128]. Therefore, EAs are less sensitive to initial weights and the networks optimised by this method are less likely to be trapped in local minima than those optimised by the gradient descent principle [133].

Although EAs globally search the connection weights of neural networks, this approach cannot always be guaranteed to find the optimal weights [140]. In order to search for a more optimal set of connection weights than that found by EAs, some researchers propose hybrid training methods which use both EAs and the back-propagation [141, 144, 145, 146, 147, 148, 149]. In this approach, the optimal weights of networks are locally searched using back-propagation after an EA fixes the weights at the end of the evolution process like the work by David and Greental mentioned above. From another point of view, it may be said that this method initialises the network weights to reduce the sensitivity of back-propagation to the initial conditions. Indeed, this hybrid method outperforms both EAs and back-propagation, as it combines the powers of both. However, in terms of computational cost, it is more expensive than either.

2.5.2 Evolving Neural Network Architecture

The architecture of a neural network is one of the most critical factors which affects network performance. On the one hand, a small network may not learn the data, because of insufficient degrees of freedom (weights, units). On the other hand, a large network may cause poor generalisation by over-fitting the training data. In short, to successfully train a network, it should have a good architecture. There are some questions to be answered before designing the architecture of a network such as (1) whether the network is fully connected; (2) if not so, how many connections there should be between the layers; (3) how many hidden layers there should be in the network and neurons on each layer. There is no simple rule or formula proposed so far to answer these questions and determine the relevant parameters. The design of a network structure is mostly based on the user experience and/or trial and error. However, there are some methods proposed to optimise network structures such as constructive (growing) [150, 151, 57], pruning [152] and EAs. These are described in more detail in the next two sections.

2.5.2.1 Constructive and Pruning Methods

Nicoletti *et al.* [153] and Sharma and Chandara [154] present a review in detail of some of the early works of the constructive and pruning methods. In the general working principle of constructive methods, a network starts training with a minimal architecture; then, its size is increased gradually during the training process. The stopping criteria of training and the maximum size of the network are defined beforehand. The training of the network starts usually with a single hidden layer and neuron. When convergence of the network error towards zero slows down or stops, the algorithm adds a new neuron to the current layer or adds a new hidden layer to the network. This process is repeated until the network reaches the stopping criteria or the predefined maximum size. New component(s) can also be added at every single epoch instead of waiting for the error to stop decreasing. Pruning methods work in the opposite direction to the constructive ones. It starts training a large network and deletes a hidden layer of the network and/or a neuron of the relevant layer at a time during training. These methods have also been applied to deep neural networks in recent years [155, 156, 157, 158].

2.5.2.2 Use of EAs to Evolve Neural Network Architecture

EAs have been one of the most commonly used methods to optimise neural network architectures since the 1990s [138, 133, 140]. Every single parameter of the EA used, such as population size, number of generations or mutation rate, affect the results of the algorithm. However, the representation scheme, genetic operators and EA used have the most important roles in determining how close the architecture found is to the optimal architecture.

Direct and indirect encodings are the most widely used representation schemes in this approach. In the direct encoding, chromosomes encode all features of networks such as the connections, neurons and activation functions. In the indirect encoding, chromosomes encode higher level information on the architectures, such as the number of hidden layers and neurons in each layer [138].

The connectivity of a network can be represented using a direct encoding scheme based on the traditional bit-string representation. Since layers in a network are connected by weight matrixes, the chromosomes encode a binary matrix for each weight matrix in the network. 1s in the chromosomes mean that there are connections between corresponding nodes; 0s in the chromosomes mean that there are no connections between such nodes. When the real weight values of connections are used lieu of 0s and 1s in the chromosomes, it is called real-valued representation, as explained in Section 2.5.1. The real-valued representation allows the algorithm to optimise the architecture *and* connection weights of the network simultaneously [133]. Both

the binary and real-valued representations with the direct encoding scheme have been employed to optimise neural network architectures. In the direct encoding, algorithms usually search the optimal design under the maximum size of a network defined by the user.

Pujol and Poli [159] improved the applicability of real-valued direct encoding scheme to neural networks. The authors propose a parallel distributed GP with new forms of genetic operators to optimise the architectures and weights of networks simultaneously. In this method, the networks are represented as graphs using linear chromosomes. The crossover operation is done by swapping sub-graph or transferring neurons. All chromosomes have the same length as the predefined maximum size of the network at the beginning of the evolution. The lengths are not changed during evolution; but, the network shapes may change as genetic operators change the connectivity of networks.

Khan *et al.* [160, 161] and Turner and Miller [162] showed that CGP was another efficient evolution technique to evolve the weights and architecture of the neural network. CGP is a form of graph-based GP developed by Miller and Thomson. In this approach, programs in the population consist of nodes which encode the input, activation function connectivity and weights. The neurons of the networks in the population do not have to be systematically connected with each other. Also, pairs of neurons in the networks may have multiple connection weights. This approach is described in detail in [161].

Stanley and Miikkulainen [163] developed an inventive method called 'NeuroEvolution of Augmenting Topologies (NEAT)' to evolve the network architecture and its weights simultaneously. The presented algorithm used the direct encoding scheme and real-valued connection weights. In this method, the chromosomes consist of two lists that contain node genes and connection-genes, respectively. Each node-gene keeps an innovation number of a neuron. When a new hidden neuron is created during evolution, an innovation number is created for it and the number inserted into the list of node-genes. A connection-gene involves every detail of the connection, including which node the connection comes from or goes to, what the value of the weight is, whether the connection is enabled and its innovation numbers. In this method, even if the selected parents do not have the same length, the crossover operator can be applied to them without suffering from the competing-conventions problem.

Direct encoding based algorithms work satisfactorily when the search space is not too big and the optimal network is in the space. Networks used in the present-day have billions of connections. Unfortunately, when a direct encoding method is applied to a large-size network, e.g., a deep network, it is difficult for the network to perform well because of the enormous search space and the correspondingly large

computational time [133, 138].

Indirect encoding schemes started to be used for the evolution of neural networks in the early 1990s [164, 165, 166]. The chromosomes with this encoding represent a coarser, higher-level, description of the network than direct encodings. Determining what are the important parameters that need encoding is challenging and may require experience. However, chromosome sizes in a direct encoding are usually much bigger than those that use indirect encoding; and thereby the evolution with the latter is computationally cheaper. Hence, the indirect encoding schemes may be a good choice to evolve large-size neural networks. For example, Sun *et al.* [167] evolved the initial weights and architectures of convolutional neural networks (CNNs) using an indirect encoding. In this method, each gene contains the type of a layer in the network (convolutional, pooling or fully connected), and summary information on the layer, such as the filter sizes, pooling type and two statistical measures (standard deviation and mean value) of the weights. The authors reported that this technique was superior to other image classifier methods they compared.

Stanley *et al.* [168] proposed a method called 'hypercube-based NEAT (HyperNEAT)' to evolve large scale neural networks. The method reduces the complexity of NEAT by employing the 'compositional pattern-producing networks (CPPN)' which is an indirect encoding. Locations of the nodes in all dimensions are mapped after assigning a coordinate between -1 and 1 to each of them. This is called substrate and is done by the user. The algorithm evolves the CPPN instead of directly evolving the network. The evolved CPPN determines the weights and connectivity between the nodes according to the position of the nodes in the substrate. The algorithm has also a constant threshold defined by the user that is used to decide whether the weight is used for the corresponding connection.

Verbancsics and Stanley [169] introduced an extended version of the HyperNEAT called 'HyperNEAT Link Expression Output (HyperNEAT-LEO)'. This method has one more output neuron in the CPPN that has a step function. It decides whether the value on the other output of the CPPN will be used as the weight for the corresponding connection or not. Another extension of the HyperNEAT is presented as 'Evolvable-Substrate HyperNEAT (ES-HyperNEAT)' [170]. This method differs from the other versions mentioned above as it evolves the number of nodes and their location in the substrate. Since the substrates evolved determines the network architecture, this algorithm no longer needs to evolve the network architecture through the connectivities of nodes.

Miikkulainen *et al.* [171] propose to apply a method called CoDeepNEAT, which had a similar representation to that used in the NEAT, to deep neural networks to evolve the network architectures. In the proposed approach the chromosomes encode lists of layer-hyper-parameters and global-hyper-parameters. Both lists

consist of boolean and real-valued variables. The algorithm evolves only the network architectures.

CNN is a widespread form of deep networks. The CNNs have been used for image or video recognition tasks in the last decade. Evolution of CNNs has become another research area in neuroevolution in recent years. Various methods have been proposed to evolve CNNs [140] such as GA with a binary string encoding [172], CGP with a graph representation [173, 174], GA with grammar-based encoding [175] and GA with a variable-length encoding [167].

Because this thesis is not focusing on the evolution of neural network architecture, this subject is not reviewed in detail.

The constructive and pruning approaches, in comparison with EAs, are faster but less effective than EAs. There may not always be a relation (linear or parabolic) between the network size and its performance. The optimal network may also be far from the starting point of the constructive method. Therefore, EAs may search the optimal network structure better than constructive methods thanks to their globally search feature.

2.5.3 Evolving Neural Network Learning Rules

Another critical element in a neural network is the learning rule used. In an MLP, the network weights are usually optimised by the back-propagation algorithm. The SBP algorithm, drawbacks of the original algorithm and well-known handcrafted methods, including learning rules proposed to overcome those drawbacks, were described in Sections 2.2 and 2.3. Most of the learning rules were manually crafted to carefully modulate the learning rate in such a way to accelerate the learning process. This process of discovering new rules is difficult and slow. Therefore, the evolution of learning rules was proposed as an alternative way to obtain effective learning algorithms in a reasonable time [176].

Bengio *et al.* [177, 25] demonstrated that a new learning rule, which can solve simple problems, can be evolved by optimisation algorithms. In the proposed methods, the learning rule is assumed to be linear combination of pre-and post-synaptic activations, current weight, etc. and their products. Each term has a coefficient. Then, the optimiser chosen evolves the coefficients. Earlier work [177] used GA; and later work [25] used the gradient descent, GA and Simulated Annealing (SA) [178] as the optimiser to evolve learning rules. Chalmers [26] presented a similar method by applying GA to single-layer neural networks. He defined the learning rule using the same terms as those used in the works of Bengio *et al.* (which were W_{ij}^{l-1}, a_i^{l-1}, a_j^l and $\sum_i W_{ji}^l \delta_i^{l+1}$ and their pairwise products). The GA optimised eleven coefficients (the first one was for the learning rate covering the

whole learning rule and others are for the terms of the learning rule) using 35-bit-length chromosomes. Some of the evolved rules in this work were variations of the delta rule [179]. Because the learning rules were evolved on single-layer networks with linearly separable problems, they were only tested on the same kind of networks and problems.

Fontanari and Meir [180] evolve a learning rule for boolean and linearly separable problems using the same approach as that proposed by Chalmers [26]. The presented method has seven parameters in the predefined parametric learning rule, which includes third-order terms too. The proposed method uses a real-valued representation for the chromosomes, but the coefficients are allowed to be only integer values.

Although these early works show that optimisation algorithms can successfully evolve learning rules for simple problems, these methods are not very effective for obtaining learning rules that can solve more complex problems. Evolved learning rules by this approach can only set the parameters of the predefined parametric rules. For instance, none of the rules evolved by the methods explained above [177, 25, 26, 180] can include a power of one of the variables because exponents are not defined in those parametric rules. Of course, it is not difficult to add the exponents or any other parameters to the parameter list. However, it is not easy to think of all possible composition of the parameters.

Bengio *et al.* and Chalmers reported that GP could be a better choice than the other optimisation algorithms to obtain effective learning rules [25, 26]. The significant advantage of GP is that it can design different rules, instead of optimising coefficients of a fixed parametric rule.

Baxter [181] evolved a learning rule, architecture and connections of a network simultaneously using GA for boolean induction problems. The weights are allowed to be one of the values in the set $\{-1, 0, +1\}$ while the learning rule can only produce the values of -1 or $+1$. This means that there is no connection with a zero weight. The proposed method may be computationally very expensive for complex problems.

Bengio *et al.* [27] evolved a relatively more complex learning rule by GP. GP uses the activations etc. of pre-synaptic and post-synaptic neurons shown above as its terminal set and uses four fundamental arithmetic operators as its function set. The evolved rule is the multiplication of the SBP by the square of derivative of the activation function with respect to the output of the post-synaptic neuron. It is more formally shown in the following equation:

$$\Delta W_{ij}^{l-1}(t) = a_i^{l-1} \delta_j^l f'(net_j^l)^2$$

They reported that the learning rule found by the GP outperformed the SBP and the rules found by the GA and SA. However, it was tested mostly on tasks of the same kind and the same network structures used in training. The networks structures used in training and testing are very small, having two input, one hidden and one output neurons. The authors used only the seven-segment problem to see performance of the evolved rule on a different task and network structure.

Radi and Poli [29, 23, 24, 121] extend the work of Bengio *et al.* [27] to obtain more general neural networks learning rules. They carried out their all experiments using three-layer neural networks and tree-based GP. The terminal set of GP consisted of W_{ij}^{l-1}, a_i^{l-1}, a_j^l, either $\sum_i W_{ji}^l \delta_i^{l+1}$ or y_j, and a few constant values. The terminal set includes $\sum_i W_{ji}^l \delta_i^{l+1}$ (error in the corresponding neuron) when the GP evolves rules for the hidden layer; and y_j (target value in the corresponding output neuron) when the GP evolves rules for the output layer. The constant values defined in an experiment were sometimes different from the ones used in another experiment. The function set is $\{-, +, \times\}$ and the same in all experiments. In their early works [28, 23], they evolved learning rules for only the output layer of the networks (connections between the hidden layer and output one). In both the evolution and testing phases, the delta rule was used for the hidden layer of the networks (weights between the hidden and input layers). They obtained the following learning rules:

$$Rule(1)^{output} = 2a_i^{l-1}(y_j^l - a_j^l) - y_j \tag{2.22}$$

and

$$Rule(2)^{output} = (y_j^l - a_j^l)(a_i^{l-1}(a_i^{l-1} + a_j^l) - y_j) \tag{2.23}$$

Radi and Poli [29], reported that those rules (1 and 2) had oscillating performance when they were tested on different data. Here, they used the same approach in previous works. The GP was used to discover learning rules only for the output layer. However, this experiment differed from the previous one since, in this work, SBP was used as the learning rule of the hidden layer. At the end of the evolution, the GP returned the delta rule as the evolved rule for the output layer. So, as a result, the authors propose to train networks using SBP rule for the hidden layer and delta rule for the output layer of the networks.

The authors extended this work by evolving learning rules for hidden layer while the rule previously evolved (the delta rule) was used for the output layer [176, 121]. They presented four different learning rules and reported that two of them are similar to each other and outperform the others. In this experiment, they considered two activation functions, the logistic and tanh. To make the rules found by GP be less sensitive to the activation function, they replaced the term of $a_j^{l+1}(1 - a_i^{l+1})$ in the rules with the derivative of the activation function used. The authors also manually

simplify the rules. The resulting forms of them are:

$$Rule(3)^{hidden} = \beta SBP + (1-\beta)HB \tag{2.24}$$

and

$$Rule(4)^{hidden} = SBP + \beta HB \tag{2.25}$$

where β is a constant and the HB is $(a_i^{l-1} - k1)(a_j^l - k2)$. Here one of the ks is zero while the other one is a constant.

They did further experiments [182, 24] with a similar approach. They used the step function as the activation function of the networks. They introduced three learning rules for each of the hidden and the output layers given by:

$$Rule(5)^{output} = \eta \left[0.1a_i^{l-1}(y_j - a_j^l) - 0.0005\right], \tag{2.26}$$

$$Rule(6)^{hidden} = \eta \left[0.11a_j^l - \left(0.1 - \sum_i W_{ji}^l \delta_i^{l+1}\right)(a_i^{l-1}a_j^l - a_i^{l-1})\right], \tag{2.27}$$

$$Rule(7)^{output} = \eta \left[0.1a_i^{l-1}(y_j - a_j^l) - 0.001\right], \tag{2.28}$$

$$Rule(8)^{hidden} = \eta \left[0.11 - \left(0.01 - \sum_i W_{ji}^l \delta_i^{l+1}\right)(a_i^{l-1}a_j^l - a_i^{l-1})\right] \tag{2.29}$$

$$Rule(9)^{output} = \eta \left[0.1(y_j - a_j^l)(a_i^{l-1} - 0.5)(a_i^{l-1} - 0.1)\right] \tag{2.30}$$

and

$$Rule(10)^{hidden} = \eta \left[(a_i^{l-1} - 0.51)\left(a_j^{l+1}\sum_i W_{ji}^l \delta_i^{l+1}(a_i^{l-1} - 0.1)\left(\sum_i W_{ji}^l \delta_i^{l+1} + 0.1\right)\right)\right]. \tag{2.31}$$

Radi and Poli did a substantial number of experiments to discover an ample space of learning rules, including rules where the hidden and output layers were trained differently. Experimental results indicated that this method can routinely produce learning rules that were faster than the SBP, and also rules that would work even with multi-layer networks with discontinuous activation function [24], which is a substantial achievement given that the SBP cannot train such networks. However, the generality of the evolved rules was tested on small network structures having

a single hidden layer with a limited number of problem instances including parity, multiplexer, 7-segment display, vowel recognition and sonar signal classification.

Risi and Stanley [183] extend the HyperNEAT approach, explained in Section 2.5.2, to obtain indirectly encoded learning rules. This method is called Adaptive HyperNEAT. In this method, the CNNP has three more input parameters which are W_{ij}^{l-1}, a_i^{l-1} and a_j^l as well as the locations of the neurons in the substrate. The output of the evolved CPPN provides the ΔW_{ij}^{l-1}. Here, the CPPN was used as a learning rule of an autonomous robot and performed well. However, this approach was not tested on other problem sets.

Runarsson and Jonsson [184] demonstrated a different approach to obtain a neural network learning rule. The authors evolved the weights and learning rate of an ANN using ES. The input layer of the ANN consists of the a_i^{l-1}, a_j^l and t_j of another neural network while the output neuron of the ANN is used as the ΔW_{ij}^{l-1} for the connection between the a_i^{l-1} and a_j^l. It is reported that the results are promising. However, this method was tested only on synthetic linearly separable problems, so it is unclear if it would work also on real-world problems that are, typically, not linearly separable.

Orchard and Wang [185, 186] used a similar approach to that used by Runarsson and Jonssons [184] to evolve learning rules. They applied the GA to three-layer neural networks to optimise the weights of the networks. In their early work [185], the ΔW_{ij} and ΔW_{0j} (the relevant delta weight and bias, respectively) are determined by different neural networks that are evolved. The input layer of the network consists of W_{ij}, a_i^{l-1}, $\sum_i W_{ji}$ and W_{0i} if the network is for weight update; $\sum_i W_{ij}^{l-1} a_i^{l-1}$ and W_{0i} if the network is for bias update. In the later work [186], only one network, which has two outputs, is evolved to determine the corresponding weight and bias. Its input layer consists of W_{ji}, a_i^{l-1}, a_j^l, W_{0j}^{l-1} and h_j^l (hidden factor) if applicable.

In these studies, the learning rules (weight and bias update networks) were specifically evolved to train networks that solved simulated foraging problems. The evolved training algorithms showed promising performance when tested on the reference task or even larger tasks of the same kind. However, it is hard to say whether the evolved learning rules are general since these rules were not tested outside the simulated foraging domain.

2.6 Conclusion

This chapter presented a review of improvements to the neural networks using theoretical models and EAs. ANNs with many different network architectures are among the most widely used machine learning techniques that can be applied to many tasks such as classification, regression, recognition and clustering. However,

neural networks still suffer from some problems such as slow learning, trapping by local minima and overfitting networks and difficulty training deep networks.

Since, in this thesis, we focussed on the evolution and theoretical analyses of learning and initialisation algorithms to accelerate the learning and to train deep networks, here, we reviewed mostly such improvement.

In order to speed up learning, researchers developed a variety of improvement methods such as momentum, Rprop, Adam optimiser and RMSProp. Even if these algorithms can also be considered a new learning rule, in fact, they only optimise/control the learning rate during the training. In order to develop new learning rules, some researchers emphasised the use of GP providing efficient evolved learning rules approximately two decades ago. Despite these early successes, somehow surprisingly, this approach went unnoticed by the machine learning community and not being followed up by any new research. Because the generality of those evolved rules is tested only on small networks and a limited number of problem instances, it is worth seeing performance of GP and new learning rules evolved using large networks with more complex problems.

Training deep networks was difficult because of the vanishing gradient problems. In order to overcome such problems, the use of alternative activation functions, pre-training and weight initialisation methods have been proposed. Pretraining was the first successful method to train deep networks. However, it is not easy to apply to the networks that have many hidden layers because of the computational cost. There have been several weight initialisation methods proposed that considered deep networks with the logistic activation function. Nevertheless, networks initialised with those methods perform poorly when the networks are deep (e.g. more than 5-hidden-layer networks). ReLU is the first activation function developed which does not suffer from the vanishing gradient problem. After this achievement, various ReLU based activation functions have been proposed. However, such activation functions also have some problem such as dying neuron, bias shift, noise sensitiveness. If the vanishing gradient problem of the sigmoid activation functions would be overcome, this could help networks to train deep networks without suffering from problems of the other activation functions.

Hence, in this thesis, we try to uncover learning and initialisation algorithms using GP and theoretical analysis to improve shallow and deep neural networks that use the logistic activation function.

Chapter 3

Evolving Learning Rules and a Fair Methodology for Comparing Learning Rule Performance

3.1 Introduction

A MLP is a commonly used form of ANNs. It is usually trained using gradient descent back-propagation algorithm. However, the SBP has some drawbacks as described in Section 2.3. In order to overcome those drawbacks, EAs and some other optimisation algorithms have been applied to neural networks.

Evolving learning rules through GP is a particular interest of this thesis. The most important feature that distinguishes GP from other EAs is that it can produce problem solvers, such as algorithms and computer programs, that can solve a range of problems rather than just produce solutions to a specific problem. It is thanks to this property that, approximately two decades ago, GP enabled researchers [27, 29, 23, 24, 121] to evolve new learning rules.

Bengio *et al.* [27] were the first to try this. In their work, the emphasis was on comparing alternative search techniques (GP, genetic algorithms and simulated annealing) to obtain new learning rules rather than improving over the SBP. It was found that GP was the best out of the techniques tested, and that, in one case, it managed to evolve a rule which was faster than the original SBP. The rule was a small variation of the back-propagation rule where the derivative of the activation function was cubed, changing slightly the behaviour of the rule in proximity of the 0 and 1 saturation points for the activation function.

Radi and Poli [29, 23, 24, 121] extended the work by Bengio *et al.* [27] to explore a larger space of learning rules. They reported that the GP could not simultaneously evolve learning rules for both hidden and output layers because of a large search space. Therefore, they applied the GP to one of those layers in each experiment.

Experimental results showed that this method can produce faster learning rules than SBP. However, the performance of the evolved rules was not widely tested.

Yet, as indicated in the previous chapter, there are still many open problems in ANN learning, and progress with traditional methods is difficult. These early successes received little attention from the machine learning community and no follow on work has appeared in the literature. This chapter attempts to fill this gap. In particular, it focused on using GP to evolve rules that: (1) are faster than the SBP, (2) do not require different formulations for the output and hidden layers, (3) generalise well beyond the learning tasks which were used for inducing them, (4) present a monotonic improvement in the error over time, with minimal oscillations, and (5) are rigorously compared against their competitors. This last point is particularly difficult and required the development of the new theoretical tools described in the next section.

3.2 Comparing Learning Rules

Of course, in order to determine whether a learning rule produced by the GP system used in this thesis, which it will be termed GP-MLP hereafter, is competitive, it is not enough for it to outperform *SBP* on the training problems and with the specific choice of parameters used in the fitness function. It needs to be tested a broad range of problems and conditions.

For instance, neural networks are initialised with random weights and biases, which make their training produce stochastic results. Therefore, for a fair comparison, it is clear that one needs to perform many re-initialisations and repetitions of the training. For this reason, in Section 3.4, 200 different initial sets of weights will be used for each problem set.

Also, a fair comparison requires one to set the maximum number of epochs to be large enough for each problem and the same for each pair of comparisons so that both rules have a reasonable and the same chance to solve problems.

In the rest of this section other requirements and techniques for a fair comparison of learning rules are discussed.

3.2.1 Pairwise Comparisons

One interesting way of comparing learning rules is to use the error obtained at each epoch of the learning process (as opposed to just at the end of it) as, in principle, a rule could initially converge faster than another but the former could be more likely trapped by local optima leading to worse end-of-training performance than the latter.

However, even after having performed a large set of runs, it is not sufficient to just compare the mean performance of learning rules: one needs to compare the

corresponding *distributions* of performance obtained for each learning rules using statistical tests.

Because performance distributions are typically non-Gaussian, non-parametric tests were used for these comparisons. To increase the reliability of the tests, the learning rules under comparison used exactly the same set of seeds for the random number generator used for weight and bias initialisation. In other words, with each seed the learning rules were pitted against each other starting from the same initial conditions. This paired the corresponding error data, making it possible to use more powerful statistical tests. More specifically, the Wilcoxon signed rank test was used, which is effectively a test of statistical significance of differences between medians. For this reason, in Section 3.4 the median error at each training epoch (instead of the more traditional mean) and the corresponding p value of the test are reported.

3.2.2 Success Criteria

Multiple ways of defining success for a training run exist. For instance, one could say that if the error is below a certain threshold, a run is successful. An alternative is to set a threshold on the accuracy of the network, defined as the fraction of patterns in the training set which have been learnt correctly.

Because different dataset can be learnt to different degrees, in order to obtain meaningful success criteria, one would need to adjust error or accuracy thresholds manually for each problem, which makes success dependent on some subjective elements such as the experimenter's objectives and experience as well as to the hyper-parameters of the algorithm (learning rate, network architecture, maximum number of epochs, etc.).

Irrespective of the definition, if the performance level required for success can be formalised numerically, then each run may or not be successful. Given that random initialisation makes the training of neural networks a stochastic algorithm, it is standard practice to train a neural network multiple times until one finds one that has satisfactory peformance. It is clear that different learning rules (with the associated hyper-parameters) could have different *success rates*, interpreted as the fraction of runs in which a particular level of performance was reached.

In general when one performs multiple runs, the effort is deemed successful if the error/accuracy is acceptable, and unsuccessful otherwise. In other words, one applies one of the success criteria discussed above to decide whether the effort of multiple runs is successful.

As it will be shown below, success rates are key in defining the optimality of learning rates and number of epochs.

3.2.3 Optimal Learning Rates

Decades of experience has been accumulated on setting good values for the learning rate, number of epochs, etc. for SBP. So, typically users set the values based on such prior knowledge (for instance $\eta \leq 0.5$) and then adjust them to a limited degree in preliminary runs. This is a form of manual optimisation which normally gives good results.

However, when one evolves a new learning rule, there is no prior experience on setting η for such a rule. Also, the fitness function may promote quicker training convergence. So, evolution may implicitly tune the learning rate of the evolved rules for maximum performance (on the training problems). Thus, adopting values of η for new learning rules using the traditional values used for *SBP* may be unfair. Also, while manual optimisation of η for *SBP* based on experience is normally successful at training a network, in no way one can claim that that the adopted learning rates are optimal.

Given the above-mentioned considerations, a fair comparison requires setting the learning rate for both *SBP* and the learning rules evolved by *GP-MLP* to their respective optimal values (based on success rates, as it will be shown later).

For this reason, in Section 3.4 each network will be trained with 30 different learning rates, $\eta \in \{0.1, 0.2, \ldots, 3.0\}$, and the best learning rate will be selected for each rule and problem. Also, four-fold cross validation will be used to measure error in continuous problems (the Boolean induction problems do not require this as there is no way of overfitting their corresponding truth tables).

Because these runs require high computational cost, the learning rate ranges and the number of epochs were determined with preliminary tests conducted with the same configuration but with a small number of independent runs.

3.2.4 Optimal Number of Epochs

The success rate of each learning rule is likely to depend on the number of training epochs. Of course, the more epochs, the more the training performance will improve and, so, the more likely success will be. However, it is not clear whether one would be better off doing more training attempts with a smaller number of epochs or fewer attempts with large number of epochs or something in between.

In the following subsections, two ways will be considered in which comparisons between learning rules could account for all this. In both cases the *computational effort*, i.e., the total number of training epochs required to obtain at least one successfully trained network within a batch of runs is estimated. The first method guarantees (with a high probability) that, if one can execute all runs in parallel, at least one run will succeed within a batch of runs. The second method assumes that one executes

runs one by one and provides the expected value of the computation required to obtain a successful run. However, there are many other ways of performing runs. For example, one could do runs in small batches, checking the result of each batch and stopping as soon as one batch contains at least one solution. Or one could do runs sequentially and consider the computational effort required if one wanted to be successful with a given probability (say 99%) in at least one of the sequential runs. In these cases, the computational effort required could be computed using a hybrid of the two approaches reported below.

Here success is defined as the network achieving a certain level of *training accuracy*, which is defined as the fraction of the training patters in the training set correctly learnt.[1]

Note that computational effort depends on the training (hyper) parameters (here this work focuses on learning rate and number of epochs). The *minimal computational effort* is the minimum computational effort recorded by varying the learning rate and number of epochs in all possible ways within a large range of values. The minimum computational effort may be a fair yardstick to compare learning rules.

3.2.4.1 Minimum Computational Effort for Parallel Training

This method considers the situation where multiple attempts to train a network (with different initialisations) are performed in parallel (e.g., if multi-processing is used to train multiple instances concurrently) or in a manner equivalent to parallel training (e.g., if multiple runs are scheduled sequentially overnight, so that the results are available to the user the following morning).

In this case, relatively simple adaptations of Koza's computational effort [17][2] can tell us how much computation is needed to obtain at least one satisfactorily trained network out of a certain number of parallel training runs.

Let $p_i(a, \eta)$ be cumulative probability of training a network to the desired degree of accuracy a (the fraction of patterns in the training set the network has correctly learnt) by epoch i and with a learning rate η. Also, let z be the prescribed *success probability* with which we want to be able to solve the problem (e.g., $z = 0.95$) when doing multiple runs of the learning algorithm. It should be noted that $p_i(a, \eta)$ depends not only on a and η but also the learning problem at hand, the architecture for the network, the initialisation strategy, etc. However, for brevity, such dependencies are not explicitly represented here.

[1] Of course success could have also been defined as reaching an error below a certain threshold. However, an absolute threshold would not work well across different problems. So accuracy is used instead.

[2] Koza's notion of minimum computational effort was used in the context of genetic programming to represent problem difficulty and to allow fair comparisons between alternative genetic programming systems.

The probability of *not* solving the problem by epoch i is clearly $1 - p_i(a,\eta)$. Because each run is independent (assuming weights and biases are initialised randomly) the probability of *not* solving the problem in R runs is given by $(1 - p_i(a,\eta))^R$. Naturally, its complement, $1-(1-p_i(a,\eta))^R$, is the *probability of us being able to solve the problem in at least one run*, i.e.,

$$z = 1 - (1 - p_i(a,\eta))^R. \tag{3.1}$$

Solving for R yields

$$R(a,\eta,i,z) = \frac{\log(1-z)}{\log(1-p_i(a,\eta))}, \tag{3.2}$$

where the fact that R depends also on z (plus all the other dependencies of p_i) is made explicit.[3]

If all runs are stopped at epoch i then the expected total number of learning steps, or computational effort, required to solve the problem with probability z when executing runs in *parallel training* is simply

$$E_p(a,\eta,i,z) = i \times R(a,\eta,i,z) = i \times \frac{\log(1-z)}{\log(1-p_i(a,\eta))}, \tag{3.3}$$

where the fact that the computational effort depends also on the choice of i (i.e., the epoch at which one stops the training) and η is made explicit.

Note that normally $E_p(a,\eta,i,z)$ is a monotonically increasing function of a — the accuracy required before training is declared successful — and z — the success probability required. However, it is not necessarily a monotonic function of η, as beyond a certain value of η, gradient-descent-type of algorithms, such as *SBP*, may climb up the error surface rather than the other way around. Also, and this may appear conter-intuitive, $E_p(a,\eta,i,z)$ is not a monotonically increasing function of i. For instance, in some cases performing many short runs yields a correctly trained network with less overall effort than a few long runs. However, making runs too short collapses the success probability to zero making $E_p(a,\eta,i,z) \to \infty$. More obviously, for a fixed i, $E_p(a,\eta,i,z)$ is a monotonically increasing function of a — the accuracy required before training is declared successful— and z — the success probability required — it is not a monotonic function of i. For instance, in many cases more shorter runs cost less effort than fewer longer ones, but making runs too short collapses the success probability to zero.

The availability of $E_p(a,\eta,i,z)$ makes it possible to optimise the effort over i and

[3] In Equation (3.2), $R(a,\eta,i,z)$ is the expected number of runs required to solve the problem, which may not be integer. Obviously, in practice one cannot do a non-integer number of training attempts. So, we need to do $\lceil R(a,\eta,i,z) \rceil$ runs, which yields a success probability $1-(1-p_i(a,\eta))^{\lceil R(a,\eta,i,z) \rceil}$ that may be strictly bigger than the desired success probability z.

η, yielding the following *minimum computational effort* required to achieve success with probability z:

$$E_p^*(a,z) = \min_{i,\eta} E_p(a,\eta,i,z) \tag{3.4}$$

with corresponding optimal values for the number of epochs and the learning rate given by

$$(i^*,\eta^*) = \arg\min_{i,\eta} E_p(a,\eta,i,z), \tag{3.5}$$

which in turn allows us to determine the optimal number of training runs:

$$R^*(a,z) = \lceil R(a,\eta^*,i^*,z) \rceil. \tag{3.6}$$

3.2.4.2 Minimum Computational Effort for Sequential Training

This method considers the situation where multiple attempts to train a network (with different initialisations) are performed sequentially. Here after each attempt the training accuracy is checked against the predefined desired level, a, and further attempts are stopped upon recording the first success.

Because each attempt is independent of the others, whether or not the actual accuracy at epoch i is greater or equal than a or not is a Bernoulli trial, with success probability $p_i(a,\eta)$ (see Section 3.2.4.1). So, the expected number of runs required to obtain one that is successful one [187] is simply given by

$$R(a,\eta,i) = \frac{1}{p_i(a,\eta)}. \tag{3.7}$$

Hence, the expected computational effort and *minimal computational effort for sequential training* are given by

$$E_s(a,\eta,i) = i \times R(a,\eta,i) = \frac{i}{p_i(a,\eta)}, \tag{3.8}$$

and

$$E_s^*(a) = \min_{i,\eta} E_s(a,\eta,i), \tag{3.9}$$

respectively. The corresponding optimal parameters are, therefore,

$$(i^*,\eta^*) = \arg\min_{i,\eta} E_s(a,\eta,i), \tag{3.10}$$

which in turn allows us to determine the optimal number of training runs:

$$R^*(a) = \lceil R(a,\eta^*,i^*) \rceil. \tag{3.11}$$

3.3 GP System, Fitness Measure and Datasets

3.3.1 Linear GP System

Experiments in this chapter were performed using a linear register-based GP system [111, 188]. The basic cycle of GP-MLP is described in Algorithm 1. Step 1: The algorithm starts with randomly generating as many learning rules as the population size at random. Step 3: Each learning rule is applied to multiple problems and is applied to each problem with multiple initial sets of weights and biases to measure the rule's fitness. Step 7: Firstly, either crossover or mutation operator is selected. Then, one or two learning rules (parents) are selected based on the chosen genetic operator. Finally, an offspring is created by applying the chosen operator to the selected parent(s). Step 8: To measure the created offspring's fitness, it is applied to the same networks (problems and the initial conditions) as those used in step 3. Step 9: The offspring created is replaced with the worst learning rule among a group of rules selected from the population. Here the learning rules that are better (fitter) than the worst one in the selected group of rules do not have any chance to be discarded from the population. Therefore, this step includes the application of elitism method too. Steps 7, 8 and 9 are repeatedly and respectively executed as many as the population size for each generation. Step 12: The algorithm returns the best learning rule when it reaches the maximum number of generations.

The system uses a population of 400 individuals, each including 30 instructions, a steady-state update schedule, tournament selection with a tournament size of 3, point mutation applied with 50% probability and two-point crossover also applied with 50% probability. When mutation is used, if visits every instruction in a program replacing it with a random instruction with a low probability (15%). Runs lasted for 100 generations. While all programs in the population are of the same length, the presence of a no-operation (NOP) instruction in the primitive set allows for programs of different complexity to evolve. These parameters were determined after having conducted some preliminary experiments. The choice of a small number of instructions leaded to obtain poor learning rules while a large number caused high computational cost. Therefore, we set the number of instructions to 30 considering such situations. A small population was used because of the heavy computational load associated with the fitness evaluation (more on this later). A small tournament size was used and a high mutation rate to avoid premature convergence, given small population size.

