EvoDNN: An Evolutionary Deep Neural Network with Heterogeneous Activation Functions

by

Peiyu Cui

B.ASc. (EE), University of British Columbia, 2017

Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in the School of Computing Science Faculty of Applied Sciences

© Peiyu Cui 2019
SIMON FRASER UNIVERSITY
Fall 2019

Copyright in this work rests with the author. Please ensure that any reproduction or re-use is done in accordance with the relevant national copyright legislation.
Approval

Name: Peiyu Cui
Degree: Master of Science (Computing Science)
Title: EvoDNN: An Evolutionary Deep Neural Network with Heterogeneous Activation Functions

Examining Committee:
Chair: Mo Chen
      Assistant Professor

      Kay C. Wiese
      Senior Supervisor
      Associate Professor

      Maxwell Libbrecht
      Supervisor
      Assistant Professor

      Fred Popowich
      External Examiner
      Professor
      School of Computing Science
      Simon Fraser University

Date Defended: November 27, 2019
Abstract

Many problems in Computational Biology and Bioinformatics involve classification, such as the classification of cell samples into malignant (cancer) or benign (normal). For such tasks, we propose EvoDNN, an evolutionary deep neural network that employs an evolutionary algorithm to evolve deep heterogeneous feed-forward neural networks. While the majority of current feed-forward neural networks employ user defined homogeneous activation functions, EvoDNN creates heterogeneous multi-layer networks where each neuron’s activation function is not statically defined by the user, but dynamically optimized during evolution. The main advantage offered by EvoDNN lies in that the activation functions do not need to be differentiable. This feature gives users a great degree of flexibility over which activation functions EvoDNN can utilize. This thesis demonstrates how EvoDNN can simultaneously optimize each neuron’s weight, bias, and activation function, and empirically shows a superior performance compared to feed-forward neural networks trained with backpropagation method, random forest method, and our earlier approach EvoNN which employed a single hidden layer.

Keywords: Model Combination; Deep Learning; Evolutionary Algorithm; Evolutionary Neural Networks; Heterogeneous Neural Networks; Evolving Neural Network Activation Functions; Classification and Learning in Computational Biology and Bioinformatics.
Acknowledgements

First and foremost, I would like to express my greatest gratitude towards my senior supervisor, Dr. Kay C. Wiese, for his constant support, patience and encouragement throughout my master study. He has taught me, both consciously and unconsciously, how good computer science is done. Without his help, this thesis could not have been completed. His dedicated academic contributions and humble personality also inspire me to continue pursuing a Ph.D degree.

Then I would also like to extend my gratitude to Dr. Maxwell Libbrecht, the members of my supervisory committee, for his helpful suggestions and insightful comments. Also, I have appreciated Dr. Fred Popowich to be the external examiner and Dr. Mo Chen to be the chair of my thesis defense.

In addition, I gratefully acknowledge Dr. Boris Shabash’s previous research that made my MSc. work possible.

Lastly, I would like to thank my family and friends for their love, encouragement, and support.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approval</td>
<td>ii</td>
</tr>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>iv</td>
</tr>
<tr>
<td>Table of Contents</td>
<td>v</td>
</tr>
<tr>
<td>List of Tables</td>
<td>viii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>ix</td>
</tr>
<tr>
<td>1 Preliminaries</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.3 Organization of The Thesis</td>
<td>2</td>
</tr>
<tr>
<td>2 Neural Networks</td>
<td>3</td>
</tr>
<tr>
<td>2.1 Deep Neural Networks</td>
<td>4</td>
</tr>
<tr>
<td>2.1.1 Feed-forward Neural Networks</td>
<td>5</td>
</tr>
<tr>
<td>2.1.2 Recurrent Neural Networks</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Activation Functions</td>
<td>5</td>
</tr>
<tr>
<td>2.3 Training Neural Networks</td>
<td>6</td>
</tr>
<tr>
<td>2.3.1 Zeroth-Order Optimization</td>
<td>6</td>
</tr>
<tr>
<td>2.3.2 First-Order Optimization</td>
<td>7</td>
</tr>
<tr>
<td>2.3.3 Second-Order Optimization</td>
<td>8</td>
</tr>
<tr>
<td>2.4 Heterogeneous Neural Network Training</td>
<td>9</td>
</tr>
<tr>
<td>2.4.1 A Self-Adaptive AF</td>
<td>9</td>
</tr>
<tr>
<td>2.4.2 Predetermined AF List</td>
<td>10</td>
</tr>
<tr>
<td>2.4.3 Combination of Self-Adaptive AF and Predetermined AF List</td>
<td>11</td>
</tr>
<tr>
<td>2.4.4 Summary</td>
<td>11</td>
</tr>
<tr>
<td>3 Methodology</td>
<td>12</td>
</tr>
</tbody>
</table>
## 3.1 EvoNN

- 3.1.1 EvoNN Architecture
- 3.1.2 Evolutionary Algorithm of EvoNN

## 3.2 EvoDNN

- 3.2.1 EvoDNN Architecture
- 3.2.2 Evolutionary Algorithm

## 4 Experimental Implementation

4.1 Pattern Classification Data Sets

- 4.1.1 Abalone Data Set
- 4.1.2 Breast Cancer Wisconsin (Diagnostic) Data Set
- 4.1.3 Iris Data Set
- 4.1.4 Lymphography Data Set
- 4.1.5 Tumor Data Set
- 4.1.6 Wine Data Set
- 4.1.7 Yeast Data Set

4.2 Evaluation Metrics

4.3 EvoNN Setup

4.4 EvoDNN Setup

4.5 Gradient Based FNN Setup

4.6 Random Forest Setup

4.7 Pre-processing, Language, and Hardware Specification

## 5 Experimental Results and Analysis

5.1 Optimization Performance

- 5.1.1 Abalone Data Set
- 5.1.2 Breast Cancer Data Set
- 5.1.3 Iris Data Set
- 5.1.4 Lymphography Data Set
- 5.1.5 Tumor Data Set
- 5.1.6 Wine Data Set
- 5.1.7 Yeast Data Set
- 5.1.8 Summary

5.2 Running Time Performance

- 5.2.1 Abalone Data Set
- 5.2.2 Breast Cancer Data Set
- 5.2.3 Iris Data Set
- 5.2.4 Lymphography Data Set
- 5.2.5 Tumor Data Set
- 5.2.6 Wine Data Set
5.2.7 Yeast Data Set ............................................. 44
5.2.8 Summary ................................................. 46
5.3 Analysis of Activation Functions in EvoDNN ................... 46
  5.3.1 5-layer EvoDNN ......................................... 48
  5.3.2 10-layer EvoDNN ........................................ 48
  5.3.3 20-layer EvoDNN ........................................ 48
  5.3.4 40-layer EvoDNN ........................................ 48
  5.3.5 Summary ................................................. 55

6 Discussion and Conclusion ........................................ 56

Bibliography ....................................................... 59
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 3.1</td>
<td>Set of Hidden Neuron’s Activation Functions</td>
<td>14</td>
</tr>
<tr>
<td>Table 4.1</td>
<td>Pattern Classification Data Sets</td>
<td>23</td>
</tr>
<tr>
<td>Table 4.2</td>
<td>EvoNN Algorithm Configuration Parameters</td>
<td>25</td>
</tr>
<tr>
<td>Table 4.3</td>
<td>EvoDNN Algorithm Configuration Parameters</td>
<td>26</td>
</tr>
<tr>
<td>Table 4.4</td>
<td>Gradient Based Algorithm Configuration Parameters</td>
<td>27</td>
</tr>
<tr>
<td>Table 4.5</td>
<td>Random Forest Algorithm Configuration Parameters</td>
<td>28</td>
</tr>
</tbody>
</table>
List of Figures

Figure 2.1  A Typical Three Layer Neural Network ............................................. 4
Figure 3.1  A Sample EvoNN Neuron ................................................................. 13
Figure 3.2  EvoNN Architecture ................................................................. 13
Figure 3.3  Crossover Process ................................................................. 15
Figure 3.4  Architecture of EvoDNN (Note the different activation functions in
the hidden neurons) ................................................................. 16
Figure 3.5  Framework of One Hidden Layer in EvoDNN ($a$ is the input data
from previous layer; $f_i()$ is the activation function associated with
each neuron; the summation $x_i$ goes through $f_i()$ to produce output
$y_i$) ................................................................. 17
Figure 5.1  Optimization Results of Evaluating Abalone Data Set .............. 30
Figure 5.2  Optimization Results of Evaluating Breast Cancer Data Set ...... 31
Figure 5.3  Optimization Results of Evaluating Iris Data Set .................. 32
Figure 5.4  Optimization Results of Evaluating Lymphography Data Set ... 33
Figure 5.5  Optimization Results of Evaluating Tumor Data Set .............. 34
Figure 5.6  Optimization Results of Evaluating Wine Data Set ............... 35
Figure 5.7  Optimization Results of Evaluating Yeast Data Set ................. 36
Figure 5.8  Run-time Result of Training Abalone Data Set ......................... 39
Figure 5.9  Run-time Result of Training Breast Cancer Data Set ............... 40
Figure 5.10 Run-time Result of Training Iris Data Set ......................... 41
Figure 5.11 Run-time Result of Training Lymphography Data Set ........... 42
Figure 5.12 Run-time Result of Training Tumor Data Set ....................... 43
Figure 5.13 Run-time Result of Training Wine Data Set ....................... 44
Figure 5.14 Run-time Result of Training Yeast Data Set ....................... 45
Figure 5.15 Analysis of Activation Functions in 5-layer EvoDNN ............. 47
Figure 5.16 Analysis of Activation Functions in 10-layer EvoDNN .......... 49
Figure 5.17 Analysis of Activation Functions in 20-layer EvoDNN ........... 50
Figure 5.18 Analysis of Activation Functions in 20-layer EvoDNN (Continue) . 51
Figure 5.19 Analysis of Activation Functions in 40-layer EvoDNN ........... 52
Figure 5.20 Analysis of Activation Functions in 40-layer EvoDNN (Continue) . 53
Figure 5.21 Percentage of Each AF in X-layer EvoDNN Architectures

x
Chapter 1

Preliminaries

1.1 Introduction

In this work we introduce a deep neural network (DNN) that learns its own activation function (AF) of each neuron through evolutionary algorithm while simultaneously learns its weights and biases. We use several datasets collected from Bioinformatics and Computational Biology areas, described in Chapter 4. This has been attempted by several groups but limited either with only gradient-based method available [2] or with a small set of predefined activation functions [52]. The model that we focus on learns interesting internal topology and it is also more accurate and confident than commonly used backpropagation methods [66] when making predictions.

1.2 Motivation

Neural networks have been employed for many classification tasks in Bioinformatics and Computational Biology. EvoDNN is a framework that employs an Evolutionary Algorithm (EA) to evolve the weights, biases and AFs of a deep heterogeneous feed-forward neural network (FNN) [87] (A neural network (NN) that consists of many hidden layers, often more than five, is called a deep neural network). EvoDNN extends our earlier framework EvoNN [70], which evolved a simple, single hidden layer FNN. The method proposed in EvoNN brought many advantages over the still popular stochastic gradient descent (SGD) based method: such as backpropagation [66]. These advantages include but are not limited to: being able to achieve global optima, being less sensitive to weight initialization, being adaptable to deep FNNs, and not requiring each neuron’s activation function to be differentiable.

In EvoNN, the authors [70] create a population pool in which each individual is a simple heterogeneous FNN. This simple heterogeneous FNN is composed of three layers: an input layer, a hidden layer, and an output layer. Each neuron in the preceding layer is fully connected to the succeeding layer. This population based training of neural networks
was also a key strategy when developing AlphaStar, who beat human in the real-time strategy game StarCraft II. Nonetheless, intuitively we would expect networks with many more hidden layers to be more powerful. Therefore, in EvoDNN we expand the three-layer heterogeneous FNN to a multi-layer deep heterogeneous FNN which contains an input layer, five, ten, twenty, or forty hidden layers, and an output layer. In addition, we add a bias term for each hidden neuron.

