Multi-Objective Evolutionary Neural Architecture Search for Recurrent Neural Networks

by

Reinhard Booysen

Submitted in partial fulfillment of the requirements for the degree
Master of Science (Computer Science)
in the Faculty of Engineering, Built Environment and Information Technology
University of Pretoria, Pretoria

January 2022

Publication data:

Reinhard Booysen. Multi-Objective Evolutionary Neural Architecture Search for Recurrent Neural Networks. Master's dissertation, University of Pretoria, Department of Computer Science, Pretoria, South Africa, January 2022.

Multi-Objective Evolutionary Neural Architecture Search for Recurrent Neural Networks

by

Reinhard Booysen
E-mail: reinn.cs@gmail.com

Abstract

Artificial neural network (ANN) architecture design is a nontrivial and time-consuming task that often requires a high level of human expertise. Neural architecture search (NAS) serves to automate the design of ANN architectures, and has proven to be successful in finding ANN architectures that can outperform those manually designed by human experts. It is often the case that in real world implementations of machine learning and ANNs, a reasonable trade-off is accepted for marginally reduced model accuracy in favour of lower computational resources demanded by the model. This study investigates the use of multi-objective evolutionary algorithms as an exploration strategy for NAS to evolve recurrent neural network (RNN) architectures. This allows for the consideration of the underlying computational resource requirements of the RNN models while maintaining an acceptable model performance-related objective. Additionally, methods such as weight inheritance, early stopping, and pruning of architectural unit connections during offspring generation, are investigated in the context of RNN architecture search to allow for more efficient exploration of the RNN architecture search space.

Keywords: Recurrent Neural Networks, Neural Architecture Search, Multi-Objective Evolutionary Algorithms.

Supervisor : Dr. A. S. Bosman
Department : Department of Computer Science
Degree : Master of Science

"Time is eternity that sees its own implementations."

Plato

Acknowledgements

I would like to express my deep and sincere gratitude to the following people and institutions:

- Doctor Anna Bosman, my supervisor, for her guidance and invaluable insight throughout this journey.
- My family for their continuous support.
- The Centre for High Performance Computing (CHPC), for the use of their distributed computing architecture resources.

Contents

List of Figures

List of Algorithms

List of Tables

Chapter 1

Introduction

The artificial neuron is a mathematical function whose development draws inspiration from the neurons found in the mammalian brain [56, 73, 76]. Multiple artificial neurons can be grouped in layers and connected to other neurons in different layers to form an artificial neural network (NN). NNs are machine learning models capable of approximating non-linear mathematical functions [34].

The structure of the NN is referred to as the NN's architecture, and different NN architectures exist to solve problems such as computer vision, forecasting, natural language processing, and more [31, 35, 60].

Recurrent neural networks (RNNs) are a set of specialized NN architectures that are designed specifically to learn from data with sequential or prominent temporal structures by simulating a discrete-time dynamical system [54, 57, 67]. The RNN architecture contains a hidden state component, which serves to provide a feedback connection into the NN. This hidden state allows the RNN to retain information as it progresses through the individual time steps of a particular input sequence [57, 67], thereby allowing the RNN to have a form of memory [12].

Designing a NN architecture for a specific problem is a nontrivial task and often requires a high level of human expertise [84, 90, 96]. A number of ways have been proposed to automate the task of NN architecture design, which is currently being researched under the neural architecture search (NAS) paradigm [24, 96]. NAS aims to automatically find NN architectures for a provided dataset with minimal human intervention, and has

already been successful in finding NN architectures that outperform state-of-the-art NN architectures designed by human experts [24, 50, 88].

Different NAS techniques exist for finding well-performing NN architectures, such as reinforcement learning (RL) methods [96], evolutionary algorithm (EA) methods [52], gradient-based methods [13], and more. Evaluating the performance of multiple NN architectures can become a computationally expensive task [42, 96], and a number of techniques have been proposed that attempt to improve the efficiency of NN architecture performance evaluation in NAS [35, 41, 94].

The majority of modern NAS studies focused on convolutional neural network (CNN) architecture search [32, 88]. Aside from Bayer *et al.* [6], there have not been any significant investigations into the use of multi-objective EAs for RNN architecture search. Furthermore, most of the methods proposed to make EA-based NAS methods more efficient have not been investigated in the context of RNN architecture search, e.g., network morphism with destructive network transformations [24].

The main purpose of this work is to develop a novel multi-objective EA-based NAS method to generationally evolve RNN architectures that are capable of learning from a provided dataset. Additionally, the use of network transformations during RNN architecture offspring generation are studied to gain a better understanding of how network transformations can be used for the optimisation of RNN architecture complexity related objectives.

The rest of this chapter is organised as follows. Section 1.1 discusses the objectives of the research. The contributions of the study are presented in Section 1.2. Finally, an outline of the remainder of this dissertation is provided in Section 1.3.

1.1 Objectives

The main objective of this study is to develop a multi-objective evolutionary algorithm based NAS method for RNN architectures. The following sub-objectives have been identified in working towards achieving the main objective:

1. Provide an overview of NNs and EAs that are used in this study.

2. Provide an overview of the existing NAS methods.

3. Investigate NAS methods for RNN architectures to gain a better understanding of the RNN architecture search space and encoding schemes.

4. Investigate the techniques that have been proposed to make NN architecture performance evaluation in NAS more efficient.

5. Propose a modular RNN architecture search space and encoding scheme.

6. Propose a multi-objective EA RNN architecture search method that can explore the modular RNN architecture search space.

7. Investigate the techniques that can be implemented to make the proposed RNN architecture search method more efficient.

8. Implement the proposed multi-objective EA NAS method to search for RNN architectures for natural language processing datasets.

9. Identify any shortcomings in the proposed RNN architecture search method, and propose possible future enhancements.

1.2 Contributions

The contributions of this study are summarised as follows:

- A cell-based RNN architecture search space is introduced. The proposed search space is expressive enough to allow for the discovery of novel RNN units, and allows for constructive as well as destructive network transformations.

- Appropriate network transformations that allow for the optimisation of RNN architecture complexity related objectives are proposed. Destructive network transformations are defined that remove units from the RNN architecture, which reduces the overall complexity of the RNN architecture and effectively leads to a decrease in the computational resource demand of the model.

- A modular RNN architecture block encoding scheme that allows for low-level RNN architecture evolution is proposed. Each block represents a single unit in the RNN

architecture, and an RNN architecture can then be encoded using a number of blocks. An RNN architecture encoded by a number of connected blocks can be easily transformed by adding or removing blocks. The modularity of the block encoding scheme allows for the analysis of the low-level changes made to an RNN architecture during evolution. Additionally, by considering the number of blocks an RNN architecture contains as an architecture complexity objective, the effect of the network transformations on the resulting model accuracy can be studied specifically within a multi-objective optimisation paradigm.

- MOE/RNAS: a novel multi-objective EA-based NAS algorithm for RNN architecture search is proposed. The MOE/RNAS algorithm relies on an NSGA-II inspired multi-objective EA for the exploration of the cell-based RNN architecture search space. The MOE/RNAS algorithm employs a network morphism approach for offspring generation instead of performing any explicit crossover or mutation operators. The destructive network transformations considered by the MOE/RNAS algorithm's network morphism component allow for the optimisation of RNN architecture complexity related objectives along with the optimisation of an architecture performance objective, such as model accuracy.
- An empirical analysis of the MOE/RNAS algorithm's effectiveness to find RNN architectures for natural language processing (NLP) datasets is conducted.
- Experiments show that the proposed MOE/RNAS algorithm is capable of evolving RNN architectures to optimise multiple objectives, which includes at least one RNN architecture complexity objective.
- Empirical results show that the proposed MOE/RNAS algorithm can automatically find novel RNN architectures that dominate manually designed RNN architectures when multiple objectives are considered for RNN architecture performance evaluation.

1.3 Dissertation Outline

The remainder of the dissertation is organised as follows:

- **Chapter 2** discusses NNs, with a focus on RNNs.
- **Chapter 3** discusses EAs and multi-objective EA based problem solving.
- **Chapter 4** discusses NAS and reviews the currently available literature on EA based NAS methods.
- **Chapter 5** proposes the MOE/RNAS algorithm: a novel multi-objective EA-based NAS method to search for RNN architectures, which is presented in the form of a framework.
- **Chapter 6** empirically analyses the effectiveness of the proposed MOE/RNAS algorithm to search for RNN architectures in the NLP domain.
- **Chapter 7** provides a summary of all the findings and conclusions of the presented work, and includes a list of ideas for future work based on the presented work.

The following appendices are included:

- **Appendix A** provides a list of the important acronyms used or newly defined in the course of this work, as well as their associated definitions.
- **Appendix B** lists and defines the mathematical symbols used in this work, categorised according to the relevant chapter in which they appear.

Chapter 2

Artificial Neural Networks

Artificial neural networks (NNs) are machine learning algorithms capable of approximating various mathematical functions [34], and gained popularity as a result of their real-world applications, some of which include computer vision, forecasting, natural language processing, and more [31, 35, 60]. Underlying NNs is the artificial neuron [78]. The artificial neuron owes its existence to earlier neuroscientific research and hypotheses around the electrochemical signals in the brain that drive mental activity [56, 73, 78]. Researchers in the field of computer science derived interpretations of these hypotheses that formed the foundation of the artificial neuron [56], along with inspiration from the biological structure of the neurons found in the mammalian brain [56, 73, 76].

The rest of this chapter outlines the fundamentals of NN architectures and the importance of proper NN architecture design in the context of machine learning problem solving. Section 2.1 describes the underlying components that make up the NN structure. In Section 2.2, the recurrent neural network and its architecture are discussed in greater detail. Section 2.3 concludes the chapter.

