Novel Evolutionary Algorithms for Robust Training of Very Small Multilayer Perceptron Models

Huang, Chuan-Bin

Award date:
2020

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Data Mining Group

Novel Evolutionary Algorithms for Robust Training of Very Small Multilayer Perceptron Models

Master Thesis

Chuan-Bin Huang

Supervisors:
Dr. Decebal C. Mocanu
Prof. Dr. Mykola Pechenizkiy

Committee members:
Dr. Decebal C. Mocanu
Prof. Dr. Mykola Pechenizkiy
Dr. Odysseas Papapetrou

Version 3

Eindhoven, August 2020

Abstract

Mobile devices are increasingly called upon to implement artificial intelligence (AI) applications; however, the limited computational power and battery life of these devices means that deployment depends heavily on the ability to train small models of reasonable accuracy. Small models of this type are typically created by simplifying larger models. Our objective in the current study was to develop an algorithm to directly enable the efficient training of small models. The sparse evolutionary training (SET) algorithm has proven highly effective in continuously probing the model parameter space based on previously-trained models. This thesis proposed four novel redesigned evolutionary algorithms derived from SET and performed benchmarking using very-small densely-connected multilayer perceptron (MLP) models. Evaluation results from synthetic datasets provide a solid proof-of-concept that the proposed algorithms are capable of probing model parameter space efficiently. However, when applied to real-world datasets, data features were shown to interfere with the effectiveness of the proposed algorithms.

Preface

This original thesis is an unpublished work developed independently by the author, Chuan-Bin Huang, under the supervision of Dr. Decebal C. Mocanu at Eindhoven University of Technology, the Netherlands (TU/e).

This thesis targets readers who are familiar with computer algorithms and neural networks; however, a basic introduction to these topics is provided for readers without prior knowledge (see Chapter 2). The text is structured to enable rearrangement of the content into a publishable paper with relative ease.

This thesis introduces a number of insights applicable to the effective training of small models for the deployment of AI systems on devices with limited computational capability. The thinking, design, and execution are presented faithfully and responsibly to provide reference for future works.

I would like to thank Dr. Mocanu for his guidance in this thesis. I would also like to thank Prof. Dr. M. Pechenizkiy and Dr. O. Papapetrou for their review of this thesis. I would like to thank Dr. E.C. Huang, Dr. Q.R. Huang, Dr. L.Y. Wang, Dr. S.C. Wu, and Dr. T.W. Xie, for their valuable advice pertaining to the graphical details. I would like to thank Dr. Chern Chuang for his prompt advice pertaining to latex editing. I am indebted to TU/e for providing the resources required to complete this thesis, including the Amandus H. Lundqvist Scholarship Program (ALSP), which is funding my studies. I am also sincerely grateful to Dr. L.Y. Hsu and Ms. Mandy Liu who provided invaluable opportunities to polish my coding skills and made recommendations for my application to TU/e.

Chuan-Bin ”Bill” Huang

July 31, 2020

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Research Goals

The development of efficient artificial intelligence (AI) models for small[1] mobile devices is crucial to their adoption in everyday life. However, the limited computational power and/or battery life of these devices means that deployment depends heavily on the ability to train small[2] models of reasonable accuracy. Our primary objective in this thesis was to develop algorithms capable of training small models[3] with accuracy exceeding that of plain (conventional) training methods. Such models should facilitate the deployment of AI systems on mobile devices.

Our research goals are outlined in the following.

1. We sought to assess the effectiveness of using redesigned versions of the sparse evolutionary training (SET) algorithm proposed by Mocanu et al. [2018] to train densely connected small MLP models. For the sake of readability, these redesigned algorithms will be referred to as *evolutionary algorithms* hereafter whenever no ambiguity presents.[4] Precise definitions of the proposed algorithms are presented in Section 3.1.

2. We compared the proposed training algorithms with plain training with the aim of identifying the factors underlying model accuracy.

3. We investigated the model training history to find indicators of effective training.

4. We investigated the means by which training results are affected by the complexity of data.

Note that due to a lack of studies in this area of research, this thesis serves as a proof-of-concept rather than a precise solution to this difficult problem. Thus, this thesis takes a

[1]i.e. possessing limited computational capability.

[2]The "size" of the model is typically measured by the total number of model parameters, which corresponds to the memory space required for storage. In the case of densely connected multilayer perceptron (MLP) models, the size can be roughly measured as the total number of edges (links).

[3]Methods have also been developed to obtain a small model by simplifying a large model (see Section 2.2).

[4]The general idea of evolutionary algorithm is introduced briefly in Section 2.1.3.

qualitative perspective focusing on specific cases as a starting point for subsequent research in this field. Quantitative and general claims are left for future studies.

1.2 Research Questions

The research goals presented in Section 1.1 were translated into the following questions.

1. *Is it possible to construct a concrete example (including a small model and a reasonably-sized dataset) to demonstrate the efficacy of the proposed algorithms in training models that are more accurate than those obtained via plain training?* To construct a clear example, a synthetic dataset can be used such that the minimal model size required to achieve 100% accuracy can be determined analytically. This provides precise control over the degree of redundancy in the trained model, which is crucial to choosing a compact model size.

2. *When trained using an equal number of total epochs, do the proposed algorithms still outperform plain training in terms of model accuracy?* A meaningful comparison of algorithms requires basing assessment on the same quantile of the same total number of epochs.

3. *Do any patterns appear in training history plots,*[5] *which could be used to verify that an algorithm (or set of hyperparameters) provides an accuracy gain over plain training?* This provides potentially useful diagnostic information for further improvements.

4. *Do the conclusions of the above experiments still hold when using complex real-world datasets?* The MNIST dataset was used for real-world assessments in the current study.

1.3 Contributions

1.3.1 Domain Knowledge

This thesis sought to establish a new direction for research in the domain of small-model training based on the use of evolutionary algorithms. To the best of the author's knowledge at the time of writing, researchers have yet to develop a systematic understanding of the methods used for the direct training of small models. Furthermore, researchers have yet to report on the application of evolutionary algorithms for the training of very small MLP models. The contributions of this thesis are as follows:

1. This thesis provides a proof-of-concept that multi-stage training starting from the partial perturbation (evolution) of previously-trained results can be more effective than simply retraining re-initialized states multiple times.

2. This thesis initiates a novel research direction for adapting SET to training small models where model sparseness is not really a requirement.

[5]plot of validation accuracy versus training epoch

3. This thesis proposes a methodology to benchmark training algorithms by constructing synthetic datasets of which their minimal optimal model size are known analytically. Two datasets were constructed for this purpose.

4. This thesis demonstrates that *evolution of important (large-magnitude) weights can help improving model accuracy.* In other words, evolving only coefficients with the smallest magnitude is not necessarily superior to random evolution. This conclusion is complementary to the suggestions made by Mocanu et al. [2018], despite the fact that they initially appear contradictory, due to fundamental differences in the experiments employed in the two studies.

As mentioned in Section 1.1, the results of this thesis should be interpreted qualitatively and intuitively rather than quantitatively and strictly. Also note that the experiments were conducted under highly-specific conditions.

1.3.2 Code Framework

An experimental multi-stage training framework was implemented as a customized add-on for the Keras library[6] to compensate for a lack of built-in multi-stage training functionality. The framework mimics the *callback* logic as in Keras, which executes the evolution of weights after a model is properly trained for a specified number of epochs (hereafter referred to as a "*stage*").[7] This framework facilitates implementation of different evolutionary algorithms as multi-stage callbacks, without interfering with other parts of the code. The framework can also be extended to arbitrary multi-stage algorithms.[8]

The entire code module was implemented in Python 3.[9] The code[10] and the documentation[11] are made public on GitLab[12]. A support infrastructure for the multi-stage training scheme was also implemented in the working code, including a data pipeline and a statistical analyzer. Note however that they are specific to the purposes of this thesis, which means that they may not be immediately applicable to other types of model or dataset.

1.4 Outline

The remainder of this thesis is organized as follows.

Chapter 2 provides a brief introduction to essential domain knowledge as well as a brief review of the existing literature. Key concepts and terminology are explained in Section 2.1. A review of related works is given in Section 2.2.

Chapter 3 describes the methodology and pivotal techniques to facilitate benchmarking in this thesis. The proposed evolutionary algorithms are defined in Section 3.1. The methods

[6] Chollet and others [2015]
[7] see Section 2.1 for detailed explanations of terminologies.
[8] The possibility of integration into Keras model is left for evaluation by the open-source community.
[9] Van Rossum and Drake [2009]
[10] `https://gitlab.com/cbhuang/tue-evo-mlp`
[11] `https://cbhuang.gitlab.io/tue-evo-mlp`
[12] `https://gitlab.com`

used to enable fair comparisons of model accuracy are described in Section 3.2.

Chapter 4 details the experiments and the analysis of outcomes. The main results from each dataset are presented in detail and interpreted. Section 4.1 describes the processes of synthesizing and preprocessing the datasets used in this thesis. Section 4.2 explains how the hyperparameters were tuned, how the experiments were run, and how the results were analyzed. Section 4.3 and 4.4 present the results obtained from synthetic and real-world datasets respectively.

Chapter 5 further examines the results in Chapter 4 from a broader perspective. The insights are then interpreted and summarized.

Chapter 6 presents the main conclusions and limitations of this thesis. Potential directions for future research are also proposed.

To keep the main text concise and focused, supplementary contents are attached in the appendix. Appendix A presents the definition of the synthetic functions used in this thesis. Appendix B reports the hyperparameter tuning results of the main experiments performed in Section 4.3 and 4.4. Supplementary experiments are presented in Appendix C.

Chapter 2

Digest of Domain Background

2.1 Preliminaries

2.1.1 Supervised Learning

In computer science, the term *learning* refers to improving some performance measure P with respect to some task T based on some set of past experience E [Mitchell, 1997]. If the learning process is performed using an iterative algorithm instead of an analytical method, then it is referred to as *machine learning*. Analytical methods such as linear regression are generally not considered machine learning because they use explicit formulae rather than algorithms. The advancement in machine learning technology has facilitated effective solutions to numerous problems that are considered too difficult to be solved with non-learning type programs [Goodfellow et al., 2016]: For instance, image classification [Krizhevsky et al., 2012], speech recognition [Hinton et al., 2012], anomaly detection [Chandola et al., 2009] and task scheduling [Whiteson and Stone, 2006], to name a few.

A device for performing a desired task is referred to as a *model*. The execution of an learning algorithm is referred to as *training*. The goal of training is to find a model in which general rules are embedded to enable the prediction of new data. In other words, the information accessible via the model should exceed simple enumeration of past experience. In practice, datasets are usually divided into two parts: *training data* and *test data*. If the training data fits the model well but the test data does not, this represents the *overfitting* phenomenon where noise is fit rather then material information [Dietterich, 1995]. Such rules embedded in overfitted models are specific to the training data but not extensible to other cases.

Supervised learning is a type of machine learning in which a *label* is associated with each set of input variables [Russell and Norvig, 2010, Goodfellow et al., 2016]. In practice, this often involve the process of finding a function or mapping rule that maps a set of input variable(s) to a corresponding set of output variables. A succinct summary of classical supervised learning methods can be found in Kotsiantis [2007]. In this thesis, input variables are referred to as *independent variables* or *samples* or simply X. Output variables are referred to as *dependent variables* or *labels* or simply Y. The value mapped by a model is called a *predicted value* or *prediction* ($\hat{Y}$). In other words, (X, Y) in the training data always come in pairs for supervised learning.

There are two basic types of supervised learning problem based on labels. If the label is numerical, then the problem is referred to as a *function-fitting problem* in this thesis[1]. The accuracy of function-fitting problems is usually measured as the coefficient of determination, namely R^2, which is analogous to linear regression. If the label is categorical, then the problem is a *classification problem.* A model used for a classification problem is also referred to as a *classifier.* Accuracy in classification is measured by the overall percentage of correctly predicted labels. Both types of problem are involved in this thesis.

Note that if the labels are known for only for a subset of input data, then the problem is referred to as *semi-supervised learning.* If none of the labels are known at all, then the problem is referred to as *unsupervised learning.* Their goals of learning and methodologies and differ considerably from supervised learning [Goodfellow et al., 2016]. Neither of these problems are addressed in this thesis.

2.1.2 Multilayer Perceptron (MLP)

An MLP is a special form of *artificial neural network* (ANN), which is also referred to as *feedforward (neural) networks* [Goodfellow et al., 2016, Chen and Burrell, 2002, Maier and Dandy, 2000]. It is basically a directed acyclic graph comprised of multiple layers of interconnected neurons with the paths of information flow within the network predefined [Goodfellow et al., 2016]. In the field of machine learning, an MLP model often acts as not only a straightforward solution but also a stepping stone to more sophisticated ones. This section provides a concise introduction to MLP, including its structure, training method and auxiliary elements.

Network Structure

The topology of MLPs consists of multiple layers of nodes as well as edges connecting nodes in adjacent layers. Within the context of AI research, the nodes are called *neurons.* In this thesis, the edges are also referred to as *links.* The layers contain an *input layer*, an *output layer*, and one or more *hidden layer(s)* in between.[2] The numbers of neurons in the input and output layers should respectively match the number of input and output variables. The number of neurons in a hidden layer can be set arbitrarily. As for edges, it is only allowed to be located between a pair of neurons located in adjacent layers. In other words, neurons in two adjacent layers and edges in between together form a *bipartite graph*[3]. If every possible edge exists, then the network is said to be *densely connected* or simply *dense.* Otherwise the MLP is *sparse.* Only dense networks are employed in this thesis.

Prediction

To compute predicted labels (also referred to as a *feedforward* step), the input variables propagate through model coefficients and transformations in the MLP network, as illustrated

[1]The term *regression problem* [Goodfellow et al., 2016] is also frequently used despite its potential confusion with linear regression.

[2]not to be confused with single layer perceptron (SLP) which has no hidden layer

[3]see `https://en.wikipedia.org/wiki/Bipartite_graph` for a quick illustration

in Figure 2.1. The *weights* are associated with links, whereas *biases* and *activation functions* are associated with neurons. Propagation begins with the input layer according to Equation 2.1, and proceeds consecutively until the output layer is reached.

$$x_j^{\text{out}} = F(\sum_i (w_{ij} x_i^{\text{in}} + b_j) \tag{2.1}$$

where

$$\begin{aligned} w_{ij} &= \text{weight associated with the link pointing from neuron } i \text{ to neuron } j \\ x_i^{\text{in}} &= \text{input value associated with neuron } i \\ b_j &= \text{bias associated with neuron } j \\ F &= \text{activation function} \\ x_j^{\text{out}} &= \text{output of neuron } j \end{aligned}$$

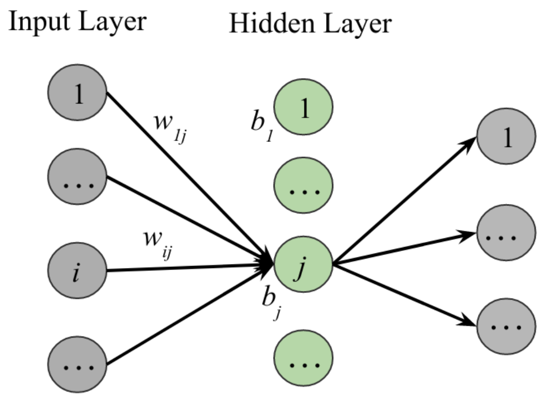


Figure 2.1: Value propagation within MLP network (i.e., feedforward step). Edges not connected to neuron j are omitted for clarity.

There are no strict rules for the selection of an activation function. A practical collection and comparison can be found in Nwankpa et al. [2018], including the ones later discussed in this section. In this thesis, the rectified linear unit (ReLU, Equation 2.2) is chosen because its effectiveness has been empirically demonstrated and it is computationally light.

$$\text{ReLU}(x) = \begin{cases} 0, & x \leq 0 \\ x, & x > 0 \end{cases} \tag{2.2}$$

The interpretation of output values varies according to the problem type. For function-fitting problems, the output layer comprises a number of neurons corresponding to the dimension of output variable[4] and the output value is taken as the predicted value directly.[5]

[4]For scalar-valued functions, the output layer consists of a single neuron.

[5]which may be a standardized or normalized value with respect to data preprocessing

For classification problems, the output layer comprises a number of neurons corresponding to the number of classes, such that each neuron represents a predicted class. The value of an output neuron is interpreted as a measure of the probability that a predicted class is the class represented by the neuron. A typical practice is to transform the output values using the *softmax* function (Equation 2.3) to ensure that the derived probabilities sum to unity. This transformation is not necessarily well-calibrated in a strict sense [Guo et al., 2017]; however, empirically it works well for many cases. In this thesis, a softmax activation is always appended at the output layer for classification problems.

$$\mathrm{softmax}(x_i) = \frac{\exp(x_i)}{\sum_i \exp(x_i)}, \text{ where } x_i = \text{ output value of the } i^{\text{th}} \text{ neuron} \tag{2.3}$$

Evaluation

Evaluation follows the feedforward step. The goodness-of-fit of the model is evaluated using a *loss function*, wherein a smaller loss value indicates better model fit. Typically, the mean square error (MSE) function is used for function-fitting problems, and the *sparse categorical cross-entropy*[6] function is used for classification problems. These were also chosen in this thesis.

As a common practice, data not previously used in the feedforward step were often used in model evaluation to prevent overfitting. The data used in this step is referred to as *validation data*. It represents a subset of the training dataset.[7] Validation data is not to be confused with the test data, which remains untouched throughout the entire training process.

Optimization

Obviously, the initialization of random model coefficients is unlikely to produce a good fit. Improving the fit requires an optimization algorithm to adjust the coefficients in an appropriate manner. This coefficient adjustment is referred to as the optimization step. The process of implementing changes to model coefficients based on the results of a loss function is referred to as *backpropagation* [Rumelhart et al., 1986, Cun, 1988], which can be seen as a form of *gradient descent* optimization (illustrated in LeCun et al. [2015]). During backpropagation, the magnitude of corrections for coefficients in the previous layer is determined by the gradient of the error function as well as the error value in the current layer. The correction process begins with the output layer and proceeds backward until the input layer is reached, thus completing a full cycle of input data training. The next round of data can then be loaded into the model, which ideally continues the process of improving itself until a local minimum is reached.

The model also includes non-trainable parameters, such as the number of neurons in each hidden layer, the activation function(s) and the loss function. These non-trainable parameters (referred to as *hyperparameters*) are either set before training begins or computed automatically without interruption during training. The trainable coefficients are referred to as *model parameters* in the narrowest sense. Nonetheless, depending on the context,

[6]refer to `https://en.wikipedia.org/wiki/Cross_entropy` for detailed formulation

[7]A typical choice is to take out 10% to 20% of the training dataset for validation use only.

hyperparameters are occasionally referred to as model parameters. The hyperparameters are optimized via a process referred to as *hyperparameter tuning*, which is detailed in Section 4.2.1.

Further Details

In the following, a selection of other elements crucial to the training process are described:

The term *epoch* refers to a complete pass of a dataset during training. Due to memory limitations and/or concerns pertaining to computational efficiency, data is often input into a model in small batches.[8] Each batch proceeds independently through the functional steps (feedforward, evaluation, and optimization). This approach makes it possible to complete multiple backpropagation procedures without increasing the I/O cost and potentially generating an optimal solution more quickly. Unfortunately, smaller batches often deviate from the sample distribution of the overall dataset, which can lead to unstable results. In practice, it is generally necessary to find a balance between speed and stability based on empirical analysis.[9]. In this thesis (focusing on devices with limited computational resources), we opted for a smaller batch size except in situations where instability becomes a problem.

The *batch normalization* (BN) method proposed by Ioffe and Szegedy [2015] is one approach to overcoming the problem of instability when using small-sized batches. This method basically centers and scales the data based on analysis performed during training. BN has proven effective when dealing with image data. In the current study, the decision of whether to use BN was determined empirically.

The *dropout* method proposed by Srinivas and Babu [2015] is commonly used to detect early convergence, i.e., converging too quickly to a less-than-ideal local optimum from which there is no escape. A dropout layer keeps a portion of the model coefficients away from changes due to backpropagation. Thus, the effect can be viewed as the addition of random noise to the gradient descent path in the parameter space. In the current study, whether to use this method is contingent on empirical results.

Callback function (i.e., *callback*) is a widely-adapted technique for implementing algorithmic operations before or after a complete training cycle. The Keras library offers a fully customizable framework as well as helpful built-in callbacks. In the current study, we considered the following two callbacks to enhance computational efficiency:

- *ReduceLRonPlateau* reduces the learning rate (LR) after the loss function ceases to facilitate improvements for a certain number of rounds (=”*patience*”). When used properly, this callback can help to improve optimization and prevent crude results caused by an initial learning rate that is too high.

- *EarlyStopping* stops training immediately after the loss function ceases to facilitate improvements for a certain number of rounds. This can save computational resources and prevent overfitting as long as a reasonable patience threshold is selected. In the

[8]It is also known empirically that stochasticity added to data distribution due to batch selection can help optimization, compared to the scenario where all data is trained at once.

[9]The value of the loss function for an epoch shown by Keras is the averaged value of all batches.

current study, we opted not to use this function because using a fixed number of epochs makes it easier to objectively compare training algorithms.