A significant advantage of EvoDNN is its ability to evolve the activation functions as well as the connection weights and biases simultaneously. Conventional methods of training neural networks, whether using back-propagation [66], or heuristic based training [62], have focused on either adjusting the weights and biases [58, 69] or the network’s architecture [3, 56]. However, a search of literature reveals that there has been comparatively little active research on dynamically optimizing neuron’s activation functions, with only a few notable exceptions [2, 73]. This is despite the fact that adjusting the activation function is as important as adjusting the architecture [18]. In Yao’s analysis [86], he identified that separating the evolution of architectures and weights and biases can cause fitness evaluation to mislead evolution, whereas simultaneous evolution of FNN architectures and weights and biases produces better results.

Finally, gradient descent methods struggle to train deep neural networks [29], caused by gradient vanishing and explosion [40], saddle points, and some other forces, which not only affect deep FNNs [87], but also recurrent networks [64]. However, the depth of the network has no impact on neural networks trained by EAs, because EAs are a zero-order, population-based method. This, coupled with the fact that deep neural networks are thought to be more efficient in terms of the number of neurons required to solve a problem [8], is another advantage of EvoDNN.

### 1.3 Organization of The Thesis

This thesis intends to help fill the gap of optimizing AFs as well as weights and biases simultaneously in a deep heterogeneous FNN by showing how EvoDNN can easily optimize these during evolution and doing so produces strong results.

The remainder of this paper is structured as follows. Chapter 2 discusses the overview of neural network and its applications. Chapter 3 describes the methodology undertaken using EAs to evolve heterogeneous deep FNNs, leading to the experimental implementation given in Chapter 4, and following by the experimental results and analysis demonstrated in Chapter 5. Finally, Chapter 6 discusses the overall findings with closing conclusions.
Chapter 2

Neural Networks

The fundamental concept of neural networks is that a neuron’s output involves a weighted sum of the input values. However, the neuron does not just output the weighted sum, instead there is a functional operation within the neuron performing on the weighted inputs. Often, this operation is a nonlinear function that make a neuron generate an output only if the weighted inputs reach over some thresholds, otherwise the computation from a cascade of neuron would then be a simple linear algebra operation. By analogy, this nonlinear function is called activation function.

Equation 2.1 shows an example of the computation at each layer:

\[ x^l_k = b^l_k + \sum_{i=1}^{N_{l-1}} w^{l-1}_{ik} y^{l-1}_i \]  

where

- \( x^l_k \) is the input of neuron \( k \) in layer \( l \),
- \( b^l_k \) is the bias into neuron \( k \) in layer \( l \),
- \( N_{l-1} \) is the number of neurons in layer \( l - 1 \),
- \( w^{l-1}_{ik} \) is the weight between neuron \( i \) in layer \( l - 1 \) and neuron \( k \) in layer \( l \), and
- \( y^{l-1}_i \) is the output of neuron \( i \) in layer \( l - 1 \).

The neuron then computes its output

\[ y^l_k = f(x^l_k) \]  

where \( f(\cdot) \) is a nonlinear function, or activation function of the neuron’s total input.

Fig 2.1 shows a simple structure of a computational neural network, which just stacks three layers. The neurons in the input layer receive some values and propagate them to the neurons in the middle layer, which is often referred as a "hidden layer." The weighted sums
from one or more hidden layers are finally propagated to the output layer. There is a cost function, measuring the discrepancy between the ground truth and the output produced by the system,

\[
E(y_1, \ldots, y_{NL})
\]  

(2.3)

associated with the output layer that we would like the neural network to minimize, where \( L \) is the number of layers in the neural network. The most commonly used cost function is the Mean Squared Error [80].

![A Typical Three Layer Neural Network](image)

Figure 2.1: A Typical Three Layer Neural Network

### 2.1 Deep Neural Networks

Within the domain of neural network, there is a research field called deep learning in which more than three layers are contained in a neural network (i.e., more than one hidden layer). Today, the typical number of network layers used in deep learning range from five to more than a thousand. In this thesis, we will generally use the terminology deep neural networks (DNNs) to refer to the neural network used in deep learning.

DNNs are capable of learning high-level features with more complexity and abstraction than shallower neural networks [65]. An example is to use DNN to process visual data task. The pixels of an image are fed into the first layer of a DNN, and the output from first layer can be interpreted as representing the presence of low-level features, such as lines and edge conjunctions. The subsequent layers then learn the likely presence of higher-level features from the combination of low-level features; e.g., lines are combined into shapes, which are further combined into sets of shapes. Finally, at the last layer the network provides a probability that these high-level feature comprise a particular object. The deep hierarchy enables DNNs to achieve superior performance in many tasks.
Although DNNs come in a wide variety of shapes and sizes depending on the applications, they can be divided into two major forms: feed-forward neural network and recurrent neural network.

2.1.1 Feed-forward Neural Networks

In feed-forward neural networks (FNNs), all of the computation is performed as a sequence of operations on the outputs of a previous layer, which implies the connections between neurons do not form feedback loops [29]. The final layer then generates the output of the network [68]; for example, a probability that an image contains a particular object, a bounding box around an object in an image, or a desired action that is recognized. Note that since data flows from input layer to output layer without feedback loops, FNNs having no memory implies that the output for an input is the same regardless of the sequence of inputs previously given to the network.

2.1.2 Recurrent Neural Networks

Recurrent neural networks (RNNs) are networks with feedback loops which allow data to be passed from one time-step of the network to the next [39]. In RNNs, intermediate operations generate values stored internally and used as inputs to other operations in conjunction with the processing of a later input. In other words, the network has internal memory to allow long-term dependencies to affect the output. A popular structure of RNNs, known as long short-term networks [39] uses memory cells in the structure of hidden neurons in which their inputs and outputs are controlled by gates, and these gates control flow of data to hidden neurons and preserve extracted features from previous time-steps.

2.2 Activation Functions

AFs are used to limit the signals propagated through the neural network. According to recent studies [68], statistical approaches of classification can divide AFs into three major categories: sigmoidal non-local functions, radial basis functions (RBFs), and semi-centralized functions (which either have many centers or one single center) [18].

Sigmoid functions, e.g. \( \frac{1}{1+e^{-x}} \), have non-local behavior. That is, for large input signals, they are non-zero in an infinite domain. It is also easy to calculate their derivatives. However, since calculation of exponents is much slower than simple arithmetic operations, other functions of sigmoidal shape may be used in large scale neural networks to speed up computation.

Radial basis function take the radial coordinate \( r = ||x-t|| \) for an activation, where \( t \) is the coordinate of the center. Networks based on radial functions are universal approximators [63].
Semi-centralized output functions operate on vector components of activation. For example, the Gaussian bar function proposed by Hartman et al. [34]:

$$\tilde{G}(r, b, v) = \sum_{i=1}^{N} v_i e^{-\frac{r^2}{b^2}} ; r_i = ||x_i - t_i||$$

(2.4)

where $t$ is the center, $b$ is the smoothness parameter. This function makes elimination of irrelevant inputs that do not contribute to the determination of the output values.

2.3 Training Neural Networks

Backpropagation is a main trend in neural network learning algorithm because it is conceptually simple and computationally efficient. However, when explicit expressions of the gradients are difficult or infeasible to obtain, zeroth-order (ZO) optimization is often embraced for solving machine learning problems. Second order algorithm is also an efficient technique for neural network training because of its fast convergence. Fast convergence means an algorithm requires less training time to reach a stable status where this algorithm’s performance cannot be improved significantly again.

Designing and training a network requires making many seemingly arbitrary choices; e.g., training methods, learning rates, training and test sets etc. These choices can be critical, yet there is no foolproof theory for deciding them since they are largely problem and data dependent. Still, there are some heuristics and underlying theories that can guide us to make better choices.

In the first sub-section we introduce ZO optimization and show their application to neural network training. We then describe standard backpropagation (first order optimization) and discuss a number of its significant applications. Finally, we present second-order algorithm and show a few applications that do accelerate learning.

2.3.1 Zeroth-Order Optimization

Many big data problems, such as network control and management with time-varying constraints and limited computation capacity [17, 50], parameter inference of black-box systems [26, 48, 51], and bandit optimization in which a player receives partial feedback [72, 71], fall into zeroth-order (gradient-free) optimization with respect to black-box models. This is because these problems with sophisticated data generating processes cannot be described by analytical forms but have function evaluations.

Zeroth-order (ZO) algorithm achieves gradient-free optimization by approximating the full gradient through efficient gradient estimators. A full gradient is typically approximated using either a one-point or a two-point gradient estimator, where the former acquires a gradient estimate by querying the objective (black-box) function $f(x)$ at a single random location to $x$ [23], and the latter computes a finite difference using two random function
queries [1, 61]. An excellent example of one-point gradient estimator is using evolutionary algorithm to adjust weights and biases of neural network, which is our topic in Chapter 2.4. However, the two-point gradient estimator has a lower variance than the one-point gradient estimator, and therefore has a better complexity bound of zeroth-order algorithms.

Many zeroth-order algorithms are developed for convex and non-convex optimization. However, most of these algorithms are limited to convex problems [50, 71, 20, 27, 21, 79]. For example, [20] proposes a zeroth-order mirror descent algorithm which has a convergence rate $O\left(\frac{\sqrt{d}}{\sqrt{T}}\right)$, where $d$ is the number of optimization while $T$ is the number of iterations. The same rate is also obtained from bandit convex optimization [71] and zeroth-order online alternating direction method of multipliers [50].

A large amount of recent attention has focused on non-convex zeroth-order optimization [48, 61, 33, 28, 32]. Other than the convex problems, a stationary condition is used to measure the convergence of non-convex algorithms. For example, the zeroth-order gradient descent (ZO-GD) algorithm [61] is proposed for deterministic non-convex programming, which has a convergence rate of $O\left(\frac{d}{T}\right)$. A stochastic version of ZO-GD, known as ZO-SGD, [28] achieves the convergence rate of $O\left(\frac{\sqrt{d}}{\sqrt{T}}\right)$. In [33], a zeroth-order distributed algorithm is utilized for multi-agent optimization, given a convergence rate of $O\left(\frac{1}{T} + \frac{d}{q}\right)$. In [48], an asynchronous zeroth-order stochastic coordinate descent (ZO-SCD) is developed for parallel optimization and led to the convergence rate of $O\left(\frac{\sqrt{d}}{\sqrt{T}}\right)$.

Recent researches indicate that zeroth-order algorithms agree with the iteration complexity of first-order algorithms to a small-degree polynomial of the problem size.

### 2.3.2 First-Order Optimization

Many of recent researches to automatic machine learning can be categorized as first-order gradient-based learning methods. The simplest form of neural network with first-order learning is simply a stack of layers. However, $f(\cdot)$ and $E(\cdot)$ in Equation 2.2 and 2.3 should be differentiable so $\frac{\partial E}{\partial y^L_k}$ is computable. This process consists of clamping the data from $E(\cdot)$ function and updating the parameters (the weights and biases) in the direction of gradient. That is, a multiple of the gradient of the loss relative to each weight, which is the partial derivative of the loss with respect to the weight, is used to update the weight (i.e., updated $w^{l+1}_{ik} = w^l_{ij} - \alpha(\frac{\partial L}{\partial w_{ik}})$, where $\alpha$ is called the learning rate [46]). The derivatives can be computed in detail as follows:

\[
\frac{\partial E}{\partial w^{l-1}_{ik}} = \frac{\partial E}{\partial y^l_i} y^{l-1}_i \tag{2.5}
\]

\[
\frac{\partial E}{\partial b^l_k} = \frac{\partial E}{\partial x^l_k} \tag{2.6}
\]

\[
\frac{\partial E}{\partial x^l_k} = \frac{\partial E}{\partial y^l_k} \frac{\partial y^l_k}{\partial x^l_k} \tag{2.7}
\]
\[
\frac{\partial E}{\partial y^k_l} = \begin{cases} 
\frac{\partial E}{\partial y^k_L} & \text{if } l = L \\
\sum_{i=1}^{N_l+1} \frac{\partial E}{\partial x^{i+1}_l} w_{ki} & \text{otherwise}
\end{cases}
\]

which operates by passing gradients backwards through the network based on the chain rule of calculus. Note that this gradient indicates how the weights and biases should change in order to reduce the loss. The process which repeats iteratively to reduce the overall loss is also known as backpropagation.