2.1 Neural Network Structures

The brain cells, called neurons, receive electrochemical signals as inputs. These signals can originate from a vast array of sources, which includes the nervous system, other cells, or neurons [78]. A particular neuron can activate (or fire) if some linear combination of

the inputs it receives exceeds some expected (hard or soft) threshold [76]. This event results in another electrochemical signal being sent onwards to subsequently connected neurons. The individual neurons will learn over time what best threshold is the appropriate boundary for deciding whether the neuron should activate/fire a signal onwards (i.e., send an output signal).

The rest of this section discusses the artificial neuron that was designed based on the biological neuron, and how multiple artificial neurons can be connected in a network that is capable of approximating complex mathematical functions. The artificial neuron is discussed in Section 2.1.1. Section 2.1.2 discusses activation functions, Section 2.1.3 discusses neural network architectures, and Section 2.1.4 discusses the training of neural networks.

2.1.1 Artificial Neuron

The artificial neuron, as it was introduced in 1943 by McCulloch and Pitts [56], aims to mimic a similar behaviour to that of the biological neuron, in what is fundamentally a mathematical paradigm. The artificial neuron accepts some linear combination of inputs, and returns a relevant output value that is calculated for the particular combination of inputs. A weight vector, $\mathbf{w}$, is incorporated wherein each of the neuron's inputs, x_n, has a specific weight value associated with it such that for n inputs, $\mathbf{w}_n \in \mathbb{R}$. These weight values represent the connection strength of the respective inputs [9]. The artificial neuron aggregates the weighted sum of the respective inputs and applies the result to an activation function, which determines the output of the artificial neuron [78]. A bias value is also incorporated which involves augmenting the input vector of the artificial neuron to include an additional input x_{n+1}. Setting the value of x_{n+1} to -1 allows for the corresponding weight w_{n+1} to serve as a threshold value that influences the aggregated weighted sum of the artificial neuron's inputs, thereby controlling when the activation function triggers. The net input signal of the artificial neuron is calculated by:

$$net = \sum_{i=1}^{n} x_i w_i - b \tag{2.1}$$

where b represents the bias value such that $b = x_{n+1}w_{n+1} = -w_{n+1}$ [26]. The output value of the artificial neuron is calculated by applying the net input signal to an activation

function, f, such that:

$$\hat{y} = f(net) \tag{2.2}$$

which can be seen in Figure 2.1.

Rosenblatt [73] introduced an artificial neuron, called a perceptron, that employs the step activation function. The step activation function is a linear activation function that compares the net input signal of the neuron with the threshold θ to determine the output signal. The output of the step activation function is 1 if the net input signal is more than the threshold θ, and zero otherwise. There are other activation functions that can be used instead of the step activation function, which is discussed below.

2.1.2 Activation Functions

Activation functions serve to determine the output of the artificial neuron [78], and play an important role in enabling artificial neurons to learn higher order polynomials [64]. The artificial neuron's activation function can either be linear or non-linear. Linear activation functions include the identity and linear functions, among others. Typical non-linear activation functions include the logistic (sigmoid) function, hyperbolic tangent, rectified linear unit, softmax, and more.

A single artificial neuron can only represent linearly separable functions [26]. Connecting multiple artificial neurons with non-linear activation functions in a network to form a NN allows for the representation of non-linearly separable functions [34]. Furthermore, Nwankpa *et al.* [64] noted that non-linear activation functions are differentiable which is an important requirement for back-propagated training of networks of artificial neurons (see Section 2.1.4).

The unipolar sigmoid function (also called the logistic function) is a commonly used non-linear activation function that is defined by:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.3}$$

with an output range limited to $(0, 1)$. The bipolar sigmoid function, on the other hand, has an output range of $(-1, 1)$ and is defined by:

$$\sigma(x) = \frac{1 - e^{-x}}{1 + e^{-x}}. \tag{2.4}$$

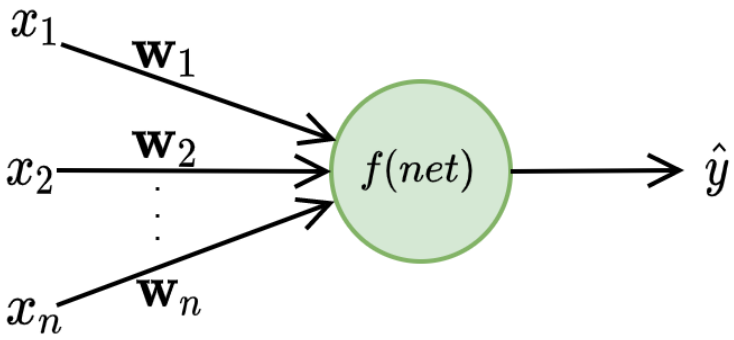


Figure 2.1: The artificial neuron.

The hyperbolic tangent function, which is also sigmoidal, is defined by:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.5}$$

with an output range of $(-1, 1)$. A comparison between the unipolar sigmoid, bipolar sigmoid, and hyperbolic tangent activation functions can be seen in Figure 2.2. Table 2.1 and Table 2.2 list the function, range and derivatives of some commonly used linear and non-linear activation functions, respectively.

Choosing an appropriate activation function will depend on the specific task or problem to be solved. Sharma *et al.* [78] noted that a general rule of thumb for selecting an activation function does not exist, and the process of selecting an activation function may require experimentation with different activation functions to determine which yield the best results. Sharma *et al.* [78] do however provide some suggestions for choosing an activation function, for example: a combination of sigmoidal functions can give better results for classification tasks and the ReLU activation function often yields better results when used as the activation function of neurons in the hidden layers of the NN [78].

2.1.3 Neural Network Architectures

Artificial neurons can be grouped in layers and connected to other neurons in different layers to form a NN. The structure of a NN, such as the number of neurons and how these neurons are connected, is referred to as the NN architecture [54]. Fundamentally, the NN architecture can either follow a feed-forward network structure or a recurrent

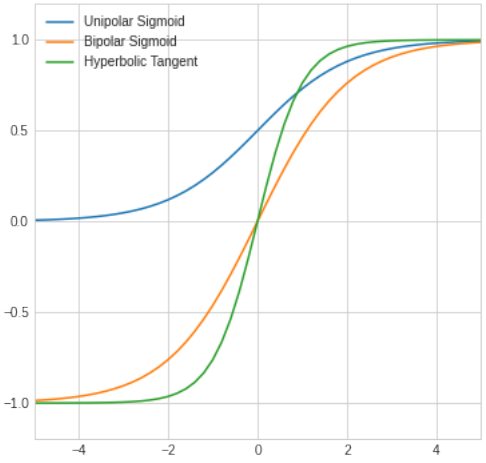


Figure 2.2: Graph depicting the unipolar sigmoidal, bipolar sigmoidal and hyperbolic tangent activation functions.

network structure [76]. The connections between neurons in a feed-forward network are orientated in a single direction, with no loops or internal state [76], an example of such a NN is shown in Figure 2.3.

The recurrent neural network (RNN) structure has connections that feed outputs back into the network as inputs [57, 76]. The recursive structure of the recurrent network architecture makes it more complicated, and more difficult to train compared to feed-forward network architectures [57, 63, 76]. RNNs are discussed in more detail in Section 2.2.

NNs with non-linear activation functions are capable of approximating any continuous function [34] requiring at minimum a single hidden layer [85]. In such a configuration, the simplest NN is composed of three layers: an input layer, a hidden layer, and an output layer. The input layer receives the raw data values and serves to send these inputs onwards to the subsequent hidden layer [78]. Hidden layers allow for increasingly complex decision-making on a more abstract level [28]. The output layer receives as its input, the output of the layer immediately preceding it. The output layer returns the final result (or output) of the NN. When neurons are connected to form a NN, the resulting model can be used to represent a non-linear function [34, 78].

Calculating the output of a NN starts by obtaining the output values of the hidden

Activation function	Function	Range	Derivative
Identity	$f(x) = x$	$(-\infty, \infty)$	$f'(x) = 1$
Linear	$f(x) = ax$	$(-\infty, \infty)$	$f'(x) = a$

Table 2.1: Linear activation functions from [64, 78, 80]

units in the NN. For a given hidden unit that accepts n linear combination of input values and a bias unit, the weighted sum of these inputs are calculated by:

$$a_j = \sum_{i=1}^{n+1} w_{ji} x_i. \tag{2.6}$$

The final activated output of the particular hidden unit, z_j, is then determined by applying the chosen activation function $g(\cdot)$ to the result of equation (2.6) such that:

$$z_j = g(a_j). \tag{2.7}$$

Given the output values for M hidden units, the output values for each of the k output units in the NN can then be calculated. As with the hidden units, this starts by obtaining the weighted sum of the output unit's inputs such that:

$$a_k = \sum_{j=1}^{M+1} w_{kj} z_j \tag{2.8}$$

The chosen activation function is then applied such that the final output of each output unit is given by:

$$\hat{y}_k = \widetilde{g}(a_k) \tag{2.9}$$

where $\widetilde{g}$ need not be similar to the activation functions chosen for the hidden units.

NN architecture design is a time-consuming task which requires expert knowledge [84, 90, 96]. The results from Jozefowicz *et al.* [37] and Merity *et al.* [59] showed the significance of NN architecture design, wherein they compared the validation and test results of different RNN architectures and found that appropriate architectural adaptations can lead to improved model performance (see Section 2.2.3). Bengio [7] conjectured that improper NN architecture design may lead to computational and statistical consequences which can result in the NN being unable to produce good outputs for new inputs that were not used during training.