2.1.3 Evolutionary Algorithms

The history of evolutionary algorithms can be traced back to the *genetic algorithm* (GA) developed since the 1950s[10] and was made popular by Holland [1992] (first published in 1975). The original idea of GA is mimicking the process of genetic mutation by *evolving* (otherwise referred to as *perturbing*) a portion of data into another value. In the simplest version, the data is selected at random for mutation; however, rule-based selection is also possible. This approach has been used with success in problems related to natural evolution and immunology [McCall, 2005], as well as climate simulation [Stanislawska et al., 2012], financial portfolio optimization [Sefiane and Benbouziane, 2012], and task scheduling [Guillaume et al., 2007], to name a few.

Evolutionary algorithm can be viewed as a superset of GA that develops the mutation methodology in numerous fashions, where units involved in mutation can interact with each other to improve the model. A comprehensive review on classical evolutionary algorithms can be found in Xin Yao [1999]. In this thesis, we focus exclusively on its application to the training procedure of ANNs. A key progress in this domain was made by [Stanley and Miikkulainen, 2002] where the topology (i.e., connectivity) of a network model was evolved.[11] The idea of topological evolution was later incorporated into reinforcement learning [Whiteson and Stone, 2006] as well as backpropagation training on ANNs [Miconi, 2016]. These progress inspired the proposal of SET [Mocanu et al., 2018], in which topological evolution was performed in a rule-based manner during the training procedure of ANNs which are large in size and sparse in connectivity. The characteristics of SET and the aforementioned evolutionary algorithms will be discussed in more detail in Section 2.2.

In the current study, the evolutionary algorithms are implemented in the form of callbacks. Note however that a self-coded multi-stage training framework was adopted instead of the standard Keras callback, as this allows evolution to proceed through multiple epochs. Timing of evolution can be tracked more easily by implementing evolution callbacks as *stage*-level callbacks rather than mixing them up with the built-in epoch-level or batch-level Keras callbacks.

2.2 Literature Review

AI research is proceeding in two different directions. One approach focuses on sophisticated algorithms and/or complex model structures to improve the state-of-the-art models, which require considerable computational resources. The other approach (model compression) focuses on reducing the number of model parameters to accommodate devices with limited computational power or battery life.

Model compression can be traced back to Buciluă et al. [2006], who trained a compressed model using the output of a more complex but relatively shallow model. The process of trans-

[10]Refer to `https://en.wikipedia.org/wiki/Genetic_algorithm` for detailed historical survey.
[11]i.e., deleting and reinserting links in a network, thereby altering the model state

ferring information from a large model to a smaller one is referred to as ”knowledge transfer” or ”knowledge distillation (KD)”. Subsequently, Ba and Caruana [2014] demonstrated the compression of deep wide networks into shallow ones. Hinton et al. [2015] managed to compress an ensemble of teacher networks into a student network of similar depth. Romero et al. [2015] proposed the *FitNet* scheme, in which a shallow wide network is compressed into a deep narrow one. In FitNet, the student mimics the full feature map of the teacher. That scheme proved highly effective in maintaining a good compression ratio without sacrificing the accuracy of the original model, as verified using a number of benchmark datasets. In the same time, approaches other than FitNet have also developed rapidly. For instance, Chen et al. [2016] proposed using ”function-preserving transformations” to accelerate the process of knowledge transfer. Korattikara et al. [2015] enables knowledge transfer between a Monte-Carlo type source and a deep neural network (DNN) student via online training. Luo et al. [2016] wisely used neurons from higher hidden layers for the representation of knowledge. Among the methodologies being developed, the attention-based learning methodology proposed by Hinton et al. [2015] has attracted the most attention, due to its intuitive premise that useful knowledge mostly likely resides in ”active” regions of the model. Naturally, researchers have sought to incorporate the fitness of regions between the teacher and student models into the loss function to ensure that the student model can learn from the most informative regions of the teacher model. Cheng et al. [2018] posited that the attention-based approach could be used to improve the efficiency of KD methods. This assertion was soon verified in numerous subsequent works based on image classification. Zagoruyko and Komodakis [2017] incorporated both activation-based as well as gradient-based attention definitions to transfer the result of activation from a teacher model to a student model. That approach permits relaxation of the assumptions on which FitNet depended and eliminates the need to increase the depth of the student network. Wu et al. [2019] incorporated a multi-teacher single-student learning scheme to enhance model accuracy. Ruiz et al. [2020] built a comprehensive framework to facilitate application of the KD method to a multiple-angle visual identification scheme using attention patterns from intermediate layers.

Reflection upon the existing literature prompts for the following question: *Why not train a small model from the beginning?* In practice, however, small models are exceptionally hard to train. The resulting models often converge too early. They are easily trapped in local optima, even with a good setting of hyperparameters.[12] Methodological studies addressing this problem are surprisingly scarce.

The sparse evolutionary training (SET) algorithm proposed by Mocanu et al. [2018] stands out as for its ability to continuously probe for better spots in the model parameter space. SET is a multi-stage training algorithm executed at the beginning of a round of plain training. It basically zeros out a fraction of connection weights bearing the smallest absolute values and then adds back the same number of properly-initialized connections. It has been claimed that SET outperforms plain training in multilayer perceptron (MLP) networks and is roughly on par with convolutional neural networks (CNNs) while using only a small fraction of the model parameters. The building blocks of SET can be traced back to the NeuroEvolution of Augmenting Topologies (NEAT) algorithm proposed by Stanley and Miikkulainen [2002], which extends the GA family by evolving model topology to achieve optimization. Later in Whiteson and Stone [2006], NEAT worked in conjunction with a reinforcement learning

[12]demonstrated in Section 4.3.1

scheme to enable *evolutionary function approximation*[13] for learning time-series functions. Miconi [2016] further made it possible for NEAT to perform gradient descent optimization on recurrent neural networks (RNNs). SET is appreciably motivated by these studies.

SET provides an alternative approach to algorithm design. First, the changes that SET makes to model parameters are generally not transient.[14] Second, SET perturbs a locally-optimized set of model parameters obtained in the previous stage of training, which is synonymous to "*combatting early convergence*" or "*breaking out of a local optimum*". Introducing a training algorithm with these characteristics greatly increases the likelihood of reaching the global optimum (or at least an improved local optimum).

The objective of this thesis was to develop multi-stage algorithms capable of perturbing model parameters of a previously-trained model that usually reside on a local optimum instead of a global optimum. We hoped that these algorithms could help to overcome the problem of early convergence in the training of small models. Note that our approach to improving computational efficiency differs from the mainstream approach based on model compression. Model compression algorithms strive to enhance model accuracy starting from a complex and accurate model, whereas our approach to model accuracy focuses solely on efficient probing in the model parameter space.

It is worth mentioning that the idea of weight manipulation has existed for years. A representative example is the DropConnect method proposed by Wan et al. [2013], which temporarily "masks out" a subset of weights chosen randomly during an epoch.[15] This is equivalent to introducing a noise source into the training process. It is a generalization of the well-known Dropout method [Hinton et al., 2012], which temporarily masks out the activation functions of a subset of neurons. Based on the success of its application to the MNIST dataset, Mobiny et al. [2019] subsequently used it to improve the performance of Baynesian deep networks. However, close inspection revealed that the DropConnect approach to weight manipulation would not be suitable for this thesis. First, DropConnect is not used to find a local optimum. Second, it does not reinitialize a new model parameter state when starting a new round of training, which means that it is unable to break out of a local optimum by itself. Third, the size of perturbation or evolution[16] is not easily tuned (and maybe not even possible), thereby making it impossible to observe progressive changes in its effectiveness.

The direction of this thesis was guided by existing literature. First, there is a clear need for fundamental research in this area. This prompted us to begin with models which are simple and commonly employed. This also prompted us to use simple synthetic datasets for which the minimal optimal networks can be derived analytically. Finally, we constructed and tested a number of redesigned algorithms derived from SET. This approach made it possible to formulate algorithms capable of probing the model parameter space efficiently. The insights attained serve as the basis for the development of models with more complex structures and data with more abundant features.

[13]representations of function approximators are selected automatically [Whiteson and Stone, 2006]

[14]affecting multiple epochs or even the entire training

[15]A mask is generated to pick out a subset of weights, where the masked weights are set to 0. The values of weights are restored to their original values when the mask is removed.

[16]the magnitude of change in the value of weights drawn to be evolved in this thesis

Chapter 3

Proposed Methods

3.1 Algorithms

Five algorithms are proposed in this thesis: Four of them are novel redesigned versions of SET, each of which retains the major characteristics of evolutionary algorithms as described in Section 2.1.3. The number of four comes from two options of the unit of evolution (link or neuron) multiplied by 2 options of unit selectivity (least-important or random). The other one left is a straightforward incremental training strategy. In the following, the premises underlying the design of the proposed algorithms are outlined:

1. This thesis focused exclusively on *dense* networks.
2. Only weights undergo evolution. None of the proposed algorithms evolves bias values.
3. *Timing of evolution was changed.* In all of the algorithms, evolution is executed after the model is fully trained, instead of after every epoch as for Mocanu et al. [2018]. This is meant to ensure that the next training stage starts from a reasonably-optimized location in the model parameter space, rather than from an under-optimized location. Note that the term *training stage* (or simply a *stage* hereafter) refers to the collection of epochs between two evolutionary actions. We adopted this approach to directly combat the situation of being stuck in local optima, as often occurs in plain training.

3.1.1 Link Evolution (LE)

The LE algorithm (Algorithm 1) resembles the SET algorithm proposed by Mocanu et al. [2018]. In addition to the changes described above, a number of adjustments were made to the original algorithm aimed at preventing unnecessary training of a model that is ill-evolved or not evolved at all. The adjustments and underlying intuitions are outlined in the following:

1. *Reasonable scaling of evolution.* When applied to the dense networks in this thesis, the remove-and-reconnect evolution method of SET would act on the same set of links,[1]

[1]In SET, the number of links removed always equals to that of reconnected in an evolution step. Since the network is dense, the links involved in an evolution step will not vary.

Algorithm 1: Link Evolution

```
   Input: $\zeta$: evolution probability, $\delta$: evolution scale
1  initialize model
2  initialize multi-stage history storage
3  for each stage do
4      for each epoch do
5          perform regular training
6      end
       /* discard worse outcomes                                    */
7      if the loss is greater than the previous best result then
8          restore the best model
9      end
       /* evolution                                                 */
10     if not the last stage then
11         for each layer in the model do
12             pick a smallest $\zeta$ fraction (rounded up) from the smallest positive weights
13             pick a largest $\zeta$ fraction (rounded up) from the largest negative weights
14             add $\pm(\delta + |N(0, \delta)|)$ to the picked weights (random sign)
15         end
16     end
17     update multi-stage history storage
18 end
```

which is not the situation originally being targeted by SET. This is equivalent to simply re-initializing those selected links. However, this would have opened up the possibility of the reinitialized coefficient deviating too little from the original value (ineffective evolution) or deviating too much (instability). Therefore, an improved strategy is proposed to gain better control over the scale of evolution (Equation 3.1).

2. *Minimal number of evolved links.* The original SET algorithm does not address the special case in which no links are drawn for evolution. Note that this situation occurs frequently when dealing with networks containing few neurons and a low probability of evolution (ζ), such as 8 neurons in a single hidden layer with $\zeta = .1$. Thus, the following rule is enforced: in the event that no neurons or links are drawn for evolution, then one of them should be drawn at random.

The design of Equation 3.1 is explained as follows: The magnitude of change on a weight undergoing evolution is defined as the minimum scale of evolution ($\delta > 0$) plus a random number drawn from $|N(0, \delta)|$. This ensures that a weight does not stall in the vicinity of its original value as long as the value of δ is sufficiently large. In the same time, the magnitude of change is proportional to δ so it would not go unreasonably large[2], thereby achieving improved control over the scale of evolution. The scaling hyperparameter δ is optimized empirically in the hyperparameter tuning procedure.

[2]The expected value of the magnitude of change is $\left(1 + \sqrt{\frac{2}{\pi}}\right)\delta$, which can be easily derived from a standard normal distribution.

$$\Delta \sim \pm(\delta + |N(0,\delta)|) \tag{3.1}$$

where

$$\begin{aligned}
\Delta &= \text{quantity adding to the evolving weight; the sign is chosen} \\
&\quad\ \text{at random} \\
\delta &= \text{std\_evo\_scale (hyperparameter name)} \in \mathbb{R}^+ \\
N(0,\delta) &= \text{normal distribution with zero mean and standard deviation } \delta
\end{aligned}$$

3.1.2 Random Link Evolution (RLE)

Algorithm 2: Random Link Evolution

```
   Input: $\zeta$: evolution probability, $\delta$: evolution scale
 1 initialize model
 2 initialize multi-stage history storage
 3 for each stage do
 4     for each epoch do
 5         perform regular training
 6     end
       /* discard worse outcomes */
 7     if the loss is greater than the previous best result then
 8         restore the best model
 9     end
       /* evolution */
10     if not the last stage then
11         for each layer in the model do
12             for each link in the layer do
13                 pick with probability $\zeta$
14             end
15             if no link was picked then
16                 pick a link randomly
17             end
18             add $\pm(\delta + |N(0,\delta)|)$ to the picked weights (random sign)
19         end
20     end
21     update multi-stage history storage
22 end
```

The RLE algorithm (Algorithm 2) is a redesigned version LE in which the links to be evolved are randomly selected. Note that in addition to the changes described at the beginning of this chapter, this algorithm includes a number of specific features.

1. *Implementation of link selection with fraction ζ.* In the original scheme, each neuron

has probability ζ of evolving, which can lead to a rounding problem when dealing with a small network. Furthermore, the original scheme draws a fixed fraction of ζ of the links to be evolved; however, this does not necessarily work as expected for small networks. For instance, $\zeta = .2$ and $\zeta = .3$ would be indistinguishable in a layer of 8 links, due to the fact that 2 links would be selected in both cases under the assumption of regular rounding rules. Neurons must be evolved individually with probability ζ in order to overcome the rounding problem. Overall, the expected total number of evolved links should be consistent with the original definition.

2. *At least one link evolves.* The use of ζ introduces the possibility that no link is selected for evolution. In that event, one link must be selected at random to avoid wasting an evolution stage.

3. *Negative and positive numbers are not differentiated.* Negative and positive integers are treated equally, due to the fact that evolution probability ζ is the same for all of the weights.

3.1.3 Neuron Evolution (NE)

The NE algorithm (Algorithm 3) is a redesigned version of LE in which so-called ”unimportant” links also undergo evolution and the selection criterion is neuron-based. In brief, NE differs from LE in the following ways:

1. *Selection of links for evolution.* All weights on links connected to a neuron[3] with its p-sum in the smallest ζ fraction of its layer are evolved.

2. *At least one neuron in every hidden layer is evolved.* The number of evolved neurons is rounded up.

3. The selection process does not differentiate between the smallest positive and largest negative values, because there is no negative p-sum.

Specifically, all of the incoming and outgoing links attached to the selected neurons undergo evolution. This approach was inspired by work in attention-based learning, in which the importance of a neuron is proxied either by the maximum absolute value of all the incoming coefficients or the p-sum formula in Equation 3.2 [Zagoruyko and Komodakis, 2017, Simonyan et al., 2014]. In other words, every incoming weight would contribute to determining the importance of a neuron. Note that $p > 1$ emphasizes links with the greatest $|\text{weight}|$ of a neuron, while $p \to \infty$ produces a result identical to rankings based only on the link with the maximum $|\text{weight}|$.

$$\text{p-sum}_{\text{original}}(i) = \sum_j |w_{ji}^p| \tag{3.2}$$

where

[3]Both incoming and outgoing links are included.

$$
\begin{aligned}
j &= \text{neuron in the previous layer} \\
i &= \text{neuron in the current layer} \\
w_{ji} &= \text{weight on the link from j to i} \\
p &= \text{positive exponent (free parameter)}
\end{aligned}
$$

Algorithm 3: Neuron Evolution

```
Input: $\zeta$: evolution probability, $\delta$: evolution scale, $p$: exponent of p-sum
1  initialize model
2  initialize multi-stage history storage
3  for each stage do
4      for each epoch do
5          perform regular training
6      end
       /* discard worse outcomes */
7      if the loss is greater than the previous best result then
8          restore the best model
9      end
       /* evolution */
10     if not the last stage then
11         for each layer in the model do
12             compute p-sum for each neuron
13             pick a smallest $\zeta$ fraction of neurons (rounded up) based on p-sum
14             add $\pm(\delta + |N(0, \delta)|)$ to the weights connected to the picked neurons
                 (random sign)
15         end
16     end
17     update multi-stage history storage
18 end
```

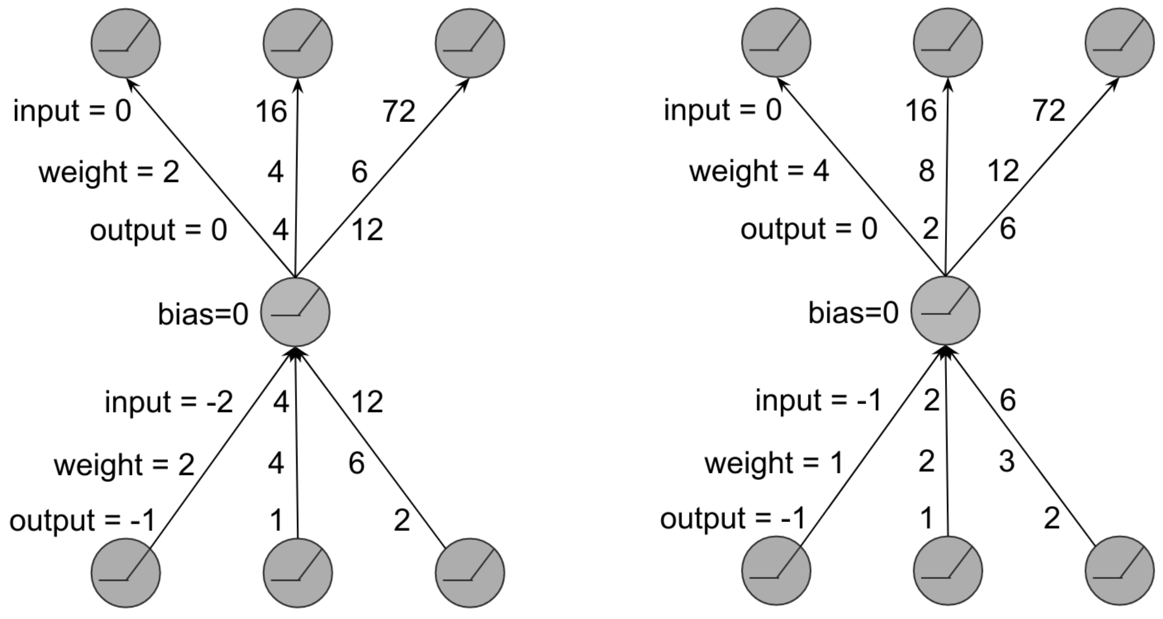
For the sake of simplicity, only p-sum with $p = 2$ was selected as a representative case in this thesis. Equation 3.2 was also modified for the MLP network, as the original equation was designed for CNNs. The modified p-sum for hidden layers in MLPs is computed using Equation 3.3. In brief, the incoming and outgoing weights both contribute to the importance of a neuron, and taking the square of both sums guards against arbitrary scaling. The reason why the geometric mean of both sides was taken is as follows: If we assume that all of the incoming weights are divided by 2, and all of the outgoing weights are multiplied by 2, then no change will be observed in the output from the outgoing layer.[4] In other words, there is freedom to scale the weights arbitrarily between the incoming and outgoing layers. This means that the importance of a neuron is not necessarily represented accurately by weights from a single side. This problem is accounted for by taking the geometric mean of the p-sum on both sides. An comprehensive illustration is given in Figure 3.1.

[4] assuming zero bias

$$\text{p-sum}(i) = \sqrt{\sum_j |w_{ji}^p| \sum_k |w_{ik}^p|} \tag{3.3}$$

where

$$\begin{aligned} j &= \text{neuron in the previous layer} \\ i &= \text{neuron in the current hidden layer} \\ k &= \text{neuron in the next layer} \\ w_{ji} &= \text{incoming weight from neuron j to i} \\ p &= \text{positive exponent (free parameter)} \end{aligned}$$



(a) Original Weights

(b) Rescaled Weights

Figure 3.1: Illustraion of adjusted p-sum: (a) A selected neuron and the links attached to it are shown. The original weights, output values from the incoming layer are given. The input values received by the neuron, the output values from the neuron, and the input values received by the outgoing layer are subsequently derived. The unadjusted p-sum ($p = 2$) over all incoming and outgoing weights is 112. (b) The incoming weights are divided by 2 while the outgoing weights are multiplied by the same number. The derived figures show that the values received by the outgoing layer is completely unaffected, implying identical importance of the selected neuron after rescaling; however, the unadjusted p-sum skyrockets to 238. This dilemma is resolved by the geometric mean adjustment introduced in Equation 3.3.

3.1.4 Random Neuron Evolution (RNE)

Algorithm 4: Random Neuron Evolution

```
   Input: $\zeta$: evolution probability, $\delta$: evolution scale, $p$: exponent of p-sum
1  initialize model
2  initialize multi-stage history storage
3  for each stage do
4      for each epoch do
5          perform regular training
6      end
       /* discard worse outcomes */
7      if the loss is greater than the previous best result then
8          restore the best model
9      end
       /* evolution */
10     if not the last stage then
11         for each layer in the model do
12             for each neuron in the layer do
13                 pick with probability $\zeta$
14             end
15             if no link was picked then
16                 pick a neuron randomly
17             end
18             add $\pm(\delta + |N(0, \delta)|)$ to the weights connected to the picked neurons
                 (random sign)
19         end
20     end
21     update multi-stage history storage
22 end
```

The RNE algorithm (Algorithm 3) is the NE algorithm with random selection of the neurons to be evolved. The RNE algorithm deviates from NE as follows:

1. P-sum is not required because the neurons are selected at random.
2. Each neuron has evolution probability ζ. As with RLE, neurons instead of links are randomly selected for evolution.
3. When the drawing prodecure with ζ fails to select a neuron to be evolved in a layer, a random neuron in that layer will be drawn at random.