The popularity of backpropagation started from ImageNet Challenges [67]. One of the challenges is an image classification task where algorithms are given an image and they must identify what is in the image. The training set consists of 1.2 million images, each of which is labeled with one of the 1000 object categories. For the evaluation stage, the algorithms must accurately identify objects in a test set of images which they have not seen previously. In 2012, Alex et al. used GPUs and a backpropagation learning method dropping the error rate by approximately 10% [45]. In 2015, the ImageNet winning entry, ResNet [36], exceeded human-level accuracy with a top-5 rate below 5%.

Except image classification, first-order optimization have also achieved significantly improvement on speech recognition [38] as well as gaining insight into a variety of diseases, such as autism, cancers, and spinal muscular atrophy [84, 89, 4, 88].

However, it is important to note that backpropagation requires intermediate outputs of the network to be preserved for the backward gradient computation, which implies this method needs a potentially large allocation of memory.

### 2.3.3 Second-Order Optimization

In the following, we will briefly introduce several second-order optimization methods, also known as Hessian optimization, which includes Gauss-Newton, Levenberg Marquardt, and the Quasi-Newton method [7, 12, 24]. There are theoretical and empirical results showing Hessian optimization converge on a minimum in fewer steps then the first-order optimization, which we will discuss below. In particular, by incorporating information about second-order changes in the cost function, it is possible to avoid many traps that can occur in first-order optimization.

Gauss-Newton algorithm is a variant of Newton algorithm; the Hessian matrix of it is approximated by the square of the Jacobian [13]. This algorithm is invariant with respect to linear transformations of the input vectors [46]. This suggests the convergence time is not affected by shifts, scaling, and rotation of input vectors. However, one of the main drawbacks is that an \( N \times N \) matrix must be stored, which takes \( O(N^3) \) per iterations and is therefore impractical for more than a few parameters.

The Levenberg Marquardt method is similar to the Gauss-Newton, but it requires a regularization parameter \( \mu \) that prevents it from growing exponentially.
The Quasi-Newton method iteratively computes an estimate of the inverse Hessian; but it works only for batch learning and requires line search. An excellent variant of Quasi-Newton algorithm is the Broyden-Fletcher-Goldfarb-Shanno (BFGS) method [37]. However, we are still required to store a $N \times N$ matrix, thus this method is only practical for small networks. Some recent studies demonstrate a few techniques to reduce the storage requirements [7].

Techniques using full Hessian information (Gauss-Newton, Levenberg-Marquardt, BFGS) can only apply to very small networks. This is because of the sheer size of the Hessian matrix. Suppose we have a neural network with $10^5$ weights and biases, then the corresponding Hessian matrix will have $10^5 \times 10^5 = 10^{10}$ entries. This makes computing second-order gradients extremely difficult in practice. For smaller problems which require accurate real-valued outputs, such as function approximation, control problems, second-order optimization offer the best combination of speed, reliability, and simplicity.

2.4 Heterogeneous Neural Network Training

To our knowledge, the majority of zeroth-order algorithms only train homogeneous FNNs. In such FNNs which have been shown capable of universal approximation, all neurons contain the same AF; they are either Sigmoid, Tanh, Gaussian, or ReLU functions. For example, Irani et al. [42] proposed a hybrid genetic algorithm-neural network strategy (GA-ANN) that uses genetic algorithm (GA) to decide the initial weights and biases of the gradient decent method so that all the initial weights and biases can be searched intelligently; then they use backpropagation (BP) to optimize the network. Note that all neurons in their network use Sigmoid AF. Other examples can be found in [53, 69].

Of those which do train heterogeneous FNNs, there are three main methods. One requires an adaptive AF with parameters that is trained along with other factors, such as weights and biases. Another chooses the AF of each neuron from a predetermined list of AFs. The last one combines the benefits of the previous two methods. While an extensive review of works in this domain is beyond the scope of this document, in the following sub-sections we categorize them into two major directions and discuss some significant prior researches from these two directions.

2.4.1 A Self-Adaptive AF

This method requires that an activation function is an algebraic expression containing a set of free parameters [19]. These parameters control the shape of AFs and therefore optimizing these parameters for each individual neuron could potentially improve the prediction performance of neural networks in many tasks.

A simple but interesting application of this method has been used by Cartesian Genetic Programming of Artificial Neural Networks (CGPANN) [55], where the widths of Gaussian
functions are optimized for each neuron. That is, the genes representing the functions of the
hidden neurons are re-purposed to specify the standard deviation for Gaussian functions.
Therefore, instead of giving different functions, the genes essentially specify different Gauss-
ian function mappings. These genes are also subject to mutation and/or crossover during
evolution training.

A more complex and recent application of this method was employed in [2] where the
authors designed a novel form of adaptive piece-wise linear (APL) activation function (Eq.
2.9) that is learned independently for each neuron using stochastic gradient descent, and
can represent both convex and non-convex functions of the input.

\[
h_i(x) = \max(0, x) + \sum_{s=1}^{S} a_s^i \max(0, -x + b_s^i)
\]  
(2.9)

The APL has three free parameters; \( S \) is a hyper-parameter set in advance while \( a_s^i, b_s^i \) are
learned using SGD during training. When comparing maxout units [31] and network-in-
network (NIN) [49] to APL, the authors find APL requires less parameters to learn any
nonlinear AFs. This difference makes APL more practical to train FNN that apply different
non-linearities at each neuron. This approach is allowed for an almost limitless variation of
activation functions.

Augusteijn et al. [6] proposed a variant of evolutionary algorithm for constructing het-
erogeneous neural networks where each neuron’s AF was itself an evolved Genetic Program.
This method allows the construction of an almost limitless variation of AFs. While only one
single hidden layer was employed in their network construction, they found that functions
other than the commonly used sigmoidal function could perform well when used as hidden
layer AFs.

2.4.2 Predetermined AF List

The second method selects the AF of each neuron from a predetermined list of AFs. Having
AFs as many as possible in the list may potentially produce a network structure with better
prediction accuracy, while this may also exponentially increase the training time as the
searching space grows larger.

Training applications which use this method include General Neural Network (GNN)
[52] which was presented by Liu and Yao in 1996. The weights and biases of this network
are adjusted by means of a combination of the Backpropagation (BP) [47] algorithm with
a random search algorithm. However, the authors chose only to apply the sigmoid and
Gaussian functions as they represent two broad classes of AFs with complementary features.

Another interesting application was studied by Turner el al. [77] where they evolved the
topology of a predefined collection of neurons in order to optimize the FNNs. Each neuron
in this network contain a function gene that represents the index in a function look-up-table
and describes the functionality of the neuron. However, the resulting FNN does not have a clear division into layers since the connectivity can occur between any two neurons.

Similar work done by Weingaertner et al. [81], with the introduction of a modified Hierarchical Co-evolutionary Genetic Algorithm (HCGA), devised a genetic algorithm to determine the number of neurons in the hidden layer and the type of AF of those neurons. However, the predefined set of AFs is composed of Linear, Signal Step, Hyperbolic Tangent, Gaussian, and RBF-Green functions which are rarely used in modern neural networks.

Recently, Shirakawa et al. [73] have experimented with the simultaneous evolution of weights and AFs, along with the evolution of dropout and other hyper-parameters in deep FNNs. Their work produced better results as compared to using a homogeneous neural network; however, their set of AFs were limited to ReLu and Tanh, which implies the potential improvement on predication accuracy may be impressing.

Our previous work EvoNN [70] also investigates the simultaneous evolution of weights and AF. It demonstrates the superior performance of applying EA to a heterogeneous FNN; however, EvoNN only contains one hidden layer which limits its ability to make a high percentage of correct predictions on complex and large data sets as well as the parameter of bias is not included in each neuron.

### 2.4.3 Combination of Self-Adaptive AF and Predetermined AF List

There are also studies that combine the key concepts of the above two methods and develop new algorithms accordingly.

The authors in [78] devise a novel neuroevolution-based learning algorithm, known as topology and weight evolving artificial neural network (TWEANN), that evolves both the topology and weights. A neuron representation unifies most of the proposed neuron variations into one. For example, a neuron has its own adjustable reaction speed (i.e., some neurons may take longer to change their outputs), which will be determined in neural network training. In addition, it has four types of AFs: random, identity, threshold, and sigmoid. The results as shown in their paper are promising. However, their proposed network only consists of one hidden layer and it is also difficult to apply its methodology to deep neural networks because the searching space of hyper-parameters may grow exponentially.

### 2.4.4 Summary

Though the examples from these three main methods showed promising results for heterogeneous FNNs, most such networks only consist of less than five hidden layers. Intuitively, we would expect networks with many more hidden layers to be more powerful. However, until now there has been little research to investigate if the ability of evolutionary algorithm to evolve heterogeneous deep FNNs provides any benefit over gradient-based methods. This is important research since if shown to be beneficial it could be easily adopted to other neural network architectures, such as recurrent neural networks (RNNs) [82].
Chapter 3

Methodology

We will first discuss the architecture of *EvoNN* [70] that our current work is based on. Then we present our recent work *EvoDNN*, a framework that employs an Evolutionary Algorithm (EA) to evolve the weights, biases and activation functions of a deep heterogeneous FNN.

3.1 EvoNN

3.1.1 EvoNN Architecture

The basic unit of EvoNN is neuron, however it consists of only two components: the weight and the AF.

\[
y^l_k = f \left( \sum_{i=1}^{N_{l-1}} w^{l-1}_{ik} y^{l-1}_i \right)
\]

(3.1)

where

- \( N_{l-1} \) is the number of neurons in layer \( l - 1 \),
- \( w^{l-1}_{ik} \) is the weight between unit \( i \) in layer \( l - 1 \) and unit \( k \) in layer \( l \),
- \( y^{l-1}_i \) is the output of unit \( i \) in layer \( l - 1 \),
- \( f(\cdot) \) is the AF, and
- \( y^l_k \) is the neuron \( k \)'s output in layer \( l \).

Figure 3.1 illustrates this process.

Conventionally, the collection of neurons in a given neural network all employ the same AF. However, EvoNN allows different neurons to have different AF (Figure 3.2). The AF of each neuron becomes another parameter to be optimized.

Each neural network in EvoNN framework is composed of three fully connected (FC) layers (Figure 3.2): input layer, hidden layer, and output layer. In an FC layer, all output activations are composed of a weighted sum of all input activations (i.e., all outputs are
\[ y_l = f(\sum w_{l-1} y_{l-1}) \]

Figure 3.1: A Sample EvoNN Neuron

connected to all inputs). The size of input layer matches the number of attributes of a given dataset; the hidden layer only contains ten hidden neurons; and the size of output layer is the same as the number of classes we need to label. This simple three-layer network is also known as a shallow neural network[65].

Figure 3.2: EvoNN Architecture

The AF of EvoNN framework are chosen from a delimited candidate set which is presented in Table 3.1 below. It is important to emphasize that we can adjust one free parameter in the Leaky Rectified Linear Unit (LReLU) function [54]: \( b \) - a small, non-zero gradient when the unit is saturated and not active. To reduce the EA’s search space, we set \( b = 0.01 \) for EvoNN framework. This value \((b = 0.01)\) is well evaluated in [54].