Activation function	Function	Range	Derivative
Logistic (sigmoid)	$f(x) = \frac{1}{1+e^{-x}}$	$(0, 1)$	$f'(x) = f(x)(1 - f(x))$
Hyperbolic tangent	$f(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$	$(-1, 1)$	$f'(x) = 1 - f(x)^2$
Softmax	$f(x)_i = \frac{e^{x_i}}{\sum_{k=1}^{K} e^{x_k}}$ for $i = 1, ..., K$	$(0, 1)$	$\frac{\partial f_i(x)}{\partial x_j} = f(x)_i(x)(\delta_{ij} - f_j(x))$
Rectified Linear Unit (ReLU)	$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases}$ $= max\{0, x\}$	$(0, \infty)$	$f'(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } x < 0 \\ \text{undefined}, & \text{if } x = 0 \end{cases}$
Leaky ReLU (LReLU)	$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0.01x, & \text{if } x < 0 \end{cases}$	$(-\infty, \infty)$	$f'(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ 0.01, & \text{if } x < 0 \end{cases}$
Exponential Linear Unit (ELU)	$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \alpha(e^{x} - 1), & \text{if } x < 0 \end{cases}$	$(-\alpha, \infty)$	$f'(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ f(x) + \alpha, & \text{if } x < 0 \end{cases}$

Table 2.2: Non-linear activation functions from [64, 78, 80]

Bengio [7] suggested that the depth of a NN is an important design decision, since deep networks can solve complex problems easier compared to shallow networks. The number of layers a NN has constitutes the depth of that particular network [67]. Deep NNs with more hidden layers have proven to be significantly more efficient at approximating functions compared to shallow NNs [7, 67].

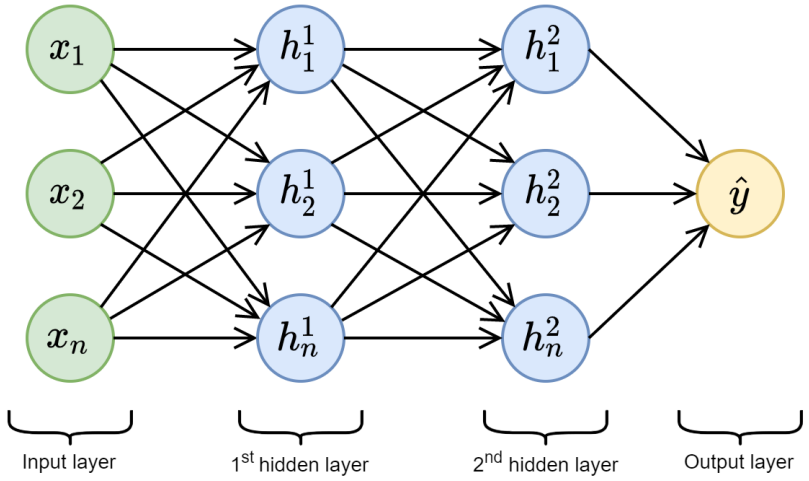


Figure 2.3: A neural network with two hidden layers.

2.1.4 Training Neural Networks

Training NNs consists of implementing and executing a specific training algorithm that iteratively updates the NN's parameters so that the NN can more closely approximate the mapping from the input values to the relevant output value(s) [30]. The parameters of a NN are its weight and bias values [30], and updating these parameters during the training process is how the NN *learns* to approximate the relevant function. The terms *training* and *learning* are commonly used interchangeably.

The training of a NN is done in either supervised, unsupervised, or reinforcement learning paradigm [9]. Supervised learning describes an environment in which the NN is trained on a labeled dataset where each input value has a known (expected) output value, called the target value [9]. Unsupervised learning, on the other hand, does not have a labeled dataset with target values. Instead, the objective of training a NN in an unsupervised setting is for the NN to learn useful structures or clusters in the data [76]. With reinforcement learning, the model is given rewards for returning good output values [76]; the model is then expected to learn which of its prior actions resulted in good outputs [63]. For this study, it is only the supervised learning paradigm that is of interest.

During training in a supervised learning environment, the labeled dataset is randomly split into two separate sets: a training set and a validation (generalisation) set. The

training set is used for training the NN, whereas the validation set is used for evaluating the NN, that is, to test its generalisation ability on the *unseen* data that the NN has not been trained on. As part of the training process, the training dataset is traversed so that each of the input patterns in the training dataset is fed forward through the NN to obtain the predicted output value for the particular input pattern. Using an error function, the error value is then computed for the particular input pattern by comparing the NN's predicted output value with the expected (target) output value. The total loss of the NN is then determined by aggregating the error values of some set of input patterns. Commonly used error functions include the sum of squared errors (SSE):

$$E = \frac{1}{2}\sum_{n=1}^{N}\sum_{k=1}^{K}(\hat{y}_{nk} - y_{nk})^2 \tag{2.10}$$

and cross-entropy:

$$E = -\sum_{n=1}^{N}\sum_{k=1}^{K}(y_{nk}\log(\hat{y}_{nk}) + (1 - y_{nk})\log(1 - \hat{y}_{nk})) \tag{2.11}$$

for $n = 1, ..., N$ examples in the labeled dataset and $k = 1, ..., K$ output nodes where $\hat{y}_{nk}$ represents the predicted output value of the k^{th} output node for the n^{th} input pattern and y_{nk} represents the target value of the k^{th} output node for the n^{th} input pattern [30].

A common approach is to use a training algorithm that employs gradient descent to explore the search space of possible NN parameter values in order to minimize the calculated loss value of the NN [30, 63]. The backpropagation training algorithm is one such gradient-based training technique that uses the chain rule for calculating the partial derivatives (gradient) of the error function with respect to the parameters of the NN [9, 30]. The gradient vector is then used to update the NN's parameters during a backward pass of the training algorithm. For a given labeled dataset containing N input patterns, consider the loss value E^m as a differentiable function such that

$$E^m = E^m(\hat{y}_1, ..., \hat{y}_k)$$

where $\hat{y}_k$ represents the predicted output value of the k^{th} output node, where $1 \leq m \leq N$ such that m defines the number of input patterns to consider for error value aggregation.

For a given weight value, w_{kj}, between units a_k and a_j, the gradient for w_{kj} is calculated by

$$\frac{\partial E^m}{\partial w_{kj}} = \frac{\partial E^m}{\partial a_k}\frac{\partial a_k}{\partial w_{kj}}\frac{\partial net_k}{\partial w_{kj}} = \frac{\partial E^m}{\partial a_k}\frac{\partial a_k}{\partial w_{kj}}\hat{y}_k \tag{2.12}$$

for the k^{th} output node $\hat{y}_k$, and the w_{kj} weight value is then updated by

$$\Delta w_{kj} \leftarrow \upsilon\left(-\frac{\partial E^m}{\partial w_{kj}}\right) \tag{2.13}$$

where υ denotes the learning rate, a hyper-parameter that is used as a multiplicative factor for scaling the gradient [9].

Updating the NN weight values can be done for every input pattern in the training dataset, which is called stochastic training. An alternative approach can be followed where the input patterns of the training dataset are split into m sized subsets. A set of m input patterns is referred to as a batch. After each of the inputs in the batch has been evaluated, a cumulative error value for the batch is calculated, which is then used for weight updates. For smaller values of m that are close to 1, gradient descent trajectory towards the minima is stochastic whereas larger values for m closer to the total number of input patterns in the training dataset result in a smoother optimization trajectory towards the minima. Batch training refers to the case where the training dataset is treated as a single batch, whereas mini-batch training considers smaller batch sizes. An illustration of the varying trajectories can be seen in Figure 2.4, which shows the best-case scenarios for each of the respective trajectories.

The training process is repeated until some pre-defined condition is met, such as the maximum number of iterations (called *epochs*), or a threshold for the calculated loss value has been reached, to name just a few. It is possible to train a NN for too long, resulting in the NN exhibiting very good performance over the training dataset, but poor performance on unseen data; this is referred to as overfitting [63]. Overfitting can also occur when the NN architecture has an unnecessarily high level of complexity, thus having more parameters than what is required [76]. Additionally, a model is said to be overfitting when it has a high variance and a low bias [9]. Variance measures the difference among predicted outputs of the model for similar input values whereas bias measures the difference between the predicted output values and the target values [9]. NNs that have not been trained for a sufficient number of epochs may exhibit poor

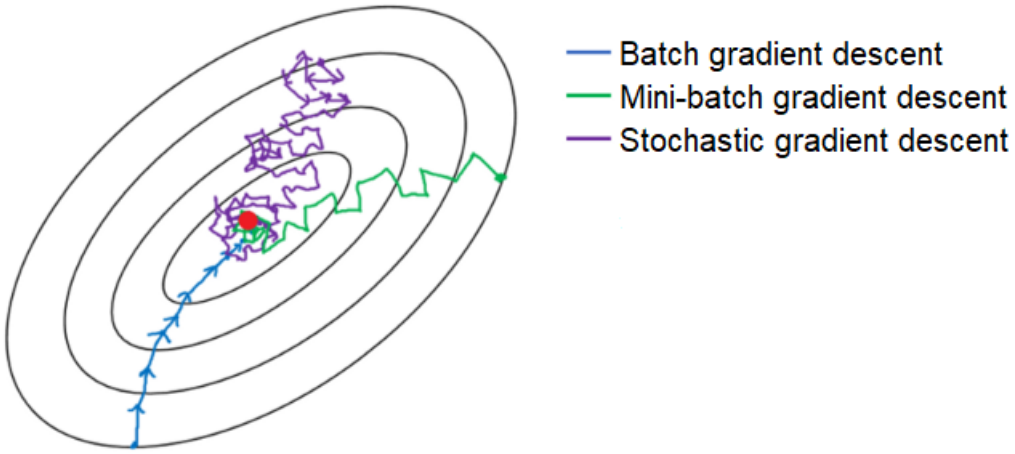


Figure 2.4: Illustration of varying gradient descent optimization trajectories based on the number of input patterns considered.

performance on both the training dataset and unseen data, which is called underfitting. Underfitting describes the scenario where a model has a low variance and a high bias [9, 27]. Underfitting can also be caused by a NN architecture with an insufficient number of parameters, which means that the NN architecture does not have enough artificial neurons to learn from the training data [7].