Table 3.1 shows the instructions used in the GP system. In order for the evolved rule not to be very complex, we used basic primitive elements in the instructions. More specifically, the terminal set consisted of the SBP, its fundamental components and a few constants while the functions set consisted of the fundamental arithmetic

algorithm 1 Pseudo-code illustrating how GP-MLP evolves learning rules

1: Generate a population of random learning rules
2: **for** all learning rules in the population **do**
3:     Apply a learning rule to all problems and conditions to measure fitness
4: **end for**
5: **for** generation = 1 **to** maximum generations **do**
6:     **for** epoch = 1 **to** population size **do**
7:         Create an offspring learning rule using mutation or crossover, selecting one or two parents, respectively
8:         Apply the offspring rule to all problems and conditions to measure fitness
9:         Insert the offspring created into the population
10:     **end for**
11: **end for**
12: **return** the fittest individual ever found

Table 3.1: Primitive set used in the experiments

Instructions	Instructions	Instructions
NOP	r0 <- -r0	r1 <- r1 + rSBP
r0 <- 0	r1 <- -r1	r0 <- r0 + r1
r1 <- 0	r0 <- r0 + ra	r1 <- r0 + r1
r0 <- 0.5	r1 <- r1 + ra	r0 <- r0 * r1
r1 <- -0.5	r0 <- r0 + ra_prev	r1 <- r0 * r1
r0 <- -0.1	r1 <- r1 + ra_prev	r0 <- r0 * r0
r1 <- 0.1	r0 <- r0 + re	r1 <- r1 * r1
r0 <- -1	r1 <- r1 + re	rs <-> r0
r1 <- 1	r0 <- r0 + rSBP	rs <-> r1

operators except the division. The system uses seven registers which are defined by: `r0` and `r1` are used for mathematical calculations; the register `rs` is used for swapping the registers. These registers are set to 0. After all instructors in the program are sequentially executed, `r0` reports the output of the program. Also, there are input registers in the table, `ra_prev`, `ra`, `re` and `rSBP`, for each of the input variables used in learning rule. These are defined as follows:

$$
\begin{aligned}
\texttt{ra_prev} &= a_i^{l-1}, \\
\texttt{ra} &= a_j^{l}, \\
\texttt{re} &= \begin{cases} e_j^k & \text{if } l = n, \\ \sum_i W_{ji}^l \delta_i^{l+1} & \text{otherwise,} \end{cases} \\
\texttt{rSBP} &= \texttt{ra_prev} \times \texttt{ra} \times (1 - \texttt{ra}) \times \texttt{re},
\end{aligned}
$$

where e_j^k is the j-th component of e^k and the last formula represents the SBP learning rule. The output produced by SBP rule is added into the input register `rSBP` to give evolution a good starting point on which to improve.

3.3.2 Fitness Measure

Assuming fitness is a function that needs to be minimised by GP-MLP, the most natural fitness for a learning rule would be the *network error* measured after having applied the rule for a number of epochs to train a network to solve a problem. However, in order to obtain general and stable learning rules one needs to use a more sophisticated strategy where fitness is the result of: (1) applying each learning rule to multiple problems, (2) applying each learning rule with multiple initial conditions (random weights and biases) for each problem, and (3) considering also the error values recorded throughout the training, not just at the end of it.

More specifically, in relation to point (1) above — applying each learning rule to multiple problems — one issue that arises when using errors measured in different problems to guide evolution is that such problems may present widely different errors due to the size of the input and output layers, the number of training examples and the degree to which a training set can actually be learnt by a MLP (e.g., because of the noise/inconsistencies in the training data or the limitations of the network architecture used). If this happens, then, of course, given a problem set, evolution will be more likely to optimise learning rules to problems with naturally larger errors than problems with smaller errors, thereby leading to rules that do not generalise well. For this reason the TSEs obtained with new learning rules were normalised to the corresponding TSEs obtained by the SBP rule, whereby normalised TSE values below 1 would indicate an improvement with respect SBP. This makes TSE values obtained in different problems comparable, and avoids unduly biasing the evolutionary search. Then evolution was forced to consider all training problems by driving it (through its fitness function) to minimise the worst normalised TSE across problems.

In relation to point (2) above — applying each learning rule with multiple initial conditions — preliminary runs showed that it was important to evaluate all learning rules with the same set of initial weights and biases, to avoid some rules gaining unduly higher fitness than others simply due to a lucky initialisation. This, however, introduces the problem that learning rules would evolve that exploited the initial sets of weights and biases, thereby producing rules that would not work well in unseen problems and/or with unseen initial conditions. So, for each problem, weights and biases had to be re-initialised n times using different seeds for the random number generator, and after each initialisation the corresponding network was trained for m epochs.

In relation to point (3) above — considering also the TSE values recorded throughout the training, not just at the end of it — preliminary runs showed that looking at only the TSE at the end of training was not enough to encourage generalisation.

The issue is that in order to reduce TSE evolution would try to amplify the learning rate of new rules, effectively obtaining rules that were not stable, and by pure luck could obtain low end-of-training TSEs after having widely oscillated. With relatively few problems and initial conditions, the likelihood of this happening in a population of hundreds of rules is quite high. Such rules would then hijack evolution and when tested with different initial conditions, problems or number of epochs, evolved rules would likely prove very poor. To prevent this from happening, penalties and rewards were also added to the fitness measure based on the directions of change of the TSE during the whole training process, effectively encouraging a monotonic descent.

More formally

$$fitness = \max(NTSE_1, \ldots, NTSE_k) \tag{3.12}$$

where k is the number of problems used in the GP run and

$$NTSE_p = \frac{\sum_{s=1}^{n}\left(NLRTSE_{p,m}^{s} + \sum_{i=1}^{m} P_{p,i}^{s}\right)}{\sum_{s=1}^{n} SBPTSE_{p,m}^{s}} \tag{3.13}$$

is the normalised TSE error in problem p, $NLRTSE_{p,m}^{s}$ is the end-of-run TSE of the *New Learning Rule* (*NLR*) being evaluated on problem p and with seed s, $SBPTSE_{p,m}^{s}$ is the corresponding end-of-run TSE of the SBP, n is the number of reinitialisations (seeds) of the network's weights, m is the number of training epochs, and $P_{p,i}^{s}$ are penalties/rewards encouraging the monotonic descent of TSEs. These are given by:

$$P_{p,i}^{s} = \begin{cases} \lambda^{+}(NLRTSE_{p,i}^{s} - NLRTSE_{p,i-1}^{s}), & \text{if } NLRTSE_{p,i}^{s} > NLRTSE_{p,i-1}^{s}, \\ \lambda^{-}(NLRTSE_{p,i}^{s} - NLRTSE_{p,i-1}^{s}), & \text{if } NLRTSE_{p,i}^{s} \leq NLRTSE_{p,i-1}^{s}, \end{cases} \tag{3.14}$$

where λ^{+} is the penalty coefficient and λ^{-} is the reward coefficient. Assuming both learning rules compared start with the same error at the first epoch and reach the same error at the last epoch, the use of the same reward and penalty coefficients does not make any sense to compare the training processes. However, a higher penalty coefficient than the reward coefficient should be used to encourage a monotonic descent of the error. In our experiments λ^{+} was set to 0.1 and λ^{-} was set to 0.001 with a small amount of trial and error.

3.3.3 Training and Test Problems

In the experiments, $k = 2$, i.e., two (training) problems were used — the parity and iris classification problems— for fitness evaluation. These problems were chosen so that they would be representatives of continuous and binary problems. Also, $n = 4$ reinitialisations and $m = 250$ epochs of each learning rule were used. Small network structures, a low number of re-initialisations and a low number of epochs

Table 3.2: Network structure and parameters for the two problems used for evolving learning rules

PROBLEM	HIDDEN LAYER SIZES	OUTPUTS	η	RUNS
PARITY	5, 5	1	0.8	4
IRIS	2, 2	3	0.1	4

Table 3.3: Information on the data sets used in this study

PROBLEM	INST.	NF	CLASSES	IM
IRIS	150	4	3	0.0000
WINE	178	13	3	0.0125
AUTHORSHIP	841	70	4	0.0833
BCW	699	9	2	0.0963
E-COLI	327	7	5	0.1089
DNA	3186	180	3	0.0776
PARITY	16	4	2	0.0000

were purposely chosen to speed up the evolution process. The network structures and other parameters used are shown in Table 3.2.

In addition to the 4-bit parity and iris classification problems that were used in the evolution process, the 7-segment display, iris, wine recognition (wine), analcatdata authorship (authorship), e-coli, breast-w (BCW) and DNA classification problems were additionally used to test performance of the new learning rules. The reasons why being selected these problems are that they are commonly used to compare algorithms and do not lead a high computational cost when we apply our comparison methods to them. These problems were taken from the Penn Machine Learning Benchmarks [189] as this repository includes a wide range of accessible real-world data for comparing machine learning algorithms. Information of these data sets — number of instances (Inst.), number of features (NF), number of classes (classes) and imbalance metric (IM) — are provided in Table 3.3. Imbalance metric shows the level of the imbalance of data. The imbalance metric values were provided by the Penn Machine Learning Benchmarks [190] and computed by the following formula:

$$I = K \sum_i \left(\frac{m_i}{M} - \frac{1}{K} \right)^2 \tag{3.15}$$

where K is the number of classes in the data, M is the number of examples in the data and m_i is the number of examples of class i in the data.

The network structures (number of neurons in the layers) used for comparing *NLR*'s and the *SBP* were manually chosen with a small amount of trial and error (to

Table 3.4: Network structures adopted for the different test problems used for comparing *NLR* and *SBP*

PROBLEM	INPUT	HIDDEN	OUTPUT
IRIS	4	10, 10	3
WINE	13	10, 10	3
AUTHORSHIP	70	30, 30	4
BCW	9	20, 20, 20	2
E-COLI	7	20, 20	5
DNA	180	100, 100, 100	3
7-SEGMENT	4	5, 5	7
THE PARITY	16	7, 7	1

balance convergence accuracy, computational load and overfitting) and are shown in Table 3.4. Note that parity and iris problems were used both for training (during evolution) and for testing the performance of evolved rules, although the networks used in testing where significantly bigger. This was done to verify that evolution had not "overfitted" the few random seeds and the tiny topologies used in training.

3.4 Experimental Results

As indicated above, GP-MLP was applied to the parity and iris classification problems with 4 different initial weights for each problem. The sigmoid activation function was used for the neurons of the output layer of networks of both the parity and iris classification problems' networks. TSE was used during the training of the networks as previously explained. With this configuration, several runs were performed, all of which produced learning rules that were competitive with SBP, although not always general. However, at the end of one run, GP-MLP returned the following new learning rule

$$NLR = SBP \times (SBP + a_j^l + 3a_i^{l-1}). \tag{3.16}$$

The generality of the rule and its performance advantages over the *SBP* will be discussed and shown below.

3.4.1 Qualitative Interpretation of Evolved Rule

To start understanding the behaviour of this rule, we can simply divide both sides of Equation (3.16) by *SBP*, obtaining:

$$\begin{aligned} \frac{NLR}{SBP} &= SBP + a_j^l + 3a_i^{l-1} && (3.17) \\ &= a_j^l(1-a_j^l)a_i^{l-1}\delta_{ij}^l + a_j^l + 3a_i^{l-1}. && (3.18) \end{aligned}$$

Table 3.5: Different effective learning rates of *NLR* as a function of the activations of the pre-synaptic and post-synaptic neurons associated with a connection weight

a_j^{l-1}	a_i^l	$\frac{NLR}{SBP}$
0.1	0.1	$0.009 \times \delta_{ij}^l + 0.4$
0.1	0.5	$0.025 \times \delta_{ij}^l + 0.8$
0.1	0.9	$0.009 \times \delta_{ij}^l + 1.2$
0.5	0.1	$0.045 \times \delta_{ij}^l + 1.6$
0.5	0.5	$0.125 \times \delta_{ij}^l + 2.0$
0.5	0.9	$0.045 \times \delta_{ij}^l + 2.4$
0.9	0.1	$0.081 \times \delta_{ij}^l + 2.8$
0.9	0.5	$0.225 \times \delta_{ij}^l + 3.2$
0.9	0.9	$0.081 \times \delta_{ij}^l + 3.6$

It is clear that where the ratio $\frac{NLR}{SBP}$ is greater than 1, the new learning rule amplifies the moves in weight-space of the *SBP*, while where the ratio is less than 1, *NLR* makes more cautious moves than *SBP*. So, we could interpret the *NLR* in Equation (3.16) as a form of *SBP* but with a variable effective learning rate, provided in Equation (3.18). This is made explicit in Table 3.5, where the effective learning rate is tabulated for different values of a_j^{l-1} and a_i^l.

From the table one sees that the bigger the activations a_j^{l-1} and a_i^l, the bigger the learning rate of the *NLR* w.r.t. *SBP*. However, for any given values of a_j^{l-1} and a_i^l, it is also clear that the effective learning rate is bigger when $\delta_{ij}^l > 0$ than when $\delta_{ij}^l < 0$. In other words, the *NLR* is asymmetric in that it attempts to reduce positive errors faster/sooner than negative ones.

Beyond this, the effect of the linear term in δ_{ij}^l will become smaller and smaller as the learning process proceeds, so that eventually the effective learning rate depends mainly only on a_j^{l-1} and a_i^l, thereby restoring symmetry.

Of course, as shown in the table, averaging over the values of δ_{ij}^l, for most values of a_j^{l-1} and a_i^l we see that the effective learning rate is bigger than 1. So, if we apply the two learning rules with the same learning rate η, we should expect the *NLR* to have a faster descent on the TSE surface. However, in terms of asymptotic values for weights and biases, we should also find that the *NLR* is not behaving in a radically different manner from a *SBP*-like gradient descent. This is illustrated for a typical run with the XOR problem in Figure 3.1. The figure reports the changes of the TSE over time (top) as well as the temporal dynamics of the weights and biases in the network (bottom) for *SBP* and *NLR*. Note how both rules start from the same TSE and weights and after training, by and large, the same asymptotic values for the parameters are achieved. *NLR*, however, reaches good parameters values much more quickly as explained above. This, of course, does not imply anything about relative performance, as it could easily be matched by the *SBP* by simply increasing

its learning rate.

3.4.2 Choice of Performance Criteria

As explained in Section 3.3.3, *NRL* was compared with the *SBP* rule on the following problems: 7-segment display, 4-bit parity, iris classification, wine classification, breast cancer classification, e-coli classification, DNA classification and authorship classification problems.

In pairwise comparisons between *NLR* and *SBP*, TSE was used as a performance indicator for the Boolean induction problems, while, for the other problems, a soft-max output layer was used to facilitate classification, in which case the average cross-entropy loss was used as a performance indicator.[4]

For the definition of success, the fraction of training patterns correctly predicted was used, setting the threshold a (see Section 3.2.4.1) to 100% for all problems except iris where it was 95%, as not all patterns in iris can be learnt without badly overfitting.

Both for sequential execution and parallel execution of training runs, the leaning rate η^* and number of epochs i^* which yielded the minimum computational effort required to solve each problem were identified using Equations (3.4) and (3.9). To compute these, $p_i(a,\eta)$ was estimated — the cumulative probability of training a network to the desired degree of accuracy a by epoch i and with a learning rate η — using 200 training attempts (with different initial sets of weights) for each problem and 30 different learning rates. It was not needed to repeat the training for different values of i as the runs used a maximum number of epochs $i_{\max}$ such that $i^* < i_{\max}$.

In the calculations of the minimal computational effort, both for sequential and parallel evaluation, a was 0.95 for the iris problem and 1.0 for all other problems. For parallel evaluation, z a probability of solving the problem was 0.99.

3.4.3 Results with Parallel Evaluation

Figure 3.2 reports results for all test problems when parallel evaluation is considered. This should be analysed jointly with Table 3.6 that provides the values of the maximum number of epochs ($i_{\max}$) used to compute cumulative probability, the required number of independent runs (R^*), the required number of epochs (i^*), the minimum computational effort ($E_p^*(a,z)$) and the best learning rate (η^*) to solve each test problem.

More specifically, each of the plots named ‘Computational Effort’ in Figure 3.2 shows: (a) the values of $p_i(a,\eta^*)$ for both *SBP* and *NLR* (blue and orange solid

[4] When the soft-max output layer is used in a network, the weights between the soft-max layer and the last hidden layer are updated using the normal operations of the soft-max layer. The other weights and biases of the network are updated according to either the *NLR* or the *SBP* learning rule.

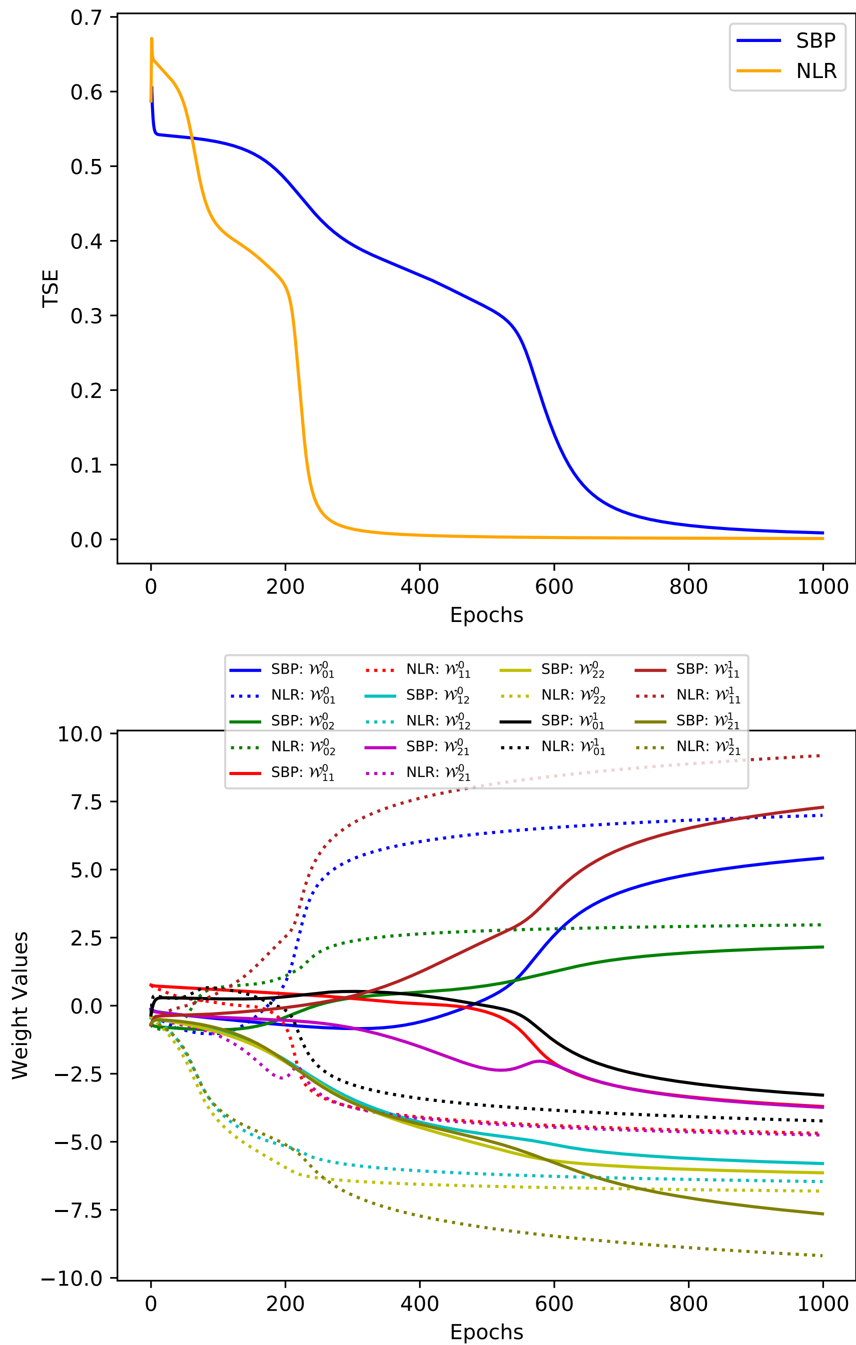


Figure 3.1: Illustrative comparison of the TSE (top) and weight (bottom) changes over time for an XOR problem with a classical 2-2-1 architecture when training is done by the *SBP* and *NLR*, both starting from the same initial set of weights and biases

lines, respectively, to be interpreted using the ordinate scale on the left) as a function of the epoch number, i; (b) the corresponding values of the computational effort $E_p(a, \eta^*, i, z)$ (purple and brown dashed lines, respectively, to be interpreted using the ordinate scale on the right); (c) the corresponding values of i^* where the minimal computational effort $E_p^*(a,z)$ is achieved for each rule (orange and blue marks, respectively). It is apparent that the probability of successfully training the network $p_i(a, \eta^*)$ for *NLR* always dominates the corresponding probability for *SBP*. The superiority of *NLR* is also illustrated by the values of the minimal computational effort $E_p^*(a,z)$ reported in Table 3.6. While the advantage of *NLR* over *SBP* is small for relatively easy problems, the gap is quite big for the harder ones (parity, BCW and e-coli).

Also, looking at values of R^* reported in Table 3.6, it is immediately clear that there is a distinction between Boolean induction problems and continuous problems. In the case of the Boolean problems, irrespective of the learning rule, it is computationally more convenient to do many short training attempts than to try to achieve success with one long run. However, with continuous problems the opposite is almost always true.

Finally, it is interesting to look at values of η^* reported in Table 3.6. Here one sees that in most cases the optimal learning rate for *SBP* is way above the values typically suggested in the literature. For instance, for the parity function, where it is notoriously difficult to avoid being trapped in sub-optimal states resulting in people using very small learning rates, the optimal learning rate is 2.

Let us now focus on the plots named 'Error' in Figure 3.2. They report the median error on the training set *vs* epoch number obtained in the 200 runs executed with the optimal learning rate η^* for the *NLR* and *SBP* (solid orange and blue lines, to be interpreted using the ordinate scale on the left), respectively, as well as the p value (green dashed line, to be interpreted using the ordinate scale on the right) obtained by applying the Wilcoxon signed rank test to the 200 pairs of errors values recorded for *NLR* and *SBP* over all training time-steps to analyse which differences are statistically significant ($p\ value \leq 0.05$). Error plots and their p-value plots obtained with each rule's optimal learning rates also show the performance advantages of *NLR* over the *SBP* on almost all problems. The only exception is the 7-segment problem, and its reason is explained below.

The plots named 'Accuracy' in Figure 3.2 show the *median* accuracy (i.e., the proportion of patterns correctly learnt) *vs* epoch number, obtained in 200 training runs executed with the optimal learning rate η^* for each learning rule. Solid orange and blue lines show the training accuracy of the *NLR* and *SBP*, respectively, while the dotted orange and blue lines show the testing accuracy of the corresponding rule. These accuracy lines were plotted according to the ordinate scale on the left.

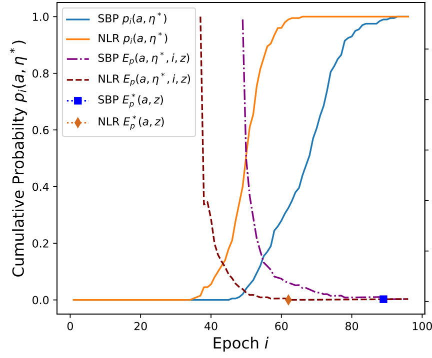
Because for Boolean induction problems there is no distinction between train and testing set, only the orange and blue lines are reported.

As seen from the accuracy plots of continuous problems, both rules, *SBP* and *NLR*, reach almost the same level of train and testing accuracy; and have a similar generalisation error on each problem at the end of the training. However, *NLR* converges to the maximum train and testing accuracy faster than *SBP* on all problems. These plots also show that neither NLR nor SBP has overfitting on iris, wine, authorship, and DNA problems, and they both have some overfitting on e-coli and BCW problems.

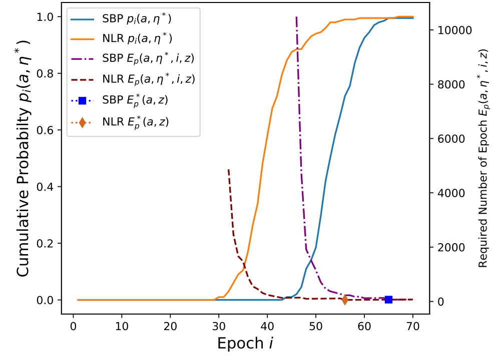
In addition, the plots report dashed green and brown lines which represent the p values obtained when doing a pairwise comparison of the accuracy data obtained at each epoch by the *NLR* and *SBP* rules, for training and (where appropriate) testing accuracy. The p value lines were plotted according to ordinate scale on the right and were obtained by applying the Wilcoxon signed rank test to the 200 pairs, similarly to what was done for the error plots in Figure 3.2.

One may think that, in Figures 3.2.21 and 3.2.23, the performance of *NLR* in the 7-segment problem looks worse than that of *SBP*. However, this ignores the fact that central tendencies (specifically, medians) in such figures are plotted. However, in relation to the performance of learning rules, the proposed approach cares more about the part of the distribution (in this case the lower tail) that solves the problem than about central tendencies. In other words, the results of the 200 independent runs performed with *SBP* and *NLR* are not equally distributed, *NLR* having a thicker tails than *SBP*. That is, the *NLR* has more successful runs than *SBP* within those 200 independent runs, but also more runs where it is significantly worse than *SBP*.

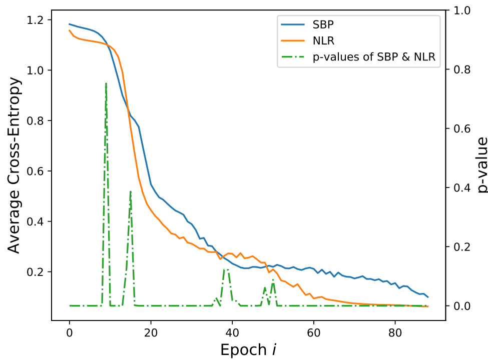
In a nutshell, *NLR* was evolved (trained) on two problems and tested on those two training (with different network structures) and six additional problems, as shown in Table 3.2 and 3.4, respectively. Figure 3.2 and Table 3.6 provide strong evidence that shows the generality of the evolved rule and its performance advantages over *SBP*. The table shows that *NLR* requires less computational effort (number of epochs) than *SBP* to solve the problems. This superiority is also shown with the computational effort plots in Figure 3.2. In the relevant plots, *NLR* $E_p(a, \eta^*, i, z)$ decreases faster than *SBP* $E_p(a, \eta^*, i, z)$ for all problems except 7-segment. Also, Figure 3.2 shows the performance advantages of *NLR* over *SBP* in terms of the training error, training accuracy and (if appropriate) testing accuracy of the test problems. The accuracy plots illustrate that while both learning rules have almost the same level of generalisation error on each problem at the end of the training, *NLR* reaches the maximum learning faster/sooner than *SBP*. Note that the boolean induction problems were not evaluated in terms of the generalisation as they cannot be divided into training and testing data. These results answer two of our research



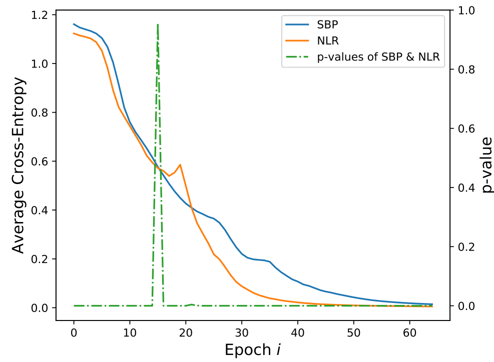
3.2.1 Iris Computational Effort



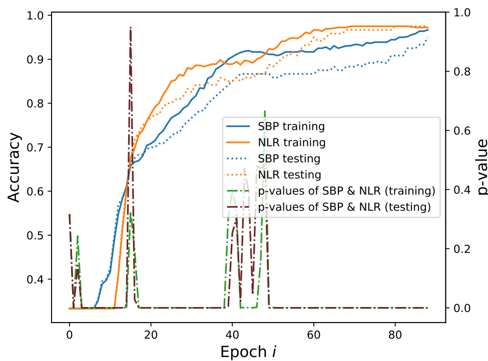
3.2.2 Wine Computational Effort



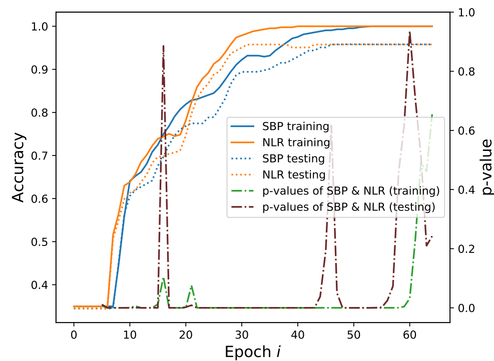
3.2.3 Iris Error



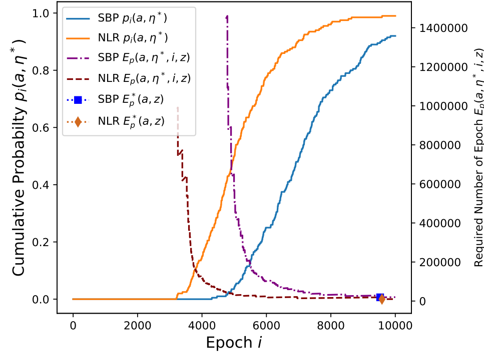
3.2.4 Wine Error



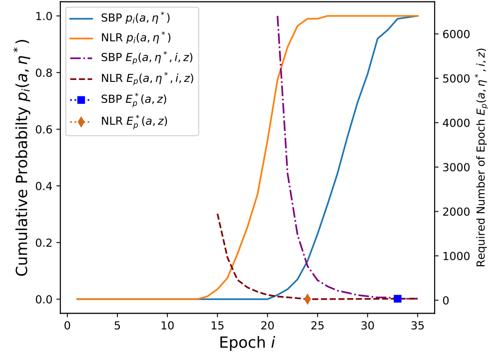
3.2.5 Iris Accuracy



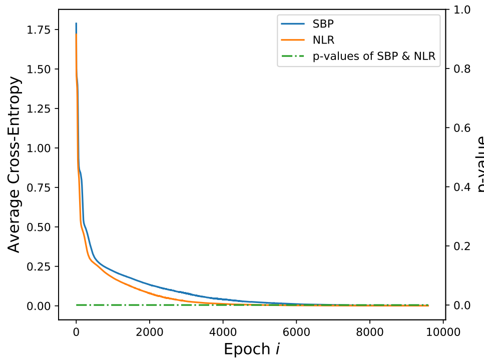
3.2.6 Wine Accuracy



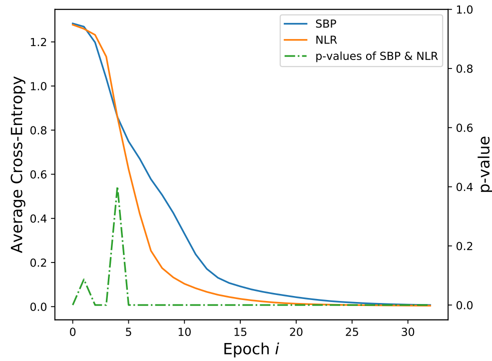
3.2.7 E-coli Computational Effort



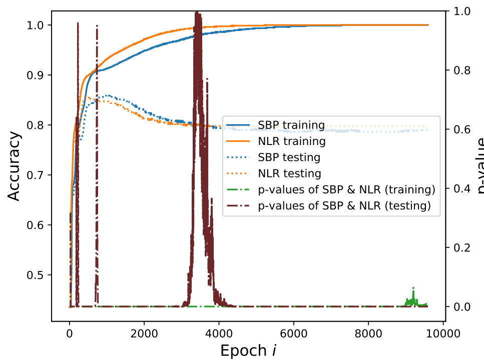
3.2.8 Authorship Computational Effort



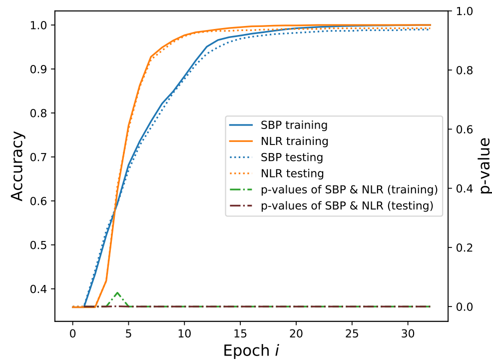
3.2.9 E-coli Error



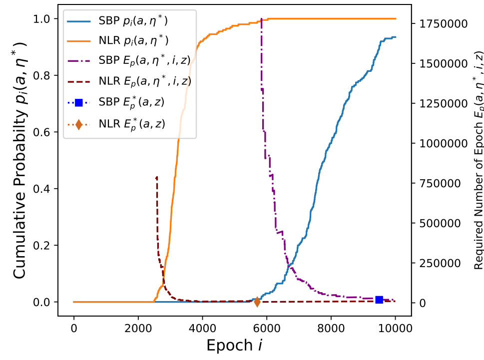
3.2.10 Authorship Error



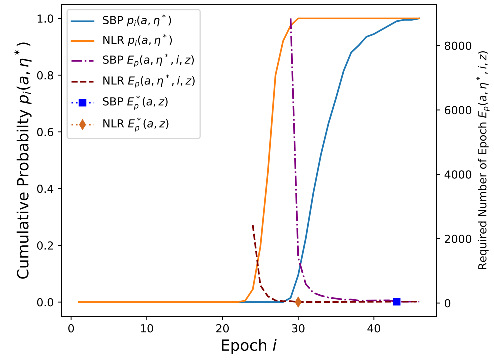
3.2.11 E-coli Accuracy



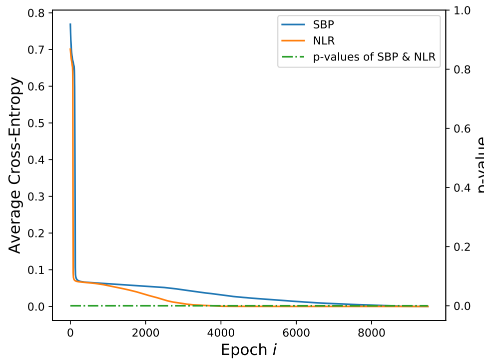
3.2.12 Authorship Accuracy



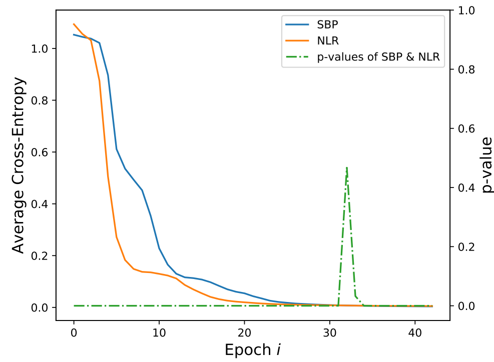
3.2.13 BCW Computational Effort



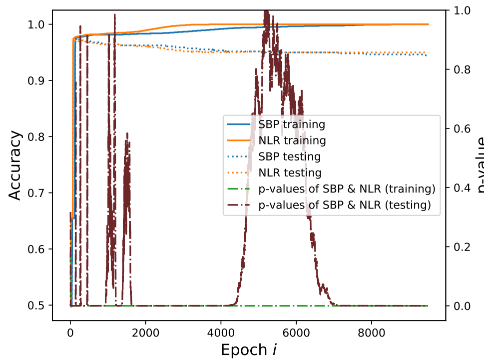
3.2.14 DNA Computational Effort



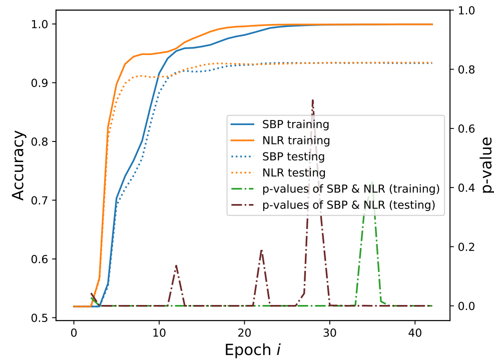
3.2.15 BCW Error



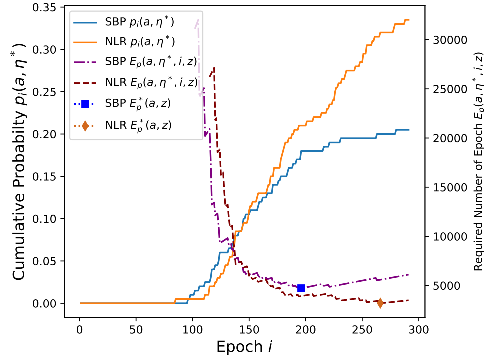
3.2.16 DNA Error



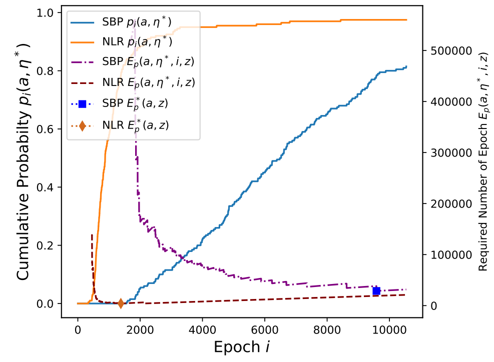
3.2.17 BCW Accuracy



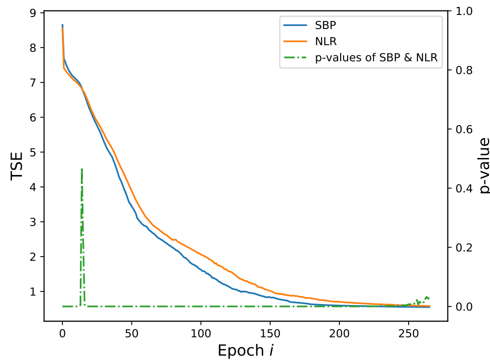
3.2.18 DNA Accuracy



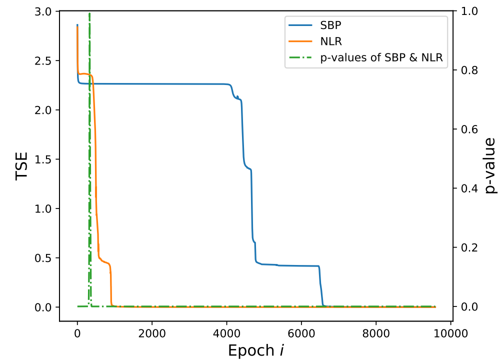
3.2.19 7-Segment Computational Effort



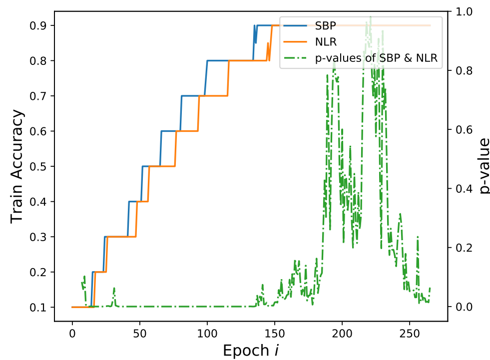
3.2.20 Parity Computational Effort



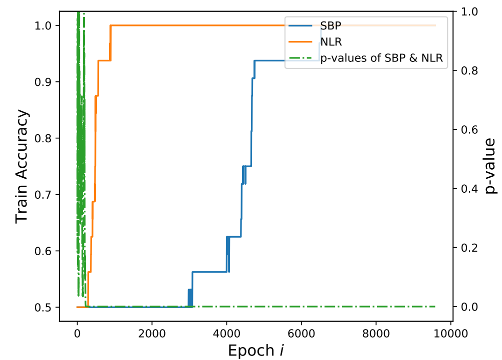
3.2.21 7-Segment Error



3.2.22 Parity Error



3.2.23 7-Segment Accuracy



3.2.24 Parity Accuracy

Figure 3.2: Results for all test problems when **parallel** evaluation is used (see text for explanations).