3.1.2 Evolutionary Algorithm of EvoNN

The training phase of EvoNN consists of three stages: population generation, mutation, and crossover as discussed below:
### Table 3.1: Set of Hidden Neuron’s Activation Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Activation Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tanh</td>
<td>$\sigma(x) = \frac{2e^{x}}{e^{x} + 1} - 1$</td>
</tr>
<tr>
<td>Sigmoid</td>
<td>$\sigma(x) = \frac{1}{1 + e^{-x}}$</td>
</tr>
<tr>
<td>LReLU</td>
<td>$\sigma(x) = \begin{cases} b \cdot x &amp; x \leq 0 \ x &amp; x &gt; 0 \end{cases}$</td>
</tr>
<tr>
<td>ReLU</td>
<td>$\sigma(x) = \begin{cases} 0.0 &amp; x \leq 0 \ x &amp; x &gt; 0 \end{cases}$</td>
</tr>
</tbody>
</table>

### Population Generation

A population size of $N$ is generated in the first place. Each individual in the population is a shallow neural network which has two key components: the weight matrix and the AF index matrix. The entries in each matrix are randomly generated via uniform distribution in a preset range. Each individual will then be evaluated through multi-class logarithmic fitness function (fitness function is discussed in section 4.2), and the one with the best fitness value will be copied to next generation (elitist selection).

### Mutation

In the context of EvoNN, mutation is divided into two types of modifications, each controlled by different hyper-parameters. Weight mutation is the process of modifying the weight connecting two neurons, while AF mutation adjusts the AF installed in a given neuron.

Weight mutation is controlled by the mutation probability ($P_m \in [0.0, 1.0]$) and the mutation radius ($R_m \in (0.0, \infty)$). The mutation probability determines the chance for each weight to be mutated, while the mutation radius dictates the severity of the mutation. For example, if $P_m = 0.5$ and $R_m = 0.1$, then each weight get a 50% chance of being mutated at each generation, and if the weight is indeed chosen for mutation, its weight will increase by a value uniformly chosen from the range $[-0.1, 0.1]$. Note that the weight may drop if the chosen value falls between -0.1 and 0.0.

AF mutation carries the similar procedure to the weight mutation but is controlled by the function mutation probability ($P_{mf} \in [0.0, 1.0]$). Much like the weight mutation probability, the function mutation probability controls the likelihood an AF within a hidden neuron is replaced. However, this mutation does not have a mutation radius equivalent. If a given hidden neuron’s AF is chosen for mutation, the new AF is uniformly chosen from the predetermined AF list. Note that the new AF for that given neuron has to be different from the old one. For example, if $P_{mf} = 0.5$, then the given hidden neuron has a 50% chance
of having its AF mutated. If the hidden neuron which contains AF \( f(x) \) is chosen for this process and the predetermined AF list has the following AFs: \{\( f(x) \), \( g(x) \), \( h(x) \), \( i(x) \)\}, then the old AF \( f(x) \) is replaced by either \( g(x) \), \( h(x) \), or \( i(x) \), each with a 33.33% chance to be chosen.

**Crossover**

The topology of each shallow neural network remains constant during the crossover process. Figure 3.3 illustrates the crossover process of hidden layer in parents and child NNs. Red neuron in child NN means this neuron is inherited from first parent and the blue one means this neuron is inherited from second parent. First, the proportion of individuals in the offspring produced through crossover is controlled by the crossover proportion parameter \( (P_c \in [0.0, 1.0]) \). For example, if \( P_c = 0.3 \), then 30% of the entire population in the offspring generation will be the result of crossover.

If an individual is chosen to be produced by crossover, then its weights and AFs are processed in the following way. (The uniform crossover [76] is applied in this step.) Each neuron in the offspring has a 50% chance to receive its weight and corresponding AF from the first parent, and a 50% chance to receive its weight and corresponding AF from the second parent.

The resulting individual has, on average, 50% of the first parent’s genotype and 50% of the second parent’s genotype.
3.2 EvoDNN

EvoDNN is an extension of EvoNN with extra attributes and improved multi-logloss values. The methodology presented on evolving EvoDNN takes two parts. The first part describes the architecture of EvoDNN and the second part gives the procedure of applying the EA to evolve EvoDNN.

3.2.1 EvoDNN Architecture

The basic unit of EvoDNN is also the neuron. However, it is composed of three components: the weights, the biases, and the activation functions. These components are summed up by the following equation:

\[ y^l_k = f(b^l_k + \sum_{i=1}^{N_{l-1}} w^{l-1}_{ik} y^{l-1}_i) \]  

(3.2)

where we use \( y^l_k \) for the output of the \( k^{th} \) neuron in the \( l^{th} \) layer, \( b^l_k \) for the bias of the \( k^{th} \) neuron in the \( l^{th} \) layer, \( w^{l-1}_{ik} \) for the weight linking from the \( i^{th} \) neuron in the \((l-1)^{th}\) layer to the \( k^{th} \) neuron in the \( l^{th} \) layer, and \( f(\cdot) \) for the AF of the \( k^{th} \) neuron in the \( l^{th} \) layer. With these notations, the output \( y^l_k \) of the \( k^{th} \) neuron in the \( l^{th} \) layer is related to the outputs in the \((l-1)^{th}\) layer where the sum is over all neurons \( i \) in the \((l-1)^{th}\) layer.

Figure 3.4: Architecture of EvoDNN (Note the different activation functions in the hidden neurons)

Figure 3.4 shows the EvoDNN model. It resembles a typical FNN; the main difference lies in the hidden layer, as its neuron may have distinct AFs \( \sigma(\cdot) \). Such functions are chosen from a delimited candidate set, which is presented in Table 3.1 above, and contains a selection of those usually employed in the construction of FNNs.
The layers in EvoDNN are codified as follows: each layer contains three matrices, one for the input weights, one for the biases, and another for the indexes of activation functions in the candidate set (Figure 3.5). This proposed architecture has two fundamental advantages. First, the EvoDNN allows each neuron to respond uniquely while homogeneous FNNs force all neurons to perform the same activation function to incoming data. However, it is not obligated to respond differently because the algorithm can select the same activation function for all neurons. Second, instead of using gradient based methods, EvoDNN uses a ZO algorithm to optimize its network. This means the activation function does not have to be differentiable, nor does their derivative need to be known ahead of training.

![Figure 3.5: Framework of One Hidden Layer in EvoDNN](image)

Figure 3.5: Framework of One Hidden Layer in EvoDNN (\(a\) is the input data from previous layer; \(f_i()\) is the activation function associated with each neuron; the summation \(x_i\) goes through \(f_i()\) to produce output \(y_i\))

### 3.2.2 Evolutionary Algorithm

The evolutionary algorithm consists of four procedures, *population pool initialization*, *crossover*, *mutation*, and *parent population generation*. Each procedure also includes several steps to avoid the moving target problem resulting from the simultaneous evolution of both architectures and weights [85]. An *early-stopping* parameter is also introduced, which will stop the evolutionary algorithm if the fitness function is not improved after a certain point. This
early-stopping feature provides guidance as to how many generations can be run before the algorithm begins to overfit.

Population Pool Initialization

A population size of $N$ is generated in the first phase. Each individual in the population is a multi-layer heterogeneous FNN which has three components for each layer, the weight matrix, the bias matrix, and the activation function index matrix. The entries in each matrix are randomly generated via uniform distribution in a preset range. Each individual will then be evaluated and the one with the best fitness value will be copied to the next generation (elitist selection).

Crossover

This procedure is for creating the offspring pool and it is similar to the EvoNN’s crossover process shown in Figure 3.3. The difference of EvoDNN’s crossover procedure to that of EvoNN’s is that the child NN inherits not only weights and AFs, but also biases. In addition, the crossover will occur in each hidden layer of EvoDNN. A crossover proportion parameter ($P_c \in [0.0, 1.0]$) controls the proportion of individuals in the offspring produced by crossover. For example, if $P_c = 0.5$, then 50% of the individuals in the offspring generation will be the result of crossover. If an individual is chosen to be produced through crossover, then its neuron layers are produced in the following way. (The uniform crossover is used in this step.) Each entry in the weight matrix, the activation function index matrix, and the bias matrix of a layer has a 50% chance to inherit the first parent’s and a 50% chance to inherit the second parent’s entry respectively. These two parents are selected from the population pool via tournament selection method [57] with tournament size 2. The resulting individual has, on average, 50% of the first parent’s genotype and 50% of the second parent’s genotype.

Mutation

In the context of EvoDNN, mutation is divided into three types of modifications, each controlled by different parameters. Weight mutation is the process of modifying the entries in the weight matrix; bias mutation modifies the entries in the bias matrix; activation function mutation varies the activation function presented in a neuron.

Weight mutation is controlled by the mutation probability ($P_m \in [0.0, 1.0]$) and the mutation radius ($R_m \in (0.0, \infty)$). The mutation probability determines the probability for each entry in a weight matrix to be mutated while the mutation radius dictates the severity of the mutation. For instance, if $P_m = 0.5$ and $R_m = 0.1$, then each entry in a weight matrix has a 50% chance of being mutated in each generation. If that entry is indeed chosen for mutation, it will vary by a value uniformly chosen from the range $[-0.1, 0.1]$.

Bias mutation is also controlled by two parameters, the mutation probability ($P_b \in [0.0, 1.0]$) and the mutation radius ($R_b \in (0.0, \infty)$). Likewise, the mutation probability gives
the probability for each neuron’s bias being mutated while the mutation radius dictates the severity of the mutation.

Activation function mutation works similarly to the above two mutations, but is controlled by only one parameter, the function mutation probability ($P_{mf} \in [0.0, 1.0]$). The function mutation probability controls the likelihood that an activation function within a neuron is modified. However, this mutation does not have a mutation radius equivalent. If a neuron’s activation function is selected for mutation, the new one is uniformly determined from the set of hidden neuron’s activation functions which are different from the function employed at the given hidden neuron. For example, if $P_{mf} = 0.5$, then each hidden neuron has a 50% chance of having its activation function mutated. If a hidden neuron which contains the function $f(x)$ is chosen for the mutation process and the set of activation functions is $\{ f(x), g(x), h(x), i(x) \}$, then the neuron’s old function is replaced by either $g(x), h(x), i(x)$, each with a 33.3% chance to be chosen respectively.

**Parent Population Generation**

In order to guarantee the solution quality obtained by the EA will not decrease from one generation to the next, elitist selection is applied in this procedure. That is, the individual with the best fitness value from the current generation will carry over to the parent population pool. The remaining individuals that will be added to the parent population are then determined by the tournament selection method [57] with tournament size 2. Afterwards, each individual in the parent population will be evaluated and the one with the best fitness value will be again copied to next generation (elitist selection).
Chapter 4

Experimental Implementation

In this chapter, we give the experimental setup for our EvoDNN framework. In section 4.1 we present details on the data sets we used in our experiments. In section 4.2 we describe the metrics that evaluate the performance of EvoDNN framework.

4.1 Pattern Classification Data Sets

The pattern classification data sets considered here are obtained from UCI machine learning repository [5]. The criterion for the pattern classification data sets we chose are:

- They are related to the bioinformatics area.
- They have fewer instances with missing values than other data sets.
- Each of the data sets has at least 100 instances with 4 features.

For the instances with missing values, we removed them from according data sets. Each data set was divided into three subsets: training set, validation set, and testing set. 60% of instances were used for training, 20% for validation, and the remaining 20% for testing. Following is the details of each benchmark and their main characteristics are summarized in Table 4.1.

4.1.1 Abalone Data Set

Abalone is a common name for any of a group of small to very large sea snails, marine gastropod molluscs in the family Haliotidae [60]. This data set consists of 4177 instances. Each instance is annotated with eight attributes and each attribute is either a real number or an integer. These attributes, or physical measurements are used to predict the age of abalone. The age is an integer range from 1 to 29, therefore this data set can be used in our experiment as a classification task. The eight attributes are: sex (Male, Female, Infant), length (longest shell measurement), diameter (perpendicular to length), height (with meat
in shell), whole weight (whole abalone), shucked weight (weight of meat), visceras weight (gut weight after bleeding), and shell weight (after being dried).