2.2 Recurrent Neural Networks

Recurrent neural networks (RNNs) are a set of specialized NN architectures that are designed specifically to learn from data with sequential or prominent temporal structures by simulating a discrete-time dynamical system [54, 57, 67]. RNNs have been applied to a number of problems which categorically include forecasting [43], speech recognition [29], natural language processing [59, 61], and more.

The RNN architecture is discussed in more detail in Section 2.2.1. Section 2.2.2 discusses RNN training and the problems often encountered when training RNNs, and Section 2.2.3 provides an overview of RNN architectures that have been designed to deal with the problems encountered during RNN training.

2.2.1 Recurrent Neural Network Architecture

The main characteristic of the RNN is the hidden state that forms part of the architecture. The hidden state serves to provide a feedback connection, which allows the NN to retain information as it progresses through the time steps of the input sequence [57, 67], thereby allowing the RNN to have a form of memory [12]. This hidden state is an essential component of the RNN architecture that enables more efficient processing of datasets with sequential attributes or significant time dependencies [12, 43]. For feed-forward NNs to process datasets with significant temporal structures, time needs to be represented explicitly by adding input nodes corresponding to the relevant time steps of the input sequence, whereas the recursive structure of the RNN allows the NN to represent time implicitly [23].

There exist, among a few, two particularly interesting ways that have been introduced to provide the aforementioned feedback connection into the network. The first one was introduced by Jordan [36], who proposed feeding the output layer back into the NN. The Jordan [36] architecture allows the network to incorporate the sequence of output values through time. Elman [23] proposed an alternative approach a few years later that feeds the hidden layer back into the NN instead. The Elman [23] approach allows the network to have better recall of previously encountered inputs.

In what is fundamentally its most basic form, the simple recurrent neural network (SRN) accepts a sequence of T inputs, $(\mathbf{x}_1, ..., \mathbf{x}_T)$, contains a hidden state $\mathbf{h}_t$, and produces an output $\hat{\mathbf{y}}_t$, where t represents a specific time step in the input sequence [12, 29, 57]. The hidden state of the RNN architecture proposed by Jordan [36] is given by:

$$\mathbf{h}_t = f_h(\mathbf{x}_t, \hat{\mathbf{y}}_{t-1})$$

where $\hat{\mathbf{y}}_{t-1}$ refers to the output of the RNN at time step $t-1$ and f_h is the activation function of the hidden state. The RNN architecture proposed by Elman [23], on the other hand, calculates the value of the hidden state by:

$$\mathbf{h}_t = f_h(\mathbf{x}_t, \mathbf{h}_{t-1})$$

where $\mathbf{h}_{t-1}$ refers to the hidden state of the preceding time step. The output of the recurrent network, $\hat{\mathbf{y}}_t$, is then given by $\hat{\mathbf{y}}_t = f_o(\mathbf{h}_t)$ where f_o can be a different activation

function to the one chosen for the hidden state. The convention followed by this study assumes the architecture that was proposed by Elman [23] as the default for the basic RNN or the simple RNN, unless otherwise stated.

More formally, the hidden state, $\mathbf{h}_t$, and the output, $\hat{\mathbf{y}}_t$, of the RNN architecture are defined by:

$$\mathbf{h}_t = f_h(\mathbf{W}_h\mathbf{x}_t + \mathbf{U}_h\mathbf{h}_{t-1} + \mathbf{b}_h), \tag{2.14}$$

$$\hat{\mathbf{y}}_t = f_o(\mathbf{W}_y\mathbf{h}_t + \mathbf{b}_y) \tag{2.15}$$

where $\mathbf{W}$ and $\mathbf{U}$ are weight matrices and $\mathbf{b}$ a bias value [29]. It is often the case that for f_h and f_o element-wise non-linear activation functions are used [15, 67]. Sigmoidal activation functions, such as the logistic (sigmoid) function or the hyperbolic tangent function, are typically used for f_h [12, 67]. There are some challenges associated with updating the $\mathbf{W}$ and $\mathbf{U}$ weight matrices during gradient-based training of the RNN as a result of the RNNs recursive structure. These challenges are discussed in the following section.

2.2.2 Training Recurrent Neural Networks

Training RNNs is often considered more complicated than training feed-forward NNs [8, 12, 18, 67, 68]. Due to the temporal structure of the input data that the RNN accepts, time-delayed updating of parameters is required during training [57]. A generalization of the backpropagation training algorithm discussed in Section 2.1.4, called backpropagation through time (BPTT), proposed by Rumelhart *et al.* [75], is commonly used for training RNNs [8, 68]. The BPTT algorithm stores the activations of the neurons as it progresses through time, and during the backward pass, these activations are recursively used for calculating the gradients with respect to the RNN parameters [8, 75]. During training, the RNN is "unrolled" in time, essentially duplicating the model for each time step [68], which can be seen in Figure 2.5. The error values for all time steps are accumulated and then used for relevant weight adjustments [68].

Bengio *et al.* [8] have shown that the RNN suffers from gradient problems during gradient-based training where input sequences with longer-term dependencies are used.

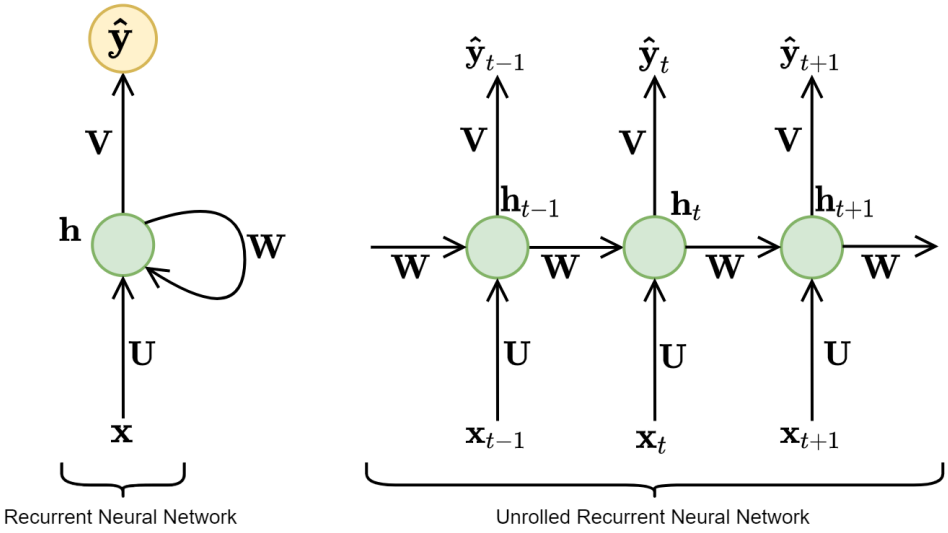


Figure 2.5: A recurrent neural network (left) and an unrolled recurrent neural network (right).

In this case, during backward propagation, the gradient values will either grow exponentially, or go exponentially fast to zero, such that they become insignificant [8, 68]. These problems are referred to as the exploding and vanishing gradients, respectively [8].

Pascanu *et al.* [68] proposed an approach called gradient-clipping to deal with the exploding gradients. This approach is presented as a mechanism that quite simply rescales the gradient values if they are larger than some predefined threshold value. Hochreiter and Schmidhuber [33] proposed a new RNN architecture that is capable of addressing the vanishing gradient problem. The architecture proposed by Hochreiter and Schmidhuber [33] is discussed in the next section.

2.2.3 Long Short-Term Memory

In an attempt to address the recurrent network's vanishing gradient problem, Hochreiter and Schmidhuber [33] introduced a novel RNN architecture dubbed Long Short-Term Memory (LSTM) [33]. The LSTM deals with the vanishing gradient problem by employing memory cells and gate units [33] with the intuition being that the respective units can each form some type of oscillating mechanism, acting like soft switches, to control the amount of information flowing through the network [43]. An illustration of

the LSTM architecture can be seen in Figure 2.6.

The various gate units of the LSTM are defined by:

$$\mathbf{f}_t = \sigma(\mathbf{W}_{xf}\mathbf{x}_t + \mathbf{W}_{hf}\mathbf{h}_{t-1} + \mathbf{b}_f), \tag{2.16}$$

$$\mathbf{i}_t = \sigma(\mathbf{W}_{xi}\mathbf{x}_t + \mathbf{W}_{hi}\mathbf{h}_{t-1} + \mathbf{b}_i), \tag{2.17}$$

$$\mathbf{o}_t = \sigma(\mathbf{W}_{xo}\mathbf{x}_t + \mathbf{W}_{ho}\mathbf{h}_{t-1} + \mathbf{b}_o), \tag{2.18}$$

$$\mathbf{g}_t = tanh(\mathbf{W}_{xg}\mathbf{x}_t + \mathbf{W}_{hg}\mathbf{h}_{t-1} + \mathbf{b}_g), \tag{2.19}$$

$$\mathbf{c}_t = \mathbf{f}_t \cdot \mathbf{c}_{t-1} + \mathbf{i}_t \cdot \mathbf{g}_t, \tag{2.20}$$

$$\mathbf{h}_t = \mathbf{o}_t \cdot tanh(\mathbf{c}_t), \tag{2.21}$$

where $\mathbf{f}_t$ is the forget gate, $\mathbf{i}_t$ the input gate, $\mathbf{o}_t$ the output gate, and $\mathbf{g}_t$ is called the input modulation gate. The sigmoid activation function is used for the $\mathbf{f}$, $\mathbf{i}$, and $\mathbf{o}$ gates, which allows the architecture to remain differentiable [40]. $\mathbf{c}_t$ is often referred to as the "memory cell" or "cell state", and contains information (memory content) from previously encountered inputs of a particular input sequence [19, 38].