3.1.5 Neuron Increment (NI)

Algorithm 5: Neuron Increment

```
1  initialize model1 (half-sized)
2  initialize multi-stage history storage
   /* Stage 1                                                    */
3  for each epoch do
4  |  perform regular training on model1
5  end
6  update multi-stage history storage
   /* Stage 2                                                    */
7  initialize model2 (full-sized)
8  copy the coefficients of model1 to the first half of neurons and associated links of
     model2
9  for each epoch do
10 |  perform regular training on model2
11 end
12 update multi-stage history storage
```

A key aspect of evolutionary algorithms is perturbing the model parameter space which was previously trained. However, the author noticed that this does not necessarily have to be done via evolution. A simpler approach would be training a fraction of the neurons at first and subsequently adding back the untrained neurons until the full model size is reached. The partial model obtained in the first stage could be regarded as a full model (with the untrained coefficients being zero), which is perturbed by randomly initializing the untrained links. The NI algorithm (Algorithm 5) is based on precisely this strategy. A two-stage NI algorithm is constructed to enhance representativeness and simplicity.*If a multi-stage NI can actually work, then it is possible that a two-stage NI can already demonstrate signs of effectiveness.* Half of the neurons in each hidden layer are trained in the first stage. The coefficients are subsequently transferred to a properly-initialized full-sized model for the last stage of training. The algorithm is designed to function as a window by which to assess the effectiveness of the proposed evolution strategies. Unlike the proposed evolutionary algorithms, the incremental training procedure is neither random- or importance-based. Thus, in the event that NI performs equally effective with an evolutionary algorithm, then the effectiveness of that evolution scheme should be questioned.

Note that *NI is not considered an redesigned version of SET* in this thesis because NI does not involve either importance- or random-based scheme of evolution as employed in LE, RLE, NE or RNE. NI does partially resemble the topological increment strategy of the NEAT algorithm [Stanley and Miikkulainen, 2002]. Topological evolution, however, is not the main focus of this thesis.

3.2 Comparison of Results

3.2.1 Accuracy

Benchmarking in this thesis centers around whether a proposed algorithm outperforms plain training. If we take LE vs Plain as an example, then the null hypothesis roughly states that "LE does not enhance accuracy compared to plain training".

The rule of thumb used to compare the efficiency of the proposed algorithms can be defined as follows: *comparing the same quantile for the total epochs spent.* This idea is illustrated in the following example. Assume that each run of the LE algorithm involves 20 stages of 100 epochs per stage and each run of plain training involves a single stage of 100 epochs. Training is performed as follows: LE (100 runs) and plain training (2,000 runs). Thus, the two algorithms run an equal number of stages and epochs. Based on a simple comparison of the median (or average) accuracy of the two algorithms, one would *not* be able to claim that one is better than the other, due to the fact that a single LE run imposes the computational load of 20 runs of plain training. The LE algorithm always retains the best stage during training; therefore, the LE results represent the top $\frac{1}{20}$ or the top 5th percentile of all stages run. Thus, the median of the LE experiments should be "the median of the top 5 percent of all results", i.e., the 97.5th percentile of all stages run.

Overall, the median accuracy of LE, RLE, NE, and RNE (20 stages each) should be compared to the 97.5th percentile of plain training, provided the total number of stages and epochs per stage are equal.[5] The null and alternative hypotheses are formalized in Equation 3.4 and 3.5.

$$\mathbf{H_0} \text{ (null hypothesis): } \operatorname{med}(\text{alg}) = Q_{97.5}(\text{plain}) \tag{3.4}$$

$$\mathbf{H_1} \text{ (alternative hypothesis): } \operatorname{med}(\text{alg}) > Q_{97.5}(\text{plain}) \tag{3.5}$$

where

$$\begin{aligned} \text{med} &= \text{median} \\ Q_x &= \text{the } x^{\text{th}} \text{ percentile} \\ \text{alg} &\in \{\text{LE, RLE, NE, RNE}\} \end{aligned}$$

Note however that 2-stage NI should be compared to the same quantile of plain training results. The underlying intuition is that one run of NI yields only one set of fully-trained model coefficients (i.e., the last round), despite completing two stages. In other words, the first stage of NI is only a partially-trained state, which could not be expected to compete with the accuracy of a fully-trained full-sized network. Therefore, a run of NI should be interpreted as a slightly modified run of plain training. The objective of this thesis was to attain the most accurate models; therefore, the 95th percentile instead of the median is selected for comparison.[6] Equation 3.6 and 3.7 present formal statements of the null and alternative hypotheses.

[5]The stage count listed is exactly the default number of stages used in this thesis. Hypotheses suitable for arbitrary number of stages can be easily derived with the same reasoning.

[6]The 97.5th percentile can also be used without altering the conclusion.

$$\mathbf{H_0} \text{ (null hypothesis): } Q_{95}(\text{NI}) = Q_{95}(\text{plain}) \tag{3.6}$$
$$\mathbf{H_1} \text{ (alternative hypothesis): } Q_{95}(\text{NI}) > Q_{95}(\text{plain}) \tag{3.7}$$

where the symbols are the same as the previous hypotheses.

The model accuracy of all proposed algorithms was evaluated using previously-untouched test data, in accordance with conventional practices. Note that training results are often unacceptably poor when dealing with small networks; therefore, the median is used rather than the mean (or truncated mean) for comparison to avoid this type of noise. Note also that the standard deviation of accuracy is considered a measure of algorithm stability. Minimum values and low quantiles of accuracy were also used to assess stability.

3.2.2 Model Training History

Training histories were recorded in the hope of gaining insight into the process of evolution. We plotted and analyzed the degree of accuracy based on validation data (i.e., *validation accuracy*). The resulting plots can be compared qualitatively via overlapping because all of the training algorithms used the same number of epochs per stage.

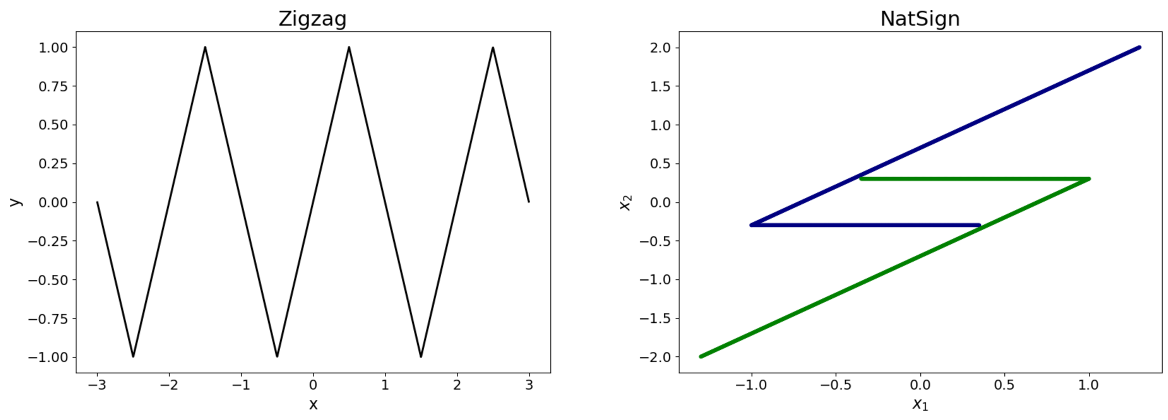
Chapter 4

Results

Detailed experiment steps along with the main results were presented in this chapter. As for the data utilized, two synthetic datasets (ZigZag and NatSign) and one real-world dataset (downsampled MNIST) were constructed, preprocessed and visualized. Experiment protocols were elaborated including hyperparameter tuning and interpretation of results. Three types of result were reported selectively: (1) benchmark of the five proposed algorithms versus plain training (2) visualization of training history and (3) visualization of goodness-of-fit (only for ZigZag). Pivotal findings and implications of the results were also discussed. Results of hyperparameter tuning for the experiments in this chapter can be found in Appendix B.

4.1 Data

4.1.1 Synthetic Data



(a) ZigZag Dataset

(b) NatSign Dataset

Figure 4.1: Synthetic datasets where $n = 2,000$ samples were randomly selected from a uniform distribution for each dataset: (a) ZigZag dataset sampled along X domain. (b) NatSign dataset is sampled along directions of its constituent lines (blue and green respectively represent $y = 0$ and $y = 1$). Detailed definitions of the datasets are given in Appendix A.

When we consider which data are suitable for assessing the effectiveness of algorithms on small networks, we must consider the following question: What defines a network as *small*? The answer depends entirely on the nature of the data.[1] For example, a dataset bearing a perfect linear relationship between the input variables and sole output variable would render any non-empty hidden layer redundant. To construct a perfectly accurate model for this dataset, it would be enough to connect the input layer directly to the output layer (consisting of one neuron).[2] This example provides the following crucial insight: If the minimal network required to reach the maximum possible accuracy is known *a priori*, then the degree of structural redundancy or deficiency in the network can be determined by comparison, giving the researcher all of the information required to make an informed decision of model size. Thus, the experiments in this study began with synthetic datasets, from which the *minimal optimal models* can be derived analytically. This experiment design prevents running experiments on highly-redundant models unwittingly.

The synthetic datasets used in the current study are presented in Figure 4.1. The *ZigZag* dataset poses a univariate function-fitting problem. The *NatSign* dataset[3] poses a classification problem with two classes (blue and green) and two input variables. 2,000 samples are drawn for training, with an 80-20 train-test split, and 20% of the training data was reserved for validation during training. The formulae and the minimal model sizes are detailed in Appendix A.

During preliminary studies, a spike-shaped synthetic dataset called *Peak-1D* was also employed (Appendix A.3). It consisted of only 3 segments, which stemmed from the intention to construct an dataset as simple as possible. Unfortunately, it was too easy to achieve $R^2 \approx 1.0$ even using plain training, thereby being unsuitable for benchmarking. The results inspired the design of ZigZag that was similar to Peak-1D while being more complex and much more difficult to fit perfectly for plain training.

4.1.2 Real-world Data (MNIST)

This thesis used a downsampled subset of the MNIST dataset [LeCun and Cortes, 2010]. MNIST is a collection of 28×28 greyscale images of human-written digits and their correct labels (0, 1, ..., 9). It includes 60k train images and 10k test images. A random sample from each label is presented in Figure 4.2. Note that in the current study, the original dataset is downsampled to 7×7 based on the average value of each 2×2 block. Furthermore, only 10% of the original dataset was selected at random. This resulted in 6k records of downsampled training data and 1k records of downsampled test samples. 20% of the training data was used for validation during training.

The motivation in devising this setup was to reduce the abundance of information and thereby make it more difficult to escape from local minima of the loss function. The reason for downgrading the data is based on the fact that training based on too few data generally reduces the quality of the resulting model. In other words, training with less information can lead to situations in which the model is susceptible to becoming trapped in local optima, which is usually worse than what would have been achieved if a more comprehensive dataset

[1]Assume that the activation function is fixed.
[2]The regression coefficients are the weights of links, and the intercept is the bias of the output neuron.
[3]The shape resembles the *natural sign* in musical scores.

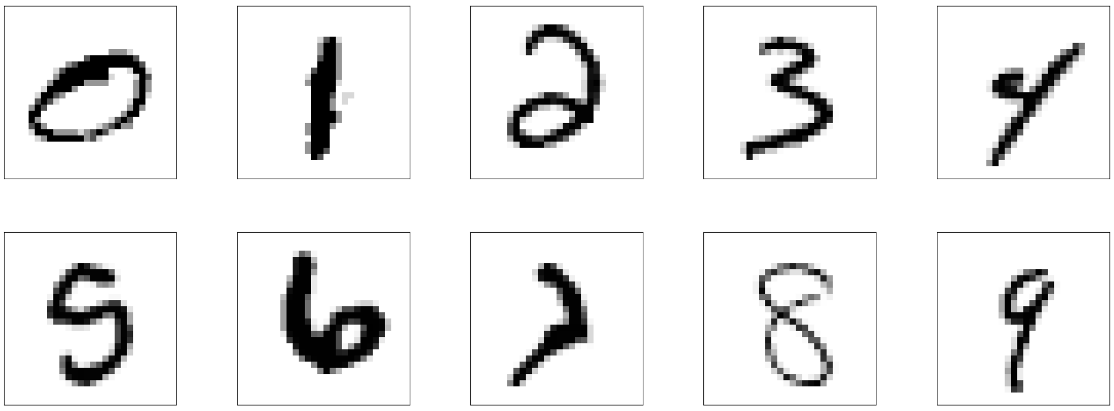


Figure 4.2: A random sample of each digit from the MNIST dataset.

were used. Note that during preliminary runs (unreported), experiments run using the full MNIST dataset achieved roughly the same accuracy for all proposed algorithms. By contrast, the formal runs utilized only 2.5% of the degree of freedom[4] from the original MNIST, and greater differences in the resulting accuracy among algorithms were indeed obtained.

4.2 Experiment Setup

4.2.1 Hyperparameter Tuning

Hyperparameter tuning is performed prior to each formal run to ensure that formal results are obtained under a reasonably optimized setup. Simple *grid search* strategy was adopted to facilitate programming and the sharing of common settings among the compared algorithms. The tuning process is separated into two rounds in order to manage the large number of combinations in the hyperparameter space. The plain (non-evolutionary) hyperparameters are first tuned using a reasonable set of candidate values. The second round involves tuning evolutionary parameters only. Table 4.1 lists the hyperparameters being tuned in each round. Such logical division of hyperparameters makes it possible to complete the task of tuning all of the proposed algorithms within a couple of days using an ordinary laptop.[5]

There are also fixed hyperparemters that do not participate in hyperparameter tuning. These are listed as follows.

- *Connectivity.* All the layers are fully connected.
- *Activation function.* ReLU (Equation 2.2) is embedded in the hidden layer(s). For the output layer, softmax (Equation 2.3) is embedded in the hidden layer for the classification problems, whereas no activation function is embedded in the function-fitting problems.

[4]10% of data multiplied by 25% of image size

[5]for a specific network size

Round	Category	Hyperparameter	Definition
1	plain	batch_size	batch size of input data
		batch_normalization	perform batch normalization or not
		dropout_rate	portion of neuron to perform dropout
		init_lr	initial learning rate
		epochs	number of epochs; 0=variable epochs by EarlyStopping (Section 2.1.2)
		optimizer	Adam, SGD or RMSProp
		reduce_lr_factor	init_lr can be reduced at most by this factor
		patience	patience for the ReduceLRonPlateau callback
2	evolutionary	zeta	portion of links or neurons to be evolved (analogous to ζ in the original SET algorithm)
		std_evo_scale (δ)	evolution scale (Equation 3.1)

Table 4.1: Procedure of hyperparameter tuning. Two rounds of hyperparameter tuning performed prior to each formal run: Round 1 involves tuning hyperparameters in plain (conventional) training. Round 2 involves tuning evolutionary hyperparameters of the proposed evolutionary algorithms. Simple grid search is performed on a collection of candidate values selected from each hyperparameter. Refer to Section 2.1.2 and 3.1.1 respectively for detailed descriptions of the plain and evolutionary hyperparameters.

- *Weight initializer.* The GlorotUniform initializer[6] (Equation B.1) is applied to all model weights. Other options for the initializer were included in hyperparameter tuning initially. However, the choice is made fixed after a hyperparameter tuning task of the ZigZag model[7]. A superior option has not been found.
- *Bias initializer.* Biases are initialized to zero. The decision was made along with the weight initializer.

4.2.2 Formal Runs

Formal experiments are performed using the optimal set of hyperparameters found in Section 4.2.1. Each set of formal experiments was based on the following setup unless otherwise indicated.

- **Plain**: 2,000 runs with 100 epochs per run. Note again that this is the baseline which we would like to outperform (Section 3.2.1).
- **LE**, **RLE**, **NE** and **RNE**: 100 runs with 20 evolutionary stages per run, and 100 epochs per stage.
- **NI**: 1,000 runs with 2 neuron increment stages per run, and 100 epochs per stage.

All settings not mentioned here followed the descriptions presented in Section 3.1 unless otherwise indicated. Accuracy values were derived using the methods described in Section 3.2.

[6]Keras default

[7]Reported in Appendix B.2

4.3 Results from Synthetic Data

4.3.1 ZigZag

algo	n	min	q05	median	q95	q975	max	avg	std	n_best	best_rate
Plain	2,000	-.0186	-.0109	.7932	.8073	.8075	.9090	.5765	.2624	0	0%
NI	1,000	.0523	.0560	.4431	.7955	.7964	.8093	.5531	.2456	0	0%
LE	100	.7754	.8746	.9991	1.0000	1.0000	1.0000	.9741	.0504	68	68%
RLE	100	.8536	.8858	.9998	1.0000	1.0000	1.0000	.9866	.0323	78	78%
NE	100	.3187	.7584	.9967	1.0000	1.0000	1.0000	.9272	.1249	54	54%
RNE	100	.7929	.8879	.9998	1.0000	1.0000	1.0000	.9821	.0405	72	72%

Table 4.2: Summary of R^2 values obtained from models trained using proposed algorithms on ZigZag dataset. All models bore a single hidden layer of 16 neurons. For the reported columns: n refers to number of runs; q05, q95 and q975 indicate 5%, 95% and 97.5% quantiles; n_best indicates number of models achieving $R^2 > .99$ and best_rate = n_best/n. All models were trained with batch_size = 32, init_lr = .01 and other optimal settings derived via hyperparameter tuning. LE, RLE and RNE were trained using $\zeta = .75$ and std_evo_scale = 1. NE was trained using $\zeta = .5$ and std_evo_scale = .25. Refer to Section 3.1 for definitions of the algorithms and Section 4.2 for the experiment setup.

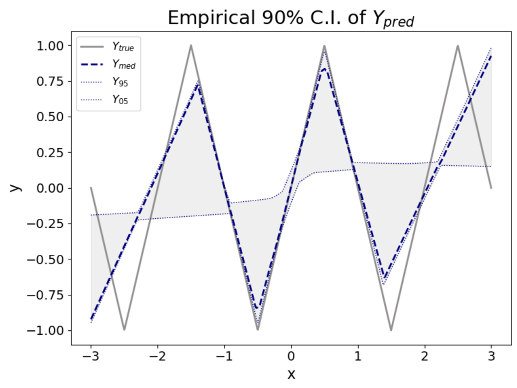
The primary benchmarking was performed on an MLP model bearing a single hidden layer of 16 neurons. Table 4.2 lists the accuracy and descriptive statistics indicating the overall performance and stability of the proposed algorithms. The results of hyperparameter tuning for this experiment can be found in Appendix B.2.

This experiment was meant to establish the ability of the evolutionary algorithms to escape from detrimental local optima. The 97.5th percentile of plain training R^2 (.8075) was significantly outperformed by the median of LE (.9991), RLE (.9998), NE (.9967) and RNE (.9998). In other words, the proposed evolutionary algorithms boosted up R^2 by at least .1892 when evaluated by the comparison protocol defined in Section 3.2.1. More remarkably, over half of the models achieved $R^2 > .99$ (hereafter referred to as "*best fit*") for all evolutionary algorithms[8]. Overall, RLE was the strongest (78%) and NE was the weakest (54%). These results cannot be achieved with the same computational resources in plain training. The NI algorithm provided no benefits in terms of efficiency and actually led to a decrease overall. This is a clear indication that "partial training followed by parameter space expansion" is unable to compete with the other evolutionary algorithms examined in this experiment.

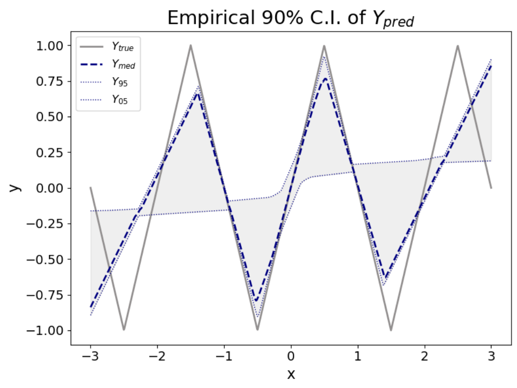
The strategies used in the selection of evolved neurons (i.e., random vs. least importance) were also compared. Surprisingly, both of the random evolution strategies (RLE and RNE) outperformed their least-importance counterparts (LE and NE). This has an significant implication: *Evolving important links can help improving model accuracy.* Due to differences in model structure and data, this was not discovered in the work of Mocanu et al. [2018].