Table 3.6: Minimal effort and associated optimal learning rate, number of epochs and number of runs for **parallel** evaluation

Problem	Rule	$i_{\max}$	R^*	i^*	$E_p^*(a,z)$	η^*
Iris	NLR	1000	1	62	*62*	1.8
	SBP	1000	1	89	*89*	3.0
Wine	NLR	200	1	56	*56*	1.9
	SBP	200	1	65	*65*	2.5
Authorship	NLR	500	1	24	*24*	1.3
	SBP	500	1	33	*33*	2.9
BCW	NLR	10000	1	5703	*5703*	0.2
	SBP	10000	2	9497	*18994*	0.3
Ecoli	NLR	10000	1	9587	*9587*	0.7
	SBP	10000	2	9530	*19060*	1.2
DNA	NLR	200	1	30	*30*	0.3
	SBP	200	1	43	*43*	0.9
7-segment	NLR	5000	12	266	*3192*	0.5
	SBP	5000	24	196	*4704*	2.3
Parity	NLR	50000	3	1381	*4143*	0.7
	SBP	50000	3	9583	*28749*	2.0

questions. More specifically, the results show that GP can produce new learning algorithms that solve problems with less computational efforts than the *SBP* and is faster than *SBP*.

3.4.4 Results with Sequential Evaluation

When *SBP* and *NLR* are used with the sequential evaluation method, results are very similar to those reported in Section 3.4.3, as illustrated in Figure A.1 (see Appendix A) and Table 3.7. The difference between these and Figure 3.2 and Table 3.6 is that here we use the computational effort definition of Section 3.2.4.2.

We should note that while the sequential evaluation computational efforts are lower than the corresponding parallel-evaluation ones, one should not conclude that one approach is faster than the other. The calculations for the two cases and the corresponding experimental results are different because they answer two different questions. As we explained at the beginning of Section 3.2.4, the first method computes the computational required to guarantee (with a high probability) at least one successful run within a batch of runs if all runs are executed in parallel. The second method assumes that runs are executed one by one and computes the expected value of the computational effort required to obtain a successful run.

Table 3.7: Minimal effort and associated optimal learning rate, number of epochs and number of runs for **sequential** evaluation

Problem	Rule	$i_{\max}$	R^*	i^*	$E_s^*(a)$	η^*
Iris	NLR	1000	1	63	*63*	1.7
	SBP	1000	1	91	*91*	2.8
Wine	NLR	200	1	65	*65*	1.5
	SBP	200	1	78	*78*	2.4
Authorship	NLR	500	1	25	*25*	1.5
	SBP	500	1	35	*35*	2.9
BCW	NLR	10000	2	1645	*3290*	0.7
	SBP	10000	2	5177	*10354*	0.7
Ecoli	NLR	10000	2	4830	*9660*	0.9
	SBP	10000	2	6788	*13576*	1.3
DNA	NLR	200	1	30	*30*	0.3
	SBP	200	1	45	*45*	1.0
7-segment	NLR	5000	5	169	*845*	1.1
	SBP	5000	6	192	*1152*	2.3
Parity	NLR	50000	2	808	*1616*	1.2
	SBP	50000	2	5905	*11810*	3.0

3.5 Summary

In this chapter, a single rule was evolved that can be applied to both the hidden and the output neurons. The NLR was evolved by the GP-MLP using one continuous and one Boolean induction problems. It was tested using relatively (1) wide set of test problems, including six real-world, large scale, problems — iris, wine, authorship, BCW, e-coli, DNA — and two boolean induction problems — 7-segment and parity — (2) networks with more than one hidden layer, (3) wide networks including up to 100 neurons in each hidden layer.

This chapter also presented theoretical tools (based on the notion of computational effort) developed to perform much more rigorous comparisons than the traditional comparison method that considers just the mean of a batch of runs. In relation to this point, a wide range of learning rates, $\eta \in \{0.1, 0.2, \ldots, 3.0\}$, was considered, always choosing the maximum number of epochs in such a way that *SBP* would not be at a disadvantage for its slower convergence; and we tested also for generalisation accuracy, not just error.

This rigorous bench-marking process showed that the evolved rule was faster than SBP and required less computational effort than the SBP to solve the problems. The generality of the evolved rule was tested on six continuous and two Boolean induction problems. The evolved rule successfully solved all problems tested. The analysis of the evolved rule showed that it was fundamentally a form of asymmetric

SBP where the learning rate of different weights was modulated both by the error and the activations of the pre- and post-synaptic neurons in such a way to reduce positive errors faster/sooner than negative ones. Thus, we can conclude that rather than learning all classes at the same time, learning the training data on a class by class basis helps to speed up the learning process.

Chapter 4

Using Unstable Learning Rules to Speed up Learning

4.1 Introduction

In this chapter, like in the previous chapter, GP was used to evolve neural network learning rules. The previous chapter successfully evolved a learning rule. However, the fitness function restricted for GP to search the search space because it used the penalty and reward functions to encourage monotonic error descent. Therefore, this chapter tried to answer the question that arose whether it would be possible to speed up learning by allowing the evolution of unstable learning rules.

As described in the previous chapter, preliminary experiments showed that the evolved rules did not perform well on previously unseen problems if the error *at the end* of the training was used as the fitness function. The point was that in order to reduce the error, the GP would try to increase the learning rate of evolved programs, effectively finding unstable rules, and it could obtain low end-of-training errors by pure luck after the error having widely oscillated. In relatively few problems and with some initial conditions, this is likely to happen in a population of hundreds of rules. Such rules would then take over evolution, and when tested with different initial conditions, problems or number of epochs they would likely perform poorly. To prevent this from happening we postulated that not only errors at the end of training but also errors obtained during training should contribute to the evaluation of the fitness of rules.

As one of the possible ways to reduce the impact of this situation, in the previous chapter, penalty and reward were added to the fitness measure based on the directions of change of the error during the whole training process. At the end of the evolution, GP provided a more general and stable learning rule than the SBP rule. This approach encourages rules in the population to perform a monotonic descent on the error surface. However, this may also lead the network to get trapped in a local

Table 4.1: Primitive set used in the experiments

Instructions	Instructions	Instructions
NOP	r1 <- -r1	r1 <- r1 + rSBP
r0 <- 0	r0 <- r0 + ra	r0 <- r0 + r1
r1 <- 0	r1 <- r1 + ra	r1 <- r0 + r1
r0 <- 0.5	r0 <- r0 + ra_prev	r0 <- r0 * r1
r1 <- -0.5	r1 <- r1 + ra_prev	r1 <- r0 * r1
r0 <- -0.1	r0 <- r0 + re	r0 <- r0 * r0
r1 <- 0.1	r1 <- r1 + re	r1 <- r1 * r1
r0 <- -1	r0 <- r0 + rd	rs <-> r0
r1 <- 1	r1 <- r1 + rd	rs <-> r1
r0 <- -r0	r0 <- r0 + rSBP	

minimum.

In this chapter, GP was employed to the intent with evolving learning rules that are fast, general and can escape local minima. To do this, in contrast to the previous chapter, we did not penalise oscillations on the error during the training because we see oscillations as a potential mechanism to escape local minima. In addition, we used a fitness measure based on the lowest errors recorded throughout the whole training process.

4.2 GP System, Fitness Measure and Datasets

4.2.1 Linear GP System

Experiments in this chapter were performed using the same GP system that was presented in the previous chapter. The only differences were that (1) the size of the GP population was increased from 400 to 600; (2) the derivative of the post-synaptic neuron was used as a terminal in the primitive set defined as follows:

$$\texttt{rd} \;=\; a_i^{l-1} \times (1 - a_i^{l-1}).$$

Although this derivative is a fundamental component of the SBP and its construction is possible with the primitive set used in the previous chapter, non of the evolved rules included this derivative. Therefore, we used it as a terminal in the experiments conducted in this chapter to help the evolution process. The primitive set used in the GP system is shown in Table 4.1. The primitives that are new in comparison with those used in Chapter 3 are highlighted in the table.

4.2.2 Fitness Measure

The main purpose of the fitness function is to minimise the training error of the learning rules in the population. However, there is not only one form of it. In this

chapter, to allow the GP to search for a wider space of possible learning rules in comparison with the previous section, we used two different fitness functions that do not penalise learning rules. These functions have some differences from that used in the previous chapter. Here, functions do not penalise and reward the rules based on the oscillations during the training. Also, instead of the end-of-run error, fitness is the sum of a fraction of the lowest errors/highest accuracies recorded during training. Removing the penalty and reward may cause errors to oscillate in the error descent during the training. The oscillations can be controlled by adjusting the fraction of errors contributing to the fitness.

Of course, like in the previous chapter in order to obtain more general learning rules, each rule is applied to multiple problems and is applied to each problem with multiple initial conditions (random weights and biases).

Here are the equations that are used to compute the fitness of the evolved learning rules:

$$fitness = \max(fitness_1, \ldots, fitness_k), \tag{4.1}$$

where k is the number of problems used in the GP run,

$$fitness_p = \frac{\sum_{s=1}^{n} \sum_{t=1}^{b} sort\left(NLRloss_p^s\right)_t}{\sum_{s=1}^{n} \sum_{t=1}^{b} sort\left(SBPloss_p^s\right)_t} \tag{4.2}$$

is the fitness contribution of problem p, $sort()$ is a function which sorts its argument and *NLRloss* and *SBPloss* are the average cross-entropy loss values of the NLR and SBP, respectively, obtained throughout the whole training process, p is the problem, n is the total number of seeds (randomly initialised set of weights and biases), and b is the number of loss values that contribute to the fitness evaluation. b is given by

$$b = \left\lceil \frac{cm}{100} \right\rceil \tag{4.3}$$

where c is percentage of errors that contribute to fitness evaluation and m is the total number of epochs. In the experiments the c was defined as 10. That is only the bottom decile of the loss values recorded in training.

As described in the previous chapter, both the parallel and the sequential comparison methods require a certain level of *training accuracy* (defined as the fraction of the training patterns in the training set correctly learnt) to declare the training successful. Therefore, in a second set of experiments in this chapter, for the fitness evaluation of the rules, training accuracy, was used instead of the error. More formally, the following equations were used:

$$fitness = \min(fitness_1, \ldots, fitness_k) \tag{4.4}$$

Table 4.2: Network structures and parameters for the problems used for evolving learning rules

PROBLEM	INPUTS	HIDDEN LAYER SIZES	OUTPUTS	η	RUNS
IRIS	4	10, 10	3	3.4	4
NOISY WINE	5	10, 10	3	2.9	4
SPIRAL	2	20, 20	3	0.4	4

and

$$fitness_p = \frac{\sum_{s=1}^{n}\sum_{t=1}^{b} sort\left(NLRaccuracy_p^s\right)_t}{\sum_{s=1}^{n}\sum_{t=1}^{b} sort\left(SBPaccuracy_p^s\right)_t}. \tag{4.5}$$

where *NLRaccuracy* and *SBPaccuracy* are accuracy values of the NLR and SBP, respectively, obtained throughout the whole training.

4.2.3 Training and Test Problems

In this work, two different data sets — noisy wine and randomly generated spiral (taken from [191]) classification problems — were additionally used during evolution. The reason why the use of these problems in the evolution (training) process is that they are challenging and small problems; thereby, GP can obtain efficient learning rules without paying high computational costs. The noisy wine was derived from the wine classification problem by reducing its input dimension from thirteen to five to make the problem harder. The spiral classification problem was randomly generated. This has three classes where sets of examples of classes are surrounding each other. Such problems are not easily separable.

Two different learning rules were evolved. In the first set of experiments, which uses error-based fitness function, $k = 3$, i.e., three (training) problems were used — iris, noisy wine and randomly generated spiral classification problems — for fitness evaluation. In the second set of experiments, which uses accuracy-based fitness function, the noisy wine, and spiral classification problems were used for fitness evaluation, i.e., $k = 2$. Also, $n = 4$ re-initialisation and $m = 500$ epochs of each learning rule were used in both sets of experiments. The networks for these problems were purposely chosen to be small to speed up evolution. The network structures and other parameters used are shown in Table 4.2.

In addition to the problems used in the evolution process, as in the previous chapter, the evolved rules were tested on different problem sets. The test problems are the same as those used in the previous study. Network structures (number of neurons in layers) used to compare *NLR*s and *SBP* are shown in Table. 3.4.

4.3 Experimental Results

As indicated above, GP-MLP was applied to the noisy wine, iris, and spiral classification problems using the error-based fitness; and applied to the noisy wine and spiral problems using the accuracy-based fitness. Each problem in each experiment was run with four different initial sets of weights. The soft-max activation function was used for the neurons of the output layer of networks. The weights and biases between the last hidden layer and the output layer were updated using the normal operations of the soft-max layer in both the evolution and testing phases. The other weights and biases of the networks were updated according to the corresponding learning rule. The average cross-entropy loss was used during the training of the networks, as previously explained. With these configurations, at the end of the runs, GP-MLP returned the following new learning rules

$$NLR(1) = SBP \times ((a_i^{l-1})^2 + a_i^{l-1}) \tag{4.6}$$

and

$$NLR(2) = SBP \times (a_i^{l-1} + a_j^l), \tag{4.7}$$

which were general and usually required less computational effort than the *SBP* according to the comparison methods presented in the previous chapter to solve the problems (as we will discuss later).

4.3.1 Qualitative Interpretation of Evolved Rules

Similar to what was found in the previous chapter, evolved rules show that GP-MLP tends to make the learning rate of *SBP* vary depending on the activation values of pre-synaptic and/or post-synaptic neurons. The following equations show the learning rates of the *SBP* provided by *NLR(1)* and *NLR(2)* respectively that are obtained by simply dividing both sides of the Equations (4.6) and (4.7) by the *SBP*.

$$\frac{NLR(1)}{SBP} = (a_i^{l-1})^2 + a_i^{l-1} \tag{4.8}$$

$$\frac{NLR(2)}{SBP} = (a_i^{l-1}) + a_j^l \tag{4.9}$$

Simple calculations show that *NLR(1)* increases the learning rate of SBP when the pre-synaptic activation value of the corresponding connection is greater than 0.618, while it decreases the learning rate when the pre-synaptic activation value is less than that. *NLR(2)* not only considers the activation values of the pre-synaptic

neuron, but it also takes into account the activation values of the post-synaptic neuron to adjust the learning rates. More specifically, it increases the learning rate when the sum of activation values of the pre- and post-synaptic neurons is greater than 1, while it decreases the learning rate when it is less than 1.

The results show that *NLR*s usually have faster convergence than the *SBP* on the error surface when the same learning rate is chosen for them.

4.3.2 Choice of Performance Criteria

As explained in Section 4.2.3, *NRL*s were compared with the *SBP* rule on the same problems — two Boolean induction and six continuous problems — as in Chapter 3.

For the comparisons of the learning rules, the minimal computational efforts that are obtained with both *parallel* training (see Section 3.2.4.1) and the *sequential training* (see Section 3.9) were considered.

In the calculations of the minimal computational effort, both for sequential and parallel evaluation, the fraction of training patterns correctly predicted (success), a was set to 0.99. For parallel evaluation, z a probability of solving the problem was 0.99.

The leaning rate η^* and the number of epochs i^* for both sequential execution and parallel execution of training runs were identified using Equation (3.4) and (3.9). To compute these, each network was trained with 60 different learning rates, $\eta \in \{0.1, 0.2, \ldots, 6.0\}$. For each learning rate, the networks were trained 100 times (with different initial sets of weights). Cross-validation was not applied to the networks because of the heavy computational load. Instead, each data set was divided into two parts, i.e., training and testing sets.

4.3.3 Results with Parallel and Sequential Evaluations

Figure 4.1 and Figure B.1 report results for all test problems when parallel and sequential evaluation are considered respectively. They should be evaluated jointly with Table 4.3 and 4.4, respectively. The figures and tables use the same parameters and notations as those used to describe the results of the previous chapter. However, the colours of the plots in the figures are different since each graph shows the corresponding plot(s) of three learning rules (NLR(1), NLR(2) and the SBP). Also, in order to increase visibility, the accuracies of the continuous problems are shown here in two different figures, called ‘Training Accuracy’ and ‘Testing Accuracy’.

Tables 4.3 and 4.4 provide the values of the maximum number of epochs ($i_{\max}$) used to compute the cumulative probability, the required number of independent runs (R^*), the required number of epochs (i^*), the minimum computational effort ($E_p^*(a,z)$) and the best learning rate (η^*) to solve each test problem.

4.3.3.1 Parallel Evaluation

The plots named ‘Computational Effort’ in Figure 4.1 show: (a) the values of $p_i(a, \eta^*)$ for *SBP*, *NLR(1)* and *NLR(2)* (blue, orange and green solid lines, respectively, to be interpreted using the ordinate scale on the *left*) as a function of the epoch number, i; (b) the corresponding values of the computational effort $E_p(a, \eta^*, i, z)$ (purple and brown and teal dashed lines, respectively, to be interpreted using the ordinate scale on the *right*); (c) the corresponding values of i^* where the minimal computational effort $E_p^*(a, z)$ is achieved for each rule (orange, blue and green marks, respectively).

As one can see from Figure 4.1, the probabilities of successfully training the network $p_i(a, \eta^*)$ for *NLR*s increase faster than the probability for *SBP* in all testing problems/networks. Tables 4.3 provides key elements to compare these curves. The E_p^* values in the tables show that the *NLR*s usually need less computational efforts than the *SBP* to solve the testing problems. However, there are some exceptions which are that (a) *NLR(1)* and the *SBP* require almost the same computational effort on the BCW problem, and (b) *NLR(2)* requires more computational effort than the *SBP* on the e-coli problem. The amounts of advantage of *NLR*s over the *SBP* vary depending on the difficulties of the problems. Based on the average computational effort across all test problems, *NLR(1)* and *NLR(2)* need 26.0% and 19.1% less efforts than the *SBP*, respectively.

Similarly to the behaviour of *NLR* presented in the previous chapter, in Boolean induction problems, *NLR(1)* and *NLR(2)* also tend to achieve success with many short training attempts rather than with one long run.

Let us now look at the plots named ‘Error’ in Figures 4.1. The plots report the *median* error on the training set obtained in 100 runs executed with the optimal learning rate η^* for the *SBP*, *NLR(1)* and *NLR(2)* (solid blue, orange and green lines, to be interpreted using the ordinate scale on the *left*), respectively. Also, the plots report the p values obtained by applying the Wilcoxon signed rank test to the 100 pairs of errors values recorded for *NLR(1)* and *SBP* (the purple dotted line) and *NLR(2)* and *SBP* (brown dotted line) over all training time-steps to analyse which differences are statistically significant (p *value* ≤ 0.05). The p value lines are interpreted using the ordinate scale on the *right*.

As seen from the ‘Error’ figures, typically the error with *NLRs* drops more quickly than for *SBP*. These figures also show that the errors obtained with *NLR(1)* and *NLR(2)* for relatively easy problems — wine, DNA, and authorship — are statistically significantly better than for *SBP* almost for the whole duration of the training. However, for relatively harder problems — iris, e-coli, BCW and parity — error differences are usually not statistically significant after the learning rules approached

the minimum error they could reach. More specifically, error differences in iris, e-coli, BCW and parity problems are not statistically significant after approximately 100, 3600, 600 and 6500 epochs respectively. Because the errors of the learning rules for the 7-segment problem converge to the same level at the very beginning of the training (approximately epoch 25), error differences for the 7-segment problem are not statistically significant for most of the training.

The plots named 'Train Accuracy' and 'Testing Accuracy' in Figures 4.1 show the *median* accuracy of the corresponding training and testing set, respectively, *vs* epoch number obtained in 100 training runs executed with the optimal learning rate η^*. The plots also report the p values for the pairwise comparisons *NLR(1) vs SBP* and *NLR(2) vs SBP*. The learning rule and the p value lines are represented by the same colour and type of lines as those used in the 'Error' figures. The Boolean induction problems have only the 'Train Accuracy' figures as their train and testing sets are identical.

The training accuracy plots illustrate that the accuracy of the NLRs nearly always is above the corresponding accuracy for SBP, at least it increases faster than for SBP. As expected, for each problem (except for parity) and each pair of learning rules compared, the p value plots in the figures named 'Error' and corresponding figures named 'Train Accuracy' are almost the same throughout the whole training. In other words, the p value plots in the error figures and training accuracy figures are very similar for the same problem (except the parity) and the same pair of learning rules. The parity problem differs in this respect simply because in the early phases of learning all algorithms get stuck into near local minima for over 2000 epochs. However, *NLR(1)*, *NLR(2)* and *SBP* tend to get stuck on different loss values, which therefore are all statistically different. However, all three levels correspond to the same train accuracies (50%). These are therefore not statistically different.

In addition, the p value trajectories plotted in corresponding 'Testing Accuracy' and 'Train Accuracy' figures are very similar for the relatively easy problems (wine, DNA, and authorship) while the trajectories are slightly different for the relatively harder problems (iris, e-coli and BCW).

One can see from the accuracy plots, for iris, wine, authorship, DNA there is no evidence of overfitting, and once again the testing accuracy is higher for the NLRs sooner. For e-coli and BCW there is some evidence of overfitting, which suggests one should stop training sooner.

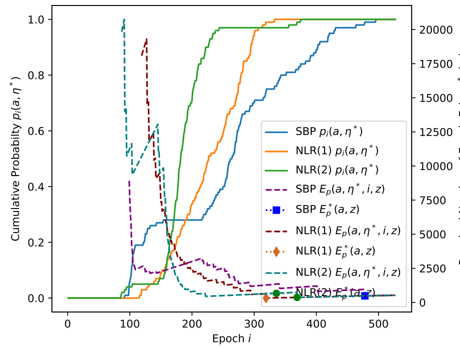
As seen in the cases of 7-segment and e-coli problems, although a learning rule is better than the other according to the comparison of minimum computational efforts provided in Tables 4.3, this superiority may sometimes not be seen in their pairwise comparison plots in Figure 4.1. As explained in Section 3.4.3, the comparison method, which computes the minimum computational effort, considers the lower

Table 4.3: Minimal effort and associated optimal learning rate, number of epochs and number of runs for **parallel** evaluation

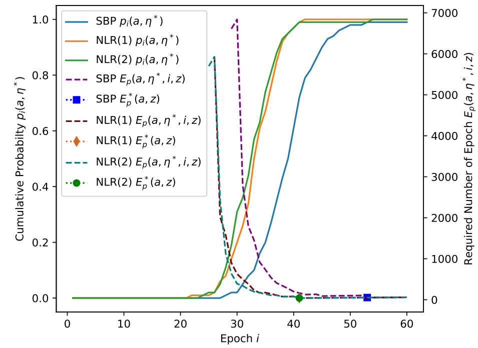
Problem	Rule	$i_{\max}$	R^*	i^*	$E_p^*(a,z)$	η^*
Iris	NLR(1)	3000	1	319	*319*	3.6
	NLR(2)	3000	1	369	*369*	3.2
	SBP	3000	1	478	*478*	4.1
Wine	NLR(1)	3000	1	41	*41*	2.8
	NLR(2)	3000	1	41	*41*	3.8
	SBP	3000	1	53	*53*	4.1
E-coli	NLR(1)	10000	1	4395	*4395*	1.2
	NLR(2)	10000	1	6938	*6938*	1.1
	SBP	10000	1	6079	*6079*	1.1
Authorship	NLR(1)	200	1	16	*16*	3.7
	NLR(2)	200	1	17	*17*	3.9
	SBP	200	1	21	*21*	4.2
BCW	NLR(1)	3000	1	1190	*1190*	3.0
	NLR(2)	3000	1	847	*847*	3.2
	SBP	3000	1	1188	*1188*	3.5
DNA	NLR(1)	200	1	17	*17*	1.4
	NLR(2)	200	1	18	*18*	1.8
	SBP	200	1	22	*22*	1.5
7-segment	NLR(1)	5000	12	216	*2592*	1.6
	NLR(2)	5000	22	128	*2816*	3.1
	SBP	5000	31	126	*3906*	4.4
Parity	NLR(1)	10000	2	7366	*14732*	2.3
	NLR(2)	10000	3	6435	*19305*	0.5
	SBP	10000	4	6652	*26608*	5.7

tail of the distribution that solves the problem while the pairwise comparisons plot the medians of those 100 runs in such figures. In a nutshell, such problem happens when the evolved rule has a higher number of successful runs and also has a higher number of overly poor runs than SBP within the 100 independent runs.

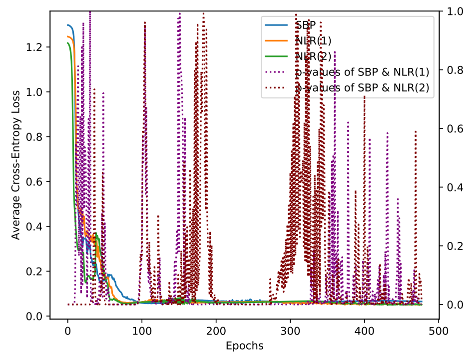
In summary, *NLRs* were compared with *SBP* using six continuous problems and two Boolean induction problems (see Table 3.4). As seen in Table 4.3, *NLRs* and *SBP* can solve all test problems. However, *NLRs* achieve this with usually less computational effort than *SBP*. The plots named 'Computational Effort' in Figure 4.1 also show this superiority. For almost all networks, the probability of successfully training networks for *NLRs* increases faster than for *SBP*, and naturally required number of epochs to solve problems for *NLRs* decreases faster than for *SBP*. Plots named 'Error', 'Train Accuracy' and 'Testing Accuracy' in Figure 4.1 show pairwise comparisons for the learning rules. *NLRs* and *SBP* reach almost the same training error, train accuracy and testing accuracy values at the end of the training. However,