The $\mathbf{f}$, $\mathbf{i}$, and $\mathbf{o}$ gates serve to control the memory cell where the $\mathbf{f}$ gate resets the cell to zero, $\mathbf{i}$ controls when the memory cell is updated, and $\mathbf{o}$ controls how much of the information is fed to the hidden state [40]. The input modulation gate, $\mathbf{g}$, is responsible for additively modifying the contents of the memory cell. The modulation of the $\mathbf{g}$ gate is what allows the gradients of the memory cell to flow backwards through time uninterrupted [38, 40].

The use of these various gate units in the LSTM architecture has inspired a number of research publications related to RNN architectures [15, 38, 68]. Additionally, other papers were published where the use of the LSTM on specific problems such as language processing, were investigated [83]. Among these published works were approaches where the LSTM was implemented in different configurations, such as using multiple stacked LSTMs in one network [43], bi-directional LSTM networks for encoding and decoding in translation tasks [15], LSTM networks that share embedding layers in language processing tasks [58], combining the LSTM architecture with convolutional neural networks for language model embedding [37], and more.

One notable alternative to the LSTM is the Gated Recurrent Unit (GRU) introduced by Cho *et al.* [15] in 2014. The premise of the GRU is that it allows the recurrent unit

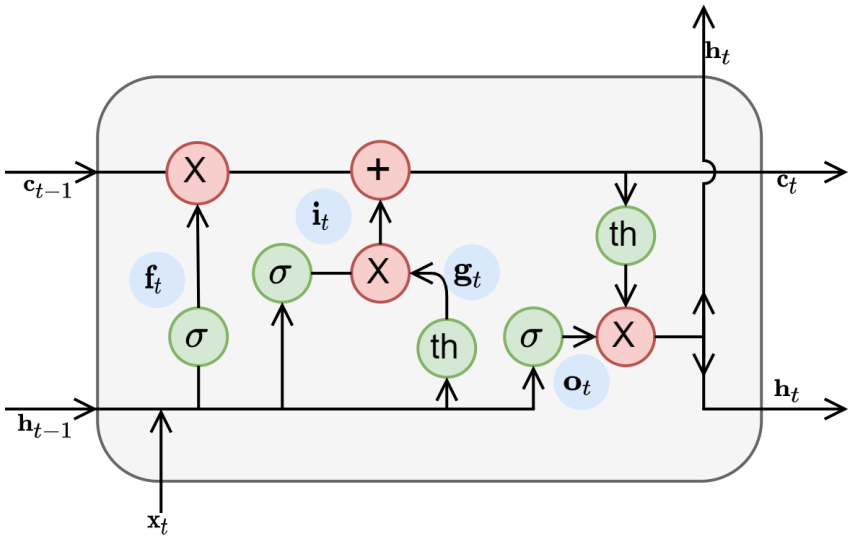


Figure 2.6: LSTM architecture [33].

to capture the dependencies of different time scales [18]. The GRU employs the same gate-unit philosophy of the LSTM, and the GRU's gate units are defined by:

$$\mathbf{z}_t = \sigma(\mathbf{W}_{xz}\mathbf{x}_t + \mathbf{W}_{hz}\mathbf{h}_{t-1} + \mathbf{b}_z), \tag{2.22}$$

$$\mathbf{r}_t = \sigma(\mathbf{W}_{xr}\mathbf{x}_t + \mathbf{W}_{hr}\mathbf{h}_{t-1} + \mathbf{b}_r), \tag{2.23}$$

$$\mathbf{n}_t = tanh(\mathbf{W}_{xn}\mathbf{x}_t + \mathbf{W}_n(\mathbf{r}_t \cdot \mathbf{h}_{t-1})), \tag{2.24}$$

$$\mathbf{h}_t = \mathbf{z}_t \cdot \mathbf{h}_{t-1} + (1 - \mathbf{z}_t) \cdot \mathbf{n}_t. \tag{2.25}$$

Unlike the LSTM, the GRU does not have a separate memory cell. The GRU uses the update gate $\mathbf{z}_t$ and reset gate $\mathbf{r}_t$ to maintain the unit's memory content, which represents the relevant information from previously encountered input steps of the particular input sequence.

The GRU's update gate $\mathbf{z}_t$ controls how much of the unit's memory content is retained and when new content gets added to the hidden state $\mathbf{h}_t$ [19]. The update gate $\mathbf{z}_t$ is therefore reminiscent of the input gate $\mathbf{i}_t$ of the LSTM. The reset gate $\mathbf{r}_t$ of the GRU serves to modulate the states of the previous step $\mathbf{h}_{t-1}$, thereby allowing the GRU to only consider the previous hidden states if they are deemed necessary [15]. The functionality of the GRU's reset gate is therefore similar to that of the LSTM's forget gate $\mathbf{f}_t$. The

LSTM uses its output gate $\mathbf{o}_t$ to modulate the exposure of the memory cell's content, whereas the GRU does not have a gate unit responsible for controlling memory content exposure, and instead exposes the full memory content at each time step [19]. The next section discusses some specific applications of RNNs which include results from studies where the LSTM and GRU were compared on specific machine learning tasks.

Applications of Recurrent Neural Networks

Both the LSTM and GRU architectures have been successfully used in a number of machine learning tasks, some of which include arithmetic problems of sequences of numbers [38], sequence prediction in context-free languages [83], classification tasks in the medical domain [5], and language modeling [19, 40], among others.

A standard language modeling task is often used for benchmarking RNN architecture performance [37, 38, 59, 96], and the objective of the standard language modeling task is to provide the model with some context, such as a sequence of consecutive words, which the model should use to predict the next word in the sequence [61]. This task can also be performed on the character-level as opposed to word-level modeling [83].

Calculating the performance of a model that is implemented for the standard language modeling task is based on how well the model is able to predict the next word (or character), which is commonly represented by a metric called *perplexity* [37, 39]. Perplexity measures how accurately a model can predict the next word, such that given a test set $D_G = d_1 d_2 ... d_Q$, the perplexity is calculated by:

$$\begin{aligned} PP(D_G) &= P(d_1 d_2 ... d_Q)^{-\frac{1}{Q}} \\ &= \sqrt[Q]{\frac{1}{P(d_1 d_2 ... d_Q)}} \end{aligned}$$

normalized by the number of words [39]. As noted by Jurafsky and Martin [39], the chain rule can be used to expand the probability of D_G such that:

$$PP(D_G) = \sqrt[Q]{\prod_{i=1}^{Q} \frac{1}{P(d_i | d_1 ... d_{i-1})}} \tag{2.26}$$

Zaremba *et al.* [92] have reported a 78.4 test perplexity using a large machine learning model configuration based on the LSTM architecture, comprising 66M parameters [59], after being trained on the Penn Treebank dataset [61]. In a later publication, Merity *et al.* [59] have proposed an alternative configuration of an LSTM-based model, wherein a test perplexity of 52.8 on the same dataset was achieved. Jozefowicz *et al.* [38] compared the LSTM and GRU architectures on the same language modeling task, and found that the GRU-based model achieved a test perplexity of 91.7 whereas the model with the LSTM architecture in their study achieved a test perplexity of 81.4 [38].

These differences in model performance are noteworthy, as it shows the impact of the overall machine learning model configurations, given that the same underlying RNN architectures were used, such as the LSTM and GRU. Designing a machine learning model implementation for such a task can be a difficult and time-consuming process [96]. Apart from merely considering the test and validation metrics of the model to evaluate the model's performance, some real-world applications may also consider other factors, such as the number of parameters a model has, the amount of computational resources required by the model, and more [17, 24, 84], all of which could contribute towards the complexity of designing and implementing the appropriate machine learning model.

2.3 Summary

This chapter discussed the fundamentals of NNs and the different NN architectures that exist. Additionally, the training of NNs was discussed along with the problems often encountered when training specialized NN architectures such as RNNs. The importance of NN architecture design was also highlighted in the context of overall machine learning model performance.

The next chapter discusses evolutionary algorithms and how they can be used for NN architecture design.

Chapter 3

Evolutionary Algorithms

Natural evolution, or *biological evolution*, has successfully produced diverse species that are capable of adapting to dynamic environments [81]. Fundamentally, natural evolution is based on a number of principles, which include a trial-and-error problem solving strategy that finds solutions by employing a survival of the fittest methodology in a process called natural selection [62, 81].

Many computational problems require searching an exceptionally large number of possibilities to find solutions [44, 62]. It is also often expected of the chosen search methodology to be adaptive to changes in its environment, such as robotic controls in variable environments [62], amongst others.

Evolutionary computation (EC) is a field of computer science research that comprises a number of natural evolutionary inspired computer-based systems and techniques for problem solving [26, 81]. Evolutionary algorithms (EAs) are a set of algorithms in EC that were designed based on natural evolutionary principles for solving problems by employing operators such as selection and recombination to find the fittest solutions in a population-based environment [26, 81].

EAs have been used for solving a number of problems, which include mathematical problems [74, 95], optimisation problems [1, 11], and more. EAs have also been successfully implemented for automated NN architecture design [3, 6].