An overall increase in the stability of the evolutionary algorithms was also observed. From a qualitative perspective, the worst results were eliminated, as indicated by the minimum and lower 5th percentile of accuracy. The lower 5th percentile accuracy values exceeded that of the 97.5th percentile in plain training (except for NE), which means that the threshold of the

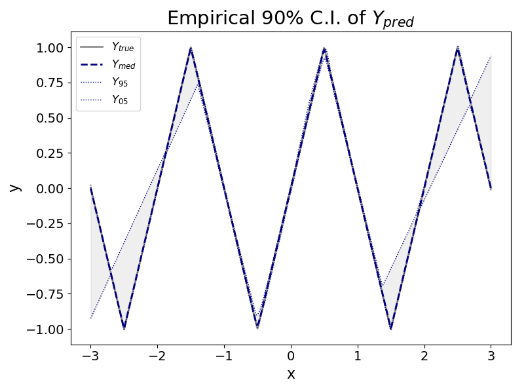
[8]refers to the four redesigned versions of SET, not including NI



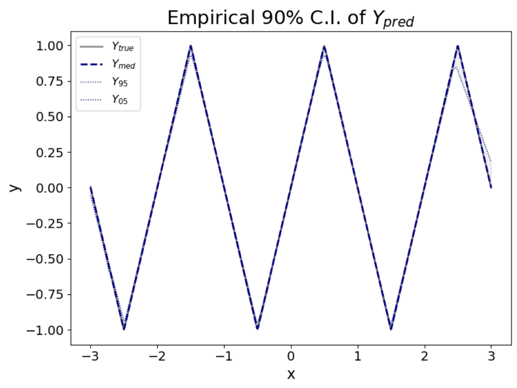
(a) Plain



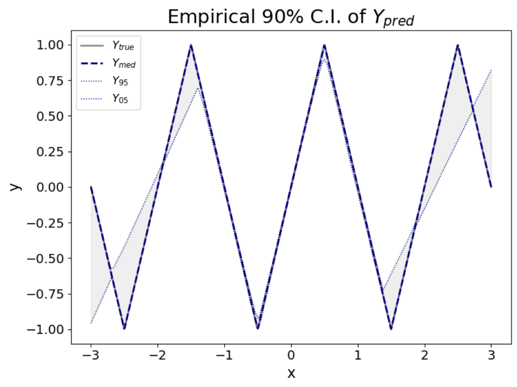
(b) NI



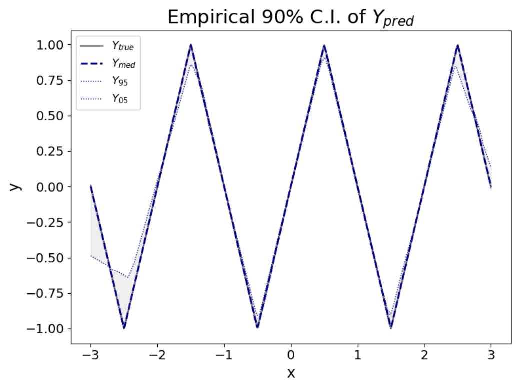
(c) LE



(d) RLE



(e) NE



(f) RNE

Figure 4.3: Empirically-derived 90% C.I. of values predicted by the proposed algorithms trained using ZigZag dataset. Grey lines are ground truth values, blue bold dashed line is median predicted value, grey shaded area is empirical symmetrical 90% C.I., and blue thin dotted lines (overlapping with blue bold dashed line frequently) are confidence limits. All models bore a single hidden layer of 16 neurons. All models were trained with batch_size = 32, init_lr = .01 and the other optimal settings derived via hyperparameter tuning. Model (c)(d)(f) were trained using $\zeta = .75$ and std_evo_scale = 1. Model (e) was trained using $\zeta = .5$ and std_evo_scale = .25. Refer to Section 3.1 for definitions of the algorithms and Section 4.2 for the experiment setup.

worst 5% of the evolutionary algorithms was better than the median of the best 5% of plain training. The likelihood that the evolutionary algorithms would produce accuracy results as low as those obtained using plain training is low.

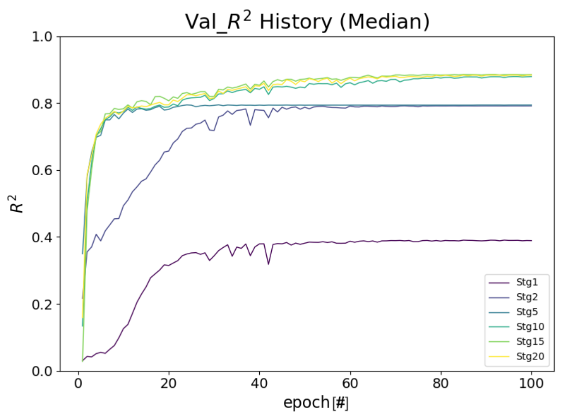
To directly visualize the goodness-of-fit of the trained models, the 90% confidence intervals (C.I.) of predicted values was plotted in Figure 4.3. The figure clearly shows poor goodness-of-fit between plain training and NI and better fit for the four evolutionary algorithms. The median result from the evolutionary algorithms was close to perfection with occasional instances of instability along the edges of the X domain. The evolutionary algorithms almost eliminated the horizontal fit lines that appeared frequently in the results from plain training and NI. The superior stability of the evolutionary algorithms is clearly indicated by the smaller shaded C.I. areas of the random evolutionary algorithms (Figure 4.3(d), 4.3(f)) compared to their non-random counterparts (Figure 4.3(c), 4.3(e)).

Training history provided valuable insights into the effectiveness of the evolutionary algorithms. Figure 4.4 shows the median multi-stage training history of the evolutionary algorithms.[9] RLE proved the most effective training strategy in terms of accuracy in the early epochs of the 5th and later training stages (Figure 4.4(b)); however, it converged far more slowly than did the other three. We posit that for an evolution step to be effective, there must be a shift from the current local optimum in the model parameter space to a new location that could be optimized to a better optimum. This can be thought of as jumping from the bottom of a shallow valley onto the hillside of another valley which is not easily reachable.[10] It is likely that the path, rate, and endpoint of convergence in the new valley are far different from those in the previous location; therefore, it is expected that the convergence of an effective step would be slower than that of an ineffective step in the original valley.

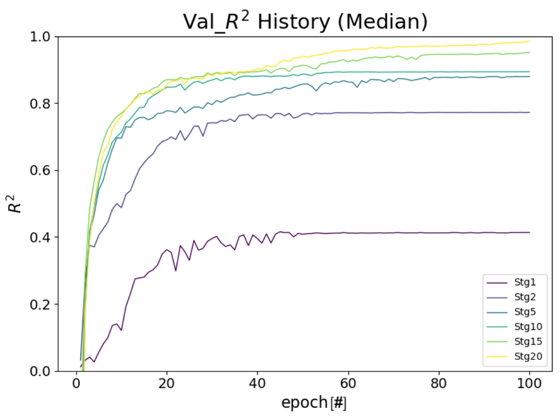
This observation can be generalized as follows: *Instances of rapid convergence may be an indication of insufficient probing of the model parameter space.* For example, the NE algorithm in Figure 4.4(c) appears to have converged more quickly in the 2nd and later stages. However, the resulting stability and the odds of reaching the best fit were far lower than with the other evolutionary algorithms. This is a clear indication of ineffective probing of the model parameter space. Nevertheless, this is only a preliminary observation offering some insight into the distribution of model accuracy.

[9]The 95th percentile of training history could also be presented, due to the fact that our focus was on the best outcomes rather than average performance. However, this would result in the plots clumping up, making them difficult to interpret while adding little in terms of new insights.

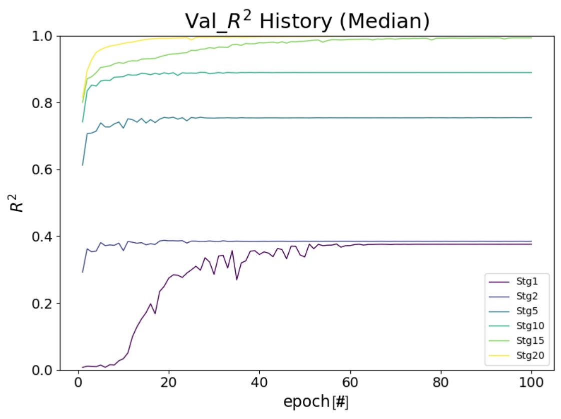
[10]A smaller loss function leads to better fit. Hence the optimization procedure is like stepping down into a valley. Evolutionary algorithms attempt to find deeper valleys over time.



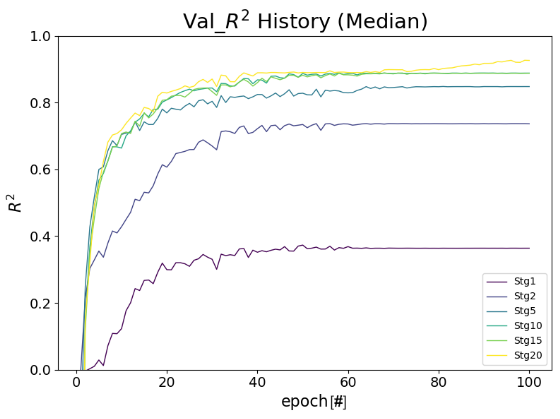
(a) LE



(b) RLE



(c) NE



(d) RNE

Figure 4.4: Multi-stage training history showing progress of median validation accuracy scores obtained from evolutionary algorithms trained using ZigZag dataset. For the sake of clarity, only the histories of stage 1, 2, 5, 10, 15 and 20 are shown. All models bore a single hidden layer of 16 neurons. All models were trained with batch_size = 32, init_lr = .01 and other optimal settings derived via hyperparameter tuning. LE, RLE and RNE were trained using $\zeta = .75$ and std_evo_scale = 1. NE was trained with $\zeta = .5$ and std_evo_scale = .25. Refer to Section 3.1 for definitions of the algorithms and Section 4.2 for the experiment setup.

4.3.2 NatSign

algo	n	min	q05	median	q95	q975	max	avg	std	n_best	best_rate
Plain	2,000	.8225	.8700	.9925	.9975	.9975	1.0000	.9625	.0524	1,409	70.5%
NI	1,000	.8025	.8675	1.0000	1.0000	1.0000	1.0000	.9617	.0589	700	70.0%
LE	75	.8800	.9950	1.0000	1.0000	1.0000	1.0000	.9977	.0138	74	98.7%
RLE	75	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	.0000	75	100.0%
NE	75	.9925	.9950	1.0000	1.0000	1.0000	1.0000	.9994	.0017	75	100.0%
RNE	75	.8425	.9950	1.0000	1.0000	1.0000	1.0000	.9958	.0230	73	97.3%

Table 4.3: Summary showing accuracy of model trained using proposed algorithms when applied to NatSign dataset. All models bore a single hidden layer of 4 neurons. For the reported columns: n indicates number of runs; q05, q95 and q975 refer to 5%, 95% and 97.5% quantiles; n_best indicates number of models that achieved 100% accuracy and best_rate = n_best/n. All models were trained with batch_size = 64 and init_lr = .03. LE, RLE, NE and RNE were run in a grid search of hyperparameters: $\zeta \in \{.25, .5, .75\}$ and std_evo_scale $\in \{.25, .5, 1, 2, 3\}$. Refer to Section 3.1 for definitions of algorithms and Section 4.2 for the experiment setup.

The experiment in Section 4.3.1 was repeated on the NatSign dataset in order to extend the proposed algorithms to classification problems. The algorithms were implemented on an MLP model with a single hidden layer of 4 neurons.[11] One critical deviation from the ZigZag experiment was that NatSign appeared too "easy" in the hyperparameter tuning phase (i.e., the results were too good), which made it difficult to distinguish the efficiency of the proposed algorithms. As a result, the formal run using a fixed set of hyperparameters was skipped. Model accuracy from the final round of hyperparameter tuning was retrieved for analysis. The summary statistics of benchmarking are listed in Table 4.3. The results of hyperparameter tuning for this experiment can be found in Appendix .

Although the median of the evolutionary algorithms reaches unity, the 97.5% quantile of plain training accuracy (.9975) also reached what was defined as the best fit ($R^2 > .99$). Hence all the proposed algorithms including plain training are working solutions to the NatSign problem. However, the nearly 30% difference in best_rate between the evolutionary and non-evolutionary algorithms clearly indicates the effectiveness of the evolutionary strategy. The outstanding stability of the evolutionary algorithms can be seen in their lower quantiles ($q05 \geq .9950$). Note that the results were obtained using a variety of evolutionary parameter combinations. This means that the conclusions should be robust to the selection of evolutionary parameters. Our findings also indicate that classification problems of greater complexity should be performed to obtain more effective benchmarks.

[11]The minimal size of the optimal MLP model is at most 4 as shown in Appendix A.2. Thus, the network used in this experiment has none or low redundancy.

4.4 Results from Real-world Dataset (MNIST)

algo	n	min	q05	median	q95	q975	max	avg	std
Plain	2,000	.8940	.9060	.9150	.9250	.9270	.9320	.9153	.0056
NI	1,000	.9120	.9200	.9300	.9380	.9400	.9470	.9292	.0054
LE	100	.9070	.9160	.9230	.9300	.9320	.9330	.9232	.0049
RLE	100	.9030	.9060	.9170	.9240	.9250	.9270	.9161	.0053
NE	100	.9200	.9220	.9310	.9390	.9400	.9420	.9313	.0049
RNE	100	.9130	.9190	.9270	.9350	.9350	.9360	.9271	.0049

Table 4.4: Summary statistics of accuracy of models trained using proposed algorithms on a downsampled subset of MNIST. All models bore a single hidden layer of 16 neurons. For the reported columns: n indicates the number of runs and q05, q95 and q975 refer to 5%, 95% and 97.5% quantiles. All models were trained with batch_size = 32, init_lr = .01 and other optimal settings derived via hyperparameter tuning. LE, RLE, NE and RNE were trained with $\zeta = .25$ and std_evo_scale = .25. Refer to Section 3.1 for definitions of algorithms and Section 4.2 for the experiment setup.

After demonstrating the effectiveness of the proposed evolutionary algorithms, we used a downsampled subset of the MNIST dataset (see Section 4.1.2) to test the performance on a real-world dataset. The MLP model used in this experiment included a single hidden layer of 16 neurons, as this was deemed ideal to demonstrate the differences among the algorithms in terms of accuracy.[12] The summary statistics of benchmarking are listed in Table 4.4. The results of hyperparameter tuning for this experiment can be found in Appendix .

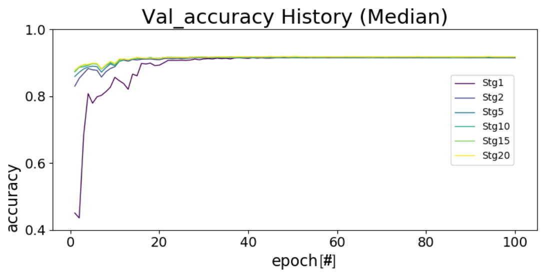
The performance of the proposed evolutionary algorithms on the real-world datasets did not match the performance on the synthetic dataset, with the result that the efficiency ranking was reversed. The 97.5% quantile of plain training accuracy (.9270) was not surpassed by the median accuracy of LE, RLE or RNE, and the NI strategy proves surprisingly efficient (q975 = .9400). Under these conditions, the most efficient[13] evolutionary algorithm was NE (.9310), whereas RLE (previously the strongest) became the weakest (.9170). Furthermore, the non-random evolutionary algorithms (LE and NE) were more efficient than their random counterparts. These results clearly demonstrate the critical role of data characteristics[14] on the effectiveness of the evolution process. Two strategies proved particularly suitable for this real-world problem: *incremental training* and *evolve-the-least-important*.

Figure 4.5 illustrates the training history of the algorithms. RLE and NE are compared for representativeness. It is clear that RLE produced more pronounced fluctuations in accuracy immediately after the evolutions in the early epochs of each training stage. This is consistent with the results obtained using the ZigZag dataset (Figure 4.4). Note however that this time, the pronounced fluctuations and slower convergence in accuracy did not have a positive effect on efficiency. Combined with the fact that NI is effective and non-random evolution is

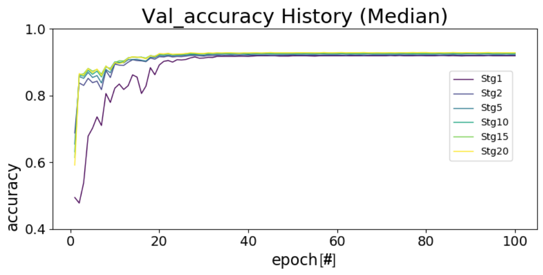
[12]Several other sizes (e.g., 8 and 32 neurons) were tested. With a large number of neurons, the results of the different algorithms were similarly good, while with only a few neurons, the results were similarly bad. The reasons for this need to be clarified through further research.

[13]in the sense defined in Section 3.2.1.

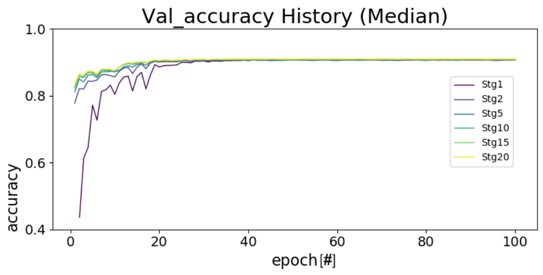
[14]The problem type (e.g. regression versus classification) may also be a vital factor; however, designing an experiment for exploring this topic is beyond the scope of this thesis.



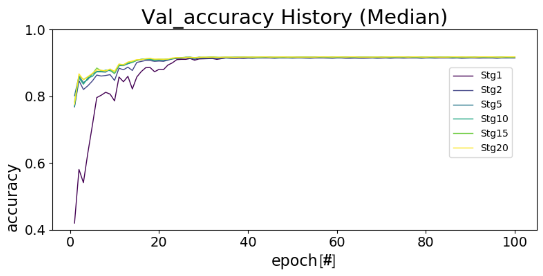
(a) LE



(b) RLE



(c) NE



(d) RNE

Figure 4.5: Multi-stage training history showing progress of median validation accuracy scores obtained from evolutionary algorithms trained on the downsampled MNIST dataset. All models included a single hidden layer of 16 neurons. Note that each algorithm performed 100 runs consisting of 20 stages. Each stage represented one round of plain training (100 epochs) following initialization (the 1st stage) or evolution (2nd stage and after). For the sake of clarity, only stages 1, 2, 5, 10, 15 and 20 (the last stage) are shown. All models were trained with batch_size = 32, init_lr = .01, std_evo_scale = .25, $\zeta = .25$ and other optimal settings derived via hyperparameter tuning. Refer to Chapter 3.1 for definitions of the algorithms.

preferred, it appears that this dataset are better-suited to incremental training rather than perturbing previously-trained coefficients. However, new insight or methodology must be introduced in order to systematically understand the interaction between training history and preference of training algorithm for a real-world dataset.

Chapter 5

Discussion

Overall, the capability of the proposed algorithms to escape local optima was strongly dependent on the characteristics of the data. From a qualitative perspective, if an algorithm proves effective even only in a special case, it is reasonable to claim that the training algorithm has potential, since a poorly-tuned algorithm would have a slim chance of success under any circumstances. Therefore, all the proposed algorithms (i.e., LE, RLE, NE, RNE and NI) reached this minimal threshold of success.

However, the difference of the effectiveness of random evolution between synthetic data and real-world data was immense (Section 4.4). Random evolution was more efficient than their non-random counterparts when trained on synthetic datasets; however, this finding was the opposite for real-world data. The gain of accuracy by introducing the evolutionary strategy also diminished greatly for real-world data. Interpretation of the convergence pattern in multi-stage training history plots looked solid for synthetic data at first, but was later challenged upon employing real-world data.

It is possible that the enormous discrepancy between the conclusions obtained from the synthetic and real-world datasets could be attributed to differences in the volume of information embedded in the data sample spaces. The synthetic datasets mainly comprised straight lines with few degrees of freedom in the sample space, which could possibly make it difficult to search for these exact degrees of freedom for the training algorithm. Note that once these degrees of freedom were found (i.e., evolution shifted into the valley containing the global optimum), subsequent optimization would not be a problem anymore. By contrast, most real-world datasets (e.g., MNIST) are too rich in terms of information content, even when downsampled and divided into subsets. Describing rich information requires numerous degrees of freedom. It is also important to consider that much of the information is not discrete, but rather follows a smooth distribution. Thus, it should be easy for a training algorithm to find a suitable number of degrees of freedom for a rich real-world dataset and produce fair results. Under these conditions, evolving the model parameter space via ”valley-jumping” can often be ineffective or even harmful.

Also note that despite multiple combinations of data and problem types were studied in this thesis,[1] the impact of problem type (as well as the joint effect of data type and problem

[1] i.e., synthetic data + function-fitting problem (ZigZag), synthetic data + classification problem (NatSign) and real-world data + classification problem (downsampled MNIST).

type) on the effectiveness of evolutionary strategy remains unresolved. The difficulty for designing an algorithm suitable for attacking this problem comes partly from the lack of systematic understanding regarding how data features progress within a neural network when a training algorithm is applied. To continue with this line of research, more sophisticated experiments must be designed to *clarify the interaction between data features and training algorithms*. Much remains to be determined within this topic. Other than direct forms of attack, promising directions may come from principles of model compression and model simplification. These topics are beyond the scope of this thesis.