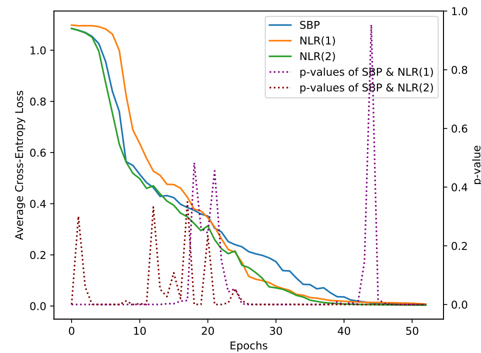
4.1.1 Iris Computational Effort



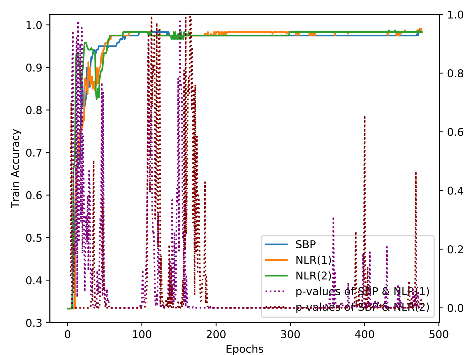
4.1.2 Wine Computational Effort



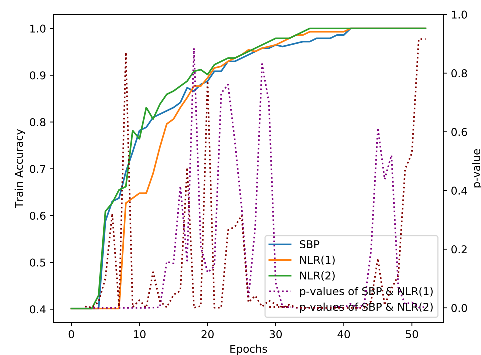
4.1.3 Iris Error



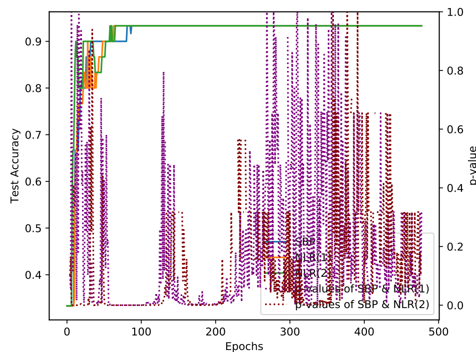
4.1.4 Wine Error



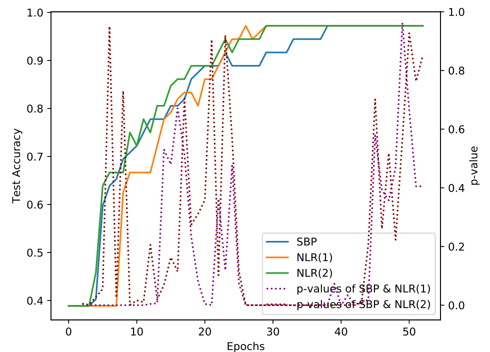
4.1.5 Iris Train Accuracy



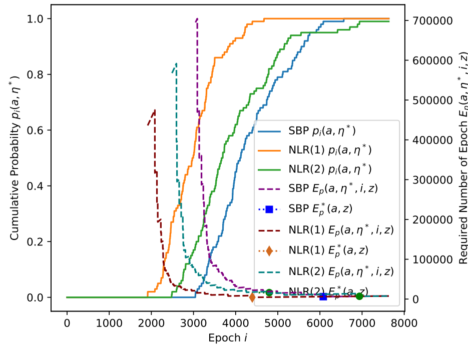
4.1.6 Wine Train Accuracy



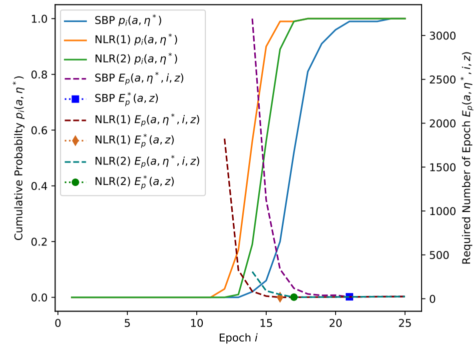
4.1.7 Iris Testing Accuracy



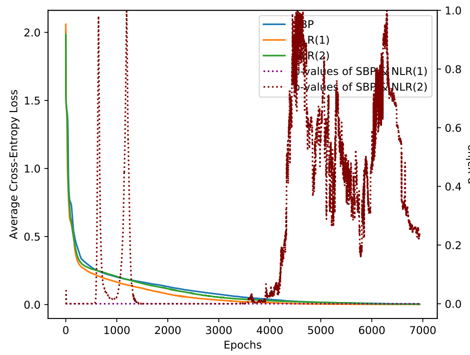
4.1.8 Wine Testing Accuracy



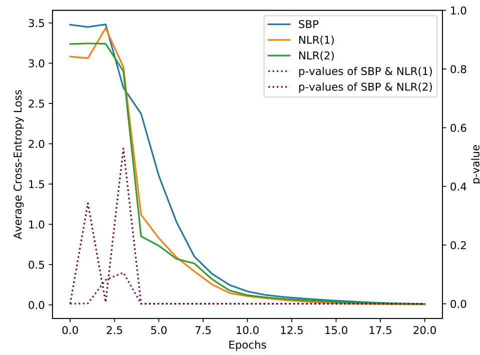
4.1.9 E-coli Computational Effort



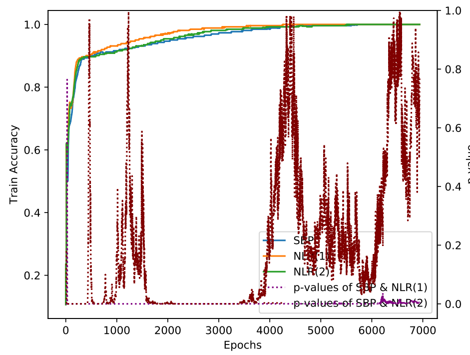
4.1.10 Authorship Computational Effort



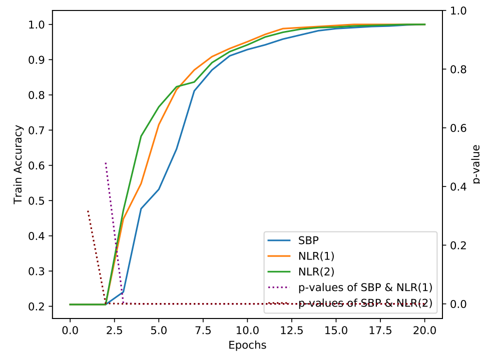
4.1.11 E-coli Error



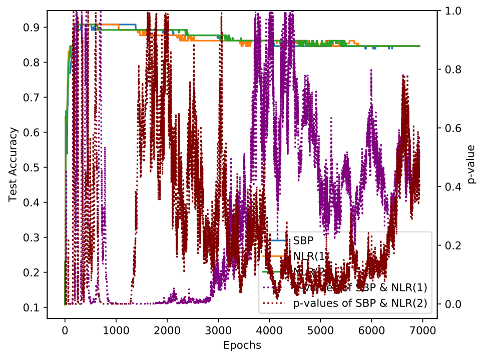
4.1.12 Authorship Error



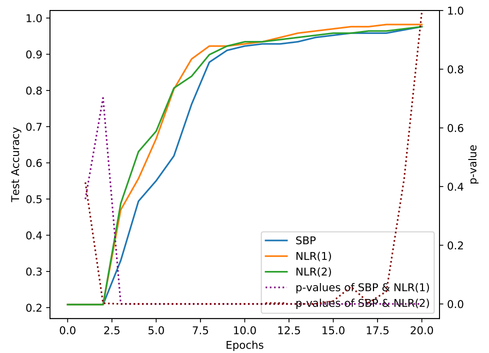
4.1.13 E-coli Train Accuracy



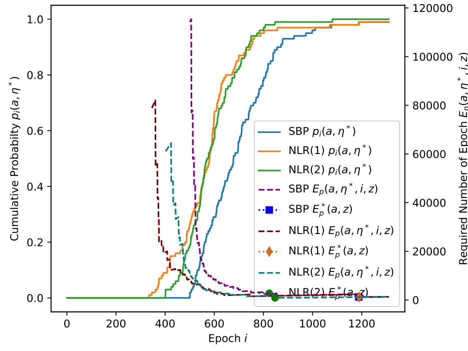
4.1.14 Authorship Train Accuracy



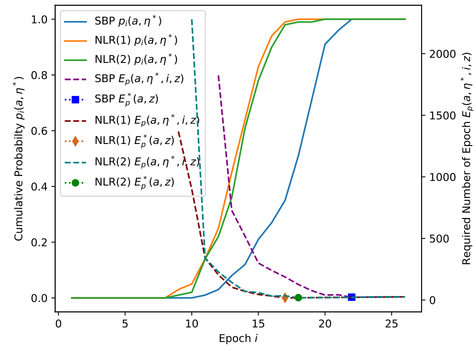
4.1.15 E-coli Testing Accuracy



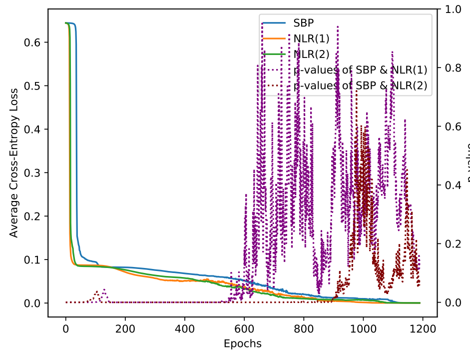
4.1.16 Authorship Testing Accuracy



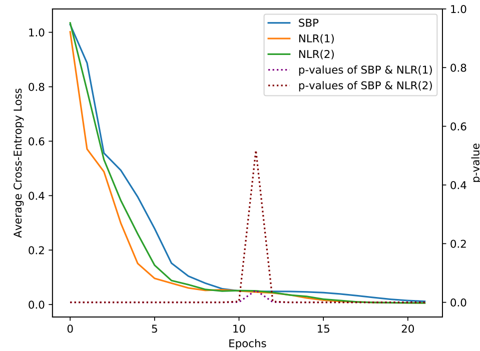
4.1.17 BCW Computational Effort



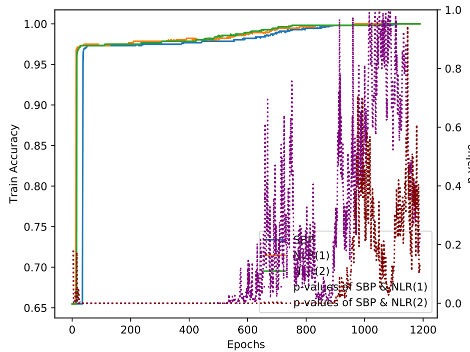
4.1.18 DNA Computational Effort



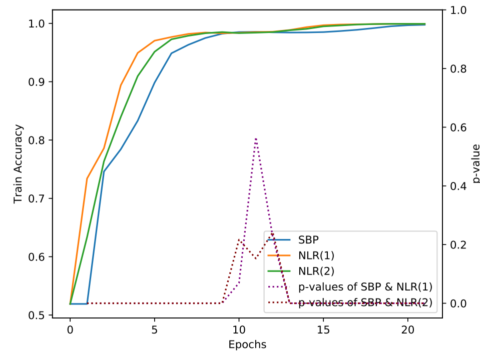
4.1.19 BCW Error



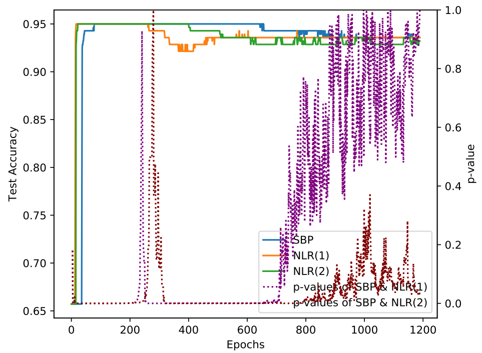
4.1.20 DNA Error



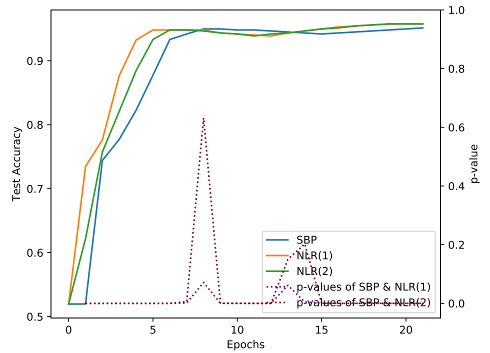
4.1.21 BCW Train Accuracy



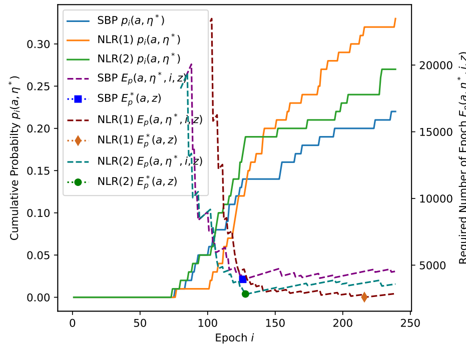
4.1.22 DNA Train Accuracy



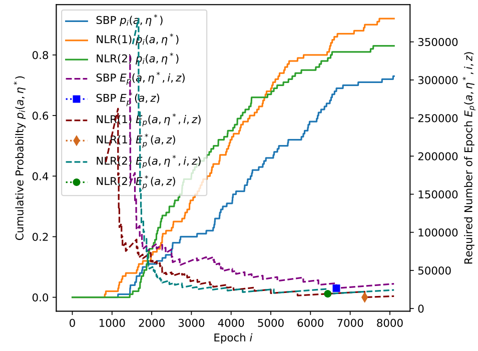
4.1.23 BCW Testing Accuracy



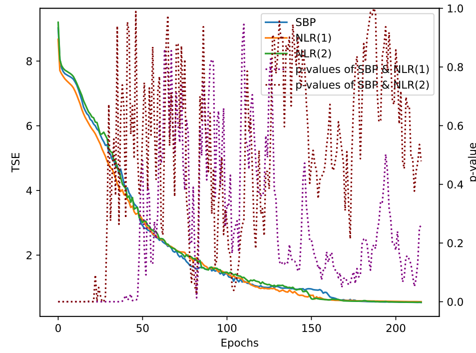
4.1.24 DNA Testing Accuracy



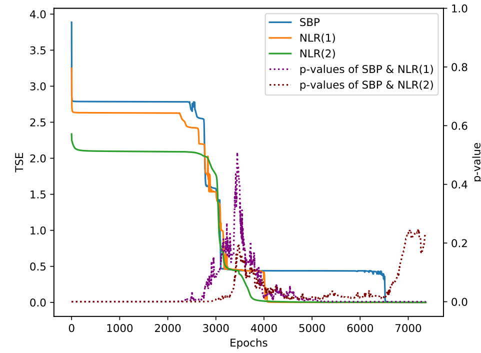
4.1.25 7-Segment Computational Effort



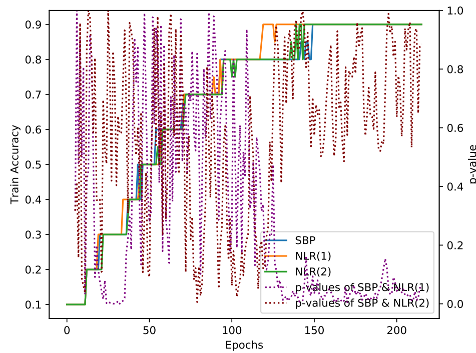
4.1.26 Parity Computational Effort



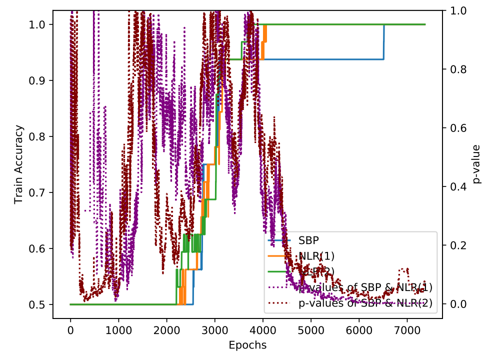
4.1.27 7-Segment Error



4.1.28 Parity Error



4.1.29 7-Segment Train Accuracy



4.1.30 Parity Train Accuracy

Figure 4.1: Results for all test problems when **parallel** evaluation is used (see text for explanations).

NLRs reach minimum error and maximum accuracy values faster than *SBP* in all problems/networks. These findings show that GP can routinely evolve learning rules that are faster than *SBP*. However, these rules evolved in this chapter, *NLR(1)* and *NLR(2)*, do not outperform *NLR*, which was evolved in the previous chapter.

4.3.3.2 Sequential Evaluation

The computational efforts of the learning rules that were compared were computed using Equation (3.11) in sequential evaluation. The results are usually very similar to those obtained with parallel evaluation when the problem sets are continuous. Figure B.1 (see Appendix B) and Table 4.4 provide the relevant plots and numerical results respectively. For the Boolean induction problem, the gains in percentage of the *NLR*s with respect to *SBP* obtained with sequential and parallel evaluation are similar. However, the computational efforts for such problems with sequential evaluation are significantly bigger than with parallel evaluation. Once again, note that it should not be concluded that one approach is superior to the other since they answer different questions.

4.4 Summary

In this chapter, GP-MLP is applied with the intent of speeding up the learning by evolving unstable learning rules. To do this, instead of penalising and/or rewarding the learning rules (individuals in the population) based on the oscillation on the error trajectory, they were allowed to oscillate.

Although in the work presented in this chapter, the training (evolution) problems are harder and the fitness functions are different than in the previous chapter, GP-MLP tends to produce learning rules that have similar behaviours. More specifically, here too, experimental results show that the GP-MLP can routinely produce faster learning rules than the *SBP* by adjusting the learning rate of the *SBP* on a weight by weight basis.

According to computational effort computed by both the parallel and sequential evaluation, *NLR(1)* and *NLR(2)* require less computational efforts than the *SBP* to solve the problems in the test suite. There are only three exceptions that NLR(2) is worse than the SBP on e-coli problem in the case of both parallel and sequential evaluations and on BCW problem in the case of parallel evaluation. However, in comparison with the *NLR* evolved in the previous chapter, they usually need more efforts to solve such problems.

GP-MLP in this chapter used less restrictive fitness function than previously used. The evolved rules in this chapter outperform the SBP on almost all testing problems. However, they are not as efficient as NLR. Its reason may be the configurations of

Table 4.4: Minimal effort and associated optimal learning rate, number of epochs and number of runs for **sequential** evaluation

Problem	Rule	$i_{\max}$	R^*	i^*	$E_p^*(a,z)$	η^*
Iris	NLR(1)	3000	1	333	*333*	3.6
	NLR(2)	3000	2	182	*364*	3.2
	SBP	3000	2	242	*484*	4.5
Wine	NLR(1)	3000	1	42	*42*	2.8
	NLR(2)	3000	1	42	*42*	3.0
	SBP	3000	1	56	*56*	2.7
E-coli	NLR(1)	10000	1	4656	*4656*	1.3
	NLR(2)	10000	2	3496	*6992*	1.4
	SBP	10000	1	6565	*6565*	1.1
Authorship	NLR(1)	200	1	17	*17*	3.5
	NLR(2)	200	1	18	*18*	3.3
	SBP	200	1	22	*22*	3.9
BCW	NLR(1)	3000	2	539	*1078*	3.3
	NLR(2)	3000	2	470	*940*	6.0
	SBP	3000	1	1262	*1262*	3.4
DNA	NLR(1)	200	1	17	*17*	1.9
	NLR(2)	200	1	19	*19*	1.4
	SBP	200	1	22	*22*	1.5
7-segment	NLR(1)	5000	4	163	*652*	1.3
	NLR(2)	5000	4	174	*696*	3.3
	SBP	5000	5	194	*970*	4.4
Parity	NLR(1)	10000	2	3557	*7114*	2.7
	NLR(2)	10000	2	3256	*6512*	0.6
	SBP	10000	2	5168	*10336*	5.2

GP-MLP such as training problems chosen, the number of training epochs and the number of errors/accuracies that contribute to the fitness evaluation. Therefore, the GP-MLP with more appropriate configuration could provide learning rules which are faster and more general than NLR.

Chapter 5

Evolving Initialisation Rules to Overcome the Vanishing Gradient Problem

5.1 Introduction

A particular interest of this thesis is to overcome the vanishing gradient problem, which is a long-standing obstacle to gradient-based training. As it was described in Chapter 2, the problem is that the gradients of the error with respect to the weights (usually in the earlier layers of a deep network) are very small or zero. It usually occurs when the network is deep, and the neurons use a sigmoidal activation function.

The previous chapters presented three different learning rules evolved to speed up the learning process of MLPs. The evolved rules (NLR, NLR(1) and NLR(2)) can successfully train the networks tested. However, they also suffer from the vanishing gradient problem when they are tested on deeper (more than four-hidden-layer) networks that use the sigmoid activation function and the standard initialisation method.

As described in Chapter 2, multi-hidden-layer networks outperformed single hidden layer networks in the early 1990s. However, they were abandoned due to the vanishing gradient problem. As reviewed in Section 2.3.3, there are a few approaches to overcome the vanishing gradient problem of deep networks such as greedy layer-wise pre-training, use of a different activation function, residual network, batch normalisation and weight initialisation. In fact, after the advent of newer activation functions such as ReLU, PReLU and LReLU that overcame this problem, research efforts on the use of the logistic function for deep networks were effectively abandoned.

An associated issue is that the training success and speed of neural networks are very sensitive to initial conditions. Several methods have been proposed to address

these issues via clever initialisation of the connection weights, but only very few considered deep networks with the logistic activation function.

The main purpose of this chapter is to show that a deep MLP which uses the logistic activation function can be trained without being affected by the vanishing gradient problem. This is obtained by a method that pre-trains the network using a weight-update rule evolved by GP to initialise the connection weights. As the learning rule evolved by GP is used only for initialisation of networks, it will be called *evolutionary initialisation method (EIM)*.

The proposed method was tested on six classification problems using various network structures. The numbers of hidden layers on the networks vary between 10 and 15 while the numbers of neurons in the hidden layers vary between 10 and 100. The method was compared with *Glorot* [30], *Kumar* [31] and *standard (SIM)* initialisation methods (as reviewed in Section 2.3.2). As we will see, when the networks were initialised by *EIM*, they could successfully learn all training problems. The networks, initialised by *Kumar* initialisation, could also learn almost all problems. However, it is slower than *EIM* when the test networks are deep and wide. None of the networks initialised by using the *SIM* and *Glorot* initialisation methods learned any of the problems.

5.2 GP System, Fitness Measure and Datasets

5.2.1 Linear GP System

In this chapter, GP was used to evolve a learning rule that was used to initialise the network weights, not for the whole training. The same GP system used in the previous chapters was used to perform the experiment in this chapter. As described in Chapter 3, the system uses a tournament selection with a tournament size of 3; point mutation applied with 50% and two-point crossover applied with 50%. Here, the population size is 600, and the program length (the number of instructions in each program) was set to 50 to obtain more complex rules because the system performed poorly when the program length was the one used in the previous chapters, i.e., 30.

The primitive sets used (see Table 5.1) consisted of some fixed constants and input registers which are the activations of pre- and post synaptic neurons, the derivative of the activation function, the error in the post-synaptic neuron and the SBP rule. Preliminary experiments showed that the modified SBP algorithm, which uses the network's layer numbers as an exponent of components of the derivative of the activation function, speeds up learning on deep networks. Considering this situation, in this chapter, two new constants and a variable were added to the standard

Table 5.1: Primitive set used in the experiments

Instructions	Instructions	Instructions
NOP	$r0 <- -r0$	$r1 <- r1+cl$
$r0 <- 0$	$r1 <- -r1$	$r0 <- r0+(1-ra)^{r1}$
$r1 <- 0$	$r0 <- r0+ra$	$r1 <- r1+(1-ra)^{r0}$
$r0 <- 0.5$	$r1 <- r1+ra$	$r0 <- r0+r1$
$r1 <- -0.5$	$r0 <- r0+ra_prev$	$r1 <- r0+r1$
$r0 <- -0.1$	$r1 <- r1+ra_prev$	$r0 <- r0*r1$
$r1 <- 0.1$	$r0 <- r0+re$	$r1 <- r0*r1$
$r0 <- -1$	$r1 <- r1+re$	$r0 <- r0*r0$
$r1 <- 1$	$r0 <- r0+rd$	$r1 <- r1*r1$
$r0 <- cL$	$r1 <- r1+rd$	$rs <-> r0$
$r1 <- cL$	$r0 <- r0+rSBP$	$rs <-> r1$
$r0 <- cRL$	$r1 <- r1+rSBP$	
$r1 <- cRL$	$r0 <- r0+cl$	

primitives previously used. They are defined as follows:

$$\begin{aligned} cl &= l, \\ cL &= L, \\ cRL &= 1/L \end{aligned}$$

where l and L are the layer number and the number of hidden layers in the network, respectively. In addition to the derivative of the activation function, its components (ra and $1-ra$) are also separately used as inputs. New instructions used in this chapter are highlighted in the table.

5.2.2 Fitness Measure

The fitness function in this experiment was defined considering the same approach as that used for experiments in Chapter 4. Fitness is the sum of a fraction of the lowest errors recorded during training. The function measures the fitness of each program (EIM) considering network errors of the whole training process (pre-training by *EIM* and main training by *SBP*). Because two different learning rules are used to complete training, for simplicity, both rules used are denoted as new training algorithm (*NTA*) in the fitness equations.

Similar to the fitness evaluation approach used in the previous chapters, each rule is applied to multiple problems and applied to each problem with multiple initial conditions (random weights and biases). Initially, the fitness values of each program were computed for every single problem. Then, each value was normalised to the corresponding fitness value evaluated with the *SBP*. Finally, the worst (highest) normalised value was assigned to the corresponding program as its fitness value.

Here are the equations that are used to compute the fitness of the rules:

$$fitness = \max(fitness_1, \ldots, fitness_k) \tag{5.1}$$

where k is the number of problems used in the GP run.

$$fitness_p = \frac{\sum_{s=1}^{n} \sum_{t=1}^{b} sort\left(NTAloss_p^s\right)_t}{\sum_{s=1}^{n} \sum_{t=1}^{b} sort\left(SBPloss_p^s\right)_t} \tag{5.2}$$

is the fitness contribution of problem p, $sort()$ is a function which sorts its argument and *NTAloss* and *SBPloss* are the average cross-entropy loss values of the NTA and SBP, respectively, obtained throughout the whole training process, n is the total number of seeds (randomly initialised set of weights and biases), and b is the number of loss values that contribute to the fitness evaluation. b is given by

$$b = \left\lceil \frac{cm}{100} \right\rceil \tag{5.3}$$

where c is percentage of errors that contribute to fitness evaluation and m is the total number of epochs. In the experiments the c was defined as 10.

In Equation (5.2), $NTAloss_{p,t}^s$ is computed by

$$NTAloss_{p,t}^s = \begin{cases} \text{error produced by} \quad EIM, & \text{if} \quad t \leq switching\ point, \\ \text{error produced by} \quad SBP, & \text{if} \quad switching\ point < t \leq m. \end{cases} \tag{5.4}$$

5.2.3 Training and Test Problems and Their Structures

In this chapter, two (training) problems — iris and wine classification problems — were used with four different initial sets of weights for fitness evaluation, i.e., $k = 2$ and $n = 4$. The networks were trained using programs in the populations until epoch 500, and then they were trained using the SBP from this point until epoch 2000, i.e., $switching\ point = 500$ and $m = 2000$. The number of pre-training epochs and the total number of training epochs were set after some preliminary experiments. The GP parameters and the network parameters, including the problems used for training (evolution) were chosen considering the low computational cost of evolution. However, relatively deep networks were chosen to obtain general rules that can successfully initialise different sizes of deep networks. Table 5.2 reports the structures and parameters of the training networks.

The proposed initialisation method, *EIM*, was compared with three different initialisation methods using six classification problems — iris, wine, authorship, DNA, BCW and 4-bit multiplexer — with two network structures. Once again,

Table 5.2: Network structure and parameters for the two problems used for evolving the initialisation rule

PROBLEM	INPUT	NUMBER OF HL	NN IN EACH HL	OUTPUT	η	NE FOR *EIM*	NE FOR *SBP*	SEEDS
IRIS	4	10	7	3	1	500	1500	4
WINE	13	10	7	3	1	500	1500	4

HL: Hidden layer. NN: Number of neuron. NE: Number of epoch.

Table 5.3: Network structures adopted for the different test problems used for comparing *NTA* and *SBP*

PROBLEM	INPUT	NUMBER OF HL	NN IN EACH HL	OUTPUT	NE FOR PRE-TRAINING
IRIS	4	10	10	3	2000
WINE	13	10	10	3	2000
AUTHORSHIP	70	10	10	4	2000
DNA	180	10	10	3	2000
BCW	9	10	10	2	2000
MUX	6	10	10	2	2000
IRIS	4	15	100	3	600
WINE	13	15	100	3	500
AUTHORSHIP	70	15	100	4	50
DNA	180	15	100	3	20
BCW	9	15	100	2	100
MUX	6	15	100	2	1000

HL: Hidden layer. NN: Number of neuron. NE: Number of epoch.

the reasons why being selected these problems are that they are commonly used to compare algorithms and do not lead a high computational cost when we used our testing architectures.

The network structures (number of neurons in the layers) used for comparisons were manually chosen with a small amount of trial and error (to balance convergence accuracy, computational load and overfitting) and are shown in Table 5.3. The input layer of each network consists of as many neurons as the features of in the problem. The output layer of each network has as many neurons as the number of classes in that problem. All networks were trained using a learning rate of $\eta = 0.25$. Each problem set has two network structures to test whether the initialisation method is sensitive to the network structure: one with depth $L = 10$ and one with depth $L = 15$. The $L = 10$ networks have relatively small size having of the order of 1,000 weights and 100 neurons, while the $L = 15$ networks, with approximately 150,000 weights and 1,000 neurons, have a much larger size.

5.3 Initialisation Methods Used for Comparison

When a network is initialised with random weights, the values are usually assigned using zero-mean normal distribution with a small standard deviation such as 0.01, 0.05, 0.1 and 0.2. To determine if EIM was competitive with other methods, here it was pitted against a standard initialisation, Glorot and Benjo's method [30] and Kumar's method [31] reviewed in Section 2.3.2.

So, the following distributions of initial weights were used for comparison:

- *SIM*: $W^l \sim \mathcal{N}(0,\ 0.01)$,
- *Glorot*: $W^l \sim \mathcal{N}(0,\ 2/(n^l + n^{l+1}))$,
- *Kumar*: $W^l \sim \mathcal{N}(0,\ 12.96/n^l)$,

n^l being number of neurons (including bias) in layer l.

The initial weights of the networks which will be initialised by EIM are assigned using the SIM before the pre-training process and then, the EIM initialises the networks by pre-training.

The input data of each network was scaled between 0 and 1, i.e., the min-max scale was applied to the input data.

5.4 Experimental Results

As indicated above, the *EIM* was evolved using GP to pre-train deep MLPs. In this experiment, iris and wine classification problems were used with 4 initial conditions (seed) for evolving the initialisation rule. The networks have 10 hidden layers, while each hidden layer of the networks has 7 neurons. As happened in the previous chapters, the last layer (softmax) of the networks were not included the evolution. In other words, the weights and biases between the last hidden layer and the output layer were updated using the standard update rule of the softmax function. The GP-MLP evaluated the individuals in the population considering both the initialisation phase (training until epoch 500 with *EIM*) and the training phase (training from epoch 501 until 2000 with *SBP*). The GP-MLP returned the following pre-training rule at the end of the evolution:

$$EIM = \left(\frac{1}{L}\right)^2 \times \left(\delta_j^l + \left(9 \times SBP^2 \times (SBP + a_j^l - 0.1) + a_j^l \times (1 - a_j^l) + \delta_j^l\right)^2\right)^2 \tag{5.5}$$

5.4.1 Qualitative Interpretation of the Evolved Initialisation Method

As one can see from Equation (5.5), the *EIM* always returns a positive value. In other words, when this rule updates a weight for a period of time, the weight always decreases (on average becoming more and more negative) as long as the delta weight is not zero. The *EIM* re-initialised the network weights that were initialised at random using a zero-mean normal distribution with a standard deviation of 0.1 at the beginning of pre-training. According to our analysis at the end of the pre-training, the weights had approximately the same standard deviation as before the pre-training (i.e., approximately 0.1) but had a negative-mean. The mean of weights at the end of pre-training varied depending on the network structure.

We should note that the task of the EIM is to pre-train the networks to initialise their weights and biases. If one uses the EIM as a learning rule for main training of a network, most probably the network cannot learn the training set since the EIM always returns a positive value.

More specifically, in the proposed method, there is a parabolic relation between the number of pre-training epochs and the optimal initial weights. In other words, the initial weights of a network approach optimal values during pre-training of the network over a period of time. However, if the networks are pre-trained with more than the desired number of epochs, the optimal or near-optimal initial weights begin to worsen.

The problem in this initialisation method is that there is no rule or formula to determine the optimal number of pre-training epochs. Determination of this number is based on trial and error and user experience. It mostly depends on network structures and the number of input features of the problem. In preliminary experiments, it was found that a relatively wide network requires a relatively small number of pre-training epoch while a deep one requires a large number of epochs.

In this experiment, the GP showed that deep MLPs needed to be initialised using an appropriate negative-centred normal distribution to solve the problems without suffering from the vanishing gradient problem. The reason why the weights of such networks require a negative-centred normal distribution and the relation between the centre value of the distribution and the network structure are the subjects of the next chapter. They will be explained theoretically there.

5.4.2 Results with the Evolved Initialisation Rule

As mentioned above, the proposed method was compared with three different initialisation methods using six classification problems and two different network structures for each problem.

Plots in Figures 5.1 (for networks with L=10 and L=15) report comparison

results of those initialisation methods that are Glorot initialisation (red lines), Kumar initialisation (orange lines), SIM (green lines) and EIM (blue lines). The solid lines are used for networks with L=10 and the dotted ones are used for networks with L=15. Since plots start from very similar positions, they usually overlap in the early epochs of graphs. The blue circle (for networks with L=10) and blue diamond shape (for networks with L=15) marks are used to show where the error is after the EIM has finished pre-training.

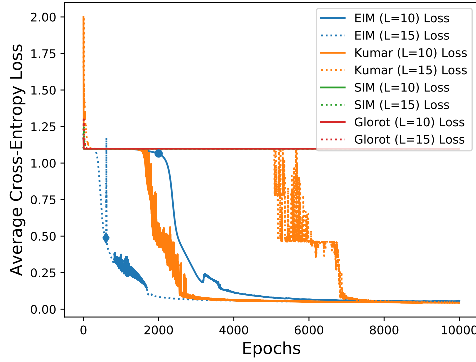
All networks were trained by *SBP* using the learning rate of 0.25 after each network was initialised by the corresponding initialisation method. The plots (losses, train accuracies and test accuracies) in the figures are the medians of values obtained in 30 independent runs (initial weights and biases).

The plots in Figure 5.1 indicate comparative results of the initialisation methods when the network sizes are similar to those used in evolution and when they are deeper and wider. Table 5.3 shows the network structures. The networks with ten hidden layers are indicated as L=10, and those with 15 hidden layers are indicated as L=15 in Figure 5.1. The plots named 'Loss' in Figure 5.1 report average cross-entropy loss of training sets obtained with SBP after networks are initialised. Note that if the initialiser is EIM, the plots report average cross-entropy loss of training sets obtained with EIM until pre-training has finished. The plots named 'Train Accuracy' and named 'Testing Accuracy' in the figure show training and testing accuracy of the corresponding problems/networks. The same learning rate (0.25) was used for each problem set and networks' pre-training phase. The number of pre-training epochs was manually adjusted depending on the networks structures and the number of input features of the problem (as shown in Table 5.3).

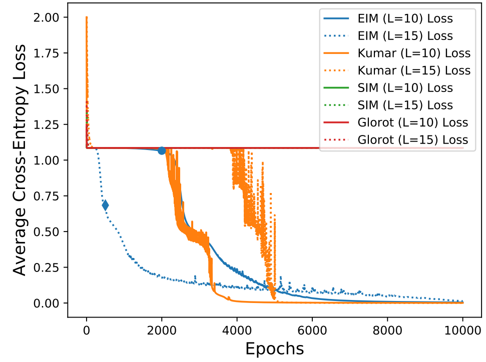
As one can see from the figure, in general, the results obtained with the large networks (with L = 15) are similar to those obtained with the networks (with L = 10). The networks initialised with the SIM and Glorot initialisation method cannot learn the data sets. When the networks were initialised using Kumar initialisation and EIM, the networks are not only successfully trained but also perform well on the testing data.

EIM and Kumar initialisation method, both, reach almost the same level of learning when the networks are trained with enough epochs. Although the plots are the median value of thirty independent runs, it is interesting that Kumar initialisation usually causes networks to oscillate on the training and testing plots. In contrast, EIM usually allows SBP to have smoother error descent and accuracy ascent than Kumar initialisation.

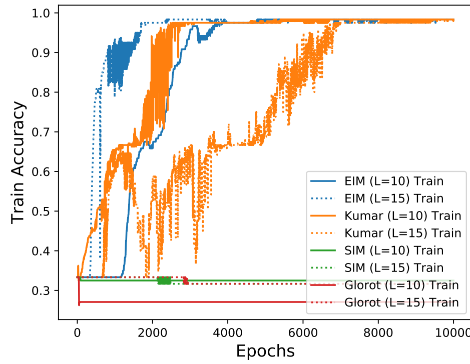
Let us compare these initialisation methods on the small networks (with L=10 and 10 neurons in each layer). The networks initialised with Kumar initialisation usually converge to the maximum performance faster than those initialised with the



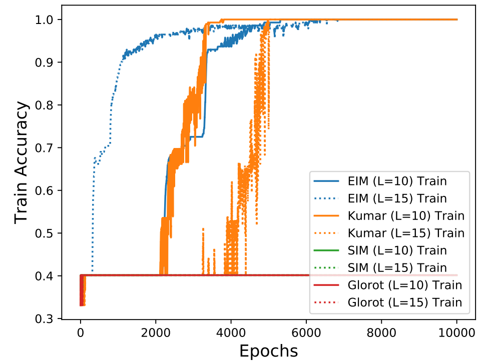
5.1.1 Iris Loss



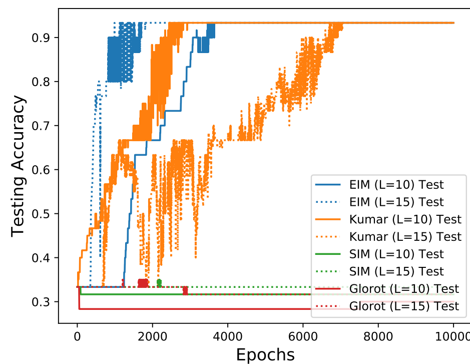
5.1.2 Wine Loss



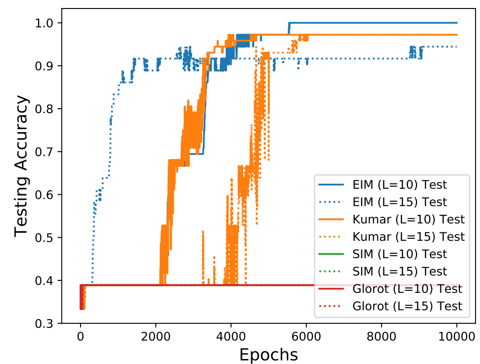
5.1.3 Iris Train Accuracy



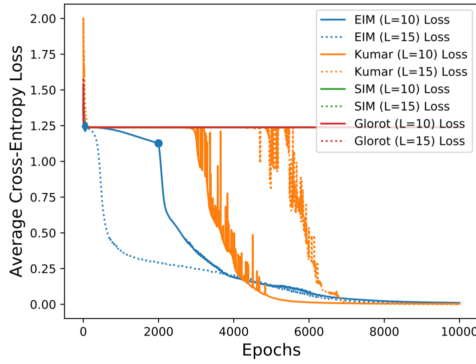
5.1.4 Wine Train Accuracy



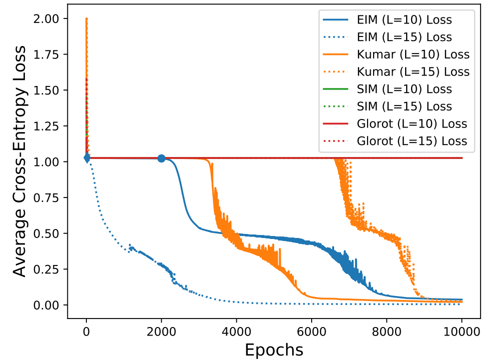
5.1.5 Iris Testing Accuracy



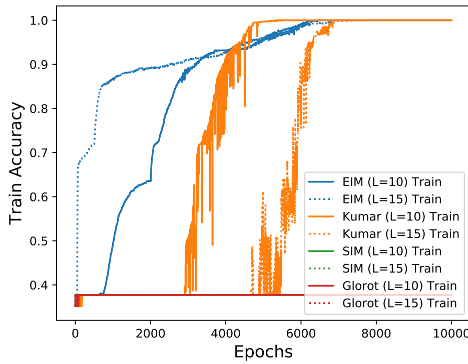
5.1.6 Wine Testing Accuracy



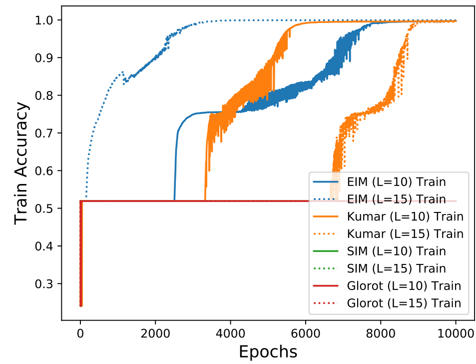
5.1.7 Authorship Loss



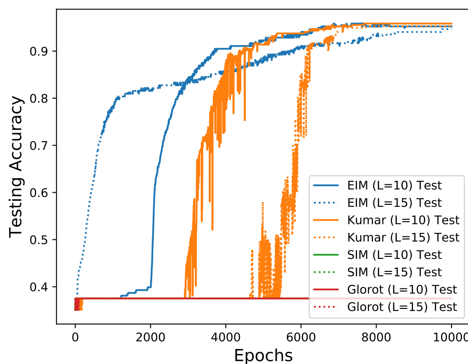
5.1.8 DNA Loss



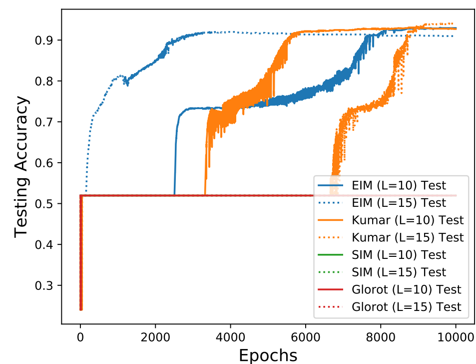
5.1.9 Authorship Train Accuracy



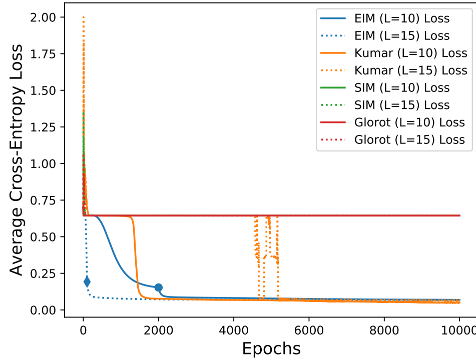
5.1.10 DNA Train Accuracy



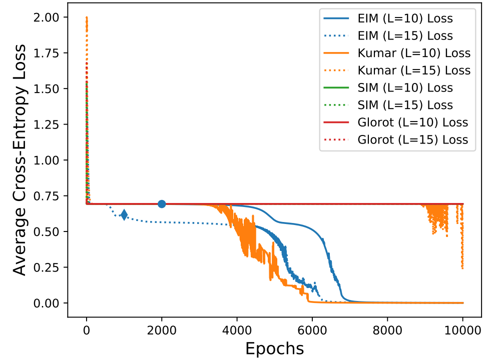
5.1.11 Authorship Testing Accuracy



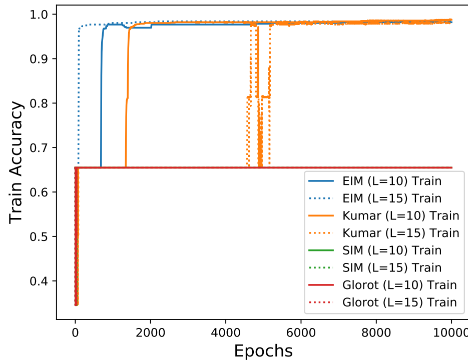
5.1.12 DNA Testing Accuracy



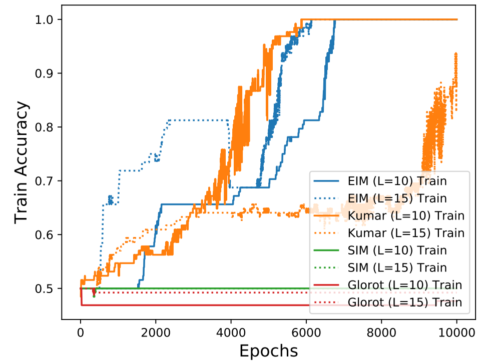
5.1.13 BCW Loss



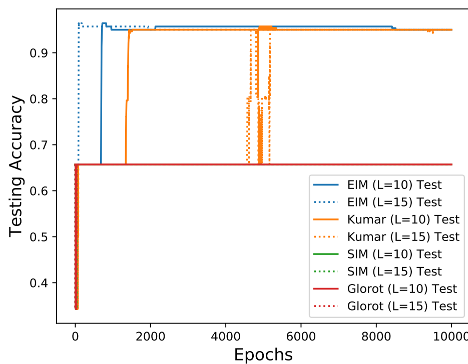
5.1.14 The Mux Loss



5.1.15 BCW Train Accuracy



5.1.16 The Mux Train Accuracy



5.1.17 BCW Testing Accuracy

Figure 5.1: Results for our six test problems when the SIM, Glorot, Kumar and EIM methods are used to initialise networks of depth L = 10 and depth L=15.