### 4.1.2 Breast Cancer Wisconsin (Diagnostic) Data Set

There are 32 features in breast cancer data set that are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass [75]. These features are all real numbers. They describe characteristics of the cell nuclei present in the image and we will only use the 9 key features in our experiments. 569 instances are included in this data set and we classify whether the tumor is benign or malignant. The 9 key features are: radius (mean of distances from center to point on the perimeter), textures ( standard deviation of gray-scale values), perimeter, area, smoothness (local variation in radius lengths), compactness \(\frac{\text{perimeter}^2}{\text{area}} - 1.0\), concavity (severity of concave portions of the contour), symmetry, and fractal dimension (coastline approximation – 1).

### 4.1.3 Iris Data Set

Iris data set perhaps is the best known database to be found in the pattern recognition literature [22]. The data set contains three classes of 50 instances each, where each class refers to a type of iris plant or flowers. The task is to predict the class of iris plant. The attributes used in our experiments are sepal length (cm), sepal width (cm), petal length (cm), and petal width (cm). They are all real values.

### 4.1.4 Lymphography Data Set

Lymphography data set is provided by the Oncology Institute [16], which has often appeared in the machine learning literature. Lymphography describes the X-ray examination of the vessels of the lymphatic system after injection of a substance opaque to X-rays. This data set is composed of 148 instances with eighteen attributes. Each instance is classified as normal find, metastases, malign lymph, or fibrosis. The goal is to predict the classes (normal find, metastases, malign lymph, fibrosis) according to these eighteen attributes. These attributes are: lymphatics (normal, arched, deformed, displaced), block of affere (no, yes), block of lymph. c (no, yes), block of lymph. s (no, yes), by pass (no, yes), extravasates (no, yes), regeneration of (no, yes), early uptake in (no, yes), lym.nodes dimin (0-3), lym.nodes enlar (1-4), changes in lym. (bean, oval, round), defect in node (no, lacunar, lac. marginal, lac. central), changes in node (no, lacunar, lac. margin, lac. central), changes in stru (no, grainy, drop-like, coarse, diluted, reticular, stripped, faint), special forms (no, chalices, vesicles), dislocation of (no, yes), exclusion of no (no, yes), and no. of nodes in (0-9, 10-19, 20-29, 30-39, 40-49, 50-59, 60-69, >=70). They all are categorical values which can be presented in integers.
4.1.5 Tumor Data Set

Tumor data set is another domain provided by the Oncology Institute [16]. It consists of 339 instances of the location of tumors. Each instance belongs to one of the twenty-two different locations. They are lung, head & neck, esophagus, thyroid, stomach, duoden & sm.int, colon, rectum, anus, salivary glands, pancreas, gallblader, liver, kidney, bladder, testis, prostate, ovary, corpus uteri, cervix uteri, vagina, breast. The data set has seventeen attributes. These attributes are: age (<30, 30-59, >=60), sex (male, female), histologic-type (epidermoid, adeno, anaplastic), degree-of-diffe (well, fairly, poorly), bone (yes, no), bone-marrow (yes, no), lung (yes, no), pleura (yes, no), peritoneum (yes, no), liver (yes, no), brain (yes, no), skin (yes, no), neck (yes, no), supraclavicular (yes, no), axillar (yes, no), mediastinum (yes, no), and abdominal (yes, no).

4.1.6 Wine Data Set

Wine data set is the results of a chemical analysis of wines grown in the same region in Italy but derived from three different cultivars [25]. The analysis determined the quantities of thirteen constituents found in each of the three types of wines. These attributes are: alcohol, malic acid, ash, alcalinity of ash, magnesium, total phenols, flavanoids, nonflavanoid phenols, proanthocyanins, color intensity, hue, OD280/OD315 of diluted wines, and proline. Wine data set is a challenging one because it has 13 features but only 178 instances; we need to find the pattern of this data set through using a small number of instances.

4.1.7 Yeast Data Set

Yeast data set is used to build model that can predict the cellular localization sites of proteins. It has 1484 instances with eight attributes [41]. These attributes are: sequence Name (Accession number for the SWISS-PROT database. This is often omitted when training model because it has limited, or no meaningful information.), mcg (McGeoch’s method for signal sequence recognition), gvh (von Heijne’s method for signal sequence recognition), alm (Score of the ALOM membrane spanning region prediction program), mit (Score of discriminant analysis of the amino acid content of the N-terminal region of mitochondrial and non-mitochondrial proteins), erl (Presence of ‘HDEL’ substring; binary attribute), pox (Peroxisomal targeting signal in the C-terminus), and vac (Score of discriminant analysis of the amino acid content of vacuolar and extracellular proteins). We then use these eight features to predict the localization site of protein. The class distribution is CYT (cytosolic or cytoskeletal), NUC (nuclear), MIT (mitochondrial), ME3 (membrane protein, no N-terminal signal), ME2 (membrane protein, uncleaved signal), ME1 (membrane protein, cleaved signal), EXC (extracellular), VAC (vacuolar), POX (peroxisomal), ERL (endoplasmic reticulum lumen).
The main characteristics of the seven classification problems considered here are presented in Table 4.1.

Table 4.1: Pattern Classification Data Sets

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Instance Number (n)</th>
<th>Feature Number (m)</th>
<th>Class Number (c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abalone</td>
<td>4177</td>
<td>8</td>
<td>29</td>
</tr>
<tr>
<td>Breast Cancer</td>
<td>569</td>
<td>30</td>
<td>2</td>
</tr>
<tr>
<td>Iris</td>
<td>150</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Lymphograph</td>
<td>148</td>
<td>18</td>
<td>4</td>
</tr>
<tr>
<td>Tumor</td>
<td>339</td>
<td>17</td>
<td>22</td>
</tr>
<tr>
<td>Wine</td>
<td>178</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>Yeast</td>
<td>1484</td>
<td>8</td>
<td>10</td>
</tr>
</tbody>
</table>

4.2 Evaluation Metrics

To be compatible with each framework, we removed the instances with missing values from the data set to construct a new dataset. Each data set was divided into three subsets: training set, validation set, and testing set. 60% of instances were used for training, 20% for validation, and the remaining 20% for testing. The experimental results reported are based on the testing set. In order to evaluate the performance of EvoDNN, two conventional algorithms were benchmarked simultaneously with the same data sets utilized in EvoDNN. As well, a comparison with single hidden layer EvoDNN network is presented.

For each data set, EvoDNN performed 50 experiments with a different random seed each time to initialize weights, biases, and AF matrices. EvoNN also performed 50 experiments with a random seed each time but only to initialize weights and AF matrices. Gradient-based FNN performed 50 experiments with a random seed each time to initialize weights and biases only. Random Forest utilized a random seed to initialize the internal structure each time when performing 50 experiments. Each experiments was a complete training, validating, and testing cases for EvoNN, EvoDNN, gradient based FNN, and random forest.

Meanwhile, we recorded the AF of each neuron in the after-training network in order to analyze if there is a pattern about AFs in the neural network. Within the 50 experiments, each data set was shuffled at random and split into training, validation, and testing subsets every 10 experiments, creating 5 "mini-experiments" of 10 repetitions each. This was done to simulate different distributions of classes within the subsets as well as to prevent any bias during the training. Also, it prevented the model from learning the permutation of the training data set. Each experiment was allowed to run until it showed no signs of improvement for 300 iterations (300 epochs for the gradient based FNN or 300 generations for EvoDNN).
and EvoNN) on the validation set, or until it reached 5,000 iterations (5,000 epochs for the gradient based FNN or 5,000 generations for EvoDNN and EvoNN). Particularly, gradient based FNN was experimented with four types of AFs in its neurons: Sigmoid, Tanh, ReLU, and LReLU.

There are twenty-two different architectures constructed for these experiments. They are [5, 10, 20, 40]-layer EvoDNNs, [5, 10, 20, 40]-layer gradient-based FNNs with Sigmoid AF, [5, 10, 20, 40]-layer gradient-based FNNs with Tanh AF, [5, 10, 20, 40]-layer gradient-based FNNs with ReLU AF, [5, 10, 20, 40]-layer gradient-based FNNs with LReLU AF, EvoNN and Random Forest. In the context of hidden layer, each layer contains ten hidden neurons.

We chose [5, 10, 20, 40]-layer networks for experiments and not continued with 80 or more layers because our purpose is to propose and experiment a novel approach instead of pursuing a state-of-the-art performance. Since as deep as 40-layer networks could show noticeable performance differences among these methods (we discovered this when we did the experiments), we did not proceed with 80 or more layer networks. In addition, we doubled layers for each configuration, not increased them linearly; this is because we could explore a large range of networks with different layers through a relatively small set of experiments. If we increased network layer linearly, we need to run more experiments in order to show the relationship between network layers and network performance.

Multi-class logarithmic loss (multi-logloss) function which was used as the fitness function to evaluate EvoDNN and EvoNN, was also used as the loss function to evaluate the gradient based FNN, such that both neural networks were making effort to optimize the same objective. The advantage of using multi-class logarithmic loss function, instead of simply computing the percentage of correct predictions, is that the smaller the multi-logloss value is, the greater the confidence of the algorithm is about its correct predictions. In other words, the smaller the multi-logloss value is, the better the performance of an algorithm is. Benchmarks that are based on the percentage of correct predictions cannot eliminate the random and lucky guesses by certain algorithms.

The multi-class logarithmic loss function is defined as

$$\text{logloss}(Y, \hat{Y}) = -\frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{c} y_{ij} \times \ln(\hat{y}_{ij})$$  \hspace{1cm} (4.1)$$

where $Y$ is the ground truth matrix of the class and $\hat{Y}$ is the prediction matrix generated by the neural network; $n$ is the number of instances in a data set; $c$ is the number of classes in a data set; $y_{ij}$ is the ground truth of instance $i$ belonging to class $j$ (1 if belongs or 0 if doesn’t) while $\hat{y}_{ij}$ is the predicted probability that instance $i$ belongs to class $j$. To prevent computational failures, $\hat{y}_{ij}$ is taken to be $\min(1 - 10^{-15}, \max(10^{-15}, p_{ij})$ where $p_{ij}$ is the neural network’s predicted probability that instance $i$ belongs to class $j$.
Just before employing multi-class logarithmic loss function, softmax function is implemented at the output layer. The softmax function (Eq. 4.2)

\[
softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{c} e^{x_j}}
\]  

assigns decimal probabilities to each class in the multi-class problem. Those decimal probabilities from the softmax function is a set of positive numbers which sum up to 1.

In the 50 experiments, only the 25 experiments with the lowest multi-logloss values were used to compute the average multi-logloss as well as the standard deviations, and then compared. This is because EvoDNN may evolve in the wrong direction due to the nature of evolutionary algorithm and thus get an unexpected multi-logloss value; the gradient based FNN may also be trapped in the local minimum and obtain an unwanted result. We need to remove these outliers.

4.3 EvoNN Setup

EvoNN has a fixed architecture of three layers: \(m\) input neurons in the first, or input layer, ten hidden neurons, and \(c\) output neurons employing the softmax function (Eq. 4.2). However, each of the ten hidden neurons could have a different AF. The AFs provided as a function "bank" were the sigmoid function, the hyperbolic tangent (tanh) function, the Rectified Linear Unit function (ReLU) [30], and the Leaky Rectified Linear Unit (LReLU) function [54]. Their properties are outlined in Table 3.1. The evolutionary parameters are outlined in Table 4.2. There is an excellent introduction to the hyper-parameters turning of evolutionary algorithm that can be found in [43].