The key findings of this thesis are summarized in the following:

- The NI, random evolution, and non-random evolution strategies proposed in this thesis are all potentially useful; however, their applicability depends largely on the characteristics of the dataset being trained. At this point, the potential benefits of the proposed algorithms in terms of efficiency can only be determined empirically.
- Datasets that resemble the simple synthetic datasets in this thesis would benefit from a strategy involving the evolution of *important links*. Furthermore, the NI approach may likely be ineffective in these situations. This does not necessarily conflict with the findings of Mocanu et al. [2018], as the models tested in that study differ from those in this thesis.
- Datasets that resemble the downsampled MNIST in this thesis would benefit from a strategy involving the evolution of *unimportant links*, as described by Mocanu et al. [2018]. Note that NI could be a viable option in these cases.
- Pattern of convergence in the multi-stage training history may carry information pertaining to the performance of the training model; however, the precise relationship depends on the characteristics of the data and remains to be clarified.

Chapter 6

Conclusions

6.1 Main Conclusions

Five novel algorithms for the training of small MLP networks were proposed in this thesis. Four of them (i.e., the proposed evolutionary algorithms: LE, RLE, NE and RNE) are re-designed versions of the sparse evolutionary training (SET) algorithm proposed by Mocanu et al. [2018]. The other one left (NI) is a simple incremental training scheme. These algorithms were benchmarked against plain (conventional) training with a comparable computational cost comprised of 2,000 traing stages with 100 epochs per stage. Two synthetic datasets for benchmarking were also constructed to adapt to the very small sizes of the networks employed (no more than 16 hidden neurons for the main experiments).

The proposed evolutionary algorithms proved highly effective when applied to synthetic data. As for the ZigZag dataset, all the proposed evolutionary algorithms not only boosted up the resulting model R^2 by at least .1892 but also enabled achieving the best fit scenario ($R^2 > .99$) with probability $\geq 54\%$, which was totally unattainable via plain training. In addition, the random evolution strategy was more effective than the "evolve the least-important weights" strategy employed by SET, suggesting that important weights can also be worth evolving. Experiments on the NatSign dataset yielded a less differentiating outcome with compatible implications. These findings were not known to Mocanu et al. [2018]. On the other hand, the accuracy gains of proposed evolutionary algorithms were far less prominent when benchmarked using real-world data (a downsampled subset of MNIST). The relative effectiveness between random and "least-important" evolution strategies were also reversed when compared to the trend of synthetic data. Neuron-based evolution and incremental training became potentially effective in this scenario. The results from real data indicated that the efficiency of the proposed algorithms may depend appreciably on the characteristics of data or problem type (classification or function-fitting). Similar phenomenon was also present in the interpretation of multi-stage training history, where slower convergence came along with better accuracy for synthesized data while the same was not observed for real-world data.

Overall, this proof-of-concept study demonstrated that evolutionary algorithms combined with gradient descent optimization could be used to overcome detrimental local optima when training small-sized networks. The findings also provided novel insights pertaining to efforts to improve the SET algorithm. That said, considerable testing on various forms of data and model structures must still be conducted to establish a more solid and systematic understanding. It is expected that these improved algorithms could eventually facilitate the deployment of more efficient and compact models on devices with limited computational capacity.

6.2 Limitations and Future Directions

This thesis was limited in a number of important areas:

1. This thesis focused exclusively on MLP networks, and all of the models employed in contained only one hidden layer. The effectiveness of the proposed algorithms on models with multiple hidden layers has yet to be elucidated.
2. This thesis used only densely-connected layers. The effectiveness of evolution in network topology (connectivity) was not explored systematically.
3. The datasets used in this thesis resided on the simplest side of the available options. For instance, functions with vector outputs or multi-color images were entirely disregarded.
4. Self-trained models were used rather than existing state-of-the-art models.
5. Sophisticated data preprocessing techniques such as data augmentation were not used.
6. The metric used for neuron evolution (p-sum with $p = 2$) was not widely explored; however, the effectiveness of random evolution reduces the significance of this limitation.

The author suggests the following directions for future studies:

1. Identifying the data features that are well-suited to evolutionary algorithms (which will no doubt require careful inspection of numerous datasets).
2. Extending the algorithm to other model types, such as CNN and recurrent neural networks (RNN).
3. Experimenting on sparse networks and with a focus on sparseness.
4. Evolving state-of-art models to see whether they can be improved.
5. Investigating the distribution and progression of model coefficients during training to elucidate how and why evolution works.

Bibliography

[1] Lei Jimmy Ba and Rich Caruana. Do Deep Nets Really Need to be Deep? In Ghahramani, Z and Welling, M and Cortes, C and Lawrence, ND and Weinberger, KQ, editor, *ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS 27 (NIPS 2014)*, volume 27 of *Advances in Neural Information Processing Systems*, 10010 NORTH TORREY PINES RD, LA JOLLA, CALIFORNIA 92037 USA, 2014. NEURAL INFORMATION PROCESSING SYSTEMS (NIPS). 11

[2] Cristian Buciluǎ, Rich Caruana, and Alexandru Niculescu-Mizil. Model Compression. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 535–541, New York, NY, USA, 2006. ACM. ISBN 1-59593-339-5. doi: 10.1145/1150402.1150464. URL `http://doi.acm.org/10.1145/1150402.1150464`. event-place: Philadelphia, PA, USA. 10

[3] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly Detection: A Survey. *ACM Comput. Surv.*, 41, 2009. doi: 10.1145/1541880.1541882. 5

[4] DQ Chen and P Burrell. On the optimal structure design of multilayer feedforward neural networks for pattern recognition. *INTERNATIONAL JOURNAL OF PATTERN RECOGNITION AND ARTIFICIAL INTELLIGENCE*, 16(4):375–398, June 2002. ISSN 0218-0014. doi: 10.1142/S0218001402001812. 6

[5] Tianqi Chen, Ian J. Goodfellow, and Jonathon Shlens. Net2Net: Accelerating Learning via Knowledge Transfer. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. URL `http://arxiv.org/abs/1511.05641`. 11

[6] Y. Cheng, D. Wang, P. Zhou, and T. Zhang. Model Compression and Acceleration for Deep Neural Networks: The Principles, Progress, and Challenges. *IEEE Signal Processing Magazine*, 35(1):126–136, January 2018. ISSN 1558-0792. doi: 10.1109/MSP.2017.2765695. 11

[7] François Chollet and others. Keras, 2015. URL `https://keras.io`. 3

[8] Yann Le Cun. A Theoretical Framework for Back-Propagation, 1988. 8

[9] Tom Dietterich. Overfitting and undercomputing in machine learning. *ACM Computing Surveys*, 27(3):326–327, September 1995. ISSN 0360-0300. doi: 10.1145/212094.212114. URL `http://doi.org/10.1145/212094.212114`. 5

[10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and D. Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13-15, 2010*, volume 9 of *JMLR Proceedings*, pages 249–256. JMLR.org, 2010. URL `http://proceedings.mlr.press/v9/glorot10a.html`. 48

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016. 5, 6

[12] A. Guillaume, S. Lee, Y. Wang, H. Zheng, R. Hovden, S. Chau, Y. Tung, and R. J. Terrile. Deep Space Network Scheduling Using Evolutionary Computational Methods. In *2007 IEEE Aerospace Conference*, pages 1–6, 2007. 10

[13] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q. Weinberger. On calibration of modern neural networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pages 1321–1330, Sydney, NSW, Australia, August 2017. JMLR.org. 8

[14] Geoffrey Hinton, Li Deng, Dong Yu, George E. Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N. Sainath, and Brian Kingsbury. Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *IEEE Signal Processing Magazine*, 29(6):82–97, November 2012. ISSN 1558-0792. doi: 10.1109/MSP.2012.2205597. Conference Name: IEEE Signal Processing Magazine. 5

[15] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the Knowledge in a Neural Network. NIPS 2014 Deep Learning Workshop, 2015. URL `https://arxiv.org/abs/1503.02531v1`. 11

[16] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580, 2012. URL `http://arxiv.org/abs/1207.0580`. _eprint: 1207.0580. 12

[17] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence.* MIT Press, 1992. ISBN 978-0-262-27555-2. doi: 10.7551/mitpress/1090.001.0001. URL `https://doi.org/10.7551/mitpress/1090.001.0001`. 10

[18] Plotly Technologies Inc. Collaborative data science, 2015. URL `https://plot.ly`. 48

[19] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *CoRR*, abs/1502.03167, 2015. URL `http://arxiv.org/abs/1502.03167`. 9

[20] Anoop Korattikara, Vivek Rathod, Kevin Murphy, and Max Welling. Bayesian Dark Knowledge. In Cortes, C and Lawrence, ND and Lee, DD and Sugiyama, M and Garnett, R, editor, *ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS 28 (NIPS 2015)*, volume 28 of *Advances in Neural Information Processing Systems*, 10010

NORTH TORREY PINES RD, LA JOLLA, CALIFORNIA 92037 USA, 2015. NEURAL INFORMATION PROCESSING SYSTEMS (NIPS). 11

[21] S. B. Kotsiantis. Supervised Machine Learning: A Review of Classification Techniques. In *Proceedings of the 2007 Conference on Emerging Artificial Intelligence Applications in Computer Engineering: Real Word AI Systems with Applications in EHealth, HCI, Information Retrieval and Pervasive Technologies*, pages 3–24, NLD, 2007. IOS Press. ISBN 978-1-58603-780-2. 5

[22] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. ImageNet Classification with Deep Convolutional Neural Networks. *Neural Information Processing Systems*, 25, 2012. doi: 10.1145/3065386. 5

[23] Yann LeCun and Corinna Cortes. MNIST handwritten digit database, 2010. URL `http://yann.lecun.com/exdb/mnist/`. 24

[24] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *NATURE*, 521(7553): 436–444, May 2015. ISSN 0028-0836. doi: 10.1038/nature14539. 8

[25] Ping Luo, Zhenyao Zhu, Ziwei Liu, Xiaogang Wang, and Xiaoou Tang. Face Model Compression by Distilling Knowledge from Neurons. In *THIRTIETH AAAI CONFERENCE ON ARTIFICIAL INTELLIGENCE*, pages 3560–3566, 2275 E BAYSHORE RD, STE 160, PALO ALTO, CA 94303 USA, 2016. ASSOC ADVANCEMENT ARTIFICIAL INTELLIGENCE. 11

[26] HR Maier and GC Dandy. Neural networks for the prediction and forecasting of water resources variables: a review of modelling issues and applications. *ENVIRONMENTAL MODELLING & SOFTWARE*, 15(1):101–124, 2000. ISSN 1364-8152. doi: 10.1016/S1364-8152(99)00007-9. 6

[27] John McCall. Genetic algorithms for modelling and optimisation. *Journal of Computational and Applied Mathematics*, 184(1):205 – 222, 2005. ISSN 0377-0427. doi: https://doi.org/10.1016/j.cam.2004.07.034. URL `http://www.sciencedirect.com/science/article/pii/S0377042705000774`. 10

[28] Thomas Miconi. Neural networks with differentiable structure, June 2016. URL `https://arxiv.org/abs/1606.06216v3`. 10, 12

[29] Tom M. Mitchell. *Machine learning, International Edition.* McGraw-Hill Series in Computer Science. McGraw-Hill, 1997. ISBN 978-0-07-042807-2. URL `https://www.worldcat.org/oclc/61321007`. 5

[30] Aryan Mobiny, Hien Van Nguyen, Supratik Moulik, Naveen Garg, and Carol C. Wu. DropConnect Is Effective in Modeling Uncertainty of Bayesian Deep Networks. *CoRR*, abs/1906.04569, 2019. URL `http://arxiv.org/abs/1906.04569`. _eprint: 1906.04569. 12

[31] Decebal Constantin Mocanu, Elena Mocanu, Peter Stone, Phuong H. Nguyen, Madeleine Gibescu, and Antonio Liotta. Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science. *Nature Communications*, 9(1):2383, June 2018. ISSN 2041-1723. doi: 10.1038/s41467-018-04316-3. URL `https://doi.org/10.1038/s41467-018-04316-3`. 1, 3, 10, 11, 13, 27, 36, 37

[32] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation Functions: Comparison of trends in Practice and Research for Deep Learning. *arXiv:1811.03378 [cs]*, November 2018. URL `http://arxiv.org/abs/1811.03378`. arXiv: 1811.03378. 7

[33] Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. FitNets: Hints for Thin Deep Nets. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL `http://arxiv.org/abs/1412.6550`. 11

[34] Idoia Ruiz, Bogdan Raducanu, Rakesh Mehta, and Jaume Amores. Optimizing speed/accuracy trade-off for person re-identification via knowledge distillation. *Engineering Applications of Artificial Intelligence*, 87:103309, 2020. ISSN 0952-1976. doi: https://doi.org/10.1016/j.engappai.2019.103309. URL `http://www.sciencedirect.com/science/article/pii/S0952197619302660`. 11

[35] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, October 1986. ISSN 1476-4687. doi: 10.1038/323533a0. URL `http://www.nature.com/articles/323533a0`. Number: 6088 Publisher: Nature Publishing Group. 8

[36] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach, Third International Edition*, pages 653–655. Pearson Education, 2010. ISBN 978-0-13-207148-2. URL `http://vig.pearsoned.com/store/product/1,1207,store-12521_isbn-0136042597,00.html`. 5

[37] Slimane Sefiane and M. Benbouziane. Portfolio Selection Using Genetic Algorithm. *Journal of Applied Finance & Banking*, 2:143–154, 2012. 10

[38] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Workshop Track Proceedings*, 2014. URL `http://arxiv.org/abs/1312.6034`. 16

[39] Suraj Srinivas and R. Venkatesh Babu. Data-free Parameter Pruning for Deep Neural Networks. In Xianghua Xie, Gary K. L. Tam, and Mark W. Jones, editors, *Proceedings of the British Machine Vision Conference (BMVC)*, pages 31.1–31.12. BMVA Press, September 2015. ISBN 1-901725-53-7. doi: 10.5244/C.29.31. URL `https://dx.doi.org/10.5244/C.29.31`. 9

[40] Karolina Stanislawska, Krzysztof Krawiec, and Zbigniew W. Kundzewicz. Modeling Global Temperature Changes with Genetic Programming. *Comput. Math. Appl.*, 64(12):3717–3728, December 2012. ISSN 0898-1221. doi: 10.1016/j.camwa.2012.02.049. URL `https://doi.org/10.1016/j.camwa.2012.02.049`. Place: USA Publisher: Pergamon Press, Inc. 10

[41] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002. ISSN 1063-6560. doi: 10.1162/106365602320169811. Place: Cambridge Publisher: Mit Press WOS:000176079100001. 10, 11, 20

[42] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009. ISBN 1-4414-1269-7. 3

[43] Li Wan, Matthew D. Zeiler, Sixin Zhang, Yann LeCun, and Rob Fergus. Regularization of Neural Networks using DropConnect. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, volume 28 of *JMLR Workshop and Conference Proceedings*, pages 1058–1066. JMLR.org, 2013. URL `http://proceedings.mlr.press/v28/wan13.html`. 12

[44] Shimon Whiteson and Peter Stone. Evolutionary function approximation for reinforcement learning. *Journal of Machine Learning Research*, 7:877–917, May 2006. ISSN 1532-4435. Place: Brookline Publisher: Microtome Publ WOS:000240173400007. 5, 10, 11, 12

[45] Ancong Wu, Wei-Shi Zheng, Xiaowei Guo, and Jian-Huang Lai. Distilled Person Re-Identification: Towards a More Scalable System. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019. 11

[46] Xin Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999. 10

[47] Sergey Zagoruyko and Nikos Komodakis. Paying More Attention to Attention: Improving the Performance of Convolutional Neural Networks via Attention Transfer. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017. URL `https://openreview.net/forum?id=Sks9_ajex`. 11, 16

Appendix A

Functions for Synthetic Datasets

A.1 ZigZag

The ZigZag function consists of three consecutive periods of a linearized sinusoidal wave (or triangle wave) centered on the origin. The wave has a period of 2 and an amplitude of 1. The function is expressed by Equation A.1.

$$y = 1 - 2|((x + .5) \bmod 2) - 1|; x \in [-3, 3] \tag{A.1}$$

The minimal optimal MLP model using ReLU activation bears a single hidden layer of 7 neurons, which is simply equal to the number of segments in the graph.

A.2 NatSign

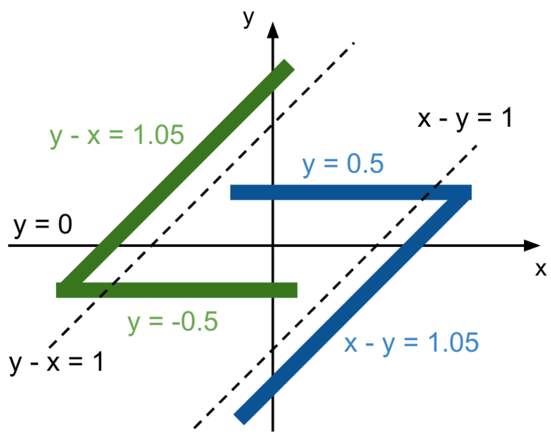
The dataset consists of two distinct labels and four lines in the input domain (i.e., the X-Y plane). The green "left elbow" in Figure A.1(a) is labeled 1 and the blue "right elbow" is labeled 0. For the left elbow, the horizontal line and slash line are derived using Equation A.2. The equation of the right elbow is derived by mirror symmetry.

$$\begin{cases} y = & -.3 \quad (x \in [-1, .35]) \quad \text{......horizontal} \\ y = & x - .7 \quad (x \in [-1, 1.3]) \quad \text{......slash} \end{cases} \tag{A.2}$$

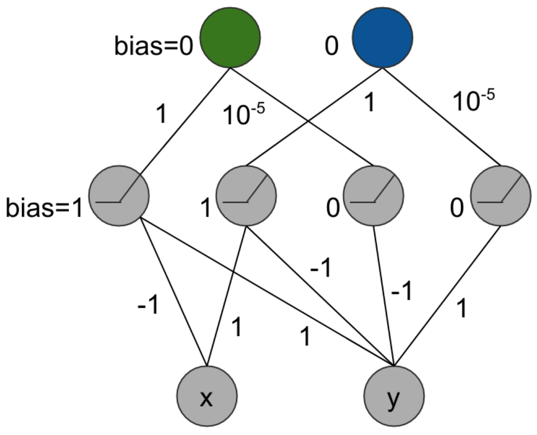
The sampling density in the x-domain of the slash lines is reduced by a factor of $1/\sqrt{2}$ compared to the horizontal lines to maintain a constant sampling density along the lines. Furthermore, the length of the slash lines ensured that for an arbitrary cut line across the origin, the number of green and blue points on one side remains approximately equal ($<$ 10% of difference). In such a balanced sample, it would not be possible to obtain an "easy score" for a blind arbitrary cut. The arm length was set in this manner to make the effectiveness of the training procedure more evident.[1].

[1]instead of always producing seemingly good results overall

The minimal optimal network consists of no more than 4 hidden neurons, as shown in Figure A.1. A different set of function parameters was used to facilitate computation and verification. This was achieved by dividing the input domain, such that positive and negative output values belong to green and blue labels, respectively. The first and second neurons left in the hidden layer give the region above $y - x = 1$ a positive value and the region below $x - y = 1$ a negative value (.05 and $-.05$ for the green and blue slope lines, respectively). The region in between yields zero after the first two neurons; however, the yielded value is expanded slightly by the third and fourth neurons. The contribution of the third and fourth neurons (on the order of 10^{-5}) was added to the value contributed by the first two neurons; however, it was insufficient to affect the sign of the value within the domain of the dataset. As a result, the green samples obtain positive outputs, whereas the blue samples obtain negative outputs. Thus, the classification problem can be solved using a hidden layer of 4 neurons.



(a) NatSign Dataset (Transformed)



(b) Optimal MLP Model

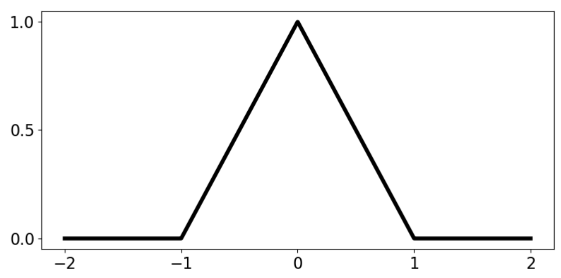
Figure A.1: A small optimal MLP for NatSign classification problem: (a) labels of output data shown in green and blue; (b) numbers associated with links and neurons are weights and biases, respectively. Links with zero associated weight are not shown for clarity.

A.3 Peak-1D

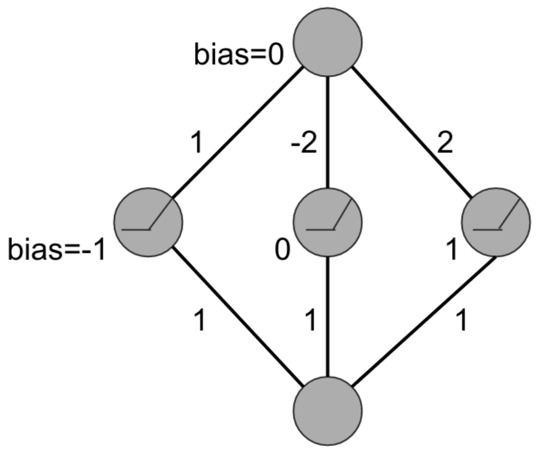
Preliminary studies of this thesis was performed using the *Peak-1D* dataset, a simple pyramid-shaped function given by Equation A.3. The function is visualized in Figure A.2(a). The minimal optimal model bears a hidden layer with 3 neurons, as shown in A.2(b).

$$\begin{cases} y = & 0 & (1 \leq |x| \leq 2) \\ y = & 1 - |x| & (|x| < 1) \end{cases} \tag{A.3}$$

The results from Peak-1D were eventually discarded as the dataset was proved too easy to train (i.e., too easy to reach the best fit even for plain training), provided that the hyperparameter tuning procedure was executed correctly. Such a dataset would be unable to distinguish between the effectiveness of the proposed algorithms.