This chapter provides an overview of EAs and how they are used for multi-objective problem solving. Section 3.1 outlines the fundamental components of the EA. Multi-

objective EA-based problem solving is discussed in Section 3.2. Section 3.3 concludes the chapter.

3.1 Evolutionary Algorithm Fundamentals

The EA describes a fundamentally stochastic search methodology for finding one or more solution(s) to a given problem by employing the principles of natural evolution [26, 62]. The general scheme of the EA involves an evolutionary search process that starts by creating and initialising a population of individuals, i.e., a collection of candidate solutions to the problem [26].

For each of the candidate solutions, a fitness value is evaluated, which represents the quality of the particular solution to the problem [26]. The fittest individuals in the population are then selected for reproduction; these individuals are referred to as parents.

The EA includes a recombination stage that comprises crossover and mutation operators for generating new individuals from the selected parents; the newly generated individuals are referred to as offspring. The crossover and mutation operators are discussed in more detail in Section 3.1.5.

The fitness for each of the individuals in the offspring population is then evaluated, and the offspring are introduced into the population to form a combined population. The fittest individuals in the combined population are then selected to form the population of the following generation. This evolutionary cycle is repeated until some predefined termination condition is satisfied. The generic EA is outlined in Algorithm 1.

The EA explores the candidate solutions for a particular problem by using the following components [26, 62, 81]:

- the encoding method that provides a formal representation of solutions to the particular problem;
- a function for evaluating the fitness of individuals;
- the method to use for creating and initialising the initial population;
- a strategy for selecting the parent solutions that will be used for recombination;

Algorithm 1 Generic Evolutionary Algorithm [26, 81]

```
Initialization of population;
Evaluate fitness of population;
while termination condition(s) not met do
    Select parents from population;
    Recombination of selected parents to create offspring;
    Evaluate the fitness of new offspring;
    Select individuals that will form the population of the next generation;
end while
```

- the recombination operators to use for generating offspring;
- a strategy to select the surviving individuals from the combined population.

The rest of this section describes each of the aforementioned components in more detail. Section 3.1.1 discusses the representation of individuals, Section 3.1.2 discusses the initial population, Section 3.1.3 discusses the fitness evaluation of individuals, Section 3.1.4 discusses the selection of individuals, Section 3.1.5 discusses the recombination stage of the EA, and Section 3.1.6 discusses the termination condition.

3.1.1 Representation of Individuals

The encoding strategy used for representing candidate solutions relates to the chromosomes of organisms found in nature [26]. Chromosomes contain a number of genes, and organisms (individuals) pass their genes on to offspring through a recombination process, also called reproduction [26, 81]. The genetic composition of an individual is referred to as the genotype, whereas the phenotype describes the forming of the object within the original problem context [26, 81].

The representation of individuals in the context of EAs for problem solving defines how possible solutions should be specified and is responsible for defining the possible collection of solutions that can exist, i.e., the search space [62, 81]. Representation of individuals creates a link between the real world problem context and the solution search space, where evolution happens [81].

A class of genetically inspired EAs, called genetic algorithms (GAs), employs a representation strategy where individuals in the search space are typically represented by a bit string. The encoding of individuals is done using n_x variables which describe an n_x dimensional search space [26]. If the variable values of the encoding structure are binary values, an individual can be defined by $\mathbf{b} = (\mathbf{b}_1, ..., \mathbf{b}_i, ..., \mathbf{b}_{n_x})$, with $\mathbf{b}_i = (b_{(i-1)n_d+1}, ..., b_{in_d})$ such that $b_l \in \{0, 1\}$ and $n_b = n_x n_d$, the total number of bits [26]. The values of the encoded bit string are not restricted to the binary domain, and there are methods for using bit strings with integer or floating point values [26]. Other methods of representation include permutations of integers [22], and trees [26], amongst others.

3.1.2 Initial Population

The first step of the EA search process involves creating and initialising the first set of candidate solutions, called the initial population. It is commonly found that the initial population is randomly generated from the defined search space [26]. Randomly initialising the initial population introduces diversity into the population, and allows for uniform sampling of the search space [22, 26]. If there exist some known solutions that are considered good, they can be included in the initial population at the discretion of the designer, given that the fitness of the known solutions is objectively good based on the chosen fitness evaluation method [22].

3.1.3 Fitness Evaluation

The fitness evaluation function is used to evaluate the absolute measure of fitness of a particular candidate solution, and the survival of an individual depends on its fitness value as determined by the fitness evaluation function [62, 81]. More formally, the fitness function, f, is used to map the individual candidate solution representation into a scalar value such that:

$$f : \Gamma^{n_x} \rightarrow \mathbb{R} \tag{3.1}$$

for a given n_x dimensional candidate solution, and the data type Γ of the elements of the representation [26].

The fitness function is used to assign a single fitness value to an individual, but can represent a number of objectives, which is discussed in more detail in Section 3.2. It is important for the fitness function to provide an accurate representation of a particular individual's fitness measure, as the fitness values of candidate solutions are used during the selection phase of the evolutionary cycle.

3.1.4 Selection

The selection process of the EA is directly related to the process of natural selection as described in the Darwinian evolution theory, and thereby has an inherent responsibility of ensuring fit individuals for subsequent generations [26]. The chosen selection method also contributes toward the diversity of candidate solutions in the population.

The *selective pressure* of a particular selection method directly influences how long it takes for the EA to produce a uniformly distributed population [26]. Selection methods with a low selective pressure may converge prematurely to solutions that can be improved on, which are referred to as suboptimal solutions [26]. Selection methods with a high selective pressure can lead to a decrease in diversity among the population [26].

There are two stages during the evolutionary cycle at which selection is performed: the selection of candidate solutions that will be used for recombination, and the selection of individuals that form the population of the following generation.

Selecting candidate solutions that will be used for recombination is done with the intention of passing their characteristics on to newer generations, to possibly find better solutions [22, 81]. In this case, the selected individuals are referred to as the parents and after being selected, some operators are used to generate offspring from these parents during the recombination (or reproduction) phase [81].

Survivor selection, on the other hand, happens at a later stage during the evolutionary cycle [81]. After the recombination phase has concluded, the fitness values of the newly generated offspring are evaluated, and the offspring are introduced into the overall population of candidate solutions to form a combined population. Survivor selection is then performed to select the individuals from the combined population that will form the population of the subsequent generation, such that a constant population size is maintained [22, 81].

Generally, candidate solutions that were not selected during the survivor selection phase, or found to have performed insufficiently compared to the other *surviving* individuals, are disposed of and therefore excluded in the following selection stages [22]. To ensure that the fittest solutions of the population survive, a process called elitism can be incorporated into the survivor selection stage, which essentially copies the fittest individuals to the following generation's population without changing any of their characteristics [26]. Selecting the surviving individuals may not necessarily be done purely based on their fitness, and will depend on the particular selection method, which is discussed in more detail below.

Selection Methods

The random selection method selects individuals from the population at random, with disregard to their respective fitness values. The random selection method has a very low selective pressure, since each individual in an n_s sized population has a $\frac{1}{n_s}$ probability of being selected [26].

An alternative selection method, called tournament selection, starts by randomly selecting n_{ts} individuals from the population, such that $n_{ts} < n_s$ [26]. Out of the n_{ts} selected individuals, those with the best fitness values are returned, resulting in a relative fitness selection as opposed to absolute fitness selection across the entire population [81]. This approach to selection allows for more control over the selective pressure compared to random selection, since an increased tournament size will lead to a relative increase in selection pressure [26, 81]. The appropriate value of n_{ts} needs to be carefully selected, as an n_{ts} value that is proportionally large compared to the overall population size, will result in only the best individuals being selected, whereas n_{ts} values that are too small may result in poorly performing individuals being selected [26, 62, 81].

A number of alternative selection methods exist, refer to [26, 81] for more extensive discussion. This study is concerned with tournament selection methods that include rank-based selection strategies within a multi-objective EA paradigm, which is further discussed in Section 3.2.

3.1.5 Recombination

Recombination describes the process wherein the characteristics of the selected parent solutions are used to generate new candidate solutions, so that the characteristics of the superior solutions can be passed on to the following generations [22, 26]. At the recombination stage, crossover and/or mutation operators are used to create new offspring solutions from the selected parent solutions [26].

The crossover operator consists of selecting the characteristics of some predefined number of parents, which are then combined to create one or more new individual candidate solution(s) [26]. The crossover operator includes an asexual operator wherein a single candidate solution is generated from one parent candidate solution, a sexual operator where one or two offspring candidate solutions are created from two parent candidate solutions, and multi-recombination that generates one or more offspring candidate solutions from a selection of more than two parent candidate solutions [26].

With typical binary string representations of individuals, the application of the crossover operator starts by selecting a position from the parent representation, which is called a point [81]. When two parent solutions are used for generating two offspring solutions, the encoding of the parent solutions are split at the selected point and the offspring solutions are created by exchanging the tails of the split parent solutions [81]. This crossover implementation is referred to as One-Point crossover, and can be seen in Figure 3.1 [81]. More than one point can be used for crossover, and offspring are then generated by taking alternative segments from the parent encodings [81]. For a more detailed discussion of alternative crossover operators, refer to [26, 81].

The mutation operator randomly changes some characteristics of a given candidate solution, and is useful for promoting diversity in the population [22, 26]. Mutation during the recombination stage is controlled by a probability value, which can be set to a higher value when exploring the first few generations to further contribute towards diversity, and the probability value can then be decreased over time as the algorithm progresses [26]. When a binary string representation of individuals is used, some number of values in the encoding are randomly selected, and the values are flipped, i.e., changed from 1 to 0 or 0 to 1 [81]. Figure 3.2 illustrates a bitwise mutation operator applied to a single individual, where three values are mutated.