EIM. This is because EIM initialises the small networks (L=10) for 2000 epochs.

Let us now focus on these methods' performance on the large networks (with L= 15 and 100 neurons in each layer). EIM allows these networks to start learning as soon as SBP has begun main training. When the problems are the iris, wine, authorship and DNA, EIM obtains an accuracy of 80% within approximately 1000 epochs; it gets there within approximately 100 epochs and 2200 epochs when the problems are BCW and mux, respectively. Note that these values include the pre-training phase too. Kumar initialisation requires networks to do thousands of epoch for the accuracy to reach 80% for each problem.

Finally, let us compare the initialisation methods' performance on small networks (solid lines), and large networks (dotted lines) plotted in Figures 5.1. It is clear that EIM accelerates learning when the networks are large (with L = 15) in comparison with its results obtained using the small networks (with L = 10). These accelerations are also shown in Table 5.4 with numerical values of comparison methods' testing accuracies obtained on the small and large networks. The third and fourth columns — Testing accuracy (epoch 2000) and (epoch 5000) — of the table indicate that large networks converge faster than the small networks in the case of EIM initialisation. The opposite of this is true in the case of Kumar method. In other words, in the comparison of the networks initialised by the Kumar initialisation, it is seen that the large networks learn the problems slower than the small networks. One of the reasons why those networks initialised with the EIM solve the problems quickly is that they require relatively fewer pre-training epochs.

Figure 5.2 reports the sum of the gradient magnitudes in the layers of networks. The plots are obtained during the training of iris problem with the small network (L=10) (Plots, for EIM, include the pre-training phases too). The layer numbers in the figure are labelled increasing from the input layer to the output layer. The last layer's gradient magnitudes are not plotted in the figure as the networks use a softmax output layer. Note that in Figure 5.2, the plots for the networks initialised by the EIM and Kumar methods are plotted using a linear scale while those initialised by SIM and Glorot are plotted using a logarithmic scale. It is expected that the network will learn relatively quickly if the layers have a relatively large gradient. Figure 5.2 shows that gradients in their layers (particularly in the hidden layers close to the input layer) do not vanish when the networks initialised with the EIM and Kumar initialisation. As a result of this, Figure 5.1 and 5.2 indicate that these initialisation methods allow SBP to train deep networks without suffering from the vanishing gradient problem. The figure also shows that learning in all layers of the network initialised using EIM immediately begins after pre-training. The network initialised by Kumar initialisation requires approximately 2,000 epoch to start learning.

In summary, the evolved initialisation method, EIM, is compared with three

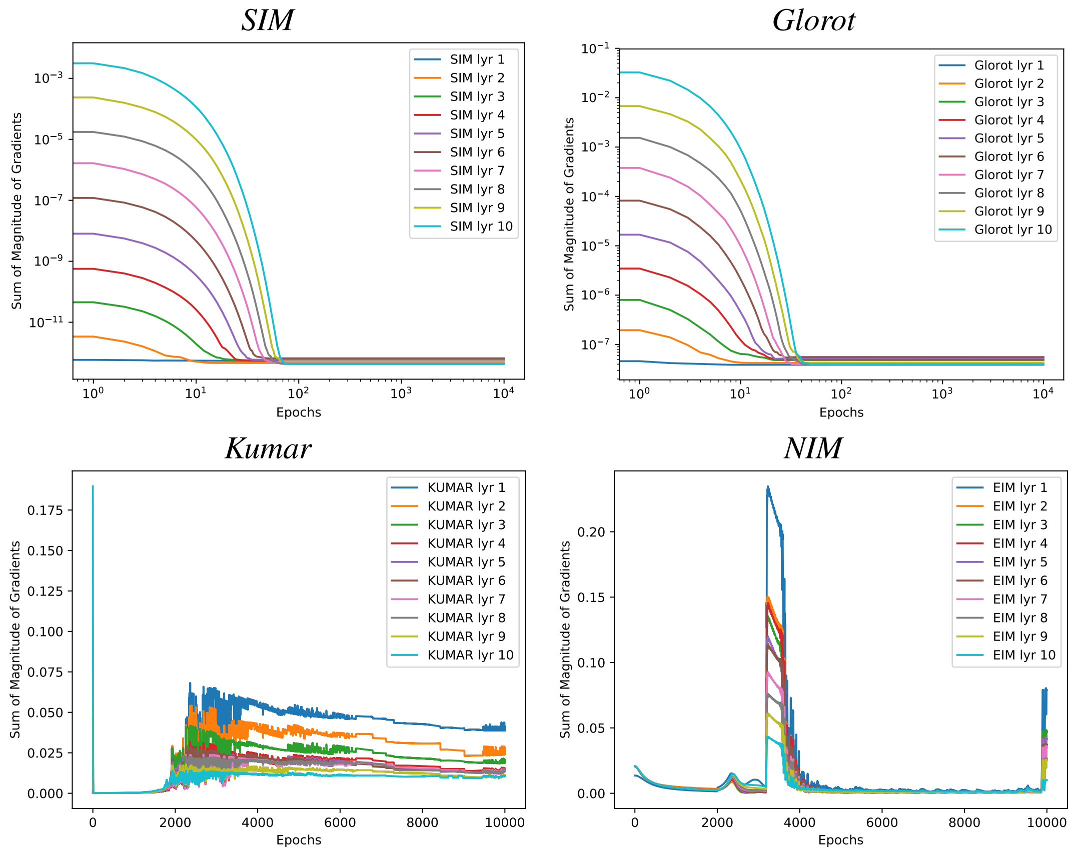


Figure 5.2: Sum of gradient magnitudes in each layer when the *EIM*, *Kumar*, *SIM* and *Glorot* initialisation methods are applied to the small (L=10) and large (L=15) networks of iris problem.

Table 5.4: Testing accuracy of the EIM and Kumar initialisation method on test networks.

Problem	Method	Testing accuracy (epoch: 2500)	Testing accuracy (epoch: 5000)	Testing accuracy (epoch: 1000)
Iris	EIM (L=10)	0.7333	0.9333	0.9333
	Kumar (L=10)	0.8667	0.9333	0.9333
	EIM (L=100)	0.9333	0.9333	0.9333
	Kumar (L=100)	0.6333	0.7333	0.9333
Wine	EIM (L=10)	0.6667	0.9722	1.00
	Kumar (L=10)	0.6667	0.9722	0.9722
	EIM (L=100)	0.9167	0.9167	0.9444
	Kumar (L=100)	0.3889	0.9167	0.9722
Authorship	EIM (L=10)	0.7560	0.9226	0.9524
	Kumar (L=10)	0.3750	0.9226	0.9583
	EIM (L=100)	0.8274	0.8810	0.9464
	Kumar (L=100)	0.3750	0.5536	0.9583
DNA	EIM (L=10)	0.5196	0.7535	0.9309
	Kumar (L=10)	0.5196	0.8556	0.9262
	EIM (L=100)	0.8964	0.9168	0.9089
	Kumar (L=100)	0.5196	0.5196	0.9403
BCW	EIM (L=10)	0.9571	0.9571	0.95
	Kumar (L=10)	0.95	0.95	0.95
	EIM (L=100)	0.95	0.95	0.95
	Kumar (L=100)	0.6571	0.95	0.95
4-bit Mux	EIM (L=10)	0.6562	0.6875	1.00
	Kumar (L=10)	0.6094	0.8750	1.00
	EIM (L=100)	0.8125	0.7969	1.00
	Kumar (L=100)	0.6250	0.6406	0.8750

different initialisation methods using six classification problem and two different network structures for each problem. As seen from Figure 5.1, only EIM and Kumar initialisation can successfully train the networks. On the small networks, the networks initialised by Kumar initialisation converge faster than those networks initialised by EIM while the opposite is almost always true for the large networks. The results show that Kumar initialisation is sensitive to the network size while EIM is more robust than Kumar initialisation. In the case of the networks initialised by Kumar method, the large networks begin and complete the learning later than the small ones. In EIM-initialised networks, learning begins immediately in both large and small networks when SBP starts training (after the pre-training phases are completed). Also, large networks usually learn problems faster than small networks. This is because EIM pre-trains the small networks for 2000 epochs and this number is usually much lower for large networks.

In conclusion, in this work, GP provided a pre-training rule (EIM) and taught us that the logistic activation function efficiently works in deep networks without suffering from the vanishing gradient problem when the networks' initial weights are initialised with a negative-mean distribution. Deep networks can successfully be trained by SBP after they are pre-trained by EIM. The drawback of this method is that the number of pre-training epochs varies depending on the networks, and it is defined by trial and error or user experience. However, after having this lesson from the GP, we theoretically developed an efficient initialisation method, that does not require a pre-training phase, to overcome the vanishing gradient problem. The details of the theoretical initialisation method will be presented in the next chapter.

5.5 Summary

In this chapter, an initialisation algorithm was evolved for deep MLPs to be successfully trained without suffering from the vanishing gradient problem. The proposed algorithm, EIM, was evolved by GP and is used as a pe-training rule, which updates the network weights. After the EIM initialises (pre-trains) the networks, the SBP takes over to complete the training of the networks.

The EIM was compared with three different initialisation methods using six classification problems. For each problem, two different network structures (L=10 and 10 neurons in each hidden layer and L=15 and 100 neurons in each hidden layer, given in Table 5.3) were used. Results showed that both networks could solve the test problems when the corresponding network is initialised using the EIM. When the networks were small, the performance of the EIM was competitive to the Kumar initialisation on all problems. However, when the networks are relatively deep and wide (i.e., L=15), the networks initialised using EIM solve the problems faster than those initialised using Kumar initialisation.

The EIM has also a drawback. There is no simple rule or formula to determine the number of pre-training epochs. It is defined based on user experience. However, one could train the networks until being sure that the network is getting worse and could record the set of weights that provide the minimum error throughout the pre-training. Then, one could use the set of weights for the initial weights of the network. This is expensive in terms of the computational time but safer and easier than trial and error.

In the last decade, the logistic activation function has been abandoned because of the vanishing gradient problem which prevents the deep networks from learning. This work showed that the logistic activation does not suffer from the vanishing gradient problem when the network is properly initialised.

Chapter 6

A Simple Initialisation Method to Train Deep Neural Networks Using the Logistic Activation Functions

6.1 Introduction

The previous chapter reported that deep MLPs which use logistic activation function can successfully be trained with the SBP when the weights in the networks are initialised with a negative-mean of distribution. This chapter is an extension of the previous chapter and proposes an initialisation rule that does not require pre-training process.

This chapter focuses on networks using sigmoid activation functions and want to verify to what extent the vanishing gradient problem depends on the choice of the *mean* (as opposed to the variance) of the initial weight distribution. This investigation is both theoretical and empirical.

This chapter: (1) presents a simple theoretical model that illustrates how the expected value of the gradient is affected by the expected value of the initial weight distribution; (2) presents a set of empirical tests that corroborate the main findings of the theory in relation to the initial gradients; (3) provides further insights on how to initialise networks to maximise such gradients; (4) further refines these insights through an analysis of variance of the initial net inputs; (5) as a result of these, develops a new initialisation strategy which sets the mean initial weights to $-\min(1, 8/\text{number of neurons in layer})$ — a method that will be called as *new initialisation method (NIM)* hereafter; (6) applies a stochastic hill-climber to the networks to optimise the magnitude of initial gradients in the first layer; and also experimentally compares NIM- and hill-climber-initialised networks, finding that NIM provides near-optimal initial weights.

6.2 Simple Theoretical Model of Initial Gradients

6.2.1 Expected Values of Network State and Parameters after First Forward and Backward Propagations

The dynamic of net_j^l, a_j^l, δ_j^{l+1} and Δw_{ij}^l were presented in Section 2.2 in Chapter 2. This section attempts to perform an analysis of the magnitude Δw_{ij}^l from these equations *at the very first iteration of the learning process*, i.e., when a training example has been propagated forward once and error has been back-propagated once, but the weights have not been updated yet, in a batch-type back-propagation learning algorithm. This analysis will be performed by introducing drastic approximations. Results will then be empirically verified for veridicity in Section 6.3.

In relation to the initialisation phase of a network, the weights can be seen as stochastic variables $w_{ij}^{l-1} \sim \mathcal{N}(\mu^l, \sigma^2)$ (normally $\mu^1 = \mu^2 = \cdots = 0$, but here we do not make this assumption, while we assume that the same σ is used for all layers of weights). Because of this, also the net input net_j^l is a stochastic variable, and so are the activations, a_j^l and the delta's, δ_j^{l+1}.

Let us consider the expectation of the net input:

$$\mathrm{E}[net_j^l] = E\left[w_{0j}^{l-1} + \sum_i a_i^{l-1} w_{ij}^{l-1}\right] = \mu^l + \sum_i E[a_i^{l-1} w_{ij}^{l-1}] = \mu^l \left(1 + \sum_i E[a_i^{l-1}]\right), \tag{6.1}$$

where the last step is the result of a_i^{l-1} and w_{ij}^{l-1} being initially uncorrelated and, so, $E[a_i^{l-1} w_{ij}^{l-1}] = E[a_i^{l-1}]E[w_{ij}^{l-1}]$.

Note that this equation shows that $\mathrm{E}[net_j^l]$ does not depend on j, so we can use the term $\mathrm{E}[net^l]$ to indicate the expected net input of any neuron in layer l. Note also, that with the exception of the input and output layers, due to symmetries, $E[a_i^l]$ does not depend on i. So, we can use the term $E[a^l]$ to indicate the expected activation of any neuron in layer l. Thus:

$$\mathrm{E}[net^l] = \mu^l \left(1 + \sum_i E[a^{l-1}]\right) = \mu^l \left(1 + n^{l-1} E[a^{l-1}]\right). \tag{6.2}$$

Let us now consider the expectation for of a_j^l. By definition, we have that

$$E[a^l] = E\left[f(net^l)\right], \tag{6.3}$$

where we omitted the subscript j for the reasons mentioned above. This cannot be

computed readily. However, as a first order approximation we could write

$$\begin{aligned} E[a^l] &\cong f\left(E\left[net^l\right]\right) \\ &= f\left(\mu^l\left(1+n^{l-1}E[a^{l-1}]\right)\right), \end{aligned} \tag{6.4}$$

where we used Equation (6.2). We should note that this equation gives us a recursion (forward propagation) which, with initial condition $E[a^1] = E[x]$, x being the mean value of the input activation across all neurons and input patterns in the training set, allows us to estimate $E[a^l]$ for the whole network.

Turning now our attention on the expectation for the δ's, we have

$$E[\delta_j^{l+1}] = E[f'(net_j^{l+1})\sum_i w_{ji}^{l+1}\delta_i^{l+2}] \cong E[f'(net_j^{l+1})]E[\sum_i w_{ji}^{l+1}\delta_i^{l+2}], \tag{6.5}$$

where the last step is true on the assumption that $f'(net_j^{l+1})$ and $\sum_i w_{ji}^{l+1}\delta_i^{l+2}$ are weakly correlated. However, also since δ_i^{l+2} and w_{ij}^{l-1} are initially uncorrelated, this transforms into:

$$E[\delta_j^{l+1}] \cong \mu^{l+1}E[f'(net_j^{l+1})]\sum_i E[\delta_i^{l+2}]. \tag{6.6}$$

For the same reasons that $E[net_j^l]$ does not depend on j, so do any functions of net_j^l. Hence, in Equation (6.6), the subscript can be omitted in $E[net_j^{l+1}]$. So, this can be rewritten as $E[net^{l+1}]$. For reasons of symmetry also E$[\delta_j^{l+1}]$ (except for the output neurons) does not depend on the subscript j. So, hereafter, we will denote these quantity E$[\delta^{l+1}]$. Then the previous equation can be rewritten as

$$E[\delta^{l+1}] \cong E[f'(net^{l+1})]\mu^{l+1}n^{l+1}E[\delta^{l+2}]. \tag{6.7}$$

If we further assume that $E[f'(net^{l+1})] \cong f'\left(E\left[net^{l+1}\right]\right)$, then we have

$$\begin{aligned} E[\delta^{l+1}] &\cong f'\left(E\left[net^{l+1}\right]\right)\mu^{l+1}n^{l+1}E[\delta^{l+2}] \\ &= f'\left(\mu^{l+1}\left(1+n^lE[a^l]\right)\right)\mu^{l+1}n^{l+1}E[\delta^{l+2}], \end{aligned} \tag{6.8}$$

where in the last step we used Equation (6.2) once again. We should note that, once the values of $E[a^l]$ are known for all l, this equation gives us a recursion on the δ's (backward propagation). The initial conditions are $E[\delta^L] \cong E[a^L](1-E[a^L])E[e]$, where e is the error for a generic output in a generic teaching input in the training set. This recursion allows us to estimate $E[\delta^{l+1}]$ for the whole network.[1]

[1] In the initial conditions for the two recursions, we have ignored, without loss of generality, the fact that the expected $E[x]$ and $E[e]$ may differ from neuron to neuron. This is because with all weights in a layer having the same mean in the first training epoch, any neuron-to-neuron differences are averaged out at the level of first hidden layer and penultimate hidden layer, respectively.

With similar arguments we have that $\mathrm{E}[\Delta w^l_{ij}]$ will not depend on either i or j and, so, we will denote it $\mathrm{E}[\Delta w^l]$ hereafter. Then

$$E[\Delta w^l] = \eta E[a^l \delta^{l+1}] \cong \eta E[a^l] E[\delta^{l+1}], \tag{6.9}$$

on the assumption that a^l and δ^{l+1} are weakly correlated. In conjunction with the results of the previous recursions for $E[\delta^{l+1}]$ and $E[a^l]$, this equation allows us to estimate the expected gradient in weight-space for every layer in the network.

6.2.2 Relationship between the Mean of the Distribution of Initial Weights and Network Gradients

These expressions allow one to calculate the approximate magnitude of the initial gradient Δw^l for all l within a network as a function of the choice of the mean initial weights μ^l, the depth of the network L, the number of neurons in each layer n^l, the characteristics of the problem with its encoding (represented by $E[x]$ and $E[e]$) and, of course, the learning rate η. One could then numerically optimise such a function, to determine the architecture, encoding and initial conditions that produce the maximum initial gradient. Our hypothesis is that starting from such a good position, the standard back-propagation algorithm would then be able to train even very deep networks (large L).

However, this simplified analysis already suggests that if one wants to maximise the magnitude of the gradient the traditional choice of $\mu^l = 0$ for all l is poor, as it guaranteed to result in $E[\delta^l] = 0$ (see Equation (6.8)). Based on this, we should expect the optimisation process described in the previous paragraph to result in weight initialisation with non-zero means, $\mu^l \neq 0$.

Furthermore, it is clear from Equation (6.8), that to have some hope of successfully back-propagating errors, we also need to ensure that $f'\left(\mu^{l+1}\left(1+n^l E[a^l]\right)\right) \neq 0$.

In fact, the equation allows us to infer that a *necessary condition* to have *no attenuation* in the expected δ's during backward propagation (so gradients would not vanish) is

$$f'\left(\mu^{l+1}\left(1+n^l E[a^l]\right)\right)\mu^{l+1} n^{l+1} \cong \pm 1 \tag{6.10}$$

for all l.

One way to ensure that this equation is satisfied would be to require that $f'\left(\mu^{l+1}\left(1+n^l E[a^l]\right)\right)$ takes some non-zero value e.g., $\frac{1}{d}$, d being a positive (given the monotonicity of activation functions) constant, which would then allows us to

solve Equation (6.10) for μ, obtaining the relationship

$$\mu^l \cong \pm \frac{d}{n_l}. \tag{6.11}$$

This surprising result suggests that in order to preserve initial gradients, the mean of the initial weights should vary, from layer to layer, in a manner inversely proportional to the size of the hidden layers. Even more surprisingly, the theory predicts that, if $|\mu^l| > \frac{d}{n_l}$, errors can even be *amplified* as they are back-propagated.

Of course, for this strategy to work, one would need to choose d such that the expected activation in each layer $E[a^l]$ satisfied

$$f'\left(\pm \frac{d}{n^{l+1}} \left(1 + n^l E[a^l]\right)\right) = \frac{1}{d}. \tag{6.12}$$

However, typically, this is not difficult to achieve. For instance, if we use $d = 8$ and the logistic activation function, if $n^{l+1} = n^l = 5$, the equation is satisfied by $E[a^l] \cong 0.02$, if $n^{l+1} = n^l = 10$ it is satisfied by $E[a^l] \cong 0.12$, while $E[a^l] \cong 0.21$ satisfies the equation for $n^{l+1} = n^l = 100$. For our experiments conducted in this chapter, we will use $d = 8$ or $d = 6$.

The theory and its main findings will be tested in Section 6.3 with the set of problems listed in Section 6.3.1.

6.3 Corroboration of the Theory and Resulting Insights

6.3.1 Benchmark Problems and Network Structures

As indicated in Section 6.1, six classification problems were used to test the proposed initialisation method in this chapter. These problems — the iris, wine, analcat data authorship (authorship), DNA, breast cancer (BCW) and 4-bit multiplexer — are the same as those used in the previous chapter.

6.3.2 Empirical Estimation of Gradients in Deep Networks with Identically-sized Hidden Layers

This section aims to corroborate the main findings of the theory in relation to the initial gradients and provide further insights on how to initialise networks to maximise such gradients. To achieve this, it presents a set of empirical tests.

In order to do a first corroboration of the theory, deep networks that use the logistic activation function and that have the same number of neurons, n, in the hidden layers were considered and the following was done:

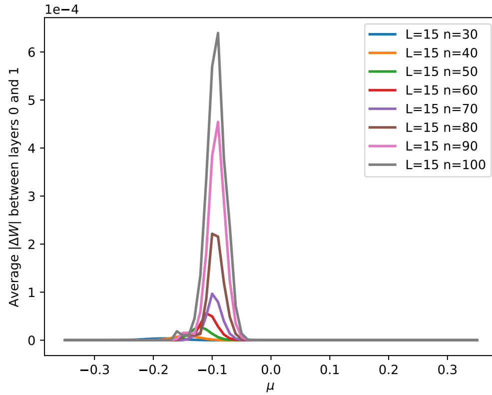
- the networks were initialised using different width (n) and depth (L) with distributions of weights having different means, μ, which were the same for all layers (as a result of all layers having the same number of neurons n)),[2]
- the weight distributions were Gaussian with standard deviation 0.1 in all cases,
- one step of forward propagation of the training set and one of error back-propagation were performed and the mean $|\Delta W|$ in all layers was recorded,
- the process was repeated 30 times with different initial random seeds and the corresponding results were averaged,
- the process described above was repeated for the 6 different problems in Section 6.3.1.

As an illustrative example, plots in Figure 6.1 show the mean $|\Delta W|$ of a particular layer of a network that uses the authorship problem, has 15 hidden layers and is initialised with different distributions of initial weight having different mean. Each plot in the subfigures shows those mean $|\Delta W|$s for a network with a different number of neurons in the hidden layers. Figure 6.1.1, 6.1.2 and 6.1.3 show the mean $|\Delta W|$ of the input layer, an intermediate layer and the output layer of the network respectively. Very similar results were obtained for other problems and depths.

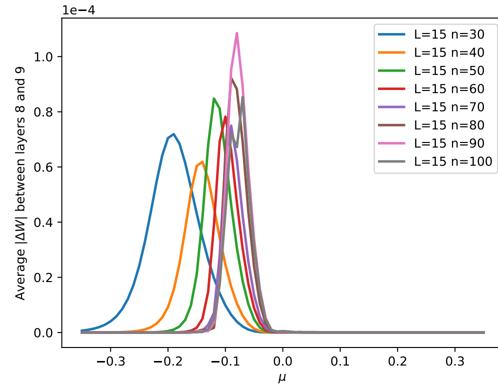
Figure 6.2 shows the heat map of those $|\Delta W|$ plotted in Figure 6.1. The scale at the bottom of each subfigure shows the relationship between the colour and $|\Delta W|$ values. The blackest is the lowest value, almost zero; the whitest is the highest value which vary from problem to problem and layer to layer. The lightest colour for the corresponding n shows the optimal μ for that n.

As indicated in Section 6.2, the input and output layers were ignored when the theory was developed. However, we should expect the theory to hold for layers far from these two layers. As one can see for an intermediate layer (Figure 6.1.2), the magnitude of the weight changes, $|\Delta W|$, is biggest for values of μ which are different from 0 (in fact, negative). Furthermore, we see how the optimal μ depends on the number of neurons in a layer, in an inversely proportional manner, following approximately the relationship $\mu = -\frac{8}{n}$, as we postulated in Section 6.2. The inverse relationship between the optimal μ and n is clearly visible in Figure 6.2.2, where as a reference we plotted the line $\mu = -\frac{6}{n}$.

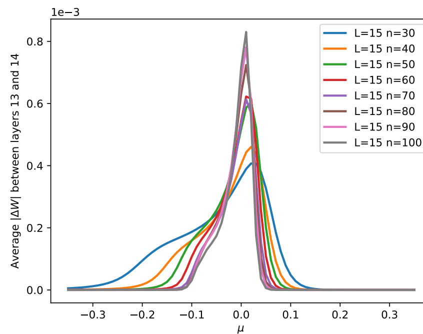
[2]This of course, in general, could not apply to the input layer, for which in principle we should use a different μ based on our simplified theory. However, for homogeneity of treatment and also to avoid departing too heavily from the theory, we have constrained the values of n tested to be not too different from the number of input features of each problem. Values of n, L and μ used in these tests vary from problem to problem. In the illustrative example, authorship problem, they were set to $n \in \{30, 40, \ldots, 70\}$, L=15, $\mu \in \{-0.35, -0.34, \ldots, 0.35\}$.



6.1.1 $|\Delta W|$ between layers 0 and 1.



6.1.2 $|\Delta W|$ between layers 8 and 9.



6.1.3 $|\Delta W|$ between layers 13 and 14.

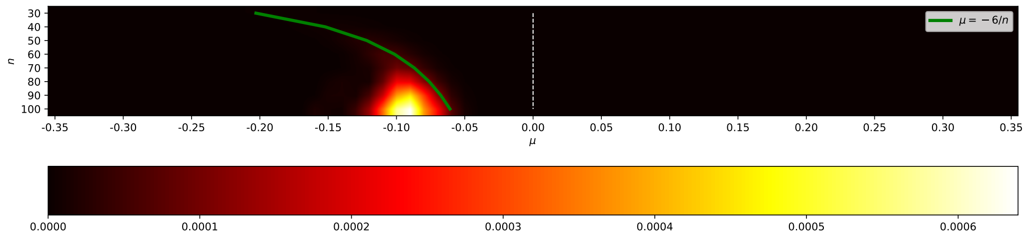
Figure 6.1: Mean $|\Delta W|$ for the authorship problem for a network with $L = 15$ layers and different numbers of neurons, n, in the hidden layers, for three representative layers.

Looking at the output layer (Figures 6.1.3 and 6.2.3), we see that the optimal μ does not depend much on n and it is close to zero or slightly positive. For the input layer, however, zero or positive values of μ produce virtually zero gradients. Instead, μ needs to be near -0.1 to produce non-zero weight changes.[3]

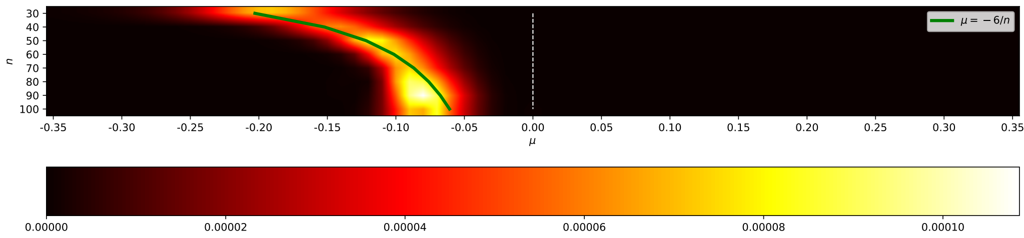
In summary, this empirical corroboration shows that the initial gradients in the input layer and hidden layers are very sensitive to the specific value of μ. While giving good initial gradients to the output layer and hidden layers near it, the standard practice of setting $\mu = 0$ would result in near-zero gradients to the input layer and many hidden layers, as one can see from Figures 6.1.1, 6.1.2, 6.2.1 and 6.2.2. On the contrary, selecting a value of μ that gives a good initial gradient in the input layer (such as $\mu = -0.1$ for this problem) produces non-zero gradients also in the hidden layers following it and the output layer.

This may be a key element in the vanishing gradient problem: the inability of the weights in the first layer to change from their initial random values due to infinitesimally small gradients may effectively prevent the propagation of the

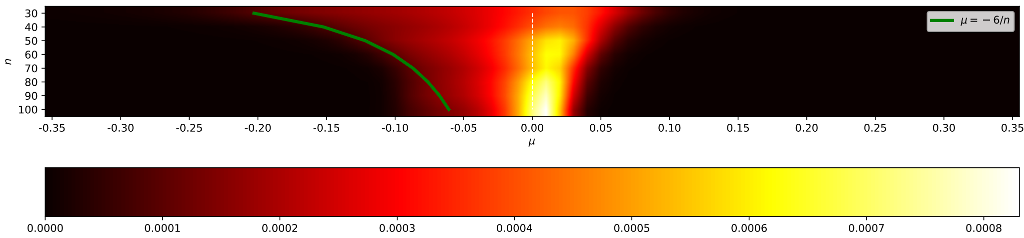
[3] Also, in this case, the optimal μ does not depend on n, but this had to be expected as n is the number of neurons in the hidden layers, while ΔW for the input layer depends on the number of inputs to the problem, which is a constant.



6.2.1 $|\Delta W|$ between layers 0 and 1.



6.2.2 $|\Delta W|$ between layers 8 and 9.



6.2.3 $|\Delta W|$ between layers 13 and 14.

Figure 6.2: Heat maps representing the mean $|\Delta W|$ for the authorship problem for a network with $L = 15$ layers and different numbers of neurons, n, in the hidden layers, for three representative layers.

information contained in the input patterns towards the output, thereby making it impossible for the network to learn. Selecting a value of μ that gives a good initial gradient in the input layer, implies non-zero activation and non-zero errors in that layer. Because errors are back-propagated, also the errors in previous layers must be non-zero and neurons in previous layers must not be saturated (else f' would be zero, zeroing the corresponding δ), implying that the activation of such neurons is non-zero. In other words, non-zero initial gradients in the input layer, imply non-zero gradients in the hidden layers and the output layer.

All of this suggests that the first layer of a network is the one where the vanishing of initial gradients needs to be solved for the learning process to have a chance of starting well. Therefore, μ should be set accordingly. However, whether this is enough to prevent the gradients from vanishing again after the training has started should need to be seen.

We should note that *in all other problems the value of* μ *that gives maximum initial gradients in the input layer is negative*, although it varies from problem to problem (approximately as a function of the number of inputs).

Finally, we should note that, in the tests reported above, the same μ were used for all layers although the number of neurons in the hidden layers (except in the case of $n = 70$) is different from that in the input layer. This is, in fact, a contradiction to the rule in Equation (6.11). So, we should expect that better (higher) initial gradients would be obtained if we follow this rule.

6.4 Refining the Initialisation Strategy

Sections 6.2 and 6.3 focused mostly on expected values. However, this section attempts to see how the variance on the initial weights affects the variance on the net inputs, as a low variance may induce symmetries and competing-conventions problems.

6.4.1 Analysis of Variance of Initial Net-inputs

From Equation (2.9) it is seen that the net input is a sum of products of stochastic variables. At the very first forward-propagation in the network all activations should on average have smaller variance than the initial weights when initialised with the rule $\mu^l = -d/n^l$. So, to a first-order approximation, we can consider net input as a weighted sum of stochastic variables $w_{ij}^{l-1} \sim \mathcal{N}(\mu^l,\ \sigma^2)$, where the a_i^{l-1} are replaced by their expectations $E[a_i^{l-1}]$. That is

$$E[net_j^l] \cong w_{0j}^{l-1} + \sum_i E[a_i^{l-1}] w_{ij}^{l-1}. \tag{6.13}$$

The proposed initialisation strategy suggests to use different μ^l for different layers so that the expected derivative of the activation function of all (hidden) neurons in a network, $E[a_i^l(1-a_i^l)]$, take one particular value, $\frac{1}{d}$. So, initially all coefficients, $E[a_i^{l-1}]$, in the weighted sum will be identical. So, we can approximate

$$net_j^l \cong w_{0j}^{l-1} + E[a^{l-1}] \sum_i w_{ij}^{l-1}. \tag{6.14}$$

Under this approximation

$$net^l \sim \mathcal{N}\left(\mu^{l-1}(1+n^{l-1}E[a^{l-1}]),\, \sigma^2(1+n^{l-1}E[a^{l-1}])\right). \tag{6.15}$$

6.4.2 New Initialisation Method

As we have seen at the end of Section 6.2.2, for the logistic activation function simple numerical approximations allows us to find a value of $E[a^l]$ that satisfies Equation (6.12) for any reasonable value of d. We specifically tested this for the pairs $(n^l, E[a^l]) \in \{(5, 0.02), (10, 0.12), (100, 0.21)\}$.

For simplicity, let us assume that all the layers in a network have an identical number of neurons, n. Substituting $\mu^l = -8/n$, for all $(n, E[a^l]) \in \{(5, 0.02), (10, 0.12), (100, 0.21)\}$ into Equation (6.15) and simplifying, we obtain

$$net^l \sim \mathcal{N}\left(-1.76, 0.22n\sigma^2\right), \tag{6.16}$$

for $n = 5$, $n = 10$ and $n = 100$ (although the relationship is general). So, the variance of the net input grows linearly with the number of neurons in the previous layer.

To see what effect this may have, let us substitute the actual value of σ used in this chapter ($\sigma = 0.1$) in Equation (6.16). We then see that $\text{StdDev}[net] = 0.047\sqrt{n}$. This indicates that narrow networks may suffer from insufficient initial variation in net inputs that translates into an even smaller variation if initial activations, potentially leading to symmetries/competing-conventions problems. For instance, if we compute the activations corresponding to the net inputs $E[net] \pm 1.96 \times \text{StdDev}[net]$, we see that 95% of initial activations fall within a very narrow interval, $[0.12, 0.17]$, for $n = 5$, but a nearly five times wider interval, $[0.06, 0.30]$, for $n = 100$.

Based on these observations, we reduce the symmetries in narrow (deep) networks, thereby increasing the likelihood of them being trainable, by slightly modifying Equation (6.11), obtaining the following *New Initialization Method (NIM)*:

$$w^l \sim \mathcal{N}\left(\max\left(\mu_{min}, -\frac{d}{n^l}\right),\ \sigma^2\right) \tag{6.17}$$

where μ_{min}, d and σ are constants that in this chapter, after having done some preliminary experiments, we set to $\mu_{min} = -1$, $d = 8$ and, once again, $\sigma = 0.1$. With such choices of μ_{min} and d, we effectively prevent the μ^l from being less than -1 in networks with 7 or fewer neurons in the hidden layers.[4]

6.5 Training Deep Networks after the Proposed New Initialisation

In both the theory developed in Section 6.2 and its corroboration in Sections 6.3 and 6.4, we focused on amplifying the initial gradient. The question we now need to address is to what degree this solves the vanishing gradient problem when we continue training the network with the SBP algorithm for many interactions. In this section, we address this question.

[4] An alternative method to counteract the reduced variance in the initial activations of narrow networks would have been to make σ a function of n^l. This is something that will be explored in future research.

6.5.1 Problems and Network Structures

We tested the NIM with a few independent runs using the test problems on deeper (up to 25 hidden layers) and/or wider (up to 150 hidden neurons) networks. The results showed that the NIM allow the SBP to train the networks successfully. Tests with such networks and with many independent runs require a high computational cost. Therefore, to make this comparison cheaper, we used the same network structures like those used in the previous chapter. They are shown in Table 5.3. Each problem has two network structures (one with depth $L = 10$ and 10 neurons in each hidden layer and one with depth $L = 15$ and 100 neuron in each hidden layer). All networks were trained using a learning rate of $\eta = 0.25$ which is also the same as that used for the networks in the previous chapter.