(a) Peak-1D Dataset



(b) Minimal Optimal Model

Figure A.2: (a) Peak-1D dataset. (b) Minimal optimal MLP model for Peak-1D.

One potential advantage for this dataset is that its definition could be easily extended to arbitrary input dimensions. This could be done by taking the minimum value of $1 - |x_i|$, subject to a lower bound of 0, where i is one of the input dimensions. For example, Peak-2D dataset would look like a real pyramid. This opens up an easily-implemented way for constructing simple multi-dimensional function-fitting problems.

Appendix B

Hyperparameter Tuning of Main Experiments

The results of hyperparameter tuning for the main experiments (Section 4.3 and Section 4.4) are presented in this section, except that the results from NatSign in Section 4.3.2 were too trivial and therefore skipped. Summary statistics of individual runs are presented in tables, while individual runs are visualized using parallel coordinates plot using Plotly library [18][1]. The actual choice of hyperparameters can be found as the bold-texted values in the tables. Adaptive adjustments that deviated from the default protocol in Section 4.2 are also explained.

B.1 Model Initializers

In the hyperparameter tuning procedure, options of initializers of model coefficients were tested.

B.1.1 Weight Initializer

The default initializer of model weights is the *GlorotUniform* formula given by Equation B.1 as proposed by Glorot and Bengio [10]. The intention is to keep the statistically expected value of the model output close to unity [10]. The initializer class was implemented in Keras.[2]

$$\begin{cases} x \sim U(\text{-limit, limit}) \\ \text{limit} = \sqrt{\dfrac{6}{n_i + n_o}} \end{cases} \tag{B.1}$$

where

[1]`https://plotly.com/python`

[2]see `https://keras.io/api/layers/initializers` for the online documentation

$$
\begin{aligned}
x &= \text{the initialized value} \\
U &= \text{uniform distribution from an interval} \\
n_i &= \text{number of input neurons connected to the layer containing} \\
&\quad \text{the neuron being initialized} \\
n_o &= \text{number of output neurons connected to the layer containing} \\
&\quad \text{the neuron being initialized}
\end{aligned}
$$

The alternative to GlorotUniform was selected to be $U(-.05, .05)$ (denoted *random_uniform* hereafter). It is not adaptive to network size, and the limit .05 can be seen as relatively small when compared to the GlorotUniform limit for $n_i = 1$ and $n_o = 16$ (.594). This option is tested in some cases of hyperparameter tuning. It is to see whether different degree of concentration of initial weight will affect the accuracy of the trained model.

A closely related formula named *GlorotNormal* could also be used.[3] However, virtually no difference from GlorotUniform was found by some preliminary studies. Hence this option is not being tested during hyperparameter tuning.

Consequently, only two options, namely glorot_uniform and random_uniform, was tuned in all hyperparameter tuning results reported. Furthermore, since GlorotNormal was winning in almost every single run, the initializer became a fixed value for most hyperparameter tuning tasks performed.

B.1.2 Bias Initializer

The default initializer of neuron bias is simple *zeros* as it is empirically known to be more efficient. An alternative to the default was the standard normal distribution $N(0, 1)$. The intuition stemmed from the fact that data are usually transformed into unity scale before training. As the bias of ReLU function defines the location of "function behavior change", the bias may also be distributed over the same range to capture different local features.

In brief, only zeros and random_normal were tested during hyperparameter tuning for some cases. However, the zeros initializer is all superior for the cases in this thesis. Therefore, the bias initializer is in fact fixed to zeros for most hyperparameter tuning tasks in this thesis.

[3]Documented in the same online manual of GlorotUniform.

B.2 ZigZag (16 Hidden Neurons)

hyperparameter	value	n	min	q05	median	q95	q975	max	avg	std
batch_size	16	240	-.0166	.0232	.1189	.3753	.4287	.8827	.1476	.1380
	32	240	.0076	.0286	.1409	.5248	.5973	.7896	.1720	.1519
	64	240	.0023	.0260	.1445	.6571	.7705	.8237	.1884	.1793
	128	240	.0041	.0265	.1173	.6714	.7626	.7877	.1677	.1742
	256	240	-.0523	-.0102	.0583	.4163	.6915	.9009	.1242	.1615
batch_normalization	True	600	-.0523	.0015	.1098	.4702	.6078	.7588	.1422	.1413
	False	600	-.0088	.0238	.1338	.6577	.7852	.9009	.1778	.1808
dropout_rate	**0**	400	-.0102	.0310	.1873	.7808	.7875	.9009	.2546	.2339
	.2	400	-.0523	.0184	.1189	.2669	.2749	.2945	.1274	.0861
	.5	400	-.0365	.0023	.1051	.1898	.1989	.2100	.0980	.0624
init_lr	.1	240	-.0166	.0297	.1735	.6915	.7808	.9009	.2145	.1790
	.03	240	-.0017	.0310	.1968	.6929	.7849	.8237	.2452	.1863
	.01	240	-.0390	.0095	.1594	.5477	.7626	.8827	.1861	.1707
	.003	240	-.0523	.0055	.0610	.2745	.3488	.6437	.0966	.0990
	.001	240	-.0164	.0052	.0466	.1418	.1859	.2922	.0574	.0449
weight_initializer	**glorot_uniform**	600	-.0166	.0248	.1298	.6567	.7808	.8827	.1792	.1809
	random_uniform	600	-.0523	.0057	.1040	.3747	.5211	.9009	.1408	.1408
bias_initializer	**zeros**	600	-.0335	.0217	.1419	.6385	.7836	.9009	.1763	.1759
	random_normal	600	-.0523	.0149	.1039	.4163	.6078	.8827	.1437	.1476
patience	5	600	-.0523	.0161	.1086	.4899	.6882	.8827	.1519	.1594
	10	600	-.0390	.0202	.1261	.5585	.7233	.9009	.1681	.1665

Table B.1: Tuning non-evolutionary hyperparameters on the ZigZag dataset with an MLP model bearing a single hidden layer of 16 neurons. A simple grid search was performed on 1,200 combinations out of 7 parameters. One trial per combination was run. The main figures reported stand for the accuracy metric, R^2. For the reported columns, n is the number of runs. q05, q95 and q975 are the 5%, 95% and 97.5% quantiles. Refer to Table 4.1 for the definition of the hyperparemeters.

Two rounds of hyperparameter tuning were performed for the ZigZag dataset with a MLP model containing a single hidden layer of 16 neurons. The results are reported in Table B.1 and B.2 for tuning the non-evolutionary and evolutionary hyperparameters respectively. The hyperparameter values in bold were chosen for subsequent runs for the same model. Values with the maximum median R^2 are chosen unless otherwise explained. The above results are also visualized in parallel coordinate axis in Figure B.1.

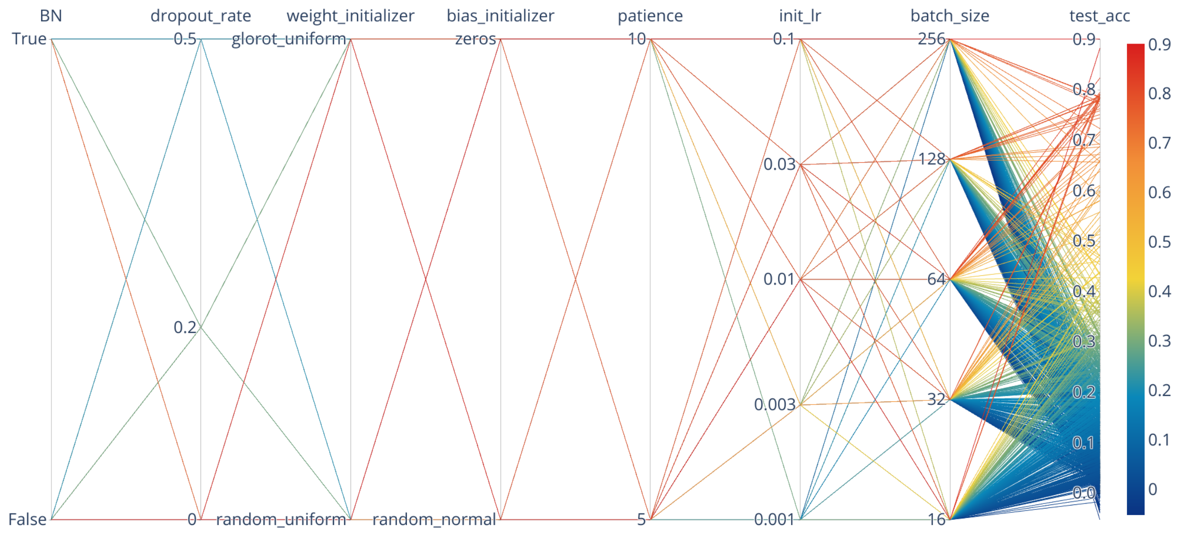
For the result of non-evolutionary parameters, an exception of the *max-median* rule was that batch_size = 32 were chosen over 64 for consistency with other results and prior experience. The median accuracy of batch_size = 32 (.1409) was not very far from that of batch_size = 64 (.1445). Also note that the low median accuracy of plain training on small networks is expected.

For the result of evolutionary parameters, one can readily observe that the median accuracy improved dramatically compared to the outcome of plain training with the evolutionary training algorithms introduced. $(\zeta) = .5$ and std_evo_scale = 1 were chosen for all algorithms except for NE taking std_evo_scale = .5. The difference of NE is understandable as NE contains an extra hyperparameter ($p = 2$ of the p-sum metric of neuron importance), so its behavior is more likely to deviate more from others.

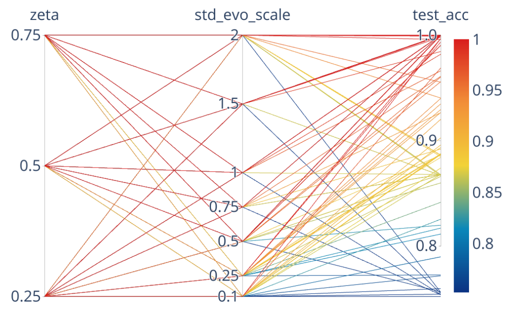
algo	hyperparameter	value	n	min	q05	median	q95	q975	max	avg	std	n_best	best_rate
LE	zeta	.25	120	.7969	.8100	.9092	1.0000	1.0000	1.0000	.9247	.0680	44	36.7%
		.5	120	.8049	.8465	.9997	1.0000	1.0000	1.0000	.9656	.0534	72	60.0%
		.75	120	.0248	.8042	.9984	1.0000	1.0000	1.0000	.9487	.1053	69	57.5%
	std_evo_scale	.1	60	.7864	.7969	.8973	1.0000	1.0000	1.0000	.8951	.0648	10	16.7%
		.25	60	.7657	.8270	.9484	1.0000	1.0000	1.0000	.9393	.0579	19	31.7%
		.5	60	.8100	.8112	.9905	1.0000	1.0000	1.0000	.9553	.0586	31	51.7%
		.75	60	.8100	.8110	.9997	1.0000	1.0000	1.0000	.9699	.0578	45	75.0%
		1	60	.8088	.8113	.9999	1.0000	1.0000	1.0000	.9654	.0562	40	66.7%
		1.5	60	.0248	.8099	.9996	1.0000	1.0000	1.0000	.9528	.1327	40	66.7%
RLE	zeta	.25	120	.7627	.8095	.9948	1.0000	1.0000	1.0000	.9531	.0655	67	55.8%
		.5	120	.7425	.7944	.9999	1.0000	1.0000	1.0000	.9597	.0680	78	65.0%
		.75	120	.6640	.7620	1.0000	1.0000	1.0000	1.0000	.9511	.0805	77	64.2%
	std_evo_scale	.1	60	.6640	.7425	.8342	.9925	.9945	.9982	.8519	.0788	4	6.7%
		.25	60	.7941	.8580	.9479	1.0000	1.0000	1.0000	.9400	.0522	17	28.3%
		.5	60	.7706	.8852	1.0000	1.0000	1.0000	1.0000	.9835	.0394	47	78.3%
		.75	60	.8846	.8866	1.0000	1.0000	1.0000	1.0000	.9895	.0297	51	85.0%
		1	60	.8786	.9297	1.0000	1.0000	1.0000	1.0000	.9937	.0236	56	93.3%
		1.5	60	.7627	.7732	1.0000	1.0000	1.0000	1.0000	.9692	.0663	47	78.3%
NE	zeta	.25	120	.4953	.7745	.8807	1.0000	1.0000	1.0000	.8769	.1018	40	33.3%
		.5	120	.7737	.7757	1.0000	1.0000	1.0000	1.0000	.9443	.0749	68	56.7%
		.75	120	.4556	.7744	.9499	1.0000	1.0000	1.0000	.9258	.0896	52	43.3%
	std_evo_scale	.1	60	.4556	.7651	.8955	1.0000	1.0000	1.0000	.9062	.0930	19	31.7%
		.25	60	.7760	.7791	.9999	1.0000	1.0000	1.0000	.9560	.0668	34	56.7%
		.5	60	.7741	.7747	1.0000	1.0000	1.0000	1.0000	.9377	.0885	36	60.0%
		.75	60	.7680	.7753	.9996	1.0000	1.0000	1.0000	.9249	.0893	32	53.3%
		1	60	.4953	.7758	.8965	1.0000	1.0000	1.0000	.9113	.1013	26	43.3%
		1.5	60	.7710	.7738	.8136	1.0000	1.0000	1.0000	.8580	.0898	13	21.7%
RNE	zeta	.25	120	.7782	.8492	.9998	1.0000	1.0000	1.0000	.9630	.0583	75	62.5%
		.5	120	.7887	.8465	.9999	1.0000	1.0000	1.0000	.9715	.0527	80	66.7%
		.75	120	.7798	.8095	.9999	1.0000	1.0000	1.0000	.9594	.0644	73	60.8%
	std_evo_scale	.1	60	.7798	.7913	.8722	.9970	.9981	1.0000	.8833	.0641	5	8.3%
		.25	60	.8080	.8614	.9701	1.0000	1.0000	1.0000	.9569	.0451	19	31.7%
		.5	60	.8801	.8814	1.0000	1.0000	1.0000	1.0000	.9882	.0293	48	80.0%
		.75	60	.7811	.8806	1.0000	1.0000	1.0000	1.0000	.9883	.0387	54	90.0%
		1	60	.8796	.8950	1.0000	1.0000	1.0000	1.0000	.9911	.0282	53	88.3%
		1.5	60	.7782	.7800	1.0000	1.0000	1.0000	1.0000	.9801	.0527	49	81.7%

Table B.2: Tuning evolutionary hyperparameters on the ZigZag dataset based on the same model and optimal hyperparemeters chosen from Table B.1. A simple grid search was performed on 18 combinations out of 2 parameters. 20 trials per combination are run. Other illustrations are the same as Table B.1.

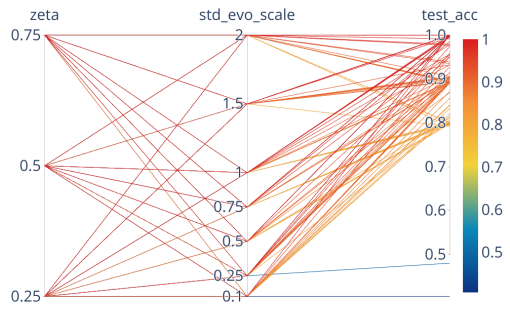
There is a note with regard to the resource cost: ca. 2.6 hours on an ordinary laptop with a 4-core-8-threaded Intel® CPU were spent on tuning the non-evolutionary parameters, whereas the evolutionary parameters took ca. 16 hours to tune for each evolutionary algorithm. The duration was simply a rough idea and not measured with precision.



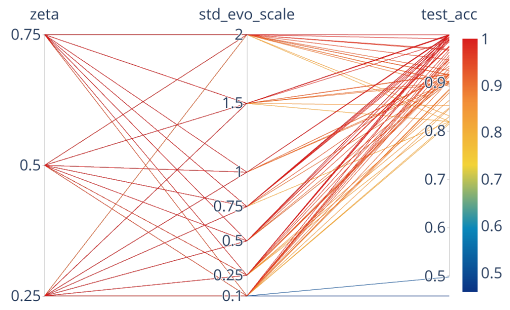
(a) Round 1



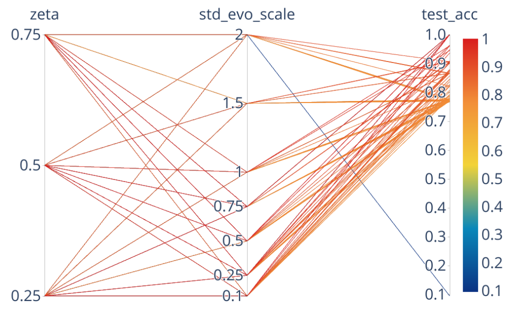
(b) Round 2 (LE)



(c) Round 2 (RLE)



(d) Round 2 (NE)



(e) Round 2 (RNE)

Figure B.1: Visualization of hyperparameter tuning on ZigZag. All models bore a single hidden layer of 16 neurons. Model accuracy values (R^2) evaluated using test data are mapped to the test_acc axis and colorized. Subfigure (a): the first round of tuning on non-evolutionary hyperparameters (BN=batch_normalization). Subfigure (b)-(e): the second round of tuning on evolutionary hyperparameters for algorithm LE, RLE, NE and RNE. Refer to Section 4.2.1 for descriptions of the tuning procedure.

B.3 NatSign (4 Hidden Neurons)

hyperparameter	value	n	min	q05	median	q95	q975	max	avg	std	n_best	best_rate
init_lr	.3	48	.8450	.8525	.8925	1.0000	1.0000	1.0000	.9075	.0525	11	22.9%
	.1	48	.8600	.8650	.8950	1.0000	1.0000	1.0000	.9260	.0531	14	29.2%
	.03	48	.8750	.8750	.9075	1.0000	1.0000	1.0000	.9320	.0491	13	27.1%
	.01	48	.8650	.8725	.8950	1.0000	1.0000	1.0000	.9221	.0505	10	20.8%
	.003	48	.8625	.8675	.8925	1.0000	1.0000	1.0000	.9154	.0470	6	12.5%
	.001	48	.7875	.8625	.8825	.9750	.9850	1.0000	.8898	.0388	2	4.2%
batch_normalization	**False**	144	.7875	.8650	.8950	1.0000	1.0000	1.0000	.9174	.0549	40	27.8%
	True	144	.8575	.8625	.8925	.9925	.9925	.9950	.9135	.0459	16	11.1%
dropout_rate	**0**	144	.7875	.8650	.9750	1.0000	1.0000	1.0000	.9448	.0558	56	38.9%
	.2	144	.8050	.8625	.8850	.9100	.9100	.9850	.8861	.0174	0	0.0%
signed_epochs	0	96	.8575	.8675	.8950	1.0000	1.0000	1.0000	.9185	.0491	17	17.7%
	50	96	.8050	.8625	.8900	1.0000	1.0000	1.0000	.9124	.0505	18	18.8%
	100	96	.7875	.8600	.8925	1.0000	1.0000	1.0000	.9154	.0523	21	21.9%
batch_size	16	72	.8525	.8625	.8950	1.0000	1.0000	1.0000	.9192	.0490	13	18.1%
	32	72	.8450	.8650	.8925	1.0000	1.0000	1.0000	.9161	.0507	14	19.4%
	64	72	.8575	.8650	.8950	1.0000	1.0000	1.0000	.9156	.0495	15	20.8%
	128	72	.7875	.8575	.8875	1.0000	1.0000	1.0000	.9109	.0530	14	19.4%

Table B.3: Tuning non-evolutionary hyperparameters on the NatSign dataset with an MLP model bearing a single hidden layer of 4 neurons. A simple grid search was performed on 288 combinations out of 5 parameters. One trial per combination was run. The main figures reported is prediction accuracy. For the reported columns, n is the number of runs. q05, q95 and q975 are the 5%, 95% and 97.5% quantiles. n_best is the number of models achieving accuracy = 1 and best_rate = n_best/n. The bold values were chosen for subsequent runs. Refer to Table 4.1 for the definition of the hyperparemeters.