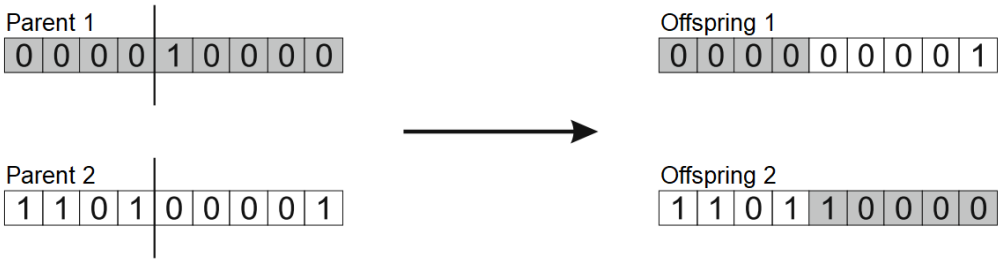


Figure 3.1: One-Point crossover of two parent solutions to generate two offspring solutions.

Smith and Eiben [81] pointed out that the recombination process is considered an important feature of the EA in general, and it is often found that the implementation of the recombination process may differ between EA subclasses. This study is concerned with the use of EAs in the context of RNN architecture design, which has some complexities associated with applying crossover and mutation operators during offspring generation. These complexities are discussed in Chapter 4, along with the concept of network morphism, which is a method that can be used for RNN architecture recombination, as opposed to explicit crossover and mutation based recombination.

3.1.6 Termination Condition

The EA will cycle through the selection, recombination, and evaluation stages until a predefined termination condition is met. This termination condition can be defined in a number of ways, which include the maximum number of generations that the evolutionary cycle should run for, or when an acceptable candidate solution was found [26, 62, 81], amongst others. Additionally, when no significant changes or improvements are observed across the population over a number of consecutive generations, the EA has converged and should be terminated [81]. Ideally, the termination condition should not cause the termination of the EA that is too early (depending on the particular problem context), and should allow for sufficient exploration of the search space across a number of generations [26].

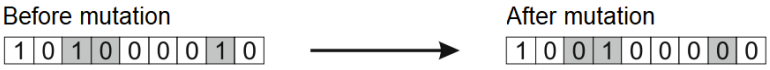


Figure 3.2: Before and after applying the mutation operator on a single candidate solution representation.

3.2 Multi-Objective Evolutionary Algorithms

A number of EA approaches exist that are capable of solving multi-objective problems [26]. In general, for multi-objective problems, assuming the goal is to minimize the respective problems, a vector-valued objective function $F : R^o \rightarrow R^n$ is defined for n objectives, where $n > 1$, and o is the dimension of the decision vector $\mathbf{x}$. The aim is then to minimize the $\mathbf{y}$ objective vector such that:

$$\mathbf{y} = F(\mathbf{x}) = (f_1(\mathbf{x}), ..., f_n(\mathbf{x})), \tag{3.2}$$

where

$$\mathbf{x} = (x_1, ..., x_o) \in R^o,$$
$$\mathbf{y} = (y_1, ..., y_n) \in R^n,$$

in an R^o parameter space and R^n objective space [74, 93, 95].

The rest of this section provides a discussion on specific EA approaches for solving multi-objective problems. Section 3.2.1 discusses the concepts related to weighted aggregation and Pareto-based approaches for multi-objective problem solving. Section 3.2.2 outlines genetic algorithms and how they can be used for solving multi-objective problems.

3.2.1 Multi-Objective Problem Solving Approaches

Smith and Eiben [81] have postulated that in practice there exist many applications of EAs that search for solutions in a single-objective paradigm when, upon further investigation, they are essentially searching for solutions that represent multiple objectives, but have been adapted to the single-objective paradigm in favour of reduced complexity [81]. Methods exist for assigning weight values to multiple objectives and then calculating the

weighted sum of the sub-objectives to obtain a single scalar-value surrogate objective value [26, 74]. The weight values for each of the sub-objectives are therefore implicitly assumed, and might have an impact on the diversity of solutions found [74, 81].

Zhang and Li [71] introduced an approach that employs a weighted aggregation method for dealing with multi-objective problems, dubbed Multiobjective Evolutionary Algorithm Based on Decomposition (MOEA/D). The MOEA/D algorithm relies on a combination of conventional multi-objective weighted aggregation and population-based approaches [81], along with the adoption of a neighbourhood relation [93]. The multi-objective problem is decomposed into a set of organically organized subproblems, also called scalar objective optimization problems (SOPs), that each represent a weighted aggregation of the respective objectives [93]. New solutions for each of the individual subproblems are generated through the use of evolutionary operators that are applied to neighboring solutions found within some proximity of the particular subproblem, which is based on the Euclidean distance between solutions [81]. If a newly generated solution is found to have achieved the best result for a subproblem, that particular solution is then kept in memory by the subproblem until a new one is found that performs better [93].

Zhang and Li [71] have shown that the MOEA/D algorithm is able to scale well with many-objective optimization problems, which generally refers to problems that have four or more objectives [20]. Zhang and Li [71] also found that for many-objective problems, the MOEA/D algorithm performs comparably to other algorithms that do not use weighted aggregation approaches.

An alternative to the aggregated weighted approach for solving multi-objective problems is the Pareto-based approach that uses the concept of dominance. Dominance states that when one solution dominates another, the dominant solution is at least as good as the other solution for all objectives and additionally, has a strictly better value for at least one of the objectives [77, 81]. Formally, given decision vector $\mathbf{a} \in R^o$, and decision vector $\mathbf{b} \in R^o$, it is said that $\mathbf{a}$ *dominates* $\mathbf{b}$ if and only if

$$\forall i \in \{1, ..., n\}\, f_i(\mathbf{a}) \leq f_i(\mathbf{b}), \text{and}\, \exists i \in \{1, ..., n\}, f_i(\mathbf{a}) < f_i(\mathbf{b}), \tag{3.3}$$

which can be written as $\mathbf{a} \prec \mathbf{b}$ [95]. If it is the case that $\mathbf{a} \prec \mathbf{b}$ or $F(\mathbf{a}) = F(\mathbf{b})$, then $\mathbf{a}$ *covers* $\mathbf{b}$, which is written as $\mathbf{a} \preceq \mathbf{b}$ [81, 95].

In the case where there are conflicting objectives, it is unlikely that one solution will exist that dominates all other solutions in the population [71, 74, 81]. Thus, a decision vector is considered nondominated if and only if it is not dominated by any of the other solutions in the population [81]. Such a decision vector that is not dominated by any other is called a *Pareto optimal* solution [71, 74]. A set of Pareto optimal solutions is called the Pareto set, and the set of corresponding objective vectors is referred to as the Pareto front [95]. Therefore, the Pareto front describes the set of objective vectors whose solutions provide the best compromise among the respective objectives. An example of a dominated and nondominated solution set for a multi-objective problem can be seen in Figure 3.3. The next section discusses genetic algorithms that have been designed to solve multi-objective problems using a Pareto-based approach.

3.2.2 Genetic Algorithms

Genetic algorithms (GAs) are a class of genetically inspired EAs [62, 81]. The classical (canonical) implementation of the GA typically employs a binary bit-string candidate solution representation structure, previously discussed in Section 3.1.1. The GA follows a fixed evolutionary process that involves a proportional fitness selection process and a recombination (reproduction) process [62, 81]. Offspring are therefore generated through the use of both crossover and mutation operators as discussed in Section 3.1.5 [62, 81]. The survival selection process of the classical GA follows a generational approach, which means the intermediary population of surviving candidate solutions replaces the previous population entirely [81].

A number of GA variants have been introduced, some of which were specifically designed and adapted so that they can be used for multi-objective problem solving. A specific GA variant that employs a Pareto-based approach to multi-objective problem solving is discussed in more detail below.

Nondominated Sorting Genetic Algorithms

The Nondominated Sorting Genetic Algorithm (NSGA) that is based on the classical GA, was introduced by Srinivas and Deb [82] for multi-objective problem solving. In contrast to the classical GA, the NSGA sorts individuals based on their nondomination

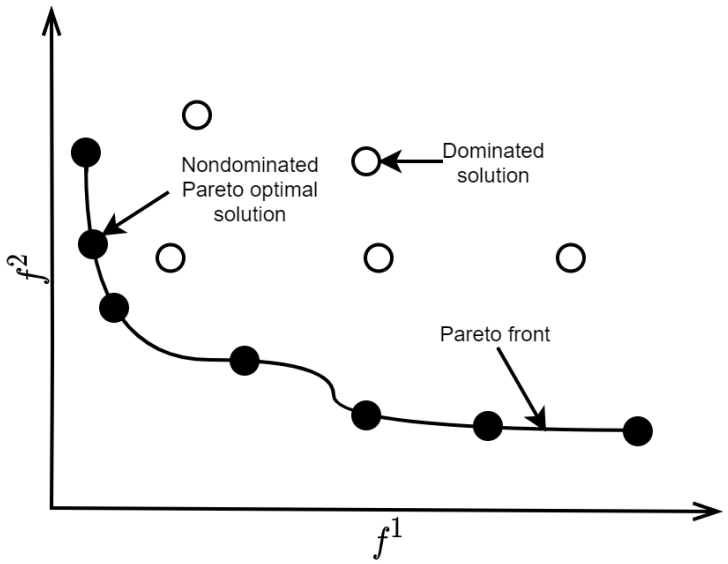


Figure 3.3: Example of a Pareto solution set with dominated and nondominated solutions for a problem with two objectives [21, 81].

with respect to the multiple objectives by using the dominance operator described in Section 3.2.1 [82]. The NSGA uses a sharing parameter to calculate a new fitness value for each of the individuals by dividing the particular individual's fitness by a value that is proportional to some predefined number of neighbouring solutions; this approach was later criticized by Deb *et al.* [21].