6.5.2 Initialisation Methods Compared

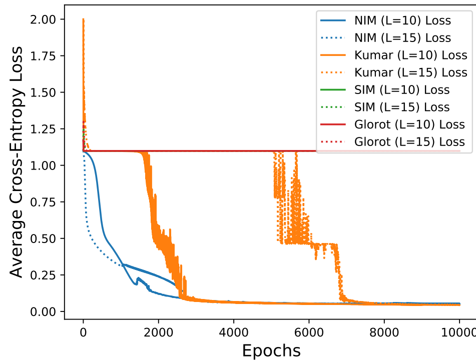
The proposed method, NIM, was compared with three different initialisation methods which were used to compare against the EIM (the initialisation method developed in the previous chapter). As listed in the previous chapter, the distributions of the initial weights of those methods are shown below. The distribution of initial weights of the NIM is also listed below.

- *SIM*: $W^l \sim \mathcal{N}(0,\ 0.01)$,
- *Glorot*: $W^l \sim \mathcal{N}(0, 2/(n^l + n^{l+1}))$,
- *Kumar*: $W^l \sim \mathcal{N}(0,\ 12.96/n^l)$,
- *NIM*: $W^l \sim \mathcal{N}(\max(-1, -\frac{d}{n^l}),\ 0.01)$

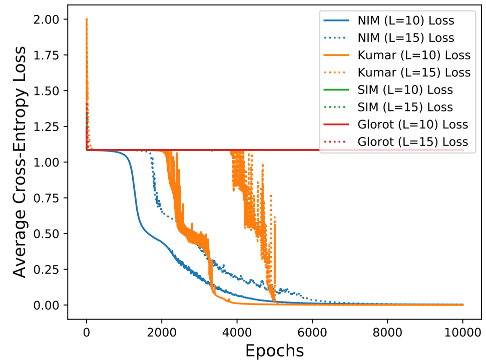
n^l being number of neurons (including bias) in layer l and $d = 8$ a constant.

Figure 6.3 shows plots of the average cross-entropy loss and (test- and training-set) accuracy when training networks initialised with the four initialisation methods mentioned above. The network structures and problems are those listed in Table 5.3. All networks were trained by *SBP* using learning rate of $\eta = 0.25$. Results are medians of 30 independent runs.

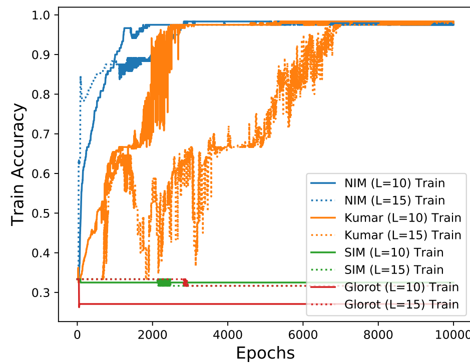
As seen from Figure 6.3, SIM- and Glorot-initialised networks could solve none of the problems applied to deep MLPs with the logistic activation function. Only the proposed simple initialisation method, NIM, and Kumar initialisation methods allow standard back-propagation to train those networks. In the case of NIM initialised networks, SBP trains the networks reliably, quickly and effectively. Furthermore, it is apparent from the figures that NIM converge faster than the Kumar initialisation method, which eventually trains the networks. The only exception is mux for $L = 15$ where with NIM initialisation the network needed more than the 10,000 epochs



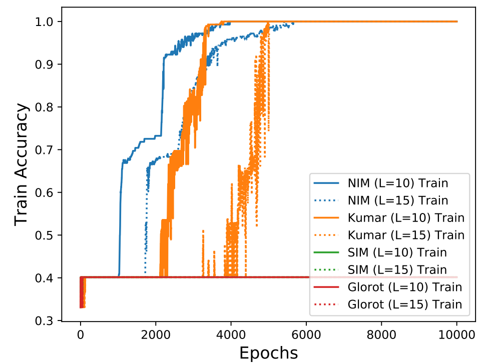
6.3.1 Iris Loss



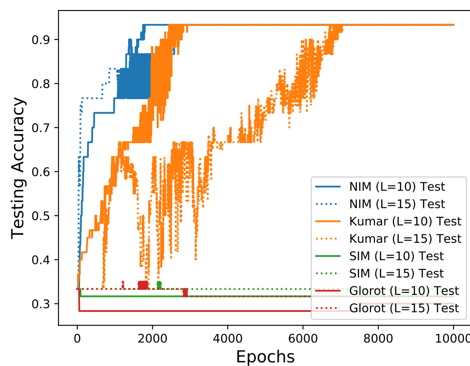
6.3.2 Wine Loss



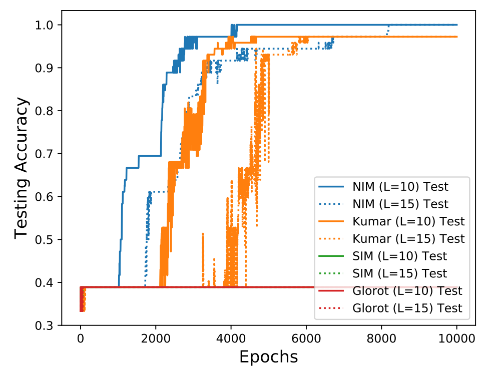
6.3.3 Iris Train Accuracy



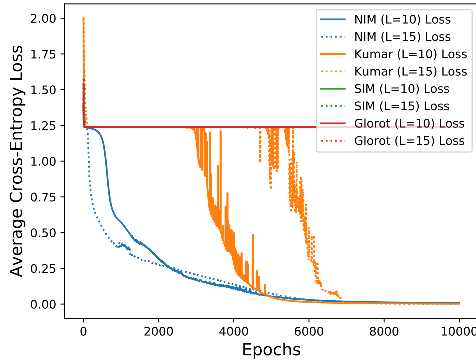
6.3.4 Wine Train Accuracy



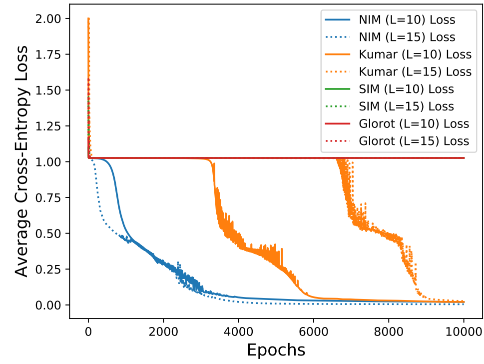
6.3.5 Iris Testing Accuracy



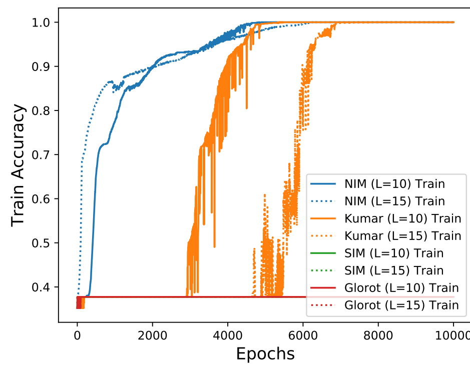
6.3.6 Wine Testing Accuracy



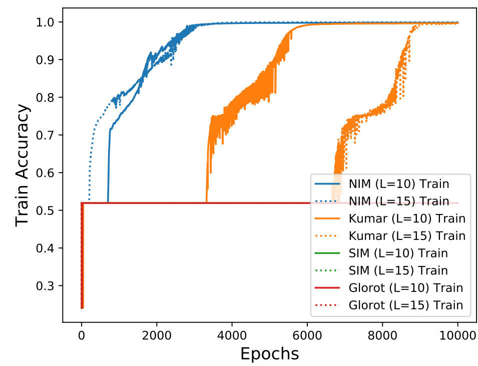
6.3.7 Authorship Loss



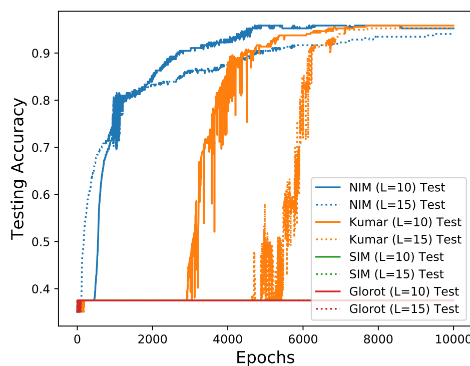
6.3.8 DNA Loss



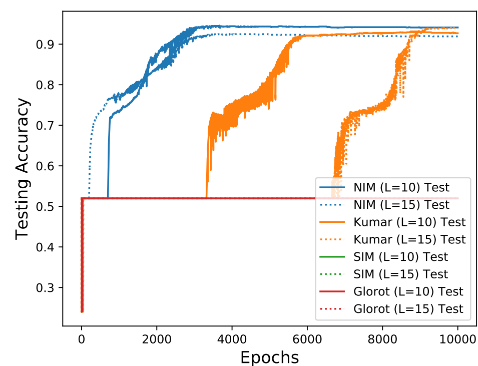
6.3.9 Authorship Train Accuracy



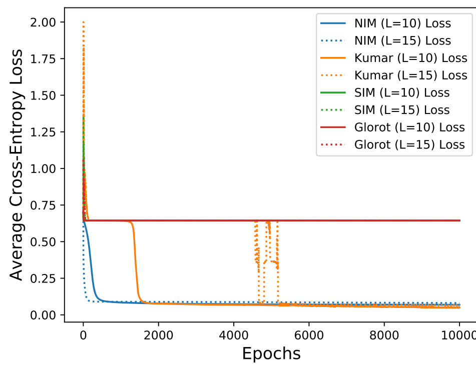
6.3.10 DNA Train Accuracy



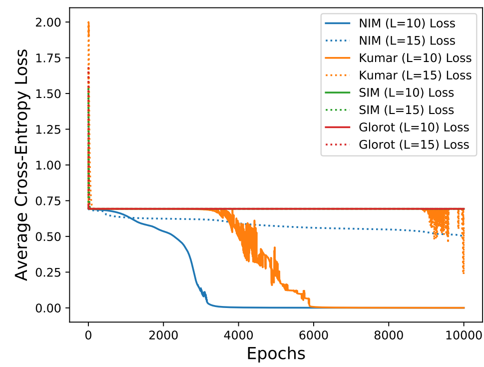
6.3.11 Authorship Testing Accuracy



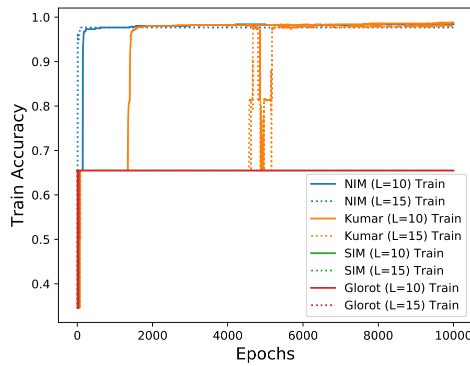
6.3.12 DNA Testing Accuracy



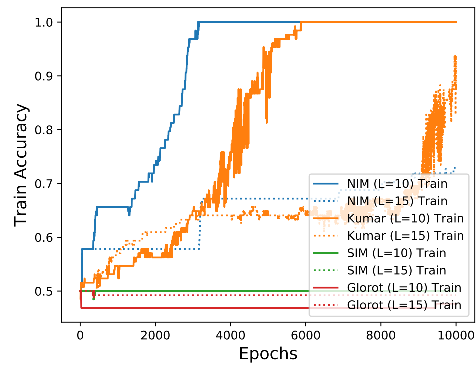
6.3.13 BCW Loss



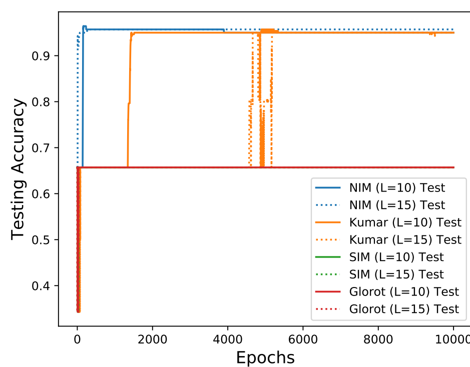
6.3.14 The Mux Loss



6.3.15 BCW Train Accuracy



6.3.16 The Mux Train Accuracy



6.3.17 BCW Testing Accuracy

Figure 6.3: Results for our six test problems when the *SIM*, *Glorot*, *Kumar* and *NIM* methods are used to initialise networks of depth $L = 10$ and depth $L = 15$

provided to reach the same accuracy as with Kumar's initialisation. We can conclude from this figure that focusing on the initial gradients is an effective approach to understand and deal with the vanishing gradient problem.

To start understanding the reasons for the success of NIM and Kumar and the failure of SIM and Glorot, we will look at the sum of the magnitudes of the gradients measured in each layer and how this changed over time. Figure 6.4 reports the results for the iris problem and a network of depth $L = 10$ for the four initialisation methods. The layers in the figure are numbered in an increasing from the input layer to the output layer. The gradient magnitudes of the last layer are not plotted as the networks use a soft-max output layer.

For networks initialised with *SIM* and *Glorot*, we used a logarithmic time scale because all of the dynamics is concentrated in the first 70 or so training epochs. As it was predicted with the theoretical model presented above, setting $\mu = 0$ produces reasonably big gradients near the output layers which progressively become smaller and smaller as we move towards the input layer. This happens for SIM, Glarot and Kumar. In all three cases, within the first tens of training epochs, such gradients become progressively even smaller, thereby apparently freezing the networks into a very-low-gradient state from which it is very hard to recover.

Surprisingly, in the case of Kumar, after a few thousand iterations the network manages to escape and gets into a phase where the first hidden layer has fairly large gradients (with the following layers having progressively smaller and smaller gradients) which is required to train the network successfully. This may happen because of the bigger variance used in Kumar when compared to SIM and Glarot, which makes it possible to have subsets of the network with negative weights large enough that gradients can locally back-propagated without too much attenuation. These trickles progressively become wider and wider until a network can learn. Some networks get to that state quickly, but most (as shown by the plot in Figure 6.3 which is flat until approximately 2,000 epochs for L=10) take 2+ thousand of epochs for L=10.

In the case of NIM, the situation is quite different. Firstly, in the first iterations, the gradients are ordered with the biggest magnitudes being in the input layer. Also, from such an initial state, the gradients grow progressively bigger rather than smaller like in (SIM, Glorot and Kumar). The dynamics is not monotonic and there are bursts where gradients become even bigger, but, in all cases, the biggest gradients are in the input layer. Therefore, for NIM, both the loss function and the accuracy grow fairly smoothly and rapidly (NIM allows the accuracy to reach 80% within tens of epochs whereas Kumar does not get there until several thousands of epochs, and SIM and Glarot never get there).

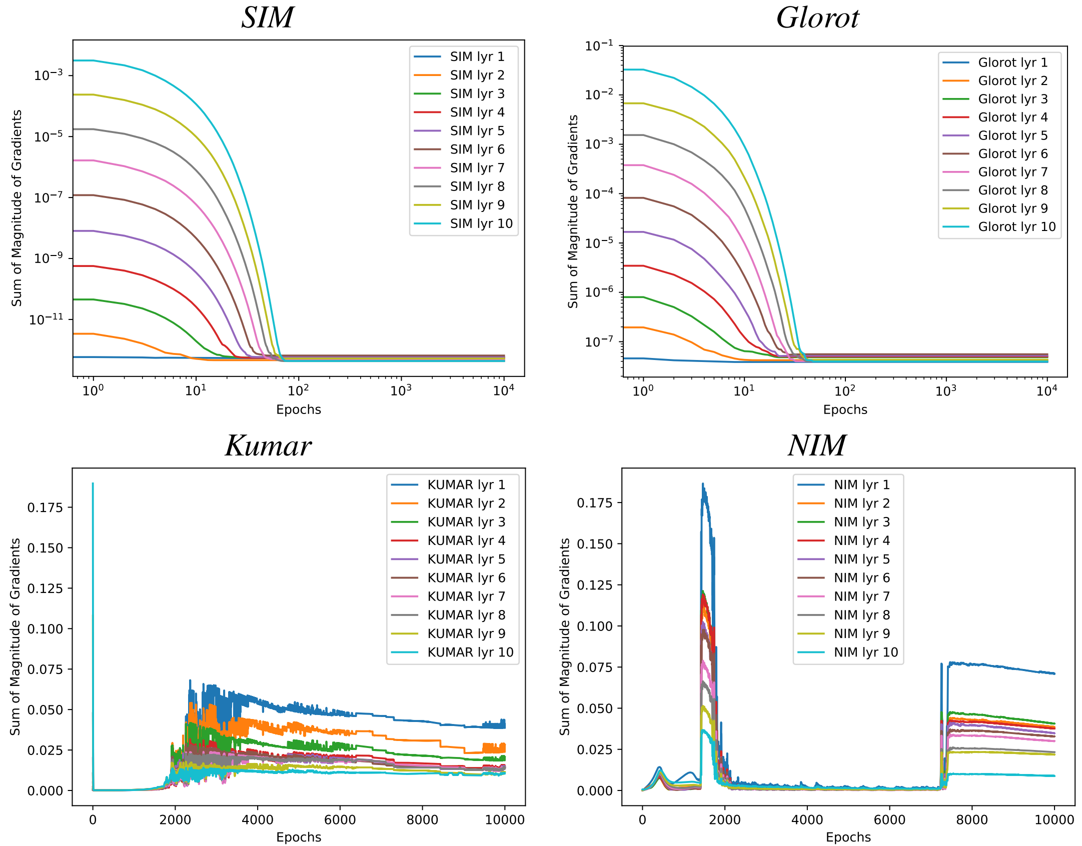


Figure 6.4: Sum of gradient magnitudes in each layer when for a network of depth $L = 10$ for four different initialisation methods in the iris problem.

6.6 Numerical Optimisation of μ

NIM has a clear advantage in terms of simplicity of use. However, it sets the value of μ only on the basis of the number of neurons in a layer, which may be sub-optimal as we have seen in Section 6.3 (e.g., see Figure 6.1) that different values of μ (including positive ones) produce maximal initial gradients in different layers.

Here, we want to remove the constraints imposed by NIM and numerically optimise the value of μ for each layer, as already suggested in Section 6.2. This might provide even bigger gradients than with NIM, as the theory in Section 6.2 is approximate.

6.6.1 Hill-climbing the Initial Gradient Surface

As a numerical optimisation algorithm, a stochastic hill-climber was used to optimise the set of μ values. To provide a good starting point, the hill-climber was initialised with the μ values proposed by NIM. The algorithm then searches for an optimal or near-optimal set of μ values by maximising mean initial $|\Delta W|$ in the first hidden layer. The search process lasts for 500 generations with 5 restarts. Figure 6.5 reports the results of these further tests for the six problems from Section 6.5.1 and a

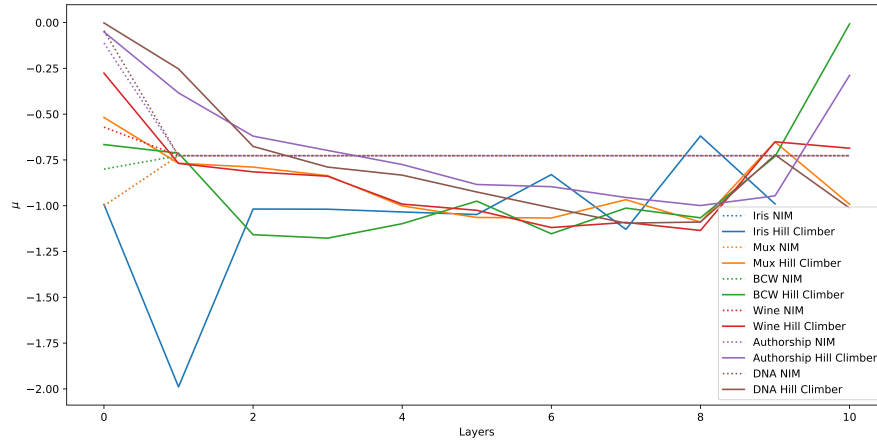


Figure 6.5: Optimal μ values prescribed by NIM in Equation (6.11) and those found by a hill-climber (optimising the mean $|\Delta W|$ in the first hidden layer) the six problems in Table 5.3 for a network with $L = 10$ layers and 10 neurons in each hidden layers.

network with $L = 10$ layers and 10 neurons in each hidden layer. The μ values for the hill-climber are averages over 30 independent runs of the algorithm.

Naturally, with the exception of layer 0 (the input layer), all μ values obtained from NIM in Equation (6.11) (where, for uniformity, we chose $d = 8$ for all problems) are identical across problems and layers $l > 0$. Interestingly, for most problems, the values of μ identified by the hill-climber are similar in magnitude and identical in sign to those prescribed by NIM. There are bigger discrepancies between the μ's for different problems and initialisation strategies for the first 2–3 hidden layers and for the last couple.

Figure 6.6 reports the same results but for a network with $L = 15$ layers and 100 neurons in each hidden layers. Here the μ values selected by the hill-climber are also all negative. Additionally, except for the DNA and BCW problems where all runs were unable to improve over the hill-climber initial position (i.e., NIM initialisation), they tend to vary a bit more from layer to layer and can be up to 2–5 times bigger in magnitude than those set by NIM.

A question that arises then is: to what degree do the discrepancies between μ values prescribed by the NIM and hill-climber translate into differences in gradient magnitude in the first hidden layer? An answer to this question comes from Figure 6.7 and 6.8, which show the mean absolute gradient in the first layer of weights obtained when using the optimal μ values prescribed by NIM and the hill-climber reported in Figures 6.5 and 6.6. As one can see from the figures, while the $|\Delta W|$'s for the two methods are of the same order of magnitude for the smaller network, the differences are much more marked (up to 3 orders of magnitude) for the larger network.

This suggests that, while for relatively small networks, NIM is a good default,

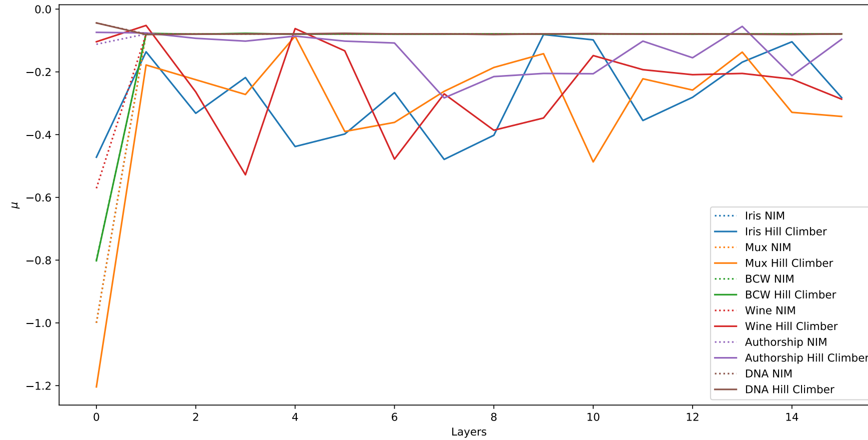


Figure 6.6: Same as in Figure 6.5 but for a network with $L = 15$ layers and 100 neurons in each hidden layers.

for larger networks and problems the small computational effort required to optimise μ with the hill-climber might pay off massively in terms of initial gradients.

6.6.2 NIM-initialised vs Hill-climber-initialised Gradient Descent

The gradient descents (controlled by the SBP algorithm) obtained for networks initialised with the μ values identified by NIM and the hill climber were pitted.

Figure 6.9 shows the average dynamics produced for the six problems in Table 5.3 for a network with $L = 10$ layers and 10 neurons in each hidden layer. As we have seen in Section 6.6.1, in this case, the μ values and, correspondingly, the initial gradients obtained with the two initialisation algorithms are not too different. It is then not surprising to see in the figure that the gradient descent produced by SBP after initialisation is very similar to that of the hill-climber having a slight edge on NIM only in the mux and wine problems.

Figure 6.10 shows the average dynamics produced for the same six problems but this time for a network with $L = 15$ layers and 100 neurons in each hidden layers. As we have seen in Section 6.6.1, in this case, the μ values obtained with the two initialisation algorithms are quite different except for the DNA and BCW problems where all hill-climber runs converged onto the NIM solution. For these two exceptions, rather obviously, the gradient descent produced by SBP with $\eta = 0.25$ after initialisation with the two algorithms is identical, as shown in the figure. However, after a sharp start, the descent (with $\eta = 0.25$) produced by hill-climber-based initialisation plateaued prematurely in all other problems and for most runs of the 30 independent runs done for each problem.

Effectively, thanks to its success at increasing the initial gradient, the hill-climber

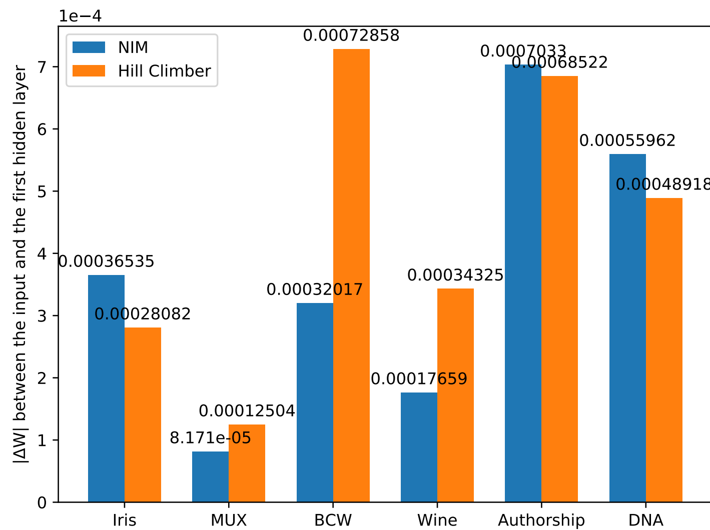


Figure 6.7: Mean absolute gradient in the first layer of weights obtained when using the optimal μ values prescribed by NIM in Equation (6.11) and those found by a hill-climber for the six problems in Table 5.3 for a network with $L = 10$ layers and 10 neurons in each hidden layers.

has made the learning process with this relatively large learning rate unstable. As Figure 6.10 shows, this problem can be solved to a significant degree just by reducing the learning rate to 0.05.

6.7 Summary

This chapter investigated to what extent the vanishing-gradient problem depends on the choice of the *mean* (as opposed to the variance) of the initial weight distribution in networks using the logistic activation function.

Focusing on the initial gradients, a simple theoretical model was developed. This model predicted that to obtain non-vanishing *initial* gradients in all layers of a network, the magnitude of the mean of the initial distribution of weights had to be inversely proportional to the number of neurons in a layer. Empirical corroboration of this prediction showed that there is an asymmetry by which gradients do not vanish only if the mean is negative. In the light of this investigation, we developed an initialisation strategy which was called NIM.

The experimental results showed that the back-propagation algorithm was both successful and efficient at training deep networks with 10 and 15 hidden layers (and 10 and 100 neurons per hidden layer, respectively) on a standard set of benchmark classification problems, including real-world, large-scale problems when the networks were initialised by the proposed method.

It was also tested whether the initialisation produced by NIM could be improved by hill climber. It was found that while this could be achieved in some problems

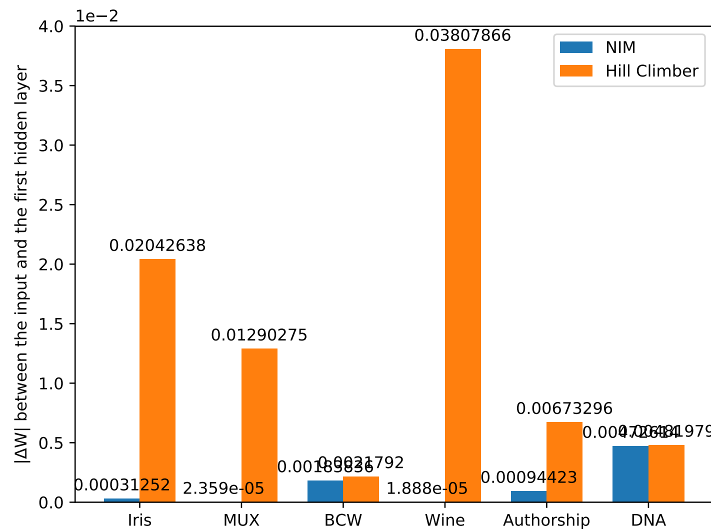
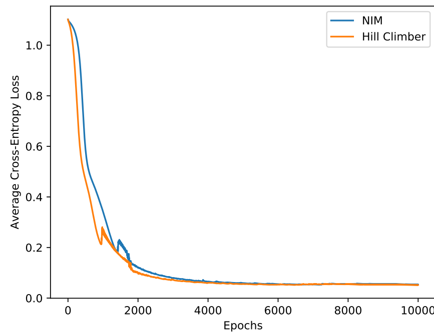


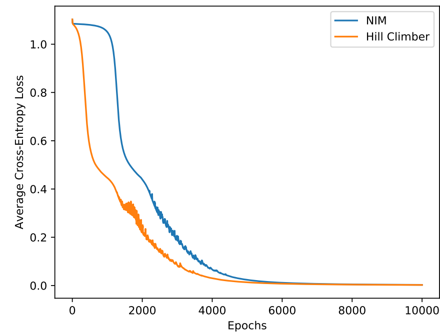
Figure 6.8: Same as in Figure 6.7 but for a network with $L = 15$ layers and 100 neurons in each hidden layers.

for the smaller architecture (10 layers, 10 neurons each), this causes too big initial gradients and failed convergence in most problems for the larger architecture (15 layers, 100 neurons each).

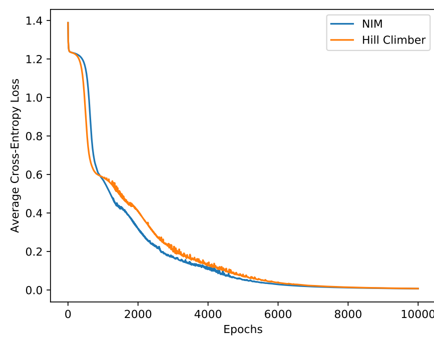
The vanishing gradient problem is a long-standing obstacle to the training of deep neural networks with neurons using the logistic activation function. Although other activation functions developed over the years overcome this problem, they have other drawbacks, as reviewed in Section 2.3.3.2. This work showed that a simple initialisation method could solve the vanishing gradient problem in deep networks that use the logistic activation function.



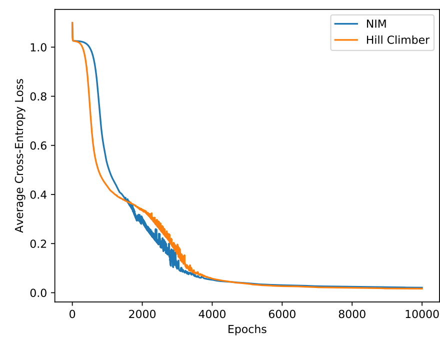
6.9.1 Gradient descent for Iris



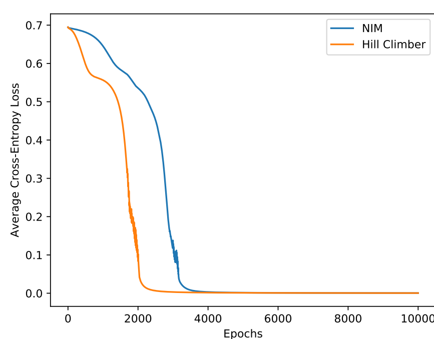
6.9.2 Gradient descent for Wine



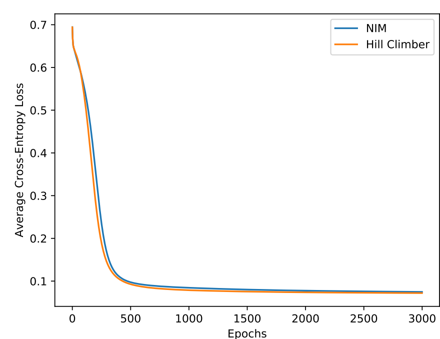
6.9.3 Gradient descent for Authorship



6.9.4 Gradient descent for DNA

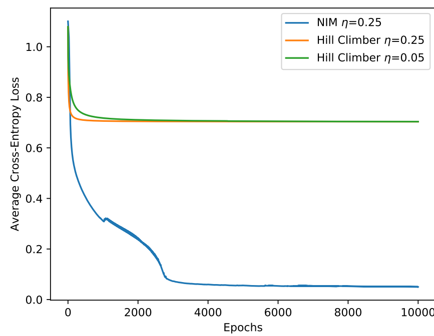


6.9.5 Gradient descent for Mux

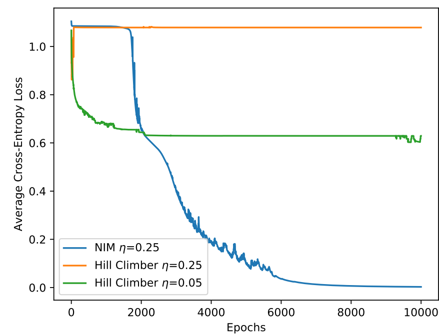


6.9.6 Gradient descent for BCW

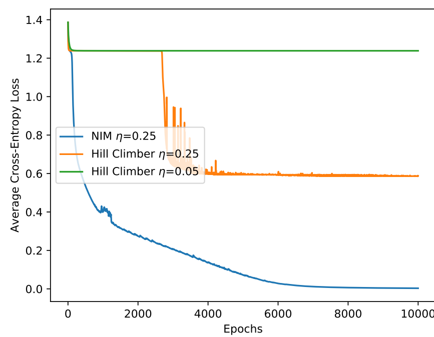
Figure 6.9: Cross-entropy loss as a function of the number of training epochs for NIM-initialised an hill-climber-initialised networks trained with the SBP for the six problems in Table 5.3 for a network with $L = 10$ layers and 10 neurons in each hidden layers. The learning rate was $\eta = 0.25$. Results are averages of 30 independent runs.



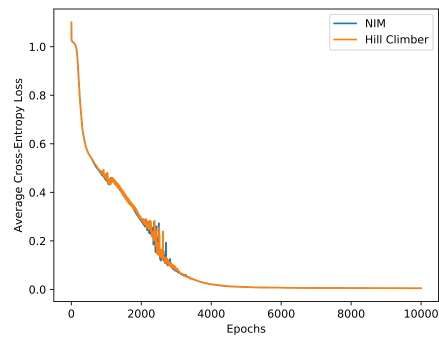
6.10.1 Gradient descent for Iris



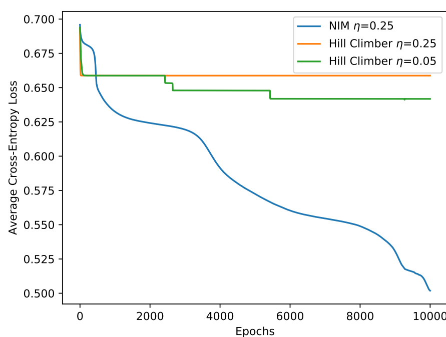
6.10.2 Gradient descent for Wine



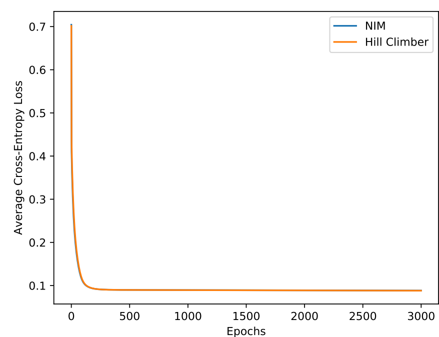
6.10.3 Gradient descent for Authorship



6.10.4 Gradient descent for DNA



6.10.5 Gradient descent for Mux



6.10.6 Gradient descent for BCW

Figure 6.10: Cross-entropy loss as a function of the number of training epochs for NIM-initialised an hill-climber-initialised networks trained with the SBP for the six problems in Table 5.3 for a network with $L = 15$ layers and 100 neurons in each hidden layers. The learning rate was $\eta = 0.25$ for NIM and both $\eta = 0.25$ and $\eta = 0.05$ for hill-climber initialisation. Results are averages of 30 independent runs.

Chapter 7

Conclusions and Future Work

Deep and shallow neural networks with many different structures have been widely used for machine learning problems. Although neural networks have been improved over time, there are still some problems to be overcome.

This thesis addressed two main problems of neural networks which are related to learning and initialisation algorithms; and tried to find answers of the following questions:

1. Would it be possible to develop fair comparison methods to compare leaning algorithms used in neural networks?
2. Can GP develop faster learning algorithms than SBP algorithm while being general?
3. Can GP develop new initialisation algorithms to make deep neural networks, that use the logistic activation function, trainable?
4. Would it be possible to find ways of initialising the weights so that the algorithms converge faster or more reliably?

7.1 Contributions of the Research

7.1.1 Comparison Methods for Learning Rules

Chapter 3 presented two general rigorous methodologies that were developed for fairly comparing the performance of learning rules.