The results of hyperparameter tuning on NatSign is reported in Table B.3 and B.4 for non-evolutionary and evolutionary hyperparameters respectively. As one can easily observe in Table B.4 (the second run), perfect prediction is reached for nearly all the test cases disregard the choice of the evolutionary parameters. Therefore, the proposed evolutionary algorithms were not executed using a fixed set of hyperparameter due to the lack of distinguishing power of the NatSign dataset for these algorithms.

algo	hyperparameter	value	n	min	q05	median	q95	q975	max	avg	std	n_best	best_rate
LE	zeta	.25	25	.8800	.9950	1.0000	1.0000	1.0000	1.0000	.9948	.0235	24	96.0%
		.5	25	.9925	.9950	1.0000	1.0000	1.0000	1.0000	.9988	.0021	25	100.0%
		.75	25	.9950	.9975	1.0000	1.0000	1.0000	1.0000	.9996	.0012	25	100.0%
	std_evo_scale	.25	15	.8800	.8800	1.0000	1.0000	1.0000	1.0000	.9917	.0299	14	93.3%
		.5	15	.9975	.9975	1.0000	1.0000	1.0000	1.0000	.9998	.0006	15	100.0%
		1	15	.9950	.9950	1.0000	1.0000	1.0000	1.0000	.9993	.0014	15	100.0%
		2	15	.9950	.9950	1.0000	1.0000	1.0000	1.0000	.9992	.0017	15	100.0%
		3	15	.9925	.9925	1.0000	1.0000	1.0000	1.0000	.9987	.0024	15	100.0%
RLE	zeta	.25	25	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	.0000	25	100.0%
		.5	25	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	.0000	25	100.0%
		.75	25	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	.0000	25	100.0%
	std_evo_scale	.25	15	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	.0000	15	100.0%
		.5	15	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	.0000	15	100.0%
		1	15	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	.0000	15	100.0%
		2	15	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	.0000	15	100.0%
		3	15	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	.0000	15	100.0%
NE	zeta	.25	25	.9975	1.0000	1.0000	1.0000	1.0000	1.0000	.9999	.0005	25	100.0%
		.5	25	.9950	.9950	1.0000	1.0000	1.0000	1.0000	.9996	.0014	25	100.0%
		.75	25	.9925	.9925	1.0000	1.0000	1.0000	1.0000	.9987	.0025	25	100.0%
	std_evo_scale	.25	15	.9925	.9925	1.0000	1.0000	1.0000	1.0000	.9983	.0025	15	100.0%
		.5	15	.9950	.9950	1.0000	1.0000	1.0000	1.0000	.9997	.0012	15	100.0%
		1	15	.9975	.9975	1.0000	1.0000	1.0000	1.0000	.9998	.0006	15	100.0%
		2	15	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	.0000	15	100.0%
		3	15	.9925	.9925	1.0000	1.0000	1.0000	1.0000	.9992	.0022	15	100.0%
RNE	zeta	.25	25	.9975	1.0000	1.0000	1.0000	1.0000	1.0000	.9999	.0005	25	100.0%
		.5	25	.9950	.9975	1.0000	1.0000	1.0000	1.0000	.9997	.0011	25	100.0%
		.75	25	.8425	.8725	1.0000	1.0000	1.0000	1.0000	.9878	.0387	23	92.0%
	std_evo_scale	.25	15	.8425	.8425	1.0000	1.0000	1.0000	1.0000	.9885	.0390	14	93.3%
		.5	15	.8725	.8725	1.0000	1.0000	1.0000	1.0000	.9912	.0317	14	93.3%
		1	15	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	.0000	15	100.0%
		2	15	.9975	.9975	1.0000	1.0000	1.0000	1.0000	.9997	.0008	15	100.0%
		3	15	.9950	.9950	1.0000	1.0000	1.0000	1.0000	.9997	.0012	15	100.0%

Table B.4: Tuning evolutionary hyperparameters on the ZigZag dataset based on the same model and optimal hyperparemeters chosen from Table B.3. A simple grid search was performed on 15 combinations out of 2 parameters. 5 trials per combination are run. Other illustrations are the same as Table B.3.

B.4 Downsampled MNIST (16 Hidden Neurons)

Panel 1: Initial Search										
hyperparameter	value	n	min	q05	median	q95	q975	max	avg	std
epochs	30	126	.0990	.1090	.7210	.8970	.9070	.9200	.5848	.3060
	60	126	.1020	.1090	.8310	.9090	.9140	.9270	.6480	.3076
	100	126	.1020	.1090	.8650	.9100	.9170	.9250	.6788	.3062
batch_size	64	126	.0990	.1090	.8730	.9200	.9210	.9270	.6632	.3345
	256	126	.1090	.1090	.8530	.9060	.9080	.9190	.6711	.3015
	1,024	126	.1020	.1090	.6520	.8740	.8780	.9060	.5773	.2799
batch_normalization	**True**	189	.1020	.2710	.8680	.9170	.9200	.9270	.7422	.2242
	False	189	.0990	.1090	.7330	.8850	.8870	.8960	.5322	.3446
dropout_rate	**0**	126	.0990	.1090	.8570	.9190	.9210	.9270	.6685	.2999
	.2	126	.1020	.1090	.8110	.9060	.9100	.9200	.6307	.3176
	.4	126	.1020	.1090	.7840	.8940	.8960	.9010	.6124	.3068
init_lr	1.0	54	.0990	.1020	.1090	.6320	.6820	.7540	.2312	.1787
	.3	54	.1020	.1020	.3130	.8980	.8980	.9190	.4452	.3438
	.1	54	.1020	.1090	.5060	.9130	.9160	.9210	.5301	.3446
	.03	54	.2920	.4930	.8670	.9140	.9200	.9270	.8001	.1371
	.01	54	.5170	.5210	.8680	.9100	.9190	.9250	.8455	.0864
	.003	54	.2890	.4180	.8700	.9060	.9070	.9250	.8197	.1344
	.001	54	.3780	.5140	.8470	.8970	.9020	.9050	.7887	.1272
Panel 2: Refined Search										
hyperparameter	value	n	min	q05	median	q95	q975	max	avg	std
batch_size	**32**	50	.8990	.9040	.9120	.9200	.9210	.9230	.9124	.0051
	64	50	.8990	.9000	.9100	.9200	.9210	.9260	.9101	.0059
	128	50	.8910	.8950	.9080	.9150	.9150	.9170	.9067	.0063
	256	50	.8870	.8920	.9040	.9130	.9130	.9210	.9037	.0063
init_lr	**.01**	100	.8920	.8980	.9100	.9210	.9210	.9260	.9098	.0066
	.003	100	.8870	.8930	.9070	.9170	.9180	.9210	.9068	.0066

Table B.5: Tuning non-evolutionary hyperparameters on the downsampled MNIST dataset with an MLP model bearing a single hidden layer of 16 neurons. The main figures reported is prediction accuracy. *Panel 1*: A simple grid search was performed on 378 combinations out of 5 parameters. One trial per combination is run. *Panel 2*: Additional grid search to better refine batch_size and init_lr. 200 trials are run and there are 25 trials per combination of parameters. For the reported columns, n is the number of runs. q05, q95 and q975 are the 5%, 95% and 97.5% quantiles. The bold values were chosen for subsequent runs. Refer to Table 4.1 for the definition of the hyperparemeters.

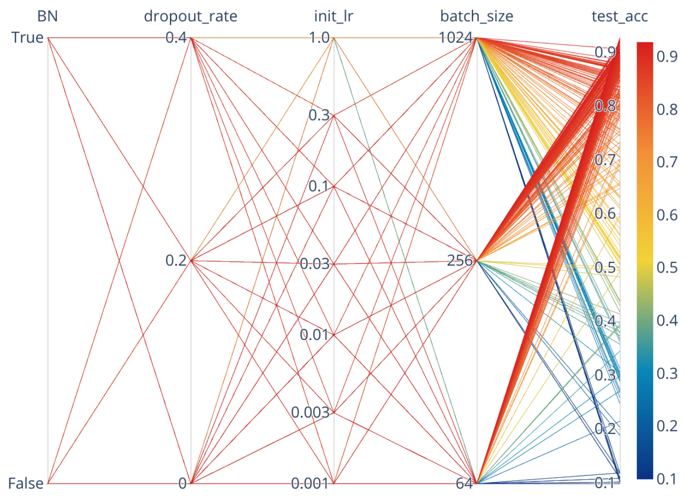
The results of hyperparameter tuning on NatSign is reported in Table B.5 and B.6 for the non-evolutionary and evolutionary hyperparameters respectively. The individual runs are visualized in Figure B.2.

For the non-evolutionary hyperparameters, note that an additional run was performed to better refine batch_size and init_lr. The need of this additional run stemmed from the much narrower range of the distribution of model accuracy, which made the optimal hyperparameter values much more difficult to be distinguished. While this could also be done in a single step by using a more refined search grid on the hyperparameters, the number of combinations to be tested would be considerably greater than the approach adopted. The main deviation from the results of the synthetic datasets is that *batch_normalization* now improves model accuracy, which is an empirically expected feature for image data.

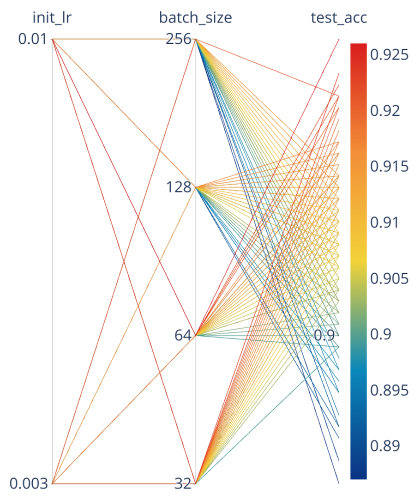
The results of evolutionary hyperparameters exhibit consistent trend over LE, RLE, NE and RNE. The evolutionary hyperparameters selected for the formal experiment are $\zeta = .25$ and std_evo_scale = .25.

algo	hyperparameter	value	n	min	q05	median	q95	q975	max	avg	std
LE	zeta	**.25**	20	.8700	.8700	.8970	.9090	.9090	.9110	.8974	.0086
		.5	20	.6460	.6460	.8970	.9070	.9070	.9080	.8764	.0581
		.75	20	.6100	.6100	.8900	.9040	.9040	.9080	.8673	.0653
	std_evo_scale	**.25**	12	.8960	.8960	.9000	.9040	.9040	.9050	.8999	.0027
		.5	12	.8880	.8880	.8970	.9080	.9080	.9080	.8987	.0063
		1	12	.8900	.8900	.9000	.9090	.9090	.9110	.9008	.0068
		2	12	.8090	.8090	.8850	.9000	.9000	.9060	.8799	.0246
		3	12	.6100	.6100	.8620	.8960	.8960	.8970	.8224	.0920
RLE	zeta	**.25**	20	.8700	.8700	.8960	.9100	.9100	.9120	.8952	.0099
		.5	20	.8330	.8330	.8950	.9130	.9130	.9140	.8909	.0189
		.75	20	.8160	.8160	.8940	.9080	.9080	.9120	.8830	.0266
	std_evo_scale	**.25**	12	.8880	.8880	.9040	.9120	.9120	.9140	.9021	.0087
		.5	12	.8920	.8920	.9020	.9070	.9070	.9130	.9021	.0054
		1	12	.8900	.8900	.8970	.9080	.9080	.9100	.8988	.0060
		2	12	.8530	.8530	.8830	.8980	.8980	.8990	.8814	.0153
		3	12	.8160	.8160	.8700	.8900	.8900	.8910	.8641	.0238
NE	zeta	**.25**	20	.8590	.8590	.8810	.8890	.8890	.8900	.8785	.0099
		.5	20	.8170	.8170	.8770	.8930	.8930	.8930	.8726	.0163
		.75	20	.7650	.7650	.8760	.8910	.8910	.8920	.8655	.0304
	std_evo_scale	**.25**	12	.8740	.8740	.8850	.8920	.8920	.8930	.8857	.0055
		.5	12	.8700	.8700	.8840	.8890	.8890	.8930	.8842	.0054
		1	12	.8700	.8700	.8780	.8860	.8860	.8880	.8796	.0053
		2	12	.8510	.8510	.8650	.8780	.8780	.8800	.8675	.0082
		3	12	.7650	.7650	.8570	.8710	.8710	.8740	.8441	.0306
RNE	zeta	**.25**	20	.8590	.8590	.8900	.9020	.9020	.9070	.8884	.0106
		.5	20	.8040	.8040	.8880	.9020	.9020	.9030	.8812	.0241
		.75	20	.8350	.8350	.8810	.8990	.8990	.9010	.8796	.0177
	std_evo_scale	**.25**	12	.8850	.8850	.8950	.9010	.9010	.9020	.8943	.0056
		.5	12	.8890	.8890	.8920	.9030	.9030	.9070	.8947	.0056
		1	12	.8800	.8800	.8900	.8990	.8990	.9020	.8903	.0073
		2	12	.8460	.8460	.8800	.8860	.8860	.8870	.8768	.0106
		3	12	.8040	.8040	.8660	.8780	.8780	.8960	.8591	.0244

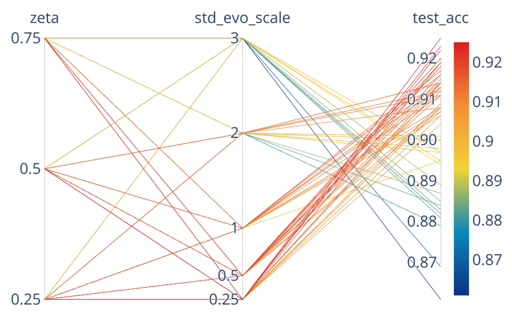
Table B.6: Tuning evolutionary hyperparameters on the downsampled MNIST dataset based on the same model and optimal hyperparemeters chosen from Table B.5. A simple grid search was performed on 15 combinations out of 2 parameters. 4 trials per combination are run. Other illustrations are the same as Table B.5.



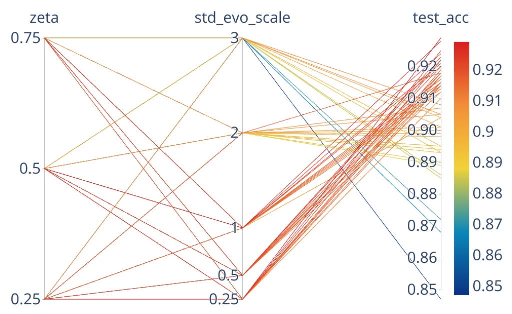
(a) Round 1



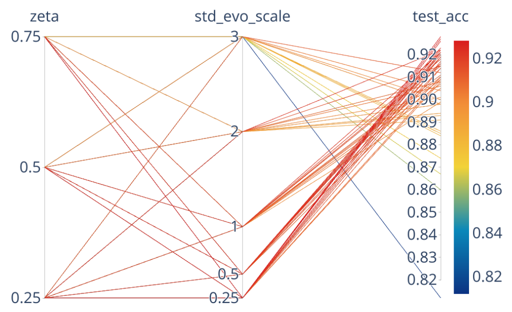
(b) Refined Round 1



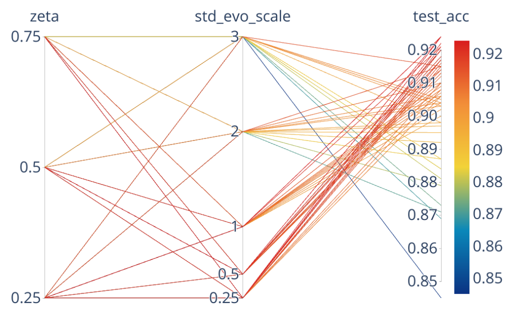
(c) Round 2 (LE)



(d) Round 2 (RLE)



(e) Round 2 (NE)



(f) Round 2 (RNE)

Figure B.2: Visualization of hyperparameter tuning on downsampled MNIST. All models bore a single hidden layer of 16 neurons. Model accuracies evaluated using test data are mapped to the test_acc axis and colorized. Subfigure (a): the first round of tuning on non-evolutionary hyperparameters (BN=batch_normalization). Subfigure (b): the refined first round on init_lr and batch_size. Subfigure (c)-(f): the second round of tuning on evolutionary hyperparameters for algorithm LE, RLE, NE and RNE. Refer to Section 4.2.1 for descriptions of the tuning procedure.

Appendix C

Supplementary Experiments

Experiments in addition to the main results were performed and reported in this section. The effect of network size was investigated for the ZigZag dataset using a smaller (8) and a greater (16×16) size compared to that of the main experiment (16) in Section 4.3.1. Summary statistics of hyperparameter tuning runs, parallel coordinate plots and summary statistic of the formal runs are presented.

C.1 ZigZag (8 Hidden Neurons)

Based on the success of the proposed evolutionary algorithms on ZigZag with 16 neurons, a follow-up question arose: *Would the effectiveness of the redesigned SET algorithms become greater for an even smaller network?* Thus, an additional experiment on a network with 8 hidden neurons was performed. The size chosen was very close to the theoretical minimal optimal network size (7) as discussed in Appendix A.1. The choice also resembled the fact that one is very unlikely to pick an exact size that equals to the theoretical minimum optimum size in a real case.

The results of hyperparameter tuning is reported in Table C.1 and Table C.2 for non-evolutionary and evolutionary parameters respectively. The individual runs are visualized in Figure C.1. To conserve experimental time, the evolutionary parameters were tuned in a slightly modified way. First, only random evolution algorithms were tuned. The non-random evolution algorithms will adopt the same choice of hyperparameters for their randomized versions.[1] Second, init_lr was re-tuned as a confirmation of the choice of the selected value. Third, std_evo_scale = 3 made the median of R^2 slightly reduced than std_evo_scale = 2 as found in a separate experiment (unreported), hence std_evo_scale > 2 was not considered.

The main findings from a model with 16 hidden neurons indeed holds for this even smaller model model. The result of the main experiment is reported in Table C.3. NI still performed worse than plain training. The median R^2 of the evolutionary algorithms outperformed the 97.5% quantile of plain training and was again able to reach the best fit ($R^2 > .99$) occasionally. The frequency of the best fit, however, is greatly reduced when compared to

[1]This was based on similarity between the results found in the past experience. Luckily, the conclusions were quite robust to the choice of hyperparameters and no serious problems appeared.

hyperparameter	value	n	min	q05	median	q95	q975	max	avg	std
batch_size	16	240	-.0194	.0413	.1194	.3345	.3860	.7790	.1381	.1074
	32	240	.0234	.0492	.1291	.4332	.5768	.8903	.1658	.1480
	64	240	-.0202	.0485	.1206	.3514	.5507	.7804	.1552	.1296
	128	240	-.0206	.0429	.1120	.3337	.3952	.7651	.1380	.0984
	256	240	-.0891	.0071	.0670	.3472	.4181	.5247	.1047	.1038
batch_normalization	True	600	-.0891	.0162	.1095	.3076	.3860	.7651	.1307	.1009
	False	600	-.0161	.0415	.1023	.4161	.5022	.8903	.1500	.1370
dropout_rate	**0**	400	-.0891	.0488	.1050	.5247	.7651	.8903	.1946	.1761
	.2	400	-.0206	.0333	.1122	.2373	.2477	.2617	.1242	.0699
	.5	400	-.0553	.0120	.0991	.1861	.1878	.2008	.1022	.0563
init_lr	.1	240	-.0202	.0456	.1682	.3698	.4400	.8749	.1805	.1199
	.03	240	.0202	.0529	.1786	.5507	.7236	.8596	.2130	.1537
	.01	240	-.0202	.0229	.1336	.3854	.4598	.8903	.1514	.1236
	.003	240	-.0346	.0213	.0697	.1828	.2180	.3019	.0855	.0505
	.001	240	-.0891	.0248	.0635	.1382	.2054	.3572	.0714	.0474
weight_initializer	**glorot_uniform**	600	-.0891	.0370	.1192	.3952	.5170	.8903	.1554	.1321
	random_uniform	600	-.0232	.0237	.0864	.3254	.4161	.7788	.1253	.1059
bias_initializer	**zeros**	600	-.0891	.0359	.1327	.4167	.4732	.7831	.1567	.1259
	random_normal	600	-.0553	.0270	.0862	.3248	.3680	.8903	.1240	.1129
patience	5	600	-.0891	.0274	.0991	.3284	.4400	.8903	.1359	.1181
	10	600	-.0232	.0317	.1133	.3854	.4997	.8749	.1448	.1231

Table C.1: Tuning non-evolutionary hyperparameters on the ZigZag dataset with an MLP model bearing a single hidden layer of 8 neurons. A simple grid search was performed on 1,200 combinations out of 7 parameters. One trial per combination was run. The main figures reported stand for the accuracy metric, R^2. For the reported columns, n is the number of runs. q05, q95 and q975 are the 5%, 95% and 97.5% quantiles. The bold values were chosen for subsequent runs. Refer to Table 4.1 for the definition of the hyperparemeters.