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>Maximum number of generations</td>
<td>5000</td>
</tr>
<tr>
<td>early_stopping</td>
<td>Avoid overfitting when training with an iterative method</td>
<td>200</td>
</tr>
<tr>
<td>node_hidden_layer</td>
<td>Number of neurons in the network’s hidden layers</td>
<td>10</td>
</tr>
<tr>
<td>num_of_hidden_layer</td>
<td>Number of hidden layers</td>
<td>1</td>
</tr>
<tr>
<td>MU</td>
<td>Number of parents in the EA</td>
<td>50</td>
</tr>
<tr>
<td>LAMBDA</td>
<td>Number of offspring in the EA</td>
<td>50</td>
</tr>
<tr>
<td>P_m</td>
<td>Chance of weight mutation occurrence</td>
<td>0.01</td>
</tr>
<tr>
<td>P_mf</td>
<td>Chance of function mutation occurrence</td>
<td>0.05</td>
</tr>
<tr>
<td>R_m</td>
<td>Weight mutation radius</td>
<td>0.1</td>
</tr>
<tr>
<td>P_c</td>
<td>Proportion of individuals to be created by crossover</td>
<td>0.3</td>
</tr>
<tr>
<td>elitism</td>
<td>Best individual copied to next generation (elitist selection)</td>
<td>True</td>
</tr>
<tr>
<td>tournament_size</td>
<td>Number of individuals selected</td>
<td>2</td>
</tr>
<tr>
<td>fitness_function</td>
<td>Objective function</td>
<td>multi-logloss</td>
</tr>
</tbody>
</table>
4.4 EvoDNN Setup

EvoDNN has a fixed architecture of three components: \( m \) input neurons in the input layer (where \( m \) is the feature number for each data set). Ten hidden neurons in each hidden layer, of which there were five, ten, twenty, forty, and \( c \) output neurons in the output layer (where \( c \) is the number of classes for each set) which employs the softmax function (Equation 4.2). Each hidden neuron, however, could use a distinct AF. The collection of AFs available were the hyperbolic tangent (tanh) function, the sigmoid function, the Leaky Rectified Linear Unit (LReLU) function, and the Rectified Linear Unit function (ReLU). Their properties are outlined in Table 3.1. The evolutionary algorithm’s parameters are listed in Table 4.3. They are empirically estimated and then fine tuned according to Nannen et al.’s research [59]. For example, the mutation probability parameters (\( P_m, P_{mf}, P_b \) in the below table) are set according to the average of mutation probability in nature. The mutation radius parameters (\( R_b, R_m \) in the below table) are set to be small. This is because if there are outlier mutations, these mutations will not cause big changes on weights and biases and hence make EvoDNN’s performance unstable.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>Maximum number of generations</td>
<td>5000</td>
</tr>
<tr>
<td>early_stopping</td>
<td>Avoid overfitting when training with an iterative method</td>
<td>200</td>
</tr>
<tr>
<td>node_per_layer</td>
<td>Number of neurons in the network's hidden layers</td>
<td>10</td>
</tr>
<tr>
<td>num_of_hidden_layer</td>
<td>Number of hidden layers</td>
<td>5,10,20,40</td>
</tr>
<tr>
<td>MU</td>
<td>Number of parents in the EA</td>
<td>50</td>
</tr>
<tr>
<td>LAMBDA</td>
<td>Number of offspring in the EA</td>
<td>50</td>
</tr>
<tr>
<td>P_m</td>
<td>Chance of weight mutation occurrence</td>
<td>0.01</td>
</tr>
<tr>
<td>P_mf</td>
<td>Chance of function mutation occurrence</td>
<td>0.01</td>
</tr>
<tr>
<td>P_b</td>
<td>Chance of bias mutation occurrence</td>
<td>0.01</td>
</tr>
<tr>
<td>R_m</td>
<td>Weight mutation radius</td>
<td>0.1</td>
</tr>
<tr>
<td>R_b</td>
<td>Bias mutation radius</td>
<td>0.1</td>
</tr>
<tr>
<td>P_c</td>
<td>Proportion of individuals to be created by crossover</td>
<td>0.3</td>
</tr>
<tr>
<td>elitism</td>
<td>Best individual copied to next generation (elitist selection)</td>
<td>True</td>
</tr>
<tr>
<td>tournament_size</td>
<td>Number of individuals selected</td>
<td>2</td>
</tr>
<tr>
<td>fitness_function</td>
<td>Objective function</td>
<td>Multi-logloss</td>
</tr>
</tbody>
</table>

The hyper-parameter setting of population size (\( MU/LAMBDA \) in the above table) is more crucial for achieving a better performance. This is because when one generation has more individuals, there is a higher chance that one potential individual evolves in the direction where its multi-logloss will approach the global minimum in the solution space.
4.5 Gradient Based FNN Setup

The gradient based FNN also has a fixed architecture consisting of three components: \( m \) input neurons in the input layer, 10 hidden neurons employing the \{Sigmoid, Tanh, ReLu, LReLu\} AFs in each hidden layer, of which there were five, ten, twenty, or forty, and \( c \) output neurons employing the softmax function in the output layer. The difference from EvoDNN is that this network has the homogeneous AFs in all hidden neurons. The parameters of this neural network are laid out in Table 4.4.

Table 4.4: Gradient Based Algorithm Configuration Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>sizes</td>
<td>Number of neurons per layer</td>
<td>10</td>
</tr>
<tr>
<td>num_of_layer</td>
<td>Number of hidden layers</td>
<td>5, 10, 20, 40</td>
</tr>
<tr>
<td>epochs</td>
<td>One pass over the full training set</td>
<td>5000</td>
</tr>
<tr>
<td>early_stopping</td>
<td>Avoid overfitting when training with an iterative method</td>
<td>300</td>
</tr>
<tr>
<td>mini_batch_size</td>
<td>A subset of data during one iteration</td>
<td>10</td>
</tr>
<tr>
<td>eta</td>
<td>Learning rate</td>
<td>2.5</td>
</tr>
<tr>
<td>loss function</td>
<td>Objective function</td>
<td>multi-logloss</td>
</tr>
</tbody>
</table>

4.6 Random Forest Setup

The parameters of random forest was configured and turned according to Boulesteix’s practical guidance of random forest methodology [14].

A random forest is a classical meta estimator that fits a number of decision tree classifiers on various sub-samples of the data set and then uses averaging to improve the predictive accuracy [15]. Due to the Law of Large Numbers [10], random forest algorithm rarely have over-fitting problem and is robust with respect to noise. That is, the generalization error for the forest converges to a limit as the number of trees in the forest becomes large. In addition, the generalization error of a forest depends on the strength of the individual trees in the forest as well as the correlation between them. The parameters of random forest are laid out in Table 4.5. These parameter are empirically estimated and then fine tuned based on [11].
Table 4.5: Random Forest Algorithm Configuration Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>n_estimator</td>
<td>The number of trees in the forest</td>
<td>300</td>
</tr>
<tr>
<td>criterion</td>
<td>The function to measure the quality of a split</td>
<td>gini</td>
</tr>
<tr>
<td>max_depth</td>
<td>The max. depth of the tree</td>
<td>based on dataset size</td>
</tr>
<tr>
<td>max_features</td>
<td>The number of features for the best split</td>
<td>30</td>
</tr>
<tr>
<td>loss function</td>
<td>Objective function</td>
<td>multi-logloss</td>
</tr>
</tbody>
</table>

4.7 Pre-processing, Language, and Hardware Specification

All experiments reported have been performed on a Dell Workstation with i7 3770 processor and 16GB RAM. The code was written in Python 3.5.4 and was compiled using CPython.

All features were scaled into the range \([-1.0, 1.0]\) by dividing the values of each feature \(i\) by \(\max(\max(\overrightarrow{x_i}), |\min(\overrightarrow{x_i})|)\), where \(\overrightarrow{x_i}\) is the vector of values in feature \(i\).
Chapter 5

Experimental Results and Analysis

In this chapter the performance of evolutionary algorithm in heterogeneous FNN training is evaluated and compared by various experiments. The experiments were carried out using training problems from a commonly used problem domain: pattern recognition (The data sets were described in last section 4.1), and evaluated by different methods with various configurations.

These methods are random forest algorithm, EvoNN, 5-layer EvoDNN, 5-layer Sigmoid FNN (FNN employs Sigmoid AF in all its hidden neurons), 5-layer Tanh FNN, 5-layer ReLU FNN, 5-layer LReLU FNN, 10-layer EvoDNN, 10-layer Sigmoid FNN, 10-layer Tanh FNN, 10-layer ReLU FNN, 10-layer LReLU FNN, 20-layer EvoDNN, 20-layer Sigmoid FNN, 20-layer Tanh FNN, 20-layer ReLU FNN, 20-layer LReLU FNN, 40-layer EvoDNN, 50-layer Sigmoid FNN, 40-layer Tanh FNN, 40-layer ReLU FNN, 40-layer LReLU FNN.

5.1 Optimization Performance

For every data set, the resulting mean values of multi-logloss for each method and the standard deviations associated with them are shown in each subsection tables.

5.1.1 Abalone Data Set

Figure 5.1 clearly shows that EvoDNN have produced very competitive accuracy compared to other methods in [5, 10, 20, 40]-layer neural networks with significantly smaller multi-logloss. In addition, [5, 10, 20, 40]-layer EvoDNN all surpass the performance of EvoNN and random forest algorithm with a relatively small standard deviations. The small standard deviation suggests that EvoDNN architecture could reach a convergent and stable status after the training is completed, even with different initial status. However, the random forest algorithm has a large standard deviation. We conjecture that this algorithm may have exponentially low convergence rates, which thus impacts the reducing of the multi-logloss value.
Figure 5.1: Optimization Results of Evaluating Abalone Data Set
Remarkably, although the depth is significantly increased, the smallest multi-logloss is achieved by 10-layer EvoDNN, instead of 40-layer EvoDNN, with a value of 2.045. We argue that this is because of overfitting. The 40-layer EvoDNN network may be unnecessarily large for this small data set (only 4177 instances). In addition, we have observed that in [5, 10, 20, 40]-layer neural networks, the performance of EvoDNN is always better than [Sigmoid, Tanh, ReLU, LReLU] FNNs.

### 5.1.2 Breast Cancer Data Set

![Figure 5.2: Optimization Results of Evaluating Breast Cancer Data Set](image)

In Figure 5.2 we compare the mean multi-logloss of these methods with Breast Cancer data set. It is clear that FNNs with Tanh AF are not suitable to classify Breast Cancer data set. In [5, 10, 20, 40]-layer neural networks, FNNs with Tanh AF obtain the worst multi-logloss compared to other same-number-layered networks, while EvoDNNs have produced the best multi-logloss values. This implies one advantage of EvoDNN over other homogeneous FNNs. That is, EvoDNN can evolve its AF in each hidden neuron in order to adapt certain data set and therefore achieve competitive accuracy.
The 5-layer EvoDNN has the lowest multi-logloss (0.068), suggesting that for a small dataset few layered neural networks is sufficient to gain significant accuracy. More importantly, random forest algorithm has an impressive low multi-logloss. We argue this is because Breast Cancer data set has an efficient feature selection so that only a few features can make correct prediction. We know that each decision tree in random forest algorithm only keeps a few features. Therefore it is clear that random forest algorithm is an alternative solution to build prediction model of Breast Cancer data set.

5.1.3 Iris Data Set

The task of Iris data set is to predict the class of iris plant/flower which has three types: Setosa, Versicolour, Virginica. It is often considered as a simple benchmark to quickly evaluate the performance of a prototype. Figure 5.3 demonstrates the accuracy of various methods to classify the type of iris flowers. Three methods reach outstanding low multi-logloss. They are 0.07, 0.11, and 0.098 multi-logloss for 5-layer EvoDNN, EvoNN, and Random Forest algorithm respectively. We argue that even though 5-layer EvoDNN has the lowest mean multi-logloss value, it is hard to say that 5-layer EvoDNN surpasses the performance of EvoNN and Random Forest algorithm. This is because these three methods have relatively
large standard deviations compared to their mean multi-logloss, which implies for a certain EvoNN or Random Forest model they may perform better than the 5-layer EvoDNN model.