Both methods compute the total number of training epochs required to obtain at least a successful run in a group of runs. The first method assumes that all runs are executed in parallel and compute how many epochs are required to obtain at least one satisfactorily trained network out of a certain number of parallel training runs using relatively simple adaptations of Koza's computational effort for GP. The second method considers the situation where multiple attempts to train a network

(with different initialisations) are performed sequentially and provides the expected number of epochs required to obtain a successfully trained network.

Computational effort depends on the training (hyper) parameters. The proposed methods consider the learning rate and the number of epochs. The maximum number of epochs is set to be large enough for each problem and the same for each pair of comparisons so that both rules have a reasonable and the same chance to solve problems. This process is repeated for both rules with different learning rates to provide optimum or near-optimal learning rate for each learning rule. The minimal computational effort is the minimum computational effort recorded by varying the learning rate and the number of epochs in all possible ways within a large range of values. These processes make the developed comparison methods fair.

7.1.2 Learning Rules Evolved for MLPs

Chapter 3 also evolved a back-propagation learning rule by applying GP to neural networks. In order to prevent programs (learning rules) in the populations from having wide oscillations on the training errors, penalty and reward functions were added to the fitness function based on the error directions during the training. GP found a form of *SBP* algorithm but with a variable effective learning rate. Chapter 3 reported that the rule was fast, stable, and general; and provided a qualitative interpretation that explains why the evolving rule performs well; and also provided strong evidence of its superiority with respect to the SBP.

Chapter 4 presented similar work that applies GP to neural networks to evolve learning rules. In Chapter 3, the search space of GP was restricted by a penalty function that considers oscillations on the error during the training. This approach encouraged learning rules in the populations to perform monotonic error descent that may cause the network to get trapped in a local minimum. Therefore Chapter 4, by changing the GP-MLP configuration, aimed to evolve learning rules that are fast, general; and that can escape local minimum. Here, the GP parameters, the training problems and network structures were changed. The most important changes were done to the fitness functions. It was designed to allow learning rules to oscillate on the training error surface. Here, in contrast to the work in Chapter 3, the fitness function did not penalise and reward the rules as it did not care about the oscillations on the training error. It was simply the sum of the lowest errors or sum of the highest accuracies of a pre-defined percentage of the number of training epochs.

Two different learning rules were evolved. The fitness function considered the error for the first one like in the previous chapter. It considered the training accuracy for the second one since the comparison methods compute the computational efforts using the training accuracy as the success criterion. The rules were compared with the SBP using the comparison methods developed in Chapter 3. The results showed

that both rules evolved outperformed the SBP but they had worse performance than the rule evolved in Chapter 3.

Each learning rule found by GP in Chapters 3 and 4 was a form of SBP with a variable learning rate that was formed from pre- and post-synaptic activations of the corresponding connection. In other words, in these experiments, GP showed us that the SBP with a variable learning rate can be faster and more general than the SBP with a constant learning rate.

7.1.3 Initialisation Rules for Deep MLPs

As described in Chapter 2, relatively deep networks suffer from the vanishing gradient problem especially when a sigmoidal activation function is used in the neurons of the networks. There are some efficient methods that solve this problem. However, they mostly do not work with the logistic activation function. This thesis also developed initialisation methods that allow deep MLPs, which use the logistic activation function, to be trained with the SBP.

In relation to this, Chapter 5 developed a method that initialised the networks by pre-training. GP was applied to deep MLPs to evolve a weight-update rule which would only be used during the pre-training process of the networks. In this model, after having completed the pre-training phase of the networks, the SBP algorithm is used for completing the training of the networks.

In order to evaluate the efficiency of the proposed algorithm, EIM, it was compared with three different initialisation methods — SIM, Glorot and Kumar — using six classification problems (four multi-class and two binary). For each problem, two networks structures (10 hidden layers with 10 neurons in each hidden layer and 15 hidden layers with 100 neurons in each hidden layer) were used. In order to make a fair comparison, the number of pre-training epochs used for the proposed model was included in the total number of training epochs. The experimental results showed that the SIM and Kumar initialisation could not be successful on the test networks and also showed that EIM and Kumar were successful and eventually reached almost the same level of accuracy/error. Because the number of pre-training epochs was relatively large for the small (L = 10) networks, Kumar initialisation provided usually faster convergence than the EIM for these networks. However, the opposite was always true for the large (L = 15) networks as they were pre-trained with a relatively small number of epochs.

Another initialisation method, NIM, was developed in Chapter 6. Because at the end of the pre-training process the EIM made the mean of the weight distributions in the networks negative, this chapter investigated to what extend the vanishing gradient problem depends on the choice of the mean of the distribution. In this investigation, the theory predicted that initial gradients in the layers are maximised

when the mean of the initial weights is negative and inversely proportional to the number of neurons in each layer. The knowledge gained on initial gradients from theory and the empirical tests suggested that a rule that sets the mean initial weights to $-\min(1, 8/\text{number of neurons in layer})$ would maximise such gradients. The developed simple initialisation rule does not require a pre-training process.

In that chapter, for comparing NIM, the same initialisation methods, the same problems and the same network structures those used for comparison of the EIM in Chapter 5 were used. The experimental results showed that SBP initialised by the NIM not only successfully trained the networks but also learned the corresponding problem efficiently and quickly.

As mentioned above, EIM initialises the networks by pre-training, while NIM initialised the networks using the initial weight distribution that is determined based on the number of neurons in the layers. Both EIM and NIM allow SBP to train deep networks that use logistic activation function without suffering from the vanishing gradient problem. However, for EIM, there is no simple rule to determine the optimal number of epoch for pre-training. Determination of this number is based on trial and error. Therefore, the NIM could be more useful than EIM since it can easily initialise the networks without waiting for pre-training.

7.2 Limitations in the Research

7.2.1 Limitations in the Works that Evolved Learning Rules

As mentioned above, in this thesis, GP evolved learning rules in Chapters 3 and 4. Although different fitness functions and slightly different primitive sets were used, the methodologies used were the same. Therefore, the limitations were almost the same.

This thesis only reported on NLR in Chapter 3 and NLR(1) and NLR(2) in Chapter 4. Of course, other rules were evolved in different runs that satisfied the criteria of speed and stability but were not always completely general (e.g., some worked well on all Boolean problems, but did not work well with continuous problems, and *vice versa*).

Also, here we took as a reference the SBP. Of course, there are many improvements that have been produced over the years to the original SBP, such as the momentum term, RMSProp, Adam optimiser, etc. The reason why the rules evolved were not tested with the momentum term is that the momentum term has one more parameter to be optimised and so the computational time for this comparison would be quite long. Since the RMSProp and Adam optimiser optimise the learning rate during the training, the evolved rules can also use such models. However, in these models, the learning rates are initialised with a relatively small number while the

comparison methodologies developed in this thesis chose the optimal learning rates from the results of runs done with a number of learning rates. Because it is quite hard to optimise the initial learning rates for the models that use RMSProp and Adam optimiser with the proposed comparison methods, in this thesis, the evolved rules were not tested using such optimiser.

Finally, the theoretical analyses were limited to the optimisation of learning rates and the number of epochs. While the extension to other hyper-parameters is possible, it is not unlikely that the extension would be of limited practical applicability due to a combinatorial explosion in the number of test conditions.

7.2.2 Limitations in the Works that Developed Initialisation Rules

Chapter 5 considered only the logistic activation function to evolve an initialisation rule for deep MLPs. Chapter 6 developed theoretical models in such a way that they could accommodate other activation functions. However, all empirical validation of ideas and study of the learning process in that chapter were also conducted only for the logistic activation function.

Both initialisation methods were tested with a limited number of benchmark problems and network structures.

Also, these studies did not consider more sophisticated forms of training, including the used of a momentum term or Rprop, RMSProp and Adam optimiser.

7.3 Future Work

This section presents some possible research that can be done in the future. While each of the limitations stated above could be considered as future work, changing the configurations in the GP-MLP system used for the experiments could also provide some interesting results.

7.3.1 Changing the Configurations Used in the Experiment

In relation to the configurations in the GP-MLP, the operator and terminal sets can be expanded. Of course, this makes the algorithm more expensive. However, this provides a larger search space; thereby, the system is more likely to evolve more efficient learning and initialisation rules.

The fitness function has a crucial role in EAs. In this thesis, the main parameter of the fitness functions was either the network error or the training accuracy. The computational efforts formulae developed in Chapter 3 can be another good choice for the fitness function.

The number of problems and runs used in evolution are also important to obtain more general rules. While a small number of problems may cause learning rules

to overfit to the problems, the large numbers may make the algorithm expensive. If at each generation, the algorithm chooses a small number of problems from a problem list predefined and forms the network structures randomly, this may allow the algorithm to produce general rules while avoiding a costly run.

7.3.2 Considering the Limitations of this Thesis

As mentioned above, all experiments in this thesis were done considering the logistic activation function. In relation to the work in Chapter 6 which developed a theoretical initialisation method, NIM, it may be a good idea to use other activation functions (such as ReLU, soft-sign and tanh) to see whether the proposed method is successful and efficient with the SBP when connection weights in deep neural networks are optimised by maximising the initial gradients.

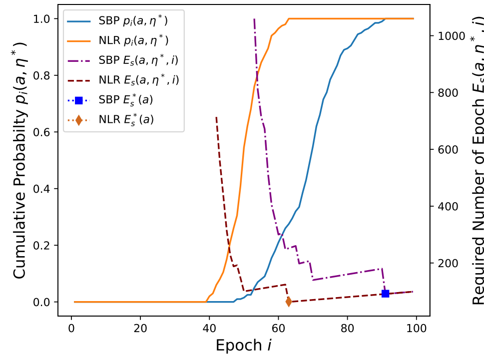
Furthermore, it is clear from the theory in Chapter 6, that it would be possible to refine our initialisation rule so as to consider $E[inputs]$. Also, while focusing on μ allowed us to make significant progress, as shown in Section 6.4.1, the standard deviation σ of the initial weight distribution does matter too. Therefore, developing a similar theory for σ would be possible and could lead to speeding up learning. This can be investigated in the future.

Finally, the vanishing gradient problem is particularly insidious in applications involving recurrent neural networks. Training of such networks could benefit from the proposed method. This can be a good research subject.

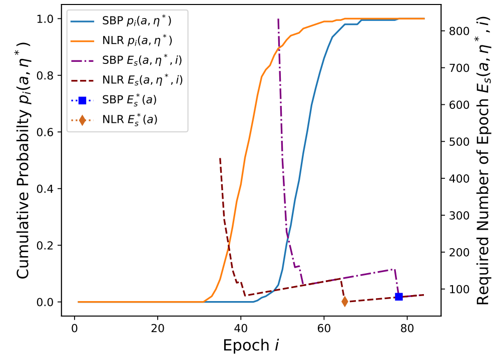
Appendix A

Additional Results from Chapter 3

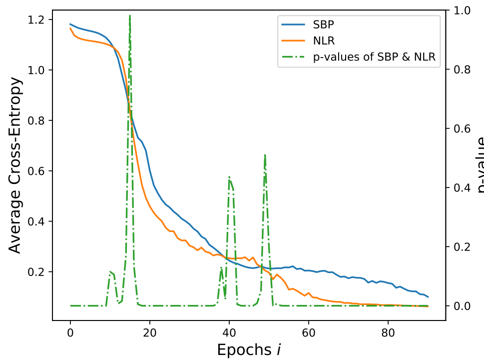
In this appendix, we report additional results that show the comparison figures of NLR and SBP for test problems when sequential evaluation is used.



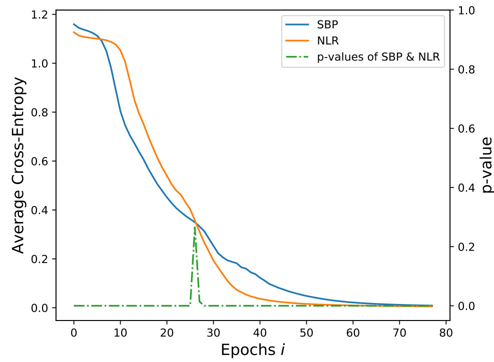
A.1.1 Iris Computational Effort



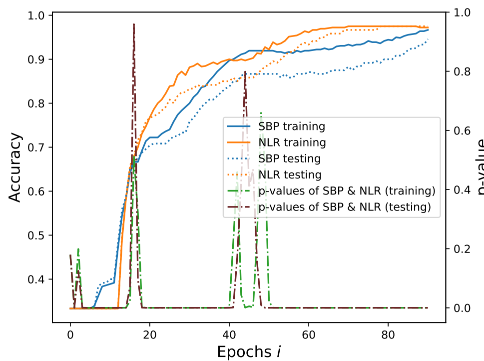
A.1.2 Wine Computational Effort



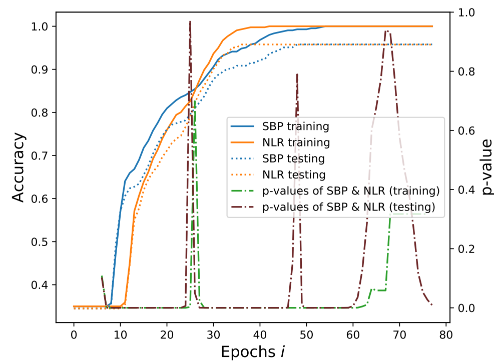
A.1.3 Iris Error



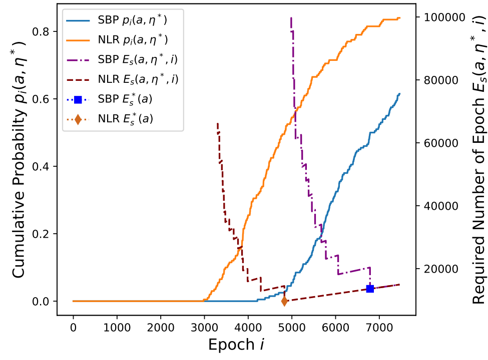
A.1.4 Wine Error



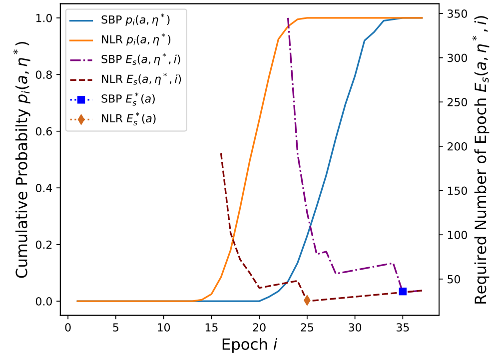
A.1.5 Iris Accuracy



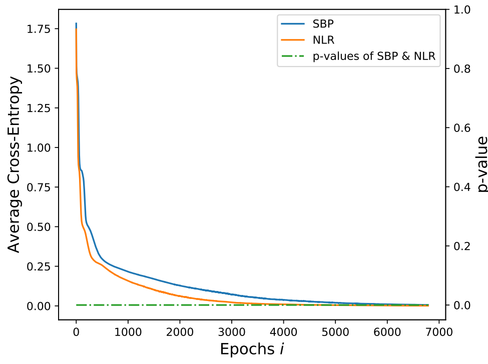
A.1.6 Wine Accuracy



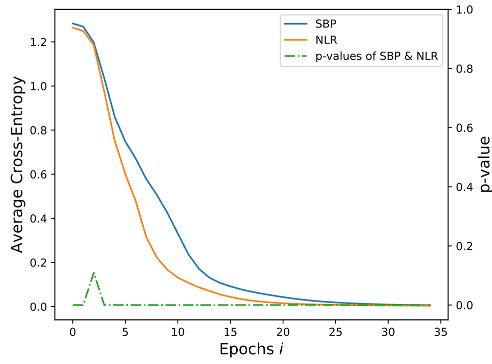
A.1.7 E-coli Computational Effort



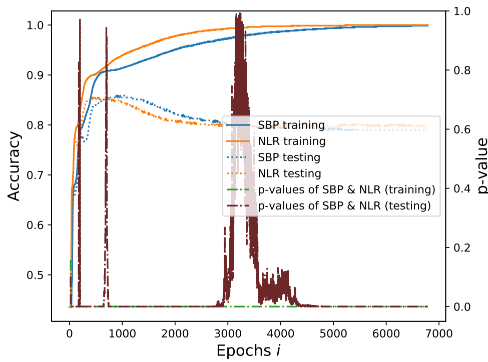
A.1.8 Authorship Computational Effort



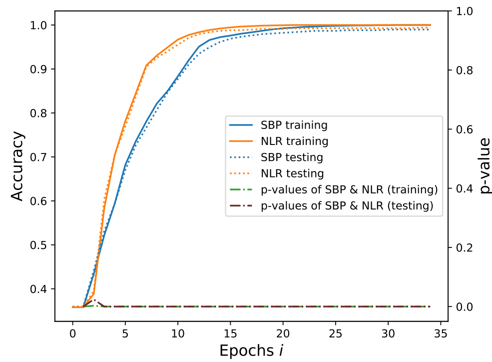
A.1.9 E-coli Error



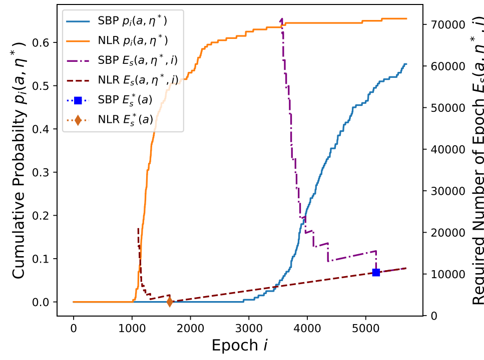
A.1.10 Authorship Error



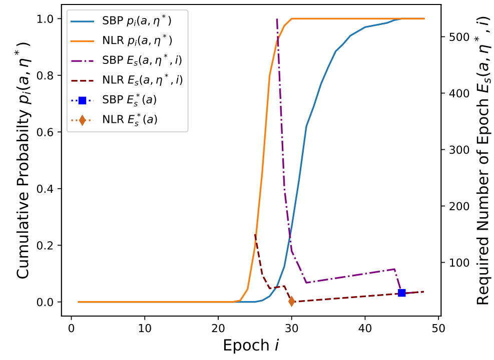
A.1.11 E-coli Accuracy



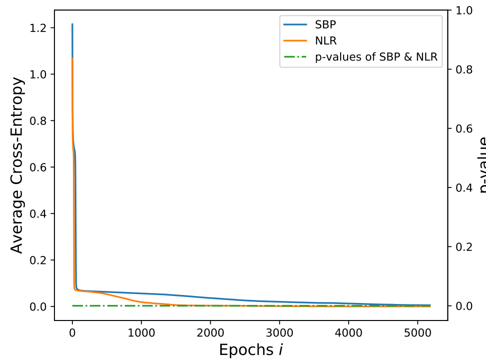
A.1.12 Authorship Accuracy



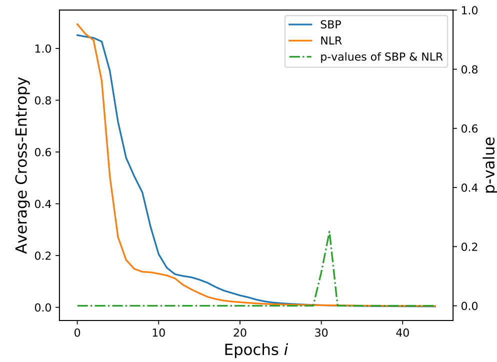
A.1.13 BCW Computational Effort



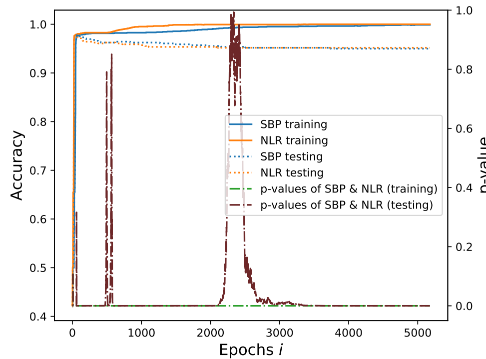
A.1.14 DNA Computational Effort



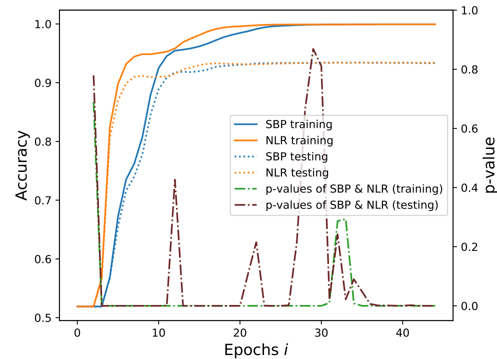
A.1.15 BCW Error



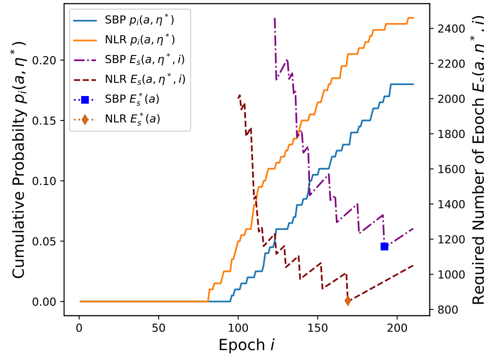
A.1.16 DNA Error



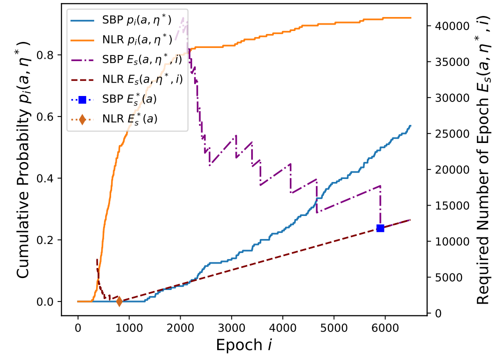
A.1.17 BCW Accuracy



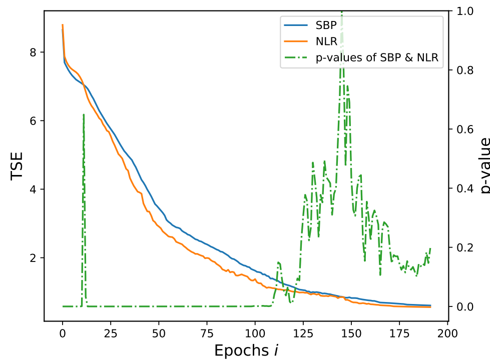
A.1.18 DNA Accuracy



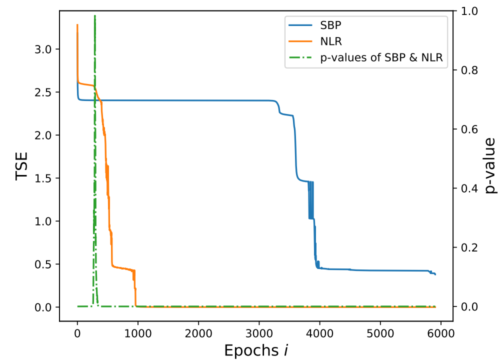
A.1.19 7-Segment Computational Effort



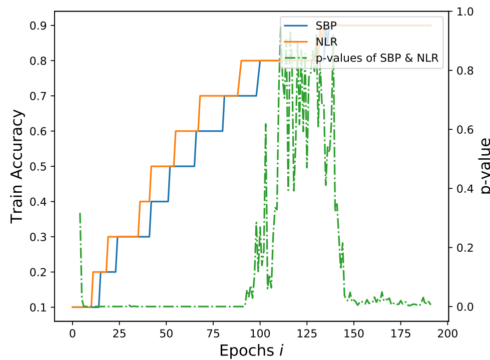
A.1.20 Parity Computational Effort



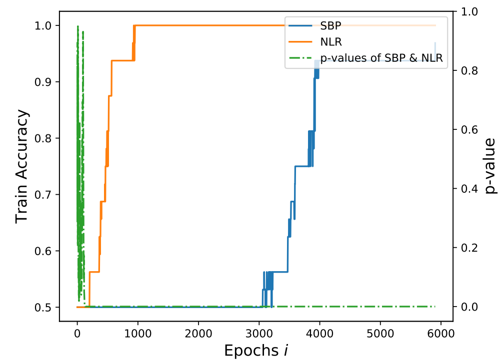
A.1.21 7-Segment Error



A.1.22 Parity Error



A.1.23 7-Segment Accuracy



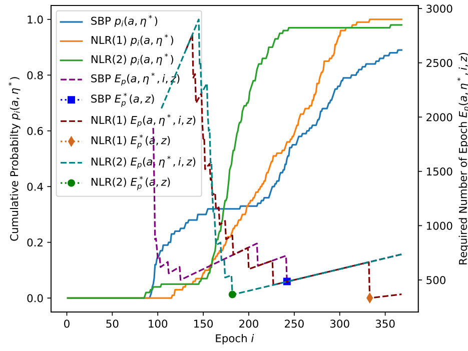
A.1.24 Parity Accuracy

Figure A.1: Comparison results of NLR and SBP for test problems when **sequential** evaluation is used.

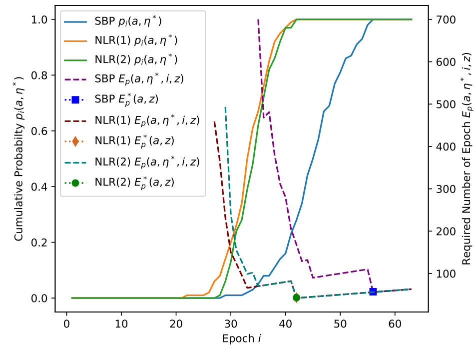
Appendix B

Additional Results from Chapter 4

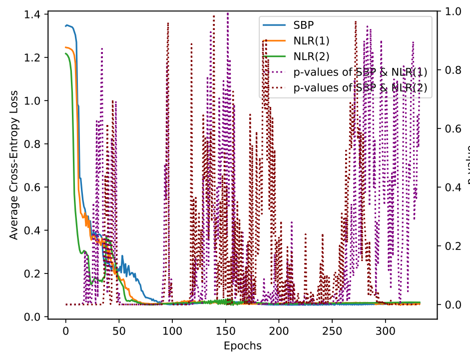
In this appendix, we report additional results that show the comparison figures of NLR(1) NLR(2) and SBP for test problems when sequential evaluation is used.