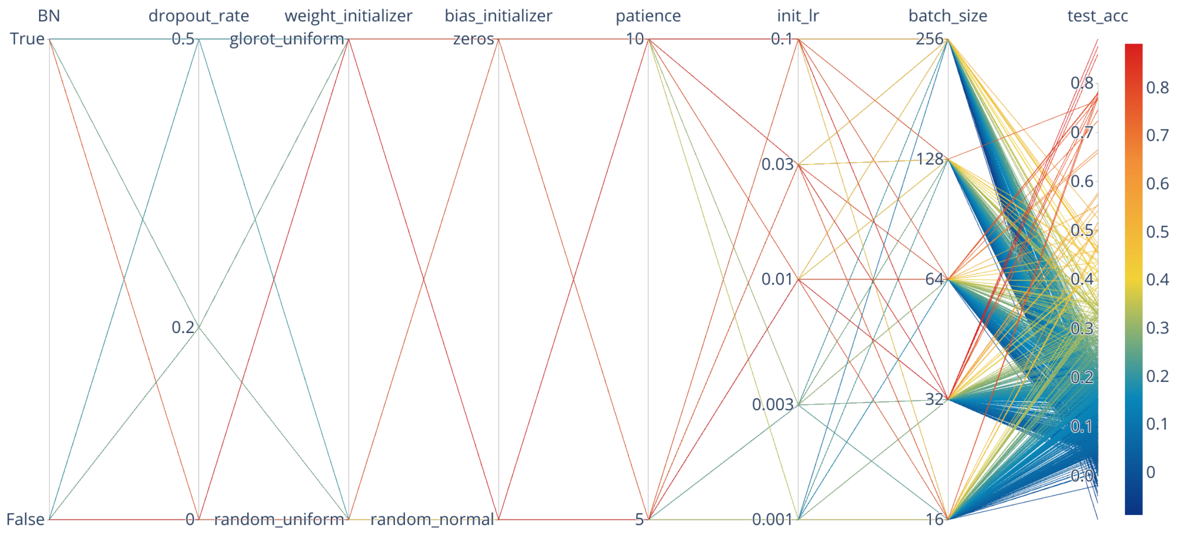
the 16-hidden-neuroned model (less than 10% vs. at least 54%). This is understandable due to the empirical fact that a small network is harder to train. Nevertheless, the evolutionary algorithms were still able to produce not only improved fitting results on average but also some opportunity to reach the global optimum fit where plain training did not stand a chance. The results from such a network again provided strong evidence on the potential of the proposed evoltionary algorithms.

algo	hyperparameter	value	n	min	q05	median	q95	q975	max	avg	std	n_best	best_rate
RLE	init_lr	1	36	-.0402	-.0401	.0741	.1919	.2033	.3198	.0594	.0997	0	0.0%
		.3	36	.0869	.0873	.3874	.8722	.8805	.8815	.3638	.2238	0	0.0%
		.1	36	.4126	.4148	.8434	1.0000	1.0000	1.0000	.7539	.2051	8	22.2%
		.03	36	.4379	.5792	.8810	1.0000	1.0000	1.0000	.8717	.1185	7	19.4%
		.01	36	.5853	.6512	.8792	.9918	.9997	1.0000	.8570	.0941	3	8.3%
	zeta	.25	60	-.0399	-.0398	.6518	1.0000	1.0000	1.0000	.5573	.3692	5	8.3%
		.5	60	-.0402	-.0398	.6515	1.0000	1.0000	1.0000	.5966	.3620	10	16.7%
		.75	60	-.0400	-.0398	.7564	.9885	1.0000	1.0000	.5895	.3355	3	5.0%
	std_evo_scale	.5	45	-.0402	-.0398	.4647	.8816	.8823	.9831	.4566	.3357	0	0.0%
		1	45	-.0401	-.0399	.7668	.9003	.9989	1.0000	.5696	.3577	3	6.7%
		1.5	45	-.0399	-.0398	.8145	1.0000	1.0000	1.0000	.6186	.3646	7	15.6%
		2	45	-.0398	.0759	.8722	1.0000	1.0000	1.0000	.6798	.3278	8	17.8%
RNE	init_lr	1	36	-.0866	-.0235	-.0234	.3109	.3119	.3189	.0661	.1192	0	0.0%
		.3	36	.0756	.1374	.3173	.8779	.8779	1.0000	.4124	.2469	1	2.8%
		.1	36	.3031	.3050	.5341	.8780	.8791	1.0000	.6417	.1992	1	2.8%
		.03	36	.5135	.5325	.8780	1.0000	1.0000	1.0000	.8637	.1249	7	19.4%
		.01	36	.5328	.6442	.8746	1.0000	1.0000	1.0000	.8694	.1034	8	22.2%
	zeta	.25	60	-.0235	-.0234	.6548	.9999	1.0000	1.0000	.5436	.3682	5	8.3%
		.5	60	-.0866	-.0234	.5858	.9997	1.0000	1.0000	.5891	.3408	5	8.3%
		.75	60	-.0234	-.0234	.5848	.9962	1.0000	1.0000	.5792	.3282	7	11.7%
	std_evo_scale	.5	45	-.0866	-.0234	.4577	.8781	.8784	.9133	.4544	.3341	0	0.0%
		1	45	-.0235	-.0234	.5325	.9999	1.0000	1.0000	.5310	.3425	4	8.9%
		1.5	45	-.0234	-.0234	.8537	.9996	1.0000	1.0000	.6343	.3437	7	15.6%
		2	45	-.0234	-.0234	.8451	.9997	1.0000	1.0000	.6629	.3257	6	13.3%

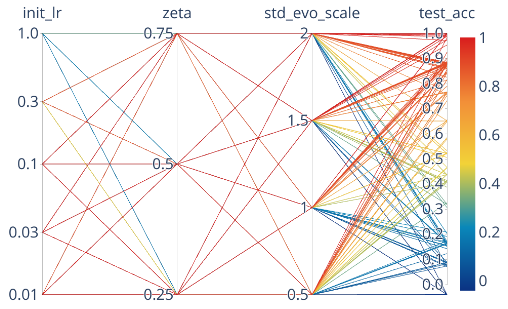
Table C.2: Tuning evolutionary hyperparameters on the ZigZag dataset based on the same model and optimal hyperparemeters chosen from Table C.1. A simple grid search was performed on 60 combinations out of 3 parameters. One trial per combination is run. Only random evolution algorithms were tuned because of the similarity of the choice of hyperparameters between random and non-random algorithms. init_lr is re-tuned as a confirmation of the choice of the selected value. Other illustrations are the same as Table C.1.

algo	n	min	q05	median	q95	q975	max	avg	std	n_best	best_rate
Plain	2,000	-.0330	.0466	.4377	.7830	.7833	.9584	.5121	.2417	0	.0%
NI	1,000	-.0227	-.0144	.3723	.7385	.7398	.8806	.4526	.2392	0	.0%
LE	100	.4605	.7454	.8624	.9818	.9953	.9998	.8520	.0742	5	5.0%
RLE	100	.7425	.7715	.8655	.9919	.9977	.9998	.8668	.0626	6	6.0%
NE	100	.4397	.7479	.8538	.9757	.9885	.9993	.8435	.0799	3	3.0%
RNE	100	.7022	.7249	.8457	.9959	.9975	.9995	.8385	.0773	7	7.0%

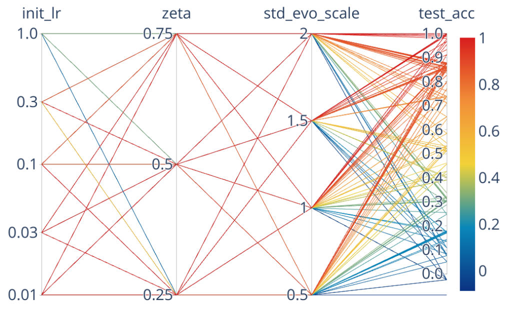
Table C.3: Summary statistics of the R^2 values of models trained by the proposed algorithms on the ZigZag dataset. All models bore a single hidden layer of 8 neurons. For the reported columns, n is the number of runs. q05, q95 and q975 are the 5%, 95% and 97.5% quantiles. n_best is the number of models achieving $R^2 > .99$ and best_rate = n_best/n. All models were trained with batch_size = 32, init_lr = .03, $\zeta = .5$, std_evo_scale = 2 and other optimal settings from hyperparameter tuning. Refer to Section 3.1 for the definition of algorithms and Section 4.2 for the experimental setup.



(a) Round 1



(b) Round 2 (RLE)



(c) Round 2 (RNE)

Figure C.1: Visualization of hyperparameter tuning on ZigZag. All models bore a single hidden layer of 8 neurons. Model accuracy values (R^2) evaluated using test data are mapped to the test_acc axis and colorized. Subfigure (a): the first round of tuning on non-evolutionary hyperparameters (BN=batch_normalization). Subfigure (b)(c): the second round of tuning on evolutionary hyperparameters respectively for RLE and RNE. Tuning of LE and NE were skipped because the results were assumed to be similar to their randomized versions based on past experience. Refer to Section 4.2.1 for descriptions of the tuning procedure.

C.2 ZigZag (16x16 Hidden Neurons)

hyperparameter	value	n	min	q05	median	q95	q975	max	avg	std	n_best	best_rate
batch_size	16	60	-.0431	-.0407	.7889	1.0000	1.0000	1.0000	.6175	.3844	12	2.0%
	32	60	-.0452	.0106	.7887	1.0000	1.0000	1.0000	.6854	.2824	6	1.0%
	64	60	-.0391	.1235	.7888	.9971	1.0000	1.0000	.7088	.2516	4	6.7%
	128	60	-.0403	.0593	.7880	.9702	.9715	1.0000	.6725	.2642	2	3.3%
	256	60	-.0385	.0780	.7878	.9570	.9879	1.0000	.6266	.2731	2	3.3%
init_lr	.3	50	-.0431	-.0409	.2521	.5971	.6126	.8507	.2479	.2158	0	0.0%
	.1	50	-.0452	.0498	.4861	.9544	.9702	1.0000	.4856	.3048	2	4.0%
	.03	50	.3860	.3866	.8882	1.0000	1.0000	1.0000	.8505	.1610	12	24.0%
	.01	50	.4353	.7866	.8103	1.0000	1.0000	1.0000	.8431	.1099	8	16.0%
	.003	50	.4284	.7871	.8045	.9956	.9960	.9999	.8308	.0859	4	8.0%
	.001	50	.0678	.2885	.7886	.9193	.9250	.9335	.7150	.2061	0	0.0%

Table C.4: Tuning non-evolutionary hyperparameters on the ZigZag dataset with an MLP model bearing double hidden layers of 16×16 neurons. A simple grid search was performed on 30 combinations out of 2 parameters. 10 trials per combination were run. The main figures reported stand for the accuracy metric, R^2. For the reported columns, n is the number of runs. q05, q95 and q975 are the 5%, 95% and 97.5% quantiles. n_best is the number of models achieving $R^2 > .99$ and best_rate = n_best/n. Refer to Table 4.1 for the definition of the hyperparemeters.

The question arose in Appendix C.1 could also be stated in a reversed manner: *Will the effectiveness of evolutionary algorithms still be eminent with multiple hidden layers?* Thus, an experiment using a double-hidden-layered MLP was performed to investigate whether the main findings could be generalized. The network structure chosen contained 16 neurons in each layer and this will be denoted as a "16×16" network hereafter. The dataset being used is still ZigZag.

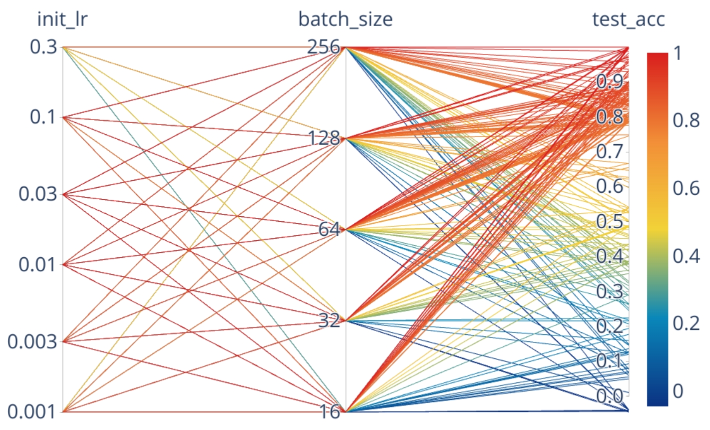
The results of hyperparameter tuning is reported in Table C.4 and C.5 for non-evolutionary and evolutionary parameters respectively. The individual runs are visualized in Figure C.2. The number of non-evolutionary parameters to be tuned was reduced to conserve execution time: batch_normalization = false, dropout_rate = 0 and patience = 5 were fixed based on prior experience. Only batch_size and init_lr were tuned. To further increase the stability of the tuning result, the number of trials per combination of hyperparameters was increased to 10 in response of the reduced total number of combinations. The non-evolutionary hyperparameters chosen are batch_size = 32 and init_lr = .01 upon consideration of the balance between stability and performance. The evolutionary hyperparameters chosen are $\zeta = .25$ and std_evo_scale = .25 for all evolutionary algorithms.

The main result of the experiment is reported in Table C.6. The effectiveness of the 97.5% quantile of plain and NI as well as the medians of the evolutionary algorithms all achieved the best fit. This implied that a 16×16 network was too easy to train for this dataset and therefore not very suitable for benchmarking between the proposed algorithms. However, the evolutionary algorithms still found an edge over plain training in terms of perfect stability. On the other hand, NI again performed worse than plain training in terms of both the median R^2 (.8707 vs. .8971) and stability (std = .1154 vs. .0965). The results were consistent with the main ZigZag experiment in Section 4.3.1.

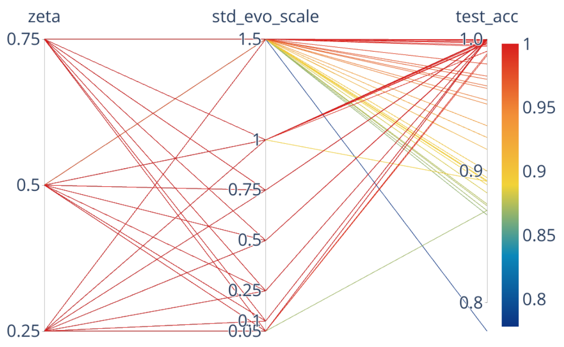
The finding in this experiment suggests that ZigZag may have very limited capability for benchmarking MLPs with sizes greater than 16×16. To further extend the use of the proposed algorithms, it is highly possible that either a suitable real-world dataset or a systematic way of generating suitable datasets will have to be found.

algo	hyperparameter	value	n	min	q05	median	q95	q975	max	avg	std	n_best	best_rate
LE	zeta	**.25**	70	.8928	.9537	1.0000	1.0000	1.0000	1.0000	.9952	.0198	66	94.3%
		.5	70	.7785	.8736	1.0000	1.0000	1.0000	1.0000	.9845	.0418	59	84.3%
		.75	70	.8701	.8921	.9999	1.0000	1.0000	1.0000	.9895	.0304	60	85.7%
	std_evo_scale	.05	30	.8701	.9876	1.0000	1.0000	1.0000	1.0000	.9949	.0234	27	90.0%
		.1	30	.9998	.9999	1.0000	1.0000	1.0000	1.0000	1.0000	.0000	30	100.0%
		.25	30	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	.0000	30	100.0%
		.5	30	.9996	.9999	1.0000	1.0000	1.0000	1.0000	1.0000	.0001	30	100.0%
		.75	30	.9998	.9998	1.0000	1.0000	1.0000	1.0000	.9999	.0001	30	100.0%
		1	30	.8922	.9908	.9997	1.0000	1.0000	1.0000	.9957	.0193	29	96.7%
		1.5	30	.7785	.8667	.9508	.9991	.9996	.9998	.9376	.0562	9	30.0%
RLE	zeta	**.25**	70	.9982	.9994	1.0000	1.0000	1.0000	1.0000	.9999	.0003	70	100.0%
		.5	70	.9566	.9884	1.0000	1.0000	1.0000	1.0000	.9980	.0074	66	94.3%
		.75	70	.9299	.9879	1.0000	1.0000	1.0000	1.0000	.9977	.0092	65	92.9%
	std_evo_scale	.05	30	.9879	.9939	1.0000	1.0000	1.0000	1.0000	.9991	.0025	29	96.7%
		.1	30	.9969	.9996	1.0000	1.0000	1.0000	1.0000	.9999	.0005	30	100.0%
		.25	30	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	.0000	30	100.0%
		.5	30	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	.0000	30	100.0%
		.75	30	.9998	.9999	1.0000	1.0000	1.0000	1.0000	1.0000	.0000	30	100.0%
		1	30	.9976	.9995	.9999	1.0000	1.0000	1.0000	.9998	.0004	30	100.0%
		1.5	30	.9299	.9566	.9985	.9999	.9999	1.0000	.9910	.0161	22	73.3%
NE	zeta	**.25**	70	.9413	.9976	1.0000	1.0000	1.0000	1.0000	.9988	.0071	68	97.1%
		.5	70	.8707	.9346	1.0000	1.0000	1.0000	1.0000	.9934	.0244	65	92.9%
		.75	70	.8813	.9487	1.0000	1.0000	1.0000	1.0000	.9936	.0226	63	90.0%
	std_evo_scale	.05	30	.9746	1.0000	1.0000	1.0000	1.0000	1.0000	.9992	.0046	29	96.7%
		.1	30	.9999	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	.0000	30	100.0%
		.25	30	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	.0000	30	100.0%
		.5	30	.9999	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	.0000	30	100.0%
		.75	30	.9998	.9998	1.0000	1.0000	1.0000	1.0000	1.0000	.0000	30	100.0%
		1	30	.9885	.9978	.9998	1.0000	1.0000	1.0000	.9993	.0021	29	96.7%
		1.5	30	.8707	.8813	.9914	.9995	.9998	.9999	.9683	.0432	18	60.0%
RNE	zeta	**.25**	70	.8976	.9971	1.0000	1.0000	1.0000	1.0000	.9976	.0133	68	97.1%
		.5	70	.8811	.9871	1.0000	1.0000	1.0000	1.0000	.9970	.0144	64	91.4%
		.75	70	.8590	.8981	1.0000	1.0000	1.0000	1.0000	.9895	.0290	59	84.3%
	std_evo_scale	.05	30	.8590	.8981	1.0000	1.0000	1.0000	1.0000	.9903	.0311	27	90.0%
		.1	30	.9952	.9996	1.0000	1.0000	1.0000	1.0000	.9998	.0009	30	100.0%
		.25	30	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	.0000	30	100.0%
		.5	30	.9999	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	.0000	30	100.0%
		.75	30	.9998	.9999	1.0000	1.0000	1.0000	1.0000	1.0000	.0000	30	100.0%
		1	30	.9644	.9960	.9998	1.0000	1.0000	1.0000	.9985	.0064	29	96.7%
		1.5	30	.8811	.8839	.9892	.9999	.9999	.9999	.9745	.0373	15	50.0%

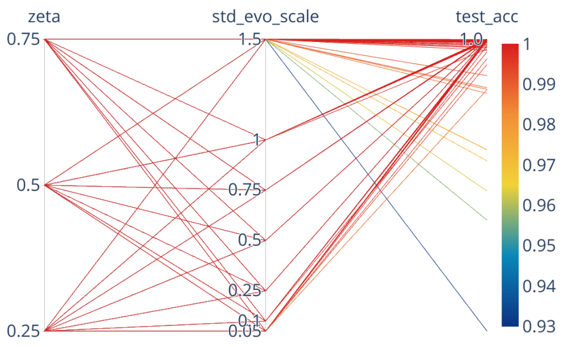
Table C.5: Tuning evolutionary hyperparameters on the ZigZag dataset based on the same model and optimal hyperparemeters chosen from Table C.4. A simple grid search was performed on 21 combinations out of 2 parameters. 10 trials per combination were run. Other illustrations are the same as Table C.4.



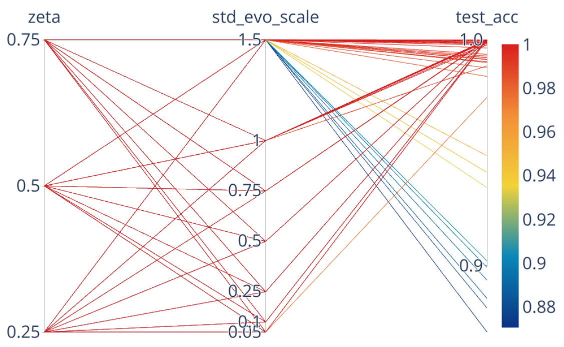
(a) Round 1



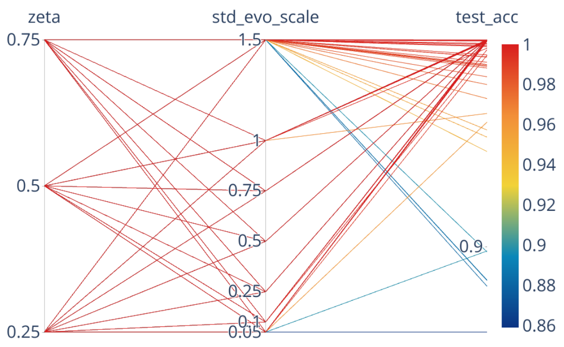
(b) Round 2 (LE)



(c) Round 2 (RLE)



(d) Round 2 (NE)



(e) Round 2 (RNE)

Figure C.2: Visualization of hyperparameter tuning on ZigZag. All models bore two hidden layers with 16 neurons per layer. Model accuracy values (R^2) evaluated using test data were mapped to the test_acc axis and colorized. Subfigure (a): the first round of tuning on non-evolutionary hyperparameters. Subfigure (b)-(e): the second round of tuning on evolutionary hyperparameters for algorithm LE, RLE, NE and RNE. Refer to Section 4.2.1 for descriptions of the tuning procedure.

algo	n	min	q05	median	q95	q975	max	avg	std	n_best	best_rate
Plain	2,000	0.3665	0.7733	0.8971	1.0000	1.0000	1.0000	0.8983	0.0965	729	36.5%
NI	1,000	0.3218	0.7499	0.8707	1.0000	1.0000	1.0000	0.8649	0.1154	324	32.4%
LE	100	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.0000	100	100.0%
RLE	100	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.0000	100	100.0%
NE	100	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.0000	100	100.0%
RNE	100	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.0000	100	100.0%

Table C.6: Summary statistics of the R^2 values of models trained by the proposed algorithms on the ZigZag dataset. All models bore double hidden layers of 16×16 neurons. For the reported columns, n is the number of runs. q05, q95 and q975 are the 5%, 95% and 97.5% quantiles. n_best is the number of models achieving $R^2 > .99$ and best_rate = n_best/n. All models are trained with batch_size = 32, init_lr = .01, $\zeta = .25$, std_evo_scale = .25 and other optimal settings from hyperparameter tuning. Refer to Section 3.1 for the definition of algorithms and Section 4.2 for the experimental setup.