In addition, we notice that FNNs with Tanh AF are also not a good fit for Iris data set. They have the worst multi-logloss values in all [5, 10, 20, 40]-layer neural networks. However, EvoDNNs have achieved the best multi-logloss values in those networks.

5.1.4 Lymphography Data Set

Lymphography is a small data set with only 148 instances; however each instance contains 18 features. The challenge of this data set is to build a model that is able to perform an efficient feature selection. This means a model can identify which features have more feature weights to decide the class of an instance while other features have less feature weights to do so. Due to the nature of random forest algorithm (a random forest is a multitude of decision trees) [15], it is hard to distinguish which features are more important than others in order to give a prediction of an instance. This optimization difficulty is likely to cause an unusual high multi-logloss. In our example, the random forest algorithm has a mean multi-logloss value of 5.776, which is approximately five times larger than other methods. This suggests that random forest algorithm is not suitable for training a data set which
has many features. It is worth to note that the standard deviation of the random forest
algorithm is also very large compared to other algorithms.

The best performance is obtained by 20-layer EvoDNN with a mean multi-logloss value
of 0.481. Again, EvoDNNs have the best performance in all [5, 10, 20, 40]-layer neural
networks.

5.1.5 Tumor Data Set

Figure 5.5: Optimization Results of Evaluating Tumor Data Set

Tumor data set is similar to the previous Lymphography data set in the way that they
both have few instances but many features. More specifically, Tumor data set has 339
instances and 17 features while Lymphography data set has 148 instances and 18 features.
However, unlike Lymphography data set which only has 4 classes, Tumor data set has
22 classes. This creates a more difficult challenge for a potential algorithm than that of
Lymphography data set, because the potential algorithm not only has to give each feature
a reasonable feature weight, but also requires to make a precise, or confident prediction of
the class of an instance. A random guess often will not work well in this scenario because
for a normal distribution, only 4.54% chance to guess the class correctly.
Figure 5.5 shows that the random forest algorithm has the worst mean multi-logloss value, which is seven times greater than other algorithms. Its standard deviation is also greater than all other algorithms, which suggests an exponentially low convergence rate and therefore reveals the high mean multi-logloss. It also implies that the random forest algorithm may be only suitable for data sets with few features and classes.

Unexpectedly, 20-layer ReLU FNN has the lowest score for mean multi-logloss. We hypothesize that it is more suitable to train Tumor data set with FNNs employing ReLU AF. However, even though 20-layer ReLU has the best performance, the second lowest score which was achieved by 40-layer EvoDNN, is only higher than 20-layer ReLU FNN by 0.096 in percentage, it is only 4% higher. Therefore, we argue that 40-layer EvoDNN performs as well as 20-layer ReLU FNN. Overall, EvoDNNs perform better than other methods.

### 5.1.6 Wine Data Set

Wine is also a data set with only 178 instances but has 14 features. The comparison of mean multi-logloss values of different algorithms is given in Fig 5.6. To our surprise, the performance of random forest algorithm is superior to the performance of other methods. We argue that unlike Lymphography and Tumor data sets, wine data set has an effective
feature selection, which means different classes of wines have very distinct values for each feature and thus only few features can make correct prediction of wine classes. Since by the nature each decision tree in random forest algorithm only contains few features, it is clear that random forest algorithm is suitable to model wine data set. Still, we have observed that it is difficult to train Tanh FNNs in order to gain competitive results.

We also noticed the fact from Figure 5.6 that as depth increases, FNNs have worse multi-logloss values. This is counter-intuitive since we would assume that by giving our model more parameters, we should be able to construct the intricacies of our data at least as well as before. However, learning better networks is not as easy as stacking more layers. An obstacle to opposing this intuition was the notorious problem of vanishing/exploding gradients [29, 9], which hamper convergence from the beginning.

Overall, EvoDNNs have produced competitive results compared to other models, even though they are not the most suitable one for wine data set.

### 5.1.7 Yeast Data Set

![Figure 5.7: Optimization Results of Evaluating Yeast Data Set](image)

From Figure 5.7 we can see that the best performance result is reached by 10-layer EvoDNN with a multi-logloss value of 1.028. Random forest algorithm also has a competitive
result while other neural networks (except 10-layer EvoDNN) have relatively high multi-logloss values.

One interesting fact from Figure 5.7, which also reveal in several previous optimization results, is that even thought EvoDNN does not exploit back-propagation method, its multi-logloss value still increases as depth increases. For example, we have observed that the 40-layer EvoDNN has higher multi-logloss value than the 10-layer EvoDNN’s, even though the solution space of the 10-layer EvoDNN is a subspace of that of the 40-layer one. We conjecture that when deeper networks are able to start converging, a degradation problem has been exposed: with the network depth increasing, accuracy get saturated and then degrades rapidly. Unexpectedly, such degradation is not caused by overfitting, because after careful examination, we found that the training errors are also worse. However, no one has a clear explanation about this problem that adding more layers to a suitably deep model leads to higher multi-logloss value, as reported in [35, 74]. The degradation problem suggests that not all models are similarly easy to optimize. The reason for such optimization difficulties will be studied in the future. However, it implies a proper number of hidden layers should be selected when designing EvoDNN. This can be done by empirically experimenting different configurations of EvoDNN and select the most suitable one.

5.1.8 Summary

Overall, EvoDNNs (allowing an EA to evolve heterogeneous deep FNNs) produced better results, on average, than the average results obtained by training homogeneous FNNs, EvoNN, and Random Forest. This is remarkable because conventional gradient descent method can not well address the gradient vanishing/explosion problem, even with the advanced ReLU and LReLU AFs. Furthermore, as seen from the above figures, certain AF may not work well on all data sets; for example, FNN with Tanh is not suitable for modeling Breast Cancer, Iris, and Wine data sets. Therefore, the gradient vanishing/explosion problem, coupled with the fact that there is no way of knowing which AF will be most suited to a given task before training, puts homogeneous FNNs at a disadvantage.

Having a great degree of flexibility over which AF to utilize, EvoDNN can search a larger solution space to find a better result, which gives a smaller multi-logloss value. In addition, we have observed that random forest algorithm only works well if a data set is less complicated and has a small number of features. However, we expect to handle large data with hundreds of thousands of instances in modern science research area; for example, ImageNet contains over 14 million annotated images [67]. Hence, random forest algorithm seems unfeasible on such big data set.

The importance of these results is also highlighted by the fact that many other architectures of neural networks are probably capable of being evolved by EAs. The evolution of heterogeneous deep FNNs may even be further improved by the inclusion of additional activation functions which are not considered here; and as the EA places no restrictions on
the types of activation functions used, \textit{i.e.} the activation function does not need to be differentiable, the range of possible activation functions is large. In addition, when constructing models, it is not always possible to attain an accurate representation of test sets due to a limited amount of data, class imbalances, or other biases related to data gathering. Having confidence in the model’s general flexibility towards distribution of examples and classes is also a benefit of the EvoDNN architecture.

5.2 Running Time Performance

So far, we have reviewed the performance of each method, and discovered that EvoDNNs produced better results, on average, than the average results gained by other methods. We now turn our attention to evaluating the running time of each method. We limit each method with only one processing thread and calculate the total amount of time for each method to complete the classification of each data set.

Training stage contributes most of the running time. This is because testing and validation stage only need to pass through the neural network once, but training stage requires to go back and forth through the neural network many times in order to update weights, biases, and AFs. Therefore, we can consider that more running time means more training time.

5.2.1 Abalone Data Set

The run-time result of Abalone data set presented in Figure 5.8 demonstrates that the EA has a larger impact on running time than gradient based FNN and random forest algorithm. It is obvious that EvoDNNs consume the most time to construct the model than other methods in [5, 10, 20, 40]-layer neural networks, and the worst run-time performance is produced by 10-layer EvoDNN, which spends approximately 37941.8 seconds to complete the classification task. However, [5, 20, 40]-layer EvoDNN do not have a big difference with the 10-layer EvoDNN when considering the large standard deviations associated with each EvoDNN. This is an interesting observation because the running time does not increase linearly when adding more hidden layers. We hypothesize this is because as more hidden layers are added, the solution space will be heavily expanded and hence it is relatively effortless to find a competitive configuration of the network. It is also possible that the most time consuming procedures in each generation are crossover and mutation; adding more hidden layers only counts up an ignorable amount of time, which makes the total amount of time do not increase linearly. Carefully analysing each procedures in a generation of EvoDNN will be an interesting direction for future work. Furthermore, it is worth to notice that EvoDNN is only slightly slower than EvoNN, but it has much better performance than EvoNN.
Figure 5.8: Run-time Result of Training Abalone Data Set
5.2.2 Breast Cancer Data Set

Figure 5.9 shows the run-time average for twenty-two optimization systems on Breast Cancer data set. As expected, the run-time performance of the gradient based FNN is still superior to the EvoDNN optimizer (e.g. the most time consuming optimizer is 40-layer EvoDNN). This is an intuitive result as in our EvoDNN setup, it requires to modify fifty individuals’ (networks’) parameters in one generation while the gradient based method only has to modify one network in one epoch. It appears to mirror the "No Free Lunch" theorem but concerning different algorithms [83]. However, to our surprise, random forest algorithm also spends a lot of time to train the Breast Cancer data set; i.e. it has the third worst run-time performance. We argue this is because random forest algorithm needs to build a very large collection of decision trees since there are thirty features in Breast Cancer data set.

![Figure 5.9: Run-time Result of Training Breast Cancer Data Set](image)

5.2.3 Iris Data Set

The mean run-time results of Iris data set is demonstrated in Figure 5.10. Unexpectedly, random forest algorithm gets the worst run-time outcome with a value of 948.6 seconds. We believe since Iris data set has a simple structure, neural-network based methods (e.g. FNNs and EvoDNNs can converge fast while random forest algorithm needs to construct a group
of decision trees that uses a lot of time. We also note that the running time of FNN does not always go up linearly as depth increases. This is because the early stopping mechanism involved in the training process. Overall, FNNs, on average, utilize less training time than EvoDNNs.

![Figure 5.10: Run-time Result of Training Iris Data Set](image)

### 5.2.4 Lymphography Data Set

From Figure 5.11 we have discovered that random forest algorithm has the worst result for run-time performance. Since Lymphography data set has eighteen features, it is intuitive to us that the size of features may also contribute to the increasing running time; *i.e.* more features means random forest algorithm needs to build more decision trees and therefore demands more running time. However, even random forest algorithm uses the most running time, it has the worst multi-logloss value. This verifies our previous hypothesis that random forest algorithm is only suited to simple structured data set.

In addition, we notice that the amount of FNN running time still significantly out-perfoms EvoDNN in all [5, 10, 20, 40]-layer neural networks; some FNN only requires as less as one quarter time of EvoDNN’s running time.
Figure 5.11: Run-time Result of Training Lymphography Data Set
5.2.5 Tumor Data Set

Tumor is a complicated data set that includes seventeen features and twenty-two classes. Therefore it requires certain algorithm to search an even larger solution space; it also indicates by search large solution space, certain algorithm may suffer small convergence rate and hence uses a lot of running time. We can see from Figure 5.12 that 40-layer EvoDNN utilizes the most seconds of running time while all FNNs continues to keep a low run-time rates. We argue that since FNNs uses stochastic gradient descending to improve their accuracy performance, in the straightforward case, it can only improve the result in one direction and cannot reverse back in the solution space. If in that direction, FNN’s optimization performance does not get better (in most cases, FNN will struggle in the saddle point), early stopping will then involve and stop the training period. This is one potential reason that FNN has low run-time rate.