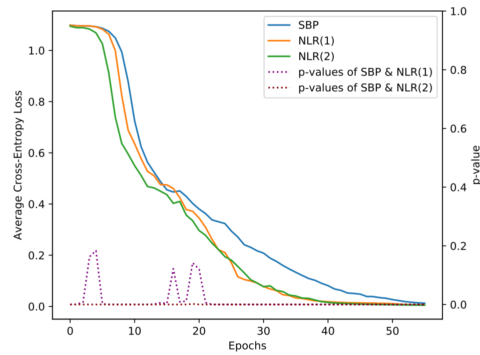
B.1.1 Iris Computational Effort



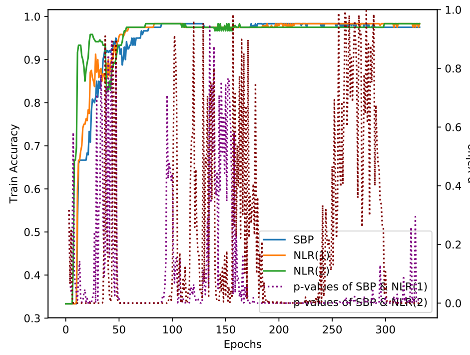
B.1.2 Wine Computational Effort



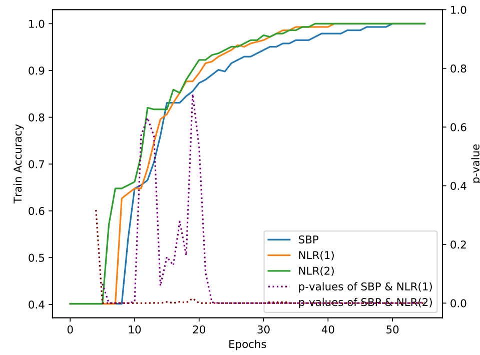
B.1.3 Iris Error



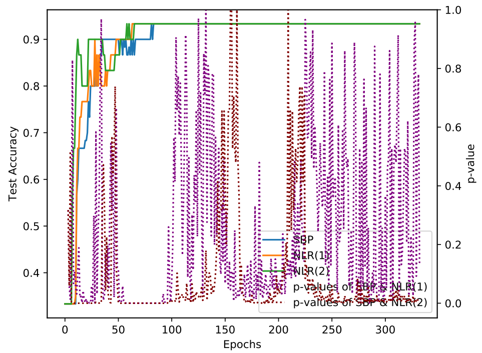
B.1.4 Wine Error



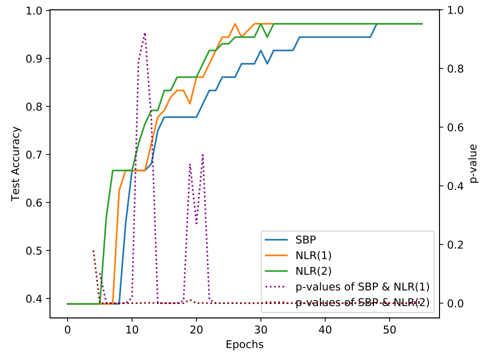
B.1.5 Iris Train Accuracy



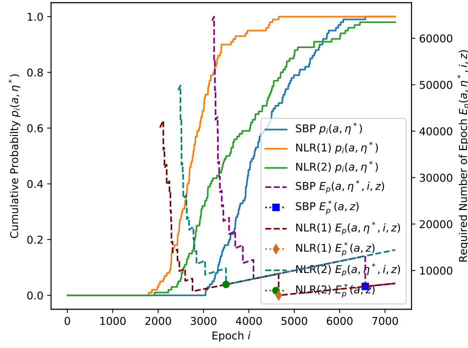
B.1.6 Wine Train Accuracy



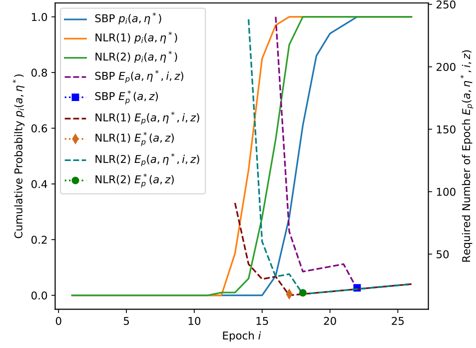
B.1.7 Iris Testing Accuracy



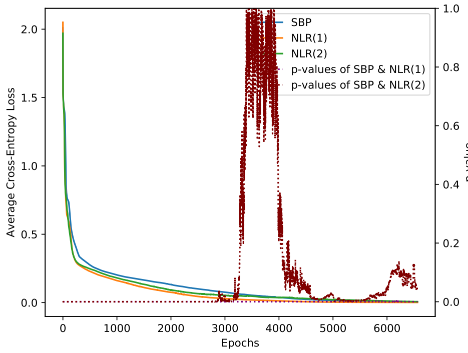
B.1.8 Wine Testing Accuracy



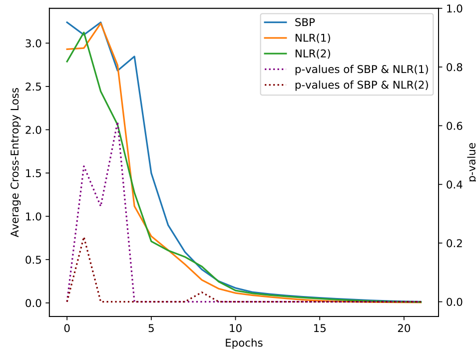
B.1.9 E-coli Computational Effort



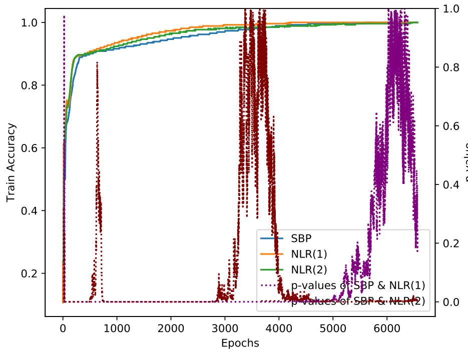
B.1.10 Authorship Computational Effort



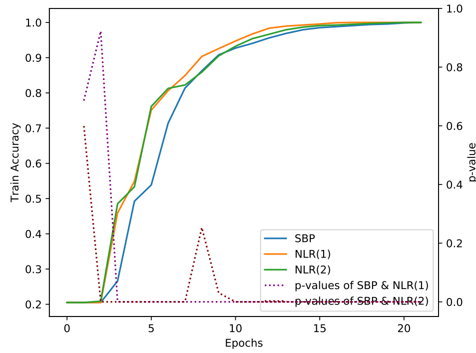
B.1.11 E-coli Error



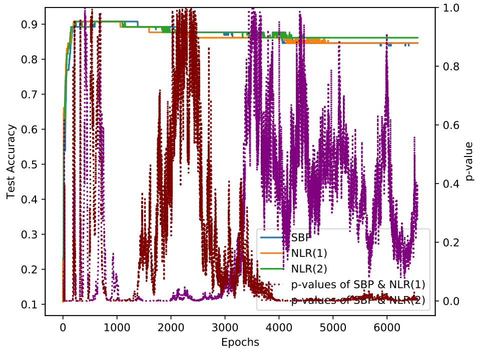
B.1.12 Authorship Error



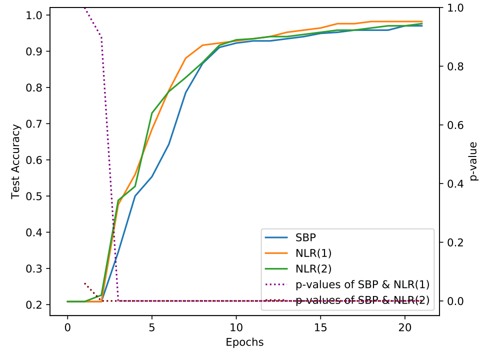
B.1.13 E-coli Train Accuracy



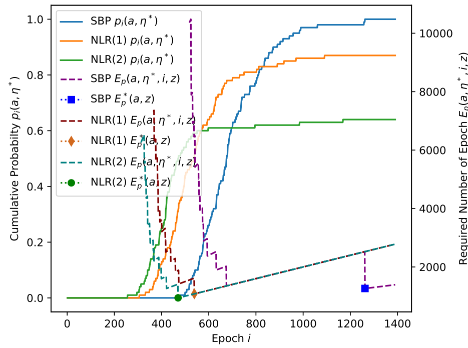
B.1.14 Authorship Train Accuracy



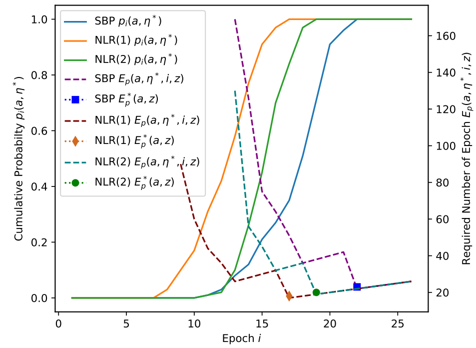
B.1.15 E-coli Testing Accuracy



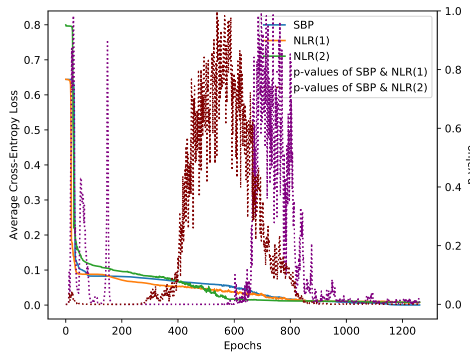
B.1.16 Authorship Testing Accuracy



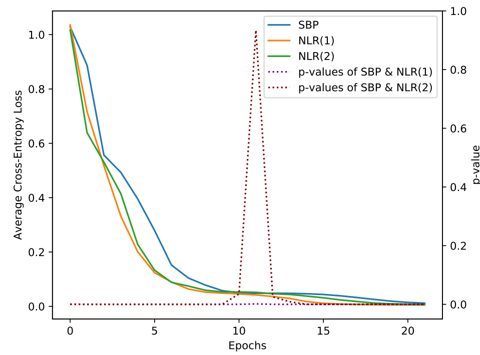
B.1.17 BCW Computational Effort



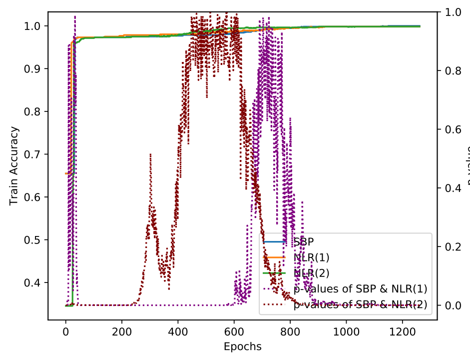
B.1.18 DNA Computational Effort



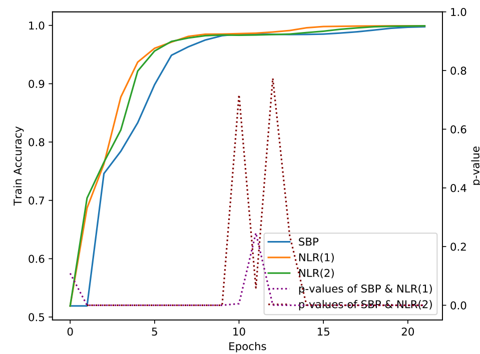
B.1.19 BCW Error



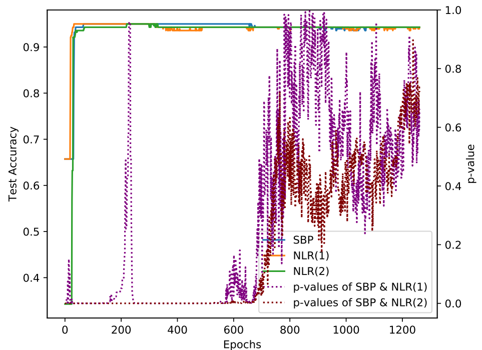
B.1.20 DNA Error



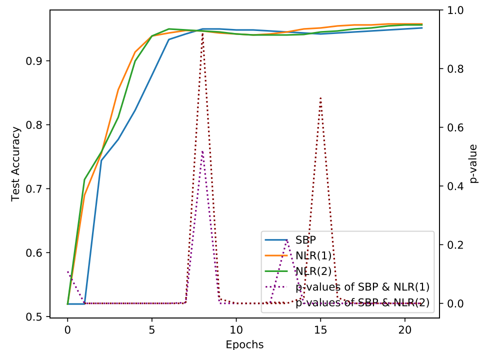
B.1.21 BCW Train Accuracy



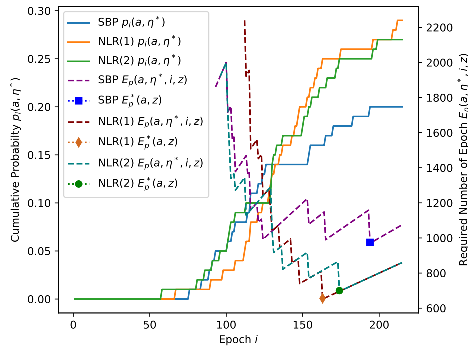
B.1.22 DNA Train Accuracy



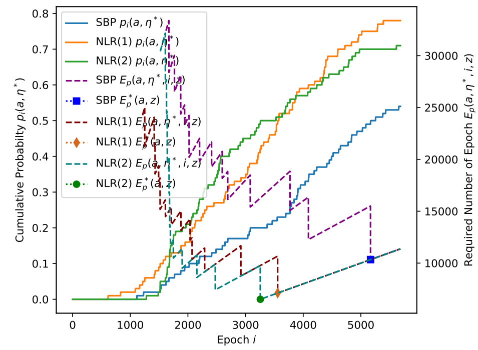
B.1.23 BCW Testing Accuracy



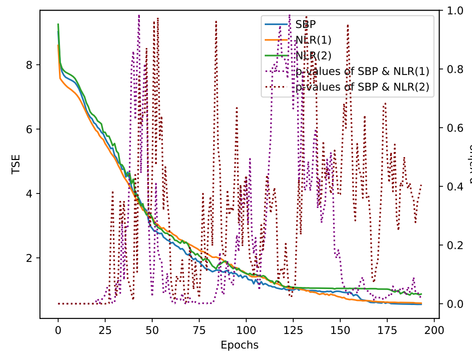
B.1.24 DNA Testing Accuracy



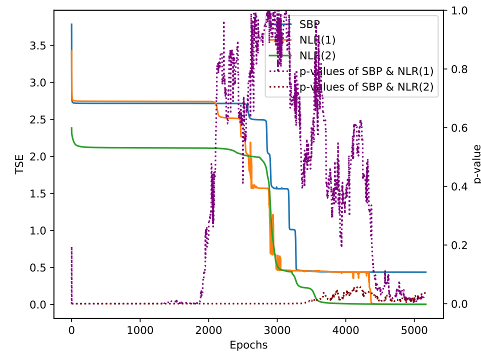
B.1.25 7-Segment Computational Effort



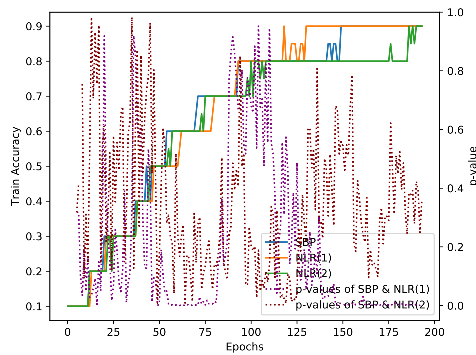
B.1.26 Parity Computational Effort



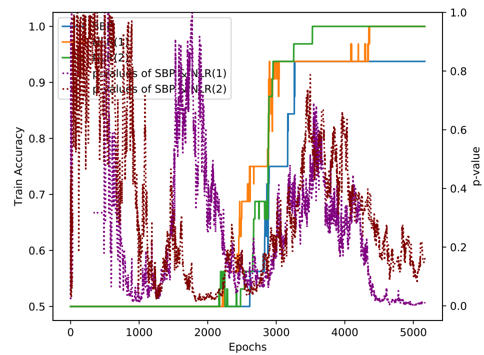
B.1.27 7-Segment Error



B.1.28 Parity Error



B.1.29 7-Segment Train Accuracy



B.1.30 Parity Train Accuracy

Figure B.1: Comparison results of NLR(1), NLR(2) and SBP for test problems when **sequential** evaluation is used.

Bibliography

[1] Michael A Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[3] Ethem Alpaydin. *Introduction to Machine Learning*. MIT Pess, 2010.

[4] Pedro Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87, 2012.

[5] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. 2018.

[6] Joaquín J Torres and Pablo Varona. Modeling biological neural networks. In *Handbook of Natural Computing*, pages 533–564. Springer, 2012.

[7] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[8] Donald Olding Hebb. *The organization of behavior: a neuropsychological theory*. J. Wiley; Chapman & Hall, 1949.

[9] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[10] Bernard Widrow. An adaptive "adaline" neuron using chemical "memistors". Technical Report 1553-2, Stanford Electronics Laboratories, 1960.

[11] Marvin Minsky and Seymour A Papert. *Perceptrons*. MIT press, 1969.

[12] D Ruhmelhart, G Hinton, and R Wiliams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[13] James L McClelland, David E Rumelhart, PDP Research Group, et al. *Parallel distributed processing*. MIT Press, 1987.

[14] James A Anderson, Edward Rosenfeld, and Andras Pellionisz. *Neurocomputing*. MIT Press, 1988.

[15] James L McClelland and David E Rumelhart. *Explorations in parallel distributed processing: A handbook of models, programs, and exercises*. MIT Press, 1989.

[16] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[17] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[18] Nichael Lynn Cramer. A representation for the adaptive generation of simple sequential programs. In *proceedings of an International Conference on Genetic Algorithms and the Applications*, pages 183–187, 1985.

[19] John R Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and computing*, 4(2):87–112, 1994.

[20] Qing Li, Weidong Cai, Xiaogang Wang, Yun Zhou, David Dagan Feng, and Mei Chen. Medical image classification with convolutional neural network. In *2014 13th International Conference on Control Automation Robotics & Vision (ICARCV)*, pages 844–848, 2014.

[21] Umut Orhan, Mahmut Hekim, and Mahmut Ozer. Eeg signals classification using the k-means clustering and a multilayer perceptron neural network model. *Expert Systems with Applications*, 38(10):13475–13481, 2011.

[22] Alexis Conneau, Holger Schwenk, Loïc Barrault, and Yann Lecun. Very deep convolutional networks for text classification. *arXiv preprint arXiv:1606.01781*, 2016.

[23] Amr Radi and Riccardo Poli. Discovery of backpropagation learning rules using genetic programming. In *1998 IEEE International Conference on Evolutionary Computation Proceedings. IEEE World Congress on Computational Intelligence (Cat. No.98TH8360)*, pages 371–375, 1998.

[24] Amr Radi and Riccardo Poli. Evolutionary discovery of learning rules for feedforward neural networks with step activation function. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation*, volume 2, pages 1178–1183, 1999.

[25] Samy Bengio, Yoshua Bengio, Jocelyn Cloutier, and Jan Gecsei. On the optimization of a synaptic learning rule. Technical report, Université de Montréal, Département d'informatique et de recherche opérationnelle, 1995.

[26] David J Chalmers. The evolution of learning: An experiment in genetic connectionism. In *Proceedings of the 1990 Connectionist Models Summer SchoolProceedings of the 1990 Connectionist Models Summer School*, pages 81–90, 1991.

[27] Samy Bengio, Yoshua Bengio, and Jocelyn Cloutier. Use of genetic programming for the search of a new learning rule for neural networks. In *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, pages 324–327, 1994.

[28] Amr Mohamed Radi and Riccardo Poli. Discovery of neural network learning rules using genetic programming. Technical report, School of Computer Science The University of Birmingham, 1997.

[29] Amr Radi and Riccardo Poli. Genetic programming can discover fast and general learning rules for neural networks. Technical report, University of Birmingham, School of Computer Science, 1998.

[30] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9, pages 249–256, 2010.

[31] Siddharth Krishna Kumar. On weight initialization in deep neural networks. *arXiv:1704.08863v2*, 2017.

[32] Dmytro Mishkin and Jiri Matas. All you need is a good init. *arXiv preprint arXiv:1511.06422v7*, 2015.

[33] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[34] Laurene V Fausett. *Fundamentals of Neural Networks: Architectures, Algorithms, and Applications*. Prentice-Hall, 1994.

[35] Shelja Kendar Pratap and Shelja. Artificial neural network (ann) inspired from biological nervous system. *International Journal of Application or Innovation in Engineering & Management (IJAIEM)*, 2(1), 2013.

[36] Neural network concepts. `https://missinglink.ai/guides/neural-network-concepts/7-types-neural-network-activation-functions-right/`. Accessed: 2019-08-28.

[37] P Sibi, S Allwyn Jones, and P Siddarth. Analysis of different activation functions using back propagation neural networks. *Journal of Theoretical and Applied Information Technology*, 47(3):1264–1268, 2013.

[38] Cenk Bircanoğlu and Nafiz Arıca. A comparison of activation functions in artificial neural networks. In *2018 26th Signal Processing and Communications Applications Conference (SIU)*, pages 1–4, 2018.

[39] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[40] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *International Conference on Machine Learning Workshop on Deep Learning for Audio, Speech, and Language Processing*, 2013.

[41] Irina Kataeva, Farnood Merrikh-Bayat, Elham Zamanidoost, and Dmitri Strukov. Efficient training algorithms for neural networks based on memristive crossbar circuits. In *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2015.

[42] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J Smola. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 661–670, 2014.

[43] Alvin J Surkan and J Clay Singleton. Neural networks for bond rating improved by multiple hidden layers. In *1990 IJCNN International Joint Conference on Neural Networks*, pages 157–162, 1990.

[44] Guang-Bin Huang. Learning capability and storage capacity of two-hidden-layer feedforward networks. *IEEE Transactions on Neural Networks*, 14(2):274–281, 2003.

[45] Shin'ichi Tamura and Masahiko Tateishi. Capabilities of a four-layered feedforward neural network: four layers versus three. *IEEE Transactions on Neural Networks*, 8(2):251–255, 1997.

[46] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, 2006.

[47] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[48] Derrick Nguyen and Bernard Widrow. Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights. In *1990 IJCNN International Joint Conference on Neural Networks*, pages 21–26 vol.3, 1990.

[49] Lodewyk FA Wessels and Etienne Barnard. Avoiding false local minima by proper initialization of connections. *IEEE Transactions on Neural Networks*, 3(6):899–905, 1992.

[50] Deniz Erdogmus, Oscar Fontenla-Romero, Jose C Principe, Amparo Alonso-Betanzos, Enrique Castillo, and Robert Jenssen. Accurate initialization of neural network weights by backpropagation of the desired response. In *Proceedings of the International Joint Conference on Neural Networks, 2003.*, volume 3, pages 2005–2010 vol.3, 2003.

[51] Jin-Song Pei, Joseph P Wright, and Andrew W Smyth. Neural network initialization with prototypes-a case study in function approximation. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 3, pages 1377–1382 vol. 3. IEEE, 2005.

[52] Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy layer-wise training of deep networks. In *Advances in Neural Information Processing Systems 19: Proceedings of the 2006 Conference*, pages 153–160, 2007.

[53] AJ Al-Shareef and Maysam F Abbod. Neural networks initial weights optimisation. In *2010 12th International Conference on Computer Modelling and Simulation*, pages 57–61, 2010.

[54] Stavros P Adam, Dimitrios A Karras, George D Magoulas, and Michael N Vrahatis. Solving the linear interval tolerance problem for weight initialization of neural networks. *Neural Networks*, 54:17–37, 2014.

[55] Mehmet Saygın Seyfioğlu and Sevgi Zübeyde Gürbüz. Deep neural network initialization methods for micro-doppler classification with low training sample support. *IEEE Geoscience and Remote Sensing Letters*, 14(12):2462–2466, 2017.

[56] MPS Bhatia and Pravin Chandra. A new weight initialization method for sigmoidal fann. *Journal of Intelligent & Fuzzy Systems*, 35(5):5193–5201, 2018.

[57] Marcus Frean. The upstart algorithm: A method for constructing and training feedforward neural networks. *Neural Computation*, 2(2):198–209, 1990.

[58] Rudy Setiono. Feedforward neural network construction using cross validation. *Neural Computation*, 13(12):2865–2877, 2001.

[59] Qi Shen, Jian-Hui Jiang, Chen-Xu Jiao, Wei-Qi Lin, Guo-Li Shen, and Ru-Qin Yu. Hybridized particle swarm algorithm for adaptive structure training of multilayer feed-forward neural network: Qsar studies of bioactivity of organic compounds. *Journal of Computational Chemistry*, 25(14):1726–1735, 2004.

[60] Teresa B Ludermir, Akio Yamazaki, and Cleber Zanchettin. An optimization methodology for neural network weights and architectures. *IEEE Transactions on Neural Networks*, 17(6):1452–1459, 2006.

[61] Florin Leon. Optimizing neural network topology using shapley value. In *2014 18th International Conference on System Theory, Control and Computing (ICSTCC)*, pages 862–867, 2014.

[62] Devika Chhachhiya, Amita Sharma, and Manish Gupta. Designing optimal architecture of neural network with particle swarm optimization techniques specifically for educational dataset. In *2017 7th International Conference on Cloud Computing, Data Science & Engineering-Confluence*, pages 52–57, 2017.

[63] Irwan Bello, Barret Zoph, Vijay Vasudevan, and Quoc V Le. Neural optimizer search with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70, pages 459–468, 2017.

[64] S Squartini, A Hussain, and F Piazza. Attempting to reduce the vanishing gradient effect through a novel recurrent multiscale architecture. In *Proceedings of the International Joint Conference on Neural Networks, 2003.*, volume 4, pages 2819–2824, 2003.

[65] Shumin Kong and Masahiro Takatsuka. Hexpo: A vanishing-proof activation function. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 2562–2567, 2017.

[66] Xin Wang, Yi Qin, Yi Wang, Sheng Xiang, and Haizhou Chen. Reltanh: An activation function with vanishing gradient resistance for sae-based dnns

and its application to rotating machinery fault diagnosis. *Neurocomputing*, 363:88–98, 2019.

[67] Yi Zhou, Yue Bai, Shuvra S Bhattacharyya, and Heikki Huttunen. Elastic neural networks for classification. In *2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pages 251–255, 2019.

[68] Li Wan, Matthew Zeiler, Sixin Zhang, Yann Le Cun, and Rob Fergus. Regularization of neural networks using dropconnect. In *in International Conference on Machine Learning*, pages 1058–1066, 2013.

[69] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[70] Fernando M Silva and Luis B Almeida. Speeding up backpropagation. In *Advanced Neural Computers*, pages 151–158. Elsevier, 1990.

[71] Martin Riedmiller and Heinrich Braun. A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *Proceedings of the IEEE International Conference on Neural Networks*, volume 1, pages 586–591, 1993.

[72] Xiao-Hu Yu, Guo-An Chen, and Shi-Xin Cheng. Dynamic learning rate optimization of the backpropagation algorithm. *IEEE Transactions on Neural Networks*, 6(3):669–677, 1995.

[73] LiMin Fu, Hui-Huang Hsu, and Jose C Principe. Incremental backpropagation learning networks. *IEEE Transactions on Neural Networks*, 7(3):757–761, 1996.

[74] Wen Jin, Zhao Jia Li, Luo Si Wei, and Han Zhen. The improvements of bp neural network learning algorithm. In *WCC 2000-ICSP 2000. 2000 5th International Conference on Signal Processing Proceedings. 16th World Computer Congress 2000*, volume 3, pages 1647–1649, 2000.

[75] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[76] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 2012.

[77] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[78] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167v3*, 2015.

[79] Tim Salimans and Durk P Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *Advances in Neural Information Processing Systems 29*, pages 901–909, 2016.

[80] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv:1609.04747*, 2016.

[81] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2011.

[82] Timothy Dozat. Incorporating nesterov momentum into adam. In *International Cconference on Learning Representations Wokshop*, 2016.

[83] Ashia C Wilson, Rebecca Roelofs, Mitchell Stern, Nati Srebro, and Benjamin Recht. The marginal value of adaptive gradient methods in machine learning. In *Advances in Neural Information Processing Systems 30*, pages 4148–4158, 2017.

[84] Georg Thimm and Emile Fiesler. High-order and multilayer perceptron initialization. *IEEE Transactions on Neural Networks*, 8(2):349–359, 1997.

[85] Scott E Fahlman. An empirical study of learning speed in back-propagation networks. Technical Report CMU-CS-88-162, School of Computer Science, Carnegie Mellon University, 1988.

[86] Egbert JW Boers and Herman Kuiper. Biological metaphors and the design of modular artificial neural networks. In *International Conference on Artificial Neural Networks, ICANN '1993*, London, 1993.

[87] Gian Paolo Drago and Sandro Ridella. Statistically controlled activation weight initialization (scawi). *IEEE Transactions on Neural Networks*, 3(4):627–631, 1992.

[88] Jim YF Yam and Tommy WS Chow. Feedforward networks training speed enhancement by optimal initialization of the synaptic coefficients. *IEEE Transactions on Neural Networks*, 12(2):430–434, 2001.

[89] LY Bottou. Reconnaissance de la parole par reseaux multi-couches. In *Proceedings of the International Workshop Neural Networks Application, Neuro-Nimes*, volume 88, pages 197–217, 1988.

[90] Frank J Smieja. Hyperplane "spin" dynamics, network plasticity and back-propagation learning. Technical Report 634, Gesellschaft fOr Mathemafik und Datenverarbeitung, St. Augustin, Germany, 1991.

[91] YK Kim and JB Ra. Weight value initialization for improving training speed in the backpropagation network. In *[Proceedings] 1991 IEEE International Joint Conference on Neural Networks*, pages 2396–2401, 1991.

[92] S Osowski. New approach to selection of initial values of weights in neural function approximation. *Electronics Letters*, 29(3):313–315, 1993.

[93] Hisashi Shimodaira. A weight value initialization method for improving learning performance of the backpropagation algorithm in neural networks. In *Proceedings Sixth International Conference on Tools with Artificial Intelligence. TAI 94*, pages 672–675, 1994.

[94] Sartaj Singh Sodhi, Pravin Chandra, and Sharad Tanwar. A new weight initialization method for sigmoidal feedforward artificial neural networks. In *2014 International Joint Conference on Neural Networks (IJCNN)*, pages 291–298, 2014.

[95] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[96] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289v5*, 2015.

[97] Jia Li, Hua Xu, Junhui Deng, and Xiaomin Sun. Hyperbolic linear units for deep convolutional neural networks. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 353–359, 2016.

[98] Alaa Sagheer and Mostafa Kotb. Unsupervised pre-training of a deep lstm-based stacked autoencoder for multivariate time series forecasting problems. *Scientific Reports*, 9(19038), 2019.

[99] James Bergstra, Guillaume Desjardins, Pascal Lamblin, and Yoshua Bengio. Quadratic polynomials learn better image features. Technical Report 1337,

Département d'Informatique et de Recherche Opérationnelle, Université de Montréal, 2009.

[100] Lawrence J Fogel, Alvin J Owens, and Michael J Walsh. *Artificial intelligence through simulated evolution*. Wiley, 1966.

[101] Hans-Paul Schwefel. *Numerical optimization of computer models*. Chichester ; New York : Wiley, 1981. Translation of Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie.

[102] Hans-Paul Paul Schwefel. *Evolution and optimum seeking: the sixth generation*. John Wiley & Sons, Inc., 1993.

[103] John Henry Holland. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence*. Ann Arbor: University of Michigan Press, 1975.

[104] Agoston E Eiben and James E Smith. *Introduction to evolutionary computing*. Springer, 2015.

[105] Charles Darwin. *The origin of species. 6th*, volume 570. John Murray, London, 1859.

[106] Derviş Karaboğa. *Yapay zeka optimizasyon algoritmaları*. Nobel Akademi Yayıncılık, 2014.

[107] Riccardo Poli, John Woodward, and Edmund K Burke. A histogram-matching approach to the evolution of bin-packing strategies. In *2007 IEEE Congress on Evolutionary Computation*, pages 3500–3507. IEEE, 2007.

[108] Edmund K Burke, Mathew R Hyde, Graham Kendall, Gabriela Ochoa, Ender Ozcan, and John R Woodward. Exploring hyper-heuristic methodologies with genetic programming. In *Computational Intelligence: Collaboration, Fusion and Emergence, Intelligent Systems Reference Library*, pages 177–201. Springer, 2009.

[109] Mohamed Bader-El-Den, Riccardo Poli, and Shaheen Fatima. Evolving timetabling heuristics using a grammar-based genetic programming hyper-heuristic framework. *Memetic Computing*, 1:205–219, 2009.

[110] William B Langdon. Genetic programming—computers using "natural selection" to generate programs. In *Genetic Programming and Data Structures. The Springer International Series in Engineering and Computer Science*, pages 9–42. Springer, 1998.

[111] Riccardo Poli, William B Langdon, Nicholas F McPhee, and John R Koza. *A field guide to genetic programming*. Lulu. com, 2008.

[112] Julian F Miller and Peter Thomson. Cartesian genetic programming. In *European Conference on Genetic Programming*, volume 1802, pages 121–132. Springer, 2000.

[113] Julian Francis Miller and Simon L. Harding. Cartesian genetic programming. In *Proceedings of the 10th Annual Conference Companion on Genetic and Evolutionary Computation*, page 2701–2726, 2008.

[114] Mihai Oltean and Crina Grosan. A comparison of several linear genetic programming techniques. *Complex Systems*, 14:285–313, 2003.

[115] Markus F Brameier and Wolfgang Banzhaf. *Linear genetic programming*. Springer US, 2007.

[116] Luca Citi, Riccardo Poli, and Caterina Cinel. High-significance averages of event-related potential via genetic programming. In *Genetic Programming Theory and Practice VII. Genetic and Evolutionary Computation*, pages 135–157. Springer, 2010.

[117] Randy L Haupt and Sue Ellen Haupt. *Practical genetic algorithms*. John Wiley & Sons, Inc, 2 edition, 2004.

[118] William B Langdon and Wolfgang Banzhaf. Repeated sequences in linear genetic programming genomes. *Complex Systems*, 15(4):285–306, 2005.

[119] Markus Brameier and Wolfgang Banzhaf. A comparison of linear genetic programming and neural networks in medical data mining. *IEEE Transactions on Evolutionary Computation*, 5(1):17–26, 2001.

[120] Ercan Öztemel. *Yapay sinir ağları*. Papatya, 2012.

[121] Amr Radi and Riccardo Poli. Discovering efficient learning rules for feed-forward neural networks using genetic programming. In *Recent Advances in Intelligent Paradigms and Applications*, pages 133–159. Physica, Heidelberg, 2003.

[122] Alexis P Wieland. Evolving neural network controllers for unstable systems. In *IJCNN-91-Seattle International Joint Conference on Neural Networks*, volume 2, pages 667–673, 1991.

[123] David J Janson and James F Frenzel. Application of genetic algorithms to the training of higher order neural networks. *Journal of Systems Engineering*, 2(4):272–276, 1992.

[124] David J Janson and James F Frenzel. Training product unit neural networks with genetic algorithms. *IEEE Expert*, 8(5):26–33, 1993.

[125] John R McDonnell and Don Waagen. Evolving neural network connectivity. In *IEEE International Conference on Neural Networks*, pages 863–868, 1993.

[126] Natarajan Saravanan and David B Fogel. Evolving neural control systems. *IEEE Expert*, 10(3):23–27, 1995.

[127] David B Fogel, Eugene C Wasson III, and Edward M Boughton. Evolving neural networks for detecting breast cancer. *Cancer letters*, 96(1):49–53, 1995.

[128] S Yao, CJ Wei, and ZY He. Evolving wavelet neural networks for function approximation. *Electronics Letters*, 32(4):360–361, 1996.

[129] Garrison W Greenwood. Training partially recurrent neural networks using evolutionary strategies. *IEEE Transactions on Speech and Audio Processing*, 5(2):192–194, 1997.

[130] Kumar Chellapilla and David B Fogel. Evolving neural networks to play checkers without relying on expert knowledge. *IEEE Transactions on Neural Networks*, 10(6):1382–1391, 1999.

[131] Yann-Chang Huang. Evolving neural nets for fault diagnosis of power transformers. *IEEE Transactions on Power Delivery*, 18(3):843–848, 2003.

[132] Mohamed Benaddy, Mohamed Wakrim, and Sultan Aljahdali. Evolutionary neural network prediction for cumulative failure modeling. In *2009 IEEE/ACS International Conference on Computer Systems and Applications*, pages 179–184, 2009.

[133] Xin Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.

[134] Alden H Wright. Genetic algorithms for real parameter optimization. In *Foundations of Genetic Algorithms*, volume 1, pages 205–218. Morgan Kaufmann, 1991.

[135] Larry J Eshelman and J David Schaffer. Real-coded genetic algorithms and interval-schemata. In *Foundations of Genetic Algorithms*, volume 2, pages 187–202. Morgan Kaufmann, 1993.

[136] Adeike A Adewuya. New methods in genetic search with real-valued chromosomes. Master's thesis, Massachusetts Institute of Technology. Dept. of Mechanical Engineering, Massachusetts Institute of Technology, 1996.

[137] Peter JB Hancock. Genetic algorithms and permutation problems: a comparison of recombination operators for neural net structure specification. In *[Proceedings] COGANN-92: International Workshop on Combinations of Genetic Algorithms and Neural Networks*, pages 108–122, 1992.

[138] Xin Yao. Evolutionary artificial neural networks. *International Journal of Neural Systems*, 4(3):203–222, 1993.

[139] Dario Floreano, Peter Dürr, and Claudio Mattiussi. Neuroevolution: from architectures to learning. *Evolutionary Intelligence*, 1:47–62, 2008.

[140] Edgar Galván and Peter Mooney. Neuroevolution in deep neural networks: Current trends and future challenges. *arXiv preprint arXiv:2006.05415*, 2020.

[141] Omid E David and Iddo Greental. Genetic algorithms for evolving deep neural networks. In *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pages 1451–1452, 2014.

[142] Sean Lander and Yi Shang. Evoae–a new evolutionary method for training autoencoders for deep learning networks. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, volume 2, pages 790–795, 2015.

[143] Gregory Morse and Kenneth O Stanley. Simple evolutionary optimization can rival stochastic gradient descent in neural networks. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2016)*, pages 477–484, 2016.

[144] Seong-Whan Lee. Off-line recognition of totally unconstrained handwritten numerals using multilayer cluster neural network. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(6):648–652, 1996.

[145] Sigeru Omatu and Michifumi Yoshioka. Self-tuning neuro-pid control and applications. In *1997 IEEE International Conference on Systems, Man, and Cybernetics. Computational Cybernetics and Simulation*, volume 3, pages 1985–1989, 1997.

[146] MI El Adawy, ME Aboul-Wafa, HA Keshk, and MM El Tayeb. A soft-backpropagation algorithm for training neural networks. In *Proceedings of the Nineteenth National Radio Science Conference*, pages 397–404, 2002.

[147] Zhengjun Liu, Aixia Liu, Changyao Wang, and Zheng Niu. Evolving neural network using real coded genetic algorithm (ga) for multispectral image classification. *Future Generation Computer Systems*, 20(7):1119–1129, 2004.

[148] A Sedki, Driss Ouazar, and E El Mazoudi. Evolving neural network using real coded genetic algorithm for daily rainfall–runoff forecasting. *Expert Systems with Applications*, 36(3):4523–4527, 2009.

[149] Shifei Ding, Chunyang Su, and Junzhao Yu. An optimizing bp neural network algorithm based on genetic algorithm. *Artificial Intelligence Review*, 36:153–162, 2011.

[150] Stephen I Gallant. Three constructive algorithms for network learning. In *Proceedings of The Eighth Annual Conference of the Cognitive Science Society*, pages 652–660, 1986.

[151] Marc Mezard and Jean-P Nadal. Learning in feedforward layered networks: The tiling algorithm. *Journal of Physics A: Mathematical and General*, 22(12):2191–2203, 1989.

[152] Michael C Mozer and Paul Smolensky. Skeletonization: A technique for trimming the fat from a network via relevance assessment. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems*, pages 107–115. Morgan-Kaufmann, 1989.

[153] Maria do Carmo Nicoletti, João R Bertini, David Elizondo, Leonardo Franco, and José M Jerez. Constructive neural network algorithms for feedforward architectures suitable for classification tasks. In *Constructive Neural Networks*, volume 258, pages 1–23. Springer, 2009.

[154] Sudhir Kumar Sharma and Pravin Chandra. Constructive neural networks: A review. *International Journal of Engineering Science and Technology*, 2(12):7847–7855, 2010.

[155] Ryad Zemouri. An evolutionary building algorithm for deep neural networks. In *2017 12th International Workshop on Self-Organizing Maps and Learning Vector Quantization, Clustering and Data Visualization (WSOM)*, pages 1–7, 2017.

[156] R Zemouri, N Omri, C Devalland, L Arnould, B Morello, N Zerhouni, and F Fnaiech. Breast cancer diagnosis based on joint variable selection and constructive deep neural network. In *2018 IEEE 4th Middle East Conference on Biomedical Engineering (MECBME)*, pages 159–164, 2018.

[157] R Zemouri, N Omri, B Morello, C Devalland, L Arnould, N Zerhouni, and F Fnaiech. Constructive deep neural network for breast cancer diagnosis. *IFAC-PapersOnLine*, 51(27):98–103, 2018.

[158] Ozan Irsoy and Ethem Alpaydin. Continuously constructive deep neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 31(4):1124–1133, 2020.

[159] João Carlos Figueira Pujol and Riccardo Poli. Evolving the topology and the weights of neural networks using a dual representation. *Applied Intelligence*, 8:73–84, 1998.

[160] Maryam Mahsal Khan, Gul Muhammad Khan, and Julian F Miller. Evolution of neural networks using cartesian genetic programming. In *IEEE Congress on Evolutionary Computation*, pages 1–8, 2010.

[161] Maryam Mahsal Khan, Arbab Masood Ahmad, Gul Muhammad Khan, and Julian F Miller. Fast learning neural networks using cartesian genetic programming. *Neurocomputing*, 121:274–289, 2013.

[162] Andrew James Turner and Julian Francis Miller. Cartesian genetic programming encoded artificial neural networks: a comparison using three benchmarks. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, pages 1005–1012, 2013.

[163] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

[164] Xin Yao and Yuhui Shi. A preliminary study on designing artificial neural networks using co-evolution. In *Proceedings of the IEEE Singapore International Conference on Intelligent Control and Instrumentation*, pages 149–154, 1995.

[165] Thomas Ragg and Steffen Gutjahr. Automatic determination of optimal network topologies based on information theory and evolution. In *EUROMICRO 97. Proceedings of the 23rd EUROMICRO Conference: New Frontiers of Information Technology (Cat. No. 97TB100167)*, pages 549–555, 1997.

[166] Claudio A Perez and Carlos A Holzmann. Improvements on handwritten digit recognition by genetic selection of neural network topology and by augmented training. In *1997 IEEE International Conference on Systems, Man, and Cybernetics. Computational Cybernetics and Simulation*, volume 2, pages 1487–1491, 1997.

[167] Yanan Sun, Bing Xue, Mengjie Zhang, and Gary G Yen. Evolving deep convolutional neural networks for image classification. *arXiv:1710.10741v3*, 2017.

[168] Kenneth O Stanley, David B D'Ambrosio, and Jason Gauci. A hypercube-based encoding for evolving large-scale neural networks. *Artificial Life*, 15(2):185–212, 2009.

[169] Phillip Verbancsics and Kenneth O Stanley. Constraining connectivity to encourage modularity in hyperneat. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2011)*, pages 1483–1490, 2011.

[170] Sebastian Risi and Kenneth O Stanley. An enhanced hypercube-based encoding for evolving the placement, density, and connectivity of neurons. *Artificial Life*, 18(4):331–363, 2012.

[171] Risto Miikkulainen, Jason Liang, Elliot Meyerson, Aditya Rawal, Daniel Fink, Olivier Francon, Bala Raju, Hormoz Shahrzad, Arshak Navruzyan, Nigel Duffy, et al. Evolving deep neural networks. In *Artificial Intelligence in the Age of Neural Networks and Brain Computing*, pages 293–312. Elsevier, 2019.

[172] Lingxi Xie and Alan Yuille. Genetic cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1388–1397, 2017.

[173] Masanori Suganuma, Shinichi Shirakawa, and Tomoharu Nagao. A genetic programming approach to designing convolutional neural network architectures. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18)*, pages 5369–5373, 2018.

[174] Masanori Suganuma, Mete Ozay, and Takayuki Okatani. Exploiting the potential of standard convolutional autoencoders for image restoration by evolutionary search. *arXiv:1803.00370*, 2018.

[175] Filipe Assunção, Nuno Lourenço, Penousal Machado, and Bernardete Ribeiro. Evolving the topology of large scale deep neural networks. In *European Conference on Genetic Programming*, volume 10781, pages 19–34, 2018.

[176] Amr Radi and Riccardo Poli. Genetic programming discovers efficient learning rules for the hidden and output layers of feedforward neural networks. In *European Conference on Genetic Programming*, volume 1598, pages 120–134, 1999.

[177] Yoshua Bengio, Samy Bengio, and Jocelyn Cloutier. Learning a synaptic learning rule. Technical report, Université de Montréal, Département d'informatique et de recherche opérationnelle, 1990.

[178] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[179] Bernard Widrow and Marcian E Hoff. Adaptive switching circuits. In *1960 IRE WESCON Convention Record*, pages 96–104, 1960.

[180] JF Fontanari and R Meir. Evolving a learning algorithm for the binary perceptron. *Network: Computation in Neural Systems*, 2(4):353–359, 1991.

[181] Jonathan Baxter. The evolution of learning algorithms for arti
cial neural networks. *Complex Systems*, pages 313–326, 1992.

[182] Amr Radi and Riccardo Poli. Discovery of general learning rules for feedforward neural networks with step activation function using genetic programming. Technical report, University of Birmingham, School of Computer Science, 1999.

[183] Sebastian Risi and Kenneth O Stanley. Indirectly encoding neural plasticity as a pattern of local rules. In *International Conference on Simulation of Adaptive Behavior*, pages 533–543, 2010.

[184] Thomas Philip Runarsson and Magnus Thor Jonsson. Evolution and design of distributed learning rules. In *2000 IEEE Symposium on Combinations of Evolutionary Computation and Neural Networks. Proceedings of the First IEEE Symposium on Combinations of Evolutionary Computation and Neural Networks (Cat. No. 00*, pages 59–63, 2000.

[185] Jeff Orchard and Lin Wang. The evolution of a generalized neural learning rule. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 4688–4694, 2016.

[186] Lin Wang and Jeff Orchard. Investigating the evolution of a neuroplasticity network for learning. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 49(10):2131–2143, 2019.

[187] Sheldon M Ross. *Introduction to probability models*. Academic Press, 2014.

[188] Wolfgang Banzhaf, Peter Nordin, Robert E Keller, and Frank D Francone. *Genetic programming: An introduction*. Morgan Kaufmann San Francisco, 1998.

[189] Randal S. Olson, William La Cava, Patryk Orzechowski, Ryan J. Urbanowicz, and Jason H. Moore. Pmlb: a large benchmark suite for machine learning evaluation and comparison. *BioData Mining*, 10(1):36, Dec 2017.

[190] Moore Jason H. Epistasislab penn-ml-benchmarks. https://github.com/EpistasisLab/penn-ml-benchmarks, 2017.

[191] Standford University Computer Science. Vanishing gradient. https://cs224d.stanford.edu/notebooks/vanishing_grad_example.html.