![Figure 5.12: Run-time Result of Training Tumor Data Set](image)

5.2.6 Wine Data Set

We provide the run-time performance of Wine data set in Figure 5.13. Random forest algorithm has the worst run-time result and 40-layer EvoDNN has the second worst one. One interesting result, which also reveals in previous figures, that in many cases the standard
deviations, both for FNN and EvoDNN, are large. This suggests that the number of genera-
tions in one EvoDNN training and the number of epoches in one FNN training are unstable. 
We argue this is because FNN may be trapped in a saddle point and in some lucky cases, 
it needs a while to escape from the saddle point; for some EvoDNN training, the algorithm 
may evolve in the relatively suited direction, but sometimes it may detour and evolve in an 
intricate direction that consumes more training time.

![Running Time](image)

Figure 5.13: Run-time Result of Training Wine Data Set

### 5.2.7 Yeast Data Set

Figure 5.14 presents the run-time result of Yeast data set. 5-layer EvoDNN uses the most 
of running time, but empirically no difference of running time between 5-layer EvoDNN and 
20-layer EvoDNN after considering the standard deviations. In addition, FNN remains the 
best of utilizing the least time to build models.
Figure 5.14: Run-time Result of Training Yeast Data Set
5.2.8 Summary

Overall, the comparisons demonstrate the excellent performance of our framework, EvoDNN has a trade-off with regards to run time. It appears that gradient based FNN always has a better run-time performance, and sometimes random forest algorithm also uses less training time than EvoDNN. EvoDNN suffers from its slow convergence and long training time in order to achieve the superior accuracy performance. However, as a user is unlikely to know, in advance of training, which AF is most suited to a given data set when employing a homogeneous neural network, the user may trial-and-error a variety of AFs before finding a good one. This disadvantage may result in more time spent on finding the most suited AF for each given task. Also, if a given task is too complicated, random forest algorithm may not work well and spend as much training time as EvoDNN does.

In addition, there are methods to reduce the training time of EvoDNN. Considering the architecture of the EA, its nature is that many steps in the evolutionary process can be parallelized. Mutation, crossover, and selection are all processes that can be applied independently and in parallel to different individuals. This feature is not available for gradient based algorithm. The gradient descent must happen sequentially since each iteration depends on the new position of the model in the loss function space. Even parallel computation is possible for random forest algorithm since it can build many decision trees simultaneously, it is unable to calculate the feature weight of each feature, which as a result may produce inferior performance. This parallel processing can, in theory, decrease the run time and allow EvoDNN delivering superior results at a reduced run time.

Generally, it may be the case that a homogeneous FNN or random forest algorithm will deliver results faster than EvoDNN. However, for application where an increased accuracy or flexibility is more important and can be traded for additional running time, EvoDNN shows superior results.

5.3 Analysis of Activation Functions in EvoDNN

For brevity, we have omitted numbers of details about the location of each AF in each layer of EvoDNNs, and we only review EvoDNNs with the best multi-logloss values in [5, 10, 20, 40]-layer neural networks.

The vertical axis represents the index of hidden layer; for example, 1 in the vertical axis of Figure 5.15a means first hidden layer and 2 means second hidden layer etc. The percentage in the horizontal axis represents the proportion of certain AF occupied in a hidden layer. From left to right, the blue color means Sigmoid AF; the orange color is Tanh AF; the grey color is LReLU AF; and the yellow color is ReLU AF.
Figure 5.15: Analysis of Activation Functions in 5-layer EvoDNN
5.3.1 5-layer EvoDNN

Figure 5.15 shows the AF distribution in each layer of the 5-layer EvoDNN of each data set. We found that only the first hidden layer of Breast Cancer data set has only two types of AFs (Tanh and LReLU); the first hidden layer of other data sets have all four types of AFs. However, only the last hidden layer of Breast Cancer, Iris data sets have complete four types of AFs; the last layer of other five data sets: Abalone, Lymphography, Tumor, Wine, and Yeast, contain only three kinds of AFs. Furthermore, four out of the five data sets, which are Abalone, Tumor, Wine, and Yeast, do not have ReLU AF. One possible explanation is that the positive x-axis side of LReLU AF can replace the functionality of ReLU AF because the positive x-axis sides of both AF are identical. In addition, the negative x-axis side of ReLU is zero, which indicates no information can be extracted.

5.3.2 10-layer EvoDNN

The architecture of 10-layer EvoDNNs are similar to that of 5-layer EvoDNNs in the case that some hidden layers of a data set only have two or three types of AFs, but the rest hidden layers have all four sorts’ AFs. One interesting phenomenon we have observed in Figure 5.16 is that Abalone data set uses more Sigmoid AFs in total than other data sets; Iris, Tumor, and Yeast data sets contain more LReLU AFs in total than other data sets; Breast Cancer and Lymphography data sets have a balanced load of each AFs. We hypothesize that if one certain type of AFs is used more than other types’ in several hidden layers of an EvoDNN model, it may have positive impact on accuracy. In other words, a data set may have preference on certain type AF.

5.3.3 20-layer EvoDNN

Figure 5.17 and Figure 5.18 display the AF distribution of 20-layer EvoDNN. We notices that each data set has a more balanced configuration of each kind AF than that of [5, 10]-layer EvoDNNs.

5.3.4 40-layer EvoDNN

We represent the AF distribution of 40-layer EvoDNN in Figure 5.19 and Figure 5.20. We found that Breast Cancer, Iris, Lymphography, Tumor, Wine, and Yeast data sets utilizes less percentage of ReLU in total compared to other AFs. As discussed before, we argue that this is because the positive x-axis of LReLU AF can replace the functionality of ReLU AF. Therefore, if a neuron has LReLU AF, EvoDNN dose not have to change its AF to ReLU in order to improve the final accuracy.
Figure 5.16: Analysis of Activation Functions in 10-layer EvoDNN
Figure 5.17: Analysis of Activation Functions in 20-layer EvoDNN
Figure 5.18: Analysis of Activation Functions in 20-layer EvoDNN (Continue)
Figure 5.19: Analysis of Activation Functions in 40-layer EvoDNN
Figure 5.20: Analysis of Activation Functions in 40-layer EvoDNN (Continue)
Figure 5.21: Percentage of Each AF in X-layer EvoDNN Architectures
5.3.5 Summary

In Figure 5.21 the vertical axis represents the architecture of EvoDNN; for example, \textit{5 Layer} in the vertical axis means it is a 5-layer EvoDNN. The percentage in the horizontal axis represents the proportion of certain AF occupied in an EvoDNN. From left to right, the blue color means Sigmoid AF; the orange color is Tanh Af; the grey color is LReLU AF; and the yellow color is ReLU AF.

We can see that overall there is no certain AF that can occupy more than half neurons of an EvoDNN model. We argue this is because different AFs can abstract different features, which therefore makes EvoDNN powerful. In addition, from previous Figure 5.15 - 5.19, we know that no one entire hidden layer is occupied by one homogeneous AF. It is also hard to have all neurons in one hidden layer with homogeneous AFs, because in EA’s mutation stage, a neuron’s AF has chance to become another AF. Furthermore, we believe that if arbitrarily one type AF is used more than other types’ in several hidden layers of an EvoDNN model, it may have positive impact on final result. However, the explanation for such optimization problem will be studied in the future.
Chapter 6
Discussion and Conclusion

This paper has demonstrated the superior performance of employing EAs to optimize the weights, biases and activation functions of a deep heterogeneous FNN by extending EvoNN’s work. In Chapter 2, we discuss the fundamental concepts of neural network and its applications. Chapter 3 explains the methodology of employing EAs to evolve heterogeneous deep FNNs, followed by Chapter 4 which gives the experimental implementation details. In Chapter 5’s analysis, we conclude that in most cases EvoDNN can achieve smaller multi-logloss values when compared to EvoNN and homogeneous FNNs with the same number of hidden layers, and random forest algorithm. In addition, it offers a number of advantages over conventional training methods, such as stochastic gradient descent.

The main advantages of EvoDNN are:

1. no major restrictions to the error function, e.g., EvoDNN is a zero-order optimization algorithm and thus non-differentiable activation function may be used.

2. no major restrictions on the regularization methods which provide ways to control overfitting for complex neural network models.

3. convergence to a global minimum can be expected (the optimization of AF in hidden layer presents an added degree of flexibility in the neural network’s exploration of the fitness space within the target problem; however the running time for convergence can be long).

4. easy tuning of the evolutionary algorithm’s parameters (e.g. the size of population, mutation rate, the size of AFs’ set).

Due to the nature of evolutionary algorithm (properties of (1) and (2)), there are no special restrictions on the performance or AFs, which means the AFs and the fitness function do not need to be differentiable. This provides researchers with the ability to design arbitrary AFs, or use optimization targets that are not necessarily differentiable, and therefore new approaches to AF and performance can be studied with a few prerequisites. In addition, properties (1) implies that EvoDNN will not struggle with gradient vanishing and explosion...
[40] problem when adding more hidden layers, which is difficult to be addressed in a deep homogeneous FNN. Property (3) permits the use of evolutionary algorithm in validation of networks trained with gradient based methods, i.e., the optimum found by the evolutionary algorithm is never worse than the initial optimum found by a gradient based method, but it is highly probable that the evolutionary algorithm converges to a better optimum if such exists.

In Section 5.3, we analyze the AFs in each hidden layer of EvoDNN. We discovered that there is no certain AF could be loaded in more than half neurons of an EvoDNN model. We argue this is because different AFs can abstract different features and one instance in certain data set may contain multiple features. It is also difficult to have all neurons in one hidden layer with homogeneous AFs. This is because in EA’s mutation stage, even we set a low mutation rate, the chance of at least one neuron’s AF in certain hidden layer changing to another AF is relatively high.

However, the benefits of EvoDNN come at the cost of additional training time. This shows there exists a trade-off between obtaining higher accuracy results and the training time dedicated to gain such accuracy. If high prediction accuracy is more crucial than training time, then it is a worthwhile trade-off and one should use EvoDNN. It is also worth to mention that as the number of hidden layer increases, the running time of EvoDNN does not increase linearly; instead, the running time only slightly increases. We argue that this is because as the number of hidden layer increases, the solution space will be expanded and therefore it is relatively effortless to find a competitive configuration of the network.

It is also important to note that there are several potential methods for reducing the training time. For example, evolutionary algorithm can be parallelized effortlessly for most instances. This is because evaluation, crossover, mutation, and selection are all actions that operate on the individual level. These actions, therefore, can be divided among the number of processor cores or threads available for the system. Additionally, since the algorithm is written in Python, it can in fact be accelerated by rewriting the code in C++, a system level language.

EvoDNN provides new interesting topics for the training of FNN, since the use of EAs to create heterogeneous deep neural networks has so far received relatively little attention. Questions, such as, why evolutionary algorithm finds a global minimum for some problem configurations while gradient-based methods do not, and what is the effect of a selected activation function on the optimum found by evolutionary algorithm could be addressed in the future work. Furthermore, although this paper has demonstrated using EAs to evolve heterogeneous FNN, it only used a limited set of activation functions and optimized a few parameters associated with each neuron over a small range of values; more research could be done on the inclusion of additional activation functions or even letting the user create activation functions. Additionally, further research could investigate the complementary relationships between different types of activation functions. Lastly, an investigation of the
possibility of evolving and deciding the appropriate number of neurons in each hidden layer of EvoDNN and the number of hidden layers could be extended.

The results presented in this paper are significant as the method described for creating and training heterogeneous neural networks is likely to be compatible with other architectures of networks, such as RNNs [82]. Many different architectures of neural networks could benefit from this approach. Also, while we demonstrated that EvoDNN is useful for classification tasks in Computational Biology such as Breast Cancer, Lymphography or Tumor classification, the general framework of EvoDNN lends itself to tasks outside of the Computational Biology domain.
Bibliography