Deb *et al.* [21] also pointed out that the NSGA lacks elitism and has a higher computational complexity than what is necessary [21]. Deb *et al.* [21] proposed a fast and elitist nondominated sorting algorithm called NSGA-II to address the problems of the NSGA. To ensure elitism, the NSGA-II performs nondominated sorting on the combined population that includes the parents and offspring [21]. The NSGA-II does not have the sharing parameter that is used by the NSGA, which makes the NSGA-II more efficient compared to the NSGA [21]. Additionally, the NSGA-II uses a mechanism, called crowding distance, with the aim of increasing the diversity of the population [21].

The crowding distance of the NSGA-II represents the Euclidean distance between candidate solutions, and is used as a density estimator to guide the algorithm towards a uniform population distribution [16, 21]. Calculating the crowding distance for one

individual is based on the absolute normalized difference in the objective values of two adjacent individuals [21], which can be represented by a cuboid as illustrated in Figure 3.4. For one specific individual j with a sort sequence n, its crowding distance, dis_j, for a single objective k is defined by:

$$dis_j = dis_j + \frac{f^k_{n+1} - f^k_{n-1}}{f^k_{max} - f^k_{min}} \tag{3.4}$$

where $k \in m$ for m objectives, f^k_n refers to the k-th objective value of the n-th individual, and f^k_{max} and f^k_{min} represent the maximum and minimum values of the k-th objective function, respectively. The crowding distance is calculated for all the objectives, which means that the dis_j value for individual j is updated as the calculation iterates through all the objectives. Individuals with the minimum and maximum values for each of the objectives are assigned an infinite crowding distance value, resulting in those individuals always being selected [16]. Algorithm 2 provides the pseudocode of the nondomination sorting function that is used by the NSGA-II algorithm. Pseudocode given in Algorithm 3 provides an overview of the NSGA-II algorithm.

Assigning the same crowding distance to individuals within a cuboid has been criticized for not having any significant contribution towards the convergence of the algorithm [16]. An alternative approach was proposed by Chu and Yu [16] wherein they suggested replacing f^k_{n-1} in the crowding distance function with f^k_n instead. This resulted in the crowding distance for an individual calculated as:

$$dis_j = dis_j + \frac{f^k_{n+1} - f^k_n}{f^k_{max} - f^k_{min}}. \tag{3.5}$$

Chu and Yu [16] have reported that this change resulted in an improved convergence of the Pareto front [16].

Deb and Jain [20] have proposed an extension of the NSGA-II algorithm, called NSGA-III. The NSGA-III employs a nondominated sorting approach to solve many-objective problems, and relies on predefined target objective values, called reference points, to guide the search direction towards Pareto-optimal solutions [20]. According to Deb and Jain [20], the need for such an algorithm stems from the increased demand for algorithms that can solve four or more objectives [20]. It has been noted by Deb and Jain [20] that the expectation of a single population-based algorithm to both maintain

Algorithm 2 Pseudocode of the nondominated sorting function that is used by the NSGA-II algorithm [21]

```
Inputs: P;
for each p ∈ P do
    S_p ← ∅
    n_p ← 0
    for each q ∈ P do
        if (p ≺ q) then
            S_p ← S_p ∪ {q}
        else if (q ≺ p) then
            n_p ← n_p + 1
        end if
        if n_p = 0 then
            p_rank ← 1
            F_1 ← F_1 ∪ {p}
        end if
    end for
end for
i ← 1
while F_i ≠ ∅ do
    Q ← ∅
    for each p ∈ F_i do
        for each q ∈ S_p do
            n_q ← n_q − 1
            if n_q = 0 then
                q_rank ← i + 1
                Q ← Q ∪ {q}
            end if
        end for
    end for
    i ← i + 1
    F_i ← Q
end while
```

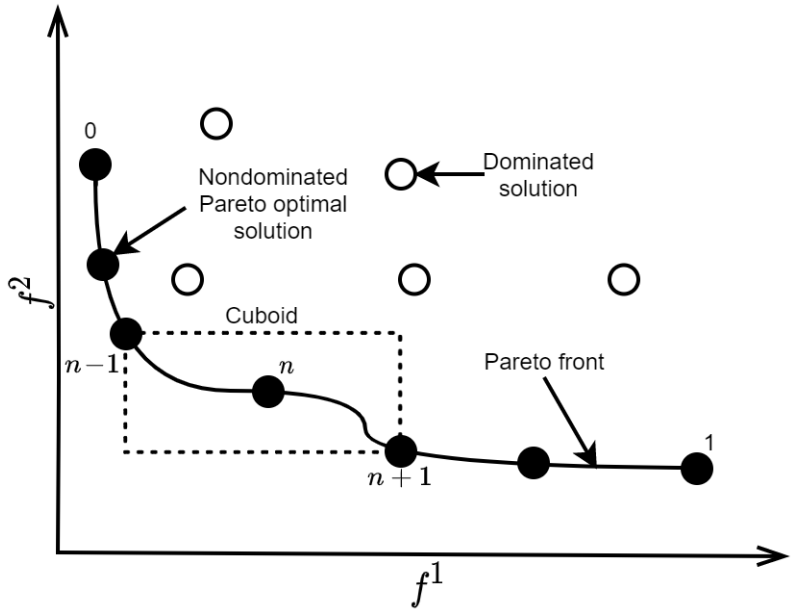


Figure 3.4: Crowding distance calculation for the i^{th} solution [21].

population diversity and reach population convergence close to the Pareto-optimal front may be too optimistic.

Predefined reference points are provided to the NSGA-III, with the premise being that the reference points would ensure diversity in the solutions found [20]. A reference line is defined for each of the reference points, which starts from the origin and then passes through the particular reference point; a visualization of this concept can be seen in Figure 3.5. The perpendicular distance between each individual in the population is calculated for each of the reference lines. A specific individual is then associated with its closest reference line [11]. Individuals that are closest to the reference lines are selected for recombination. NSGA-III performs recombination of parent solutions by using the same crossover and mutation operators that are used by the NSGA-II [20].

The NSGA-II is capable of solving many-objective problems, and does not require predefined reference points [11, 71]. The NSGA-III algorithm, on the other hand, was specifically designed to solve many-objective problems, and should therefore outperform the NSGA-II when solving problems within a many-objective paradigm [20]. The NSGA-III and MOEA/D algorithms are more efficient at solving many-objective prob-

Algorithm 3 Pseudocode of the NSGA-II algorithm [21]

Inputs: N, g, objectives;
$P \leftarrow$ randomly initialise parent population of size N;
$Q \leftarrow$ initialise offspring population to $\emptyset$;
Evaluate fitness of P; ▷ For all objectives
Nondominated sorting of P based on fitness values; ▷ Pareto-ranking
$Q \leftarrow$ generate offspring;
while termination condition g not met **do**
    Evaluate fitness of Q; ▷ For all objectives
    $R \leftarrow P \cup Q$; ▷ Combine parent and offspring population
    Nondominated sorting of R based on fitness values;
    Crowding distance assignment for population R;
    Sort solutions in nondominated fronts based on ranking;
    Select N solutions starting from the first front and only select solutions from subsequent fronts if all N population slots have not been filled;
    $Q \leftarrow$ generate offspring from selected solutions;
end while

lems compared to the NSGA-II algorithm, however, the quality of the solutions found by the NSGA-II algorithm is comparable to those found by the NSGA-III and MOEA/D algorithms [11, 71].

This study is interested in the use of multi-objective EAs for automated NN architecture design. The number of objectives that will be considered for the NN architecture candidate solution fitness evaluation are assumed to be low, with preliminary estimations suggesting a maximum of five objectives [24]. Accurate NN architecture fitness evaluation is a computationally expensive task [96], which would make the difference between the execution time of the aforementioned EAs insignificant. Therefore, the NSGA-II algorithm is the preferred candidate since it does not require predefined reference points, and it has already been successfully implemented in the context of NN architecture design [6, 52]. A more detailed discussion related to the specific NN architecture related objectives and NN architecture candidate solution fitness evaluation is

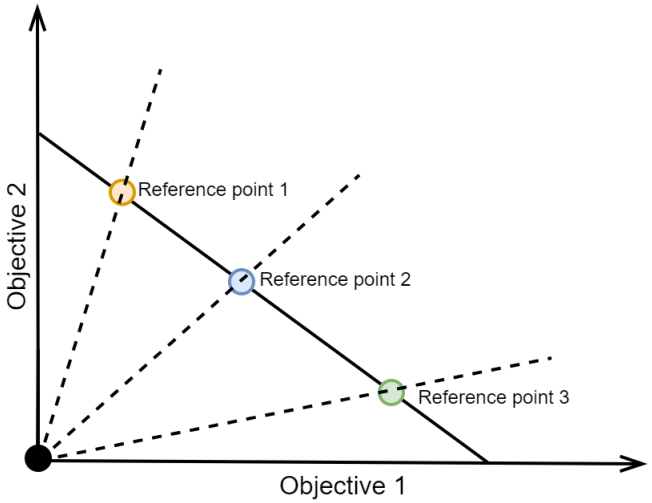


Figure 3.5: Normalized reference lines for three reference points of a two-objective problem [47].

provided in Chapter 4.

3.3 Summary

This chapter discussed EAs and how they can be used for solving multi-objective problems. A number of genetically inspired EAs were also discussed along with their respective multi-objective problem solving approaches.

Multi-objective EAs can be used for population-based automated NN architecture design. The next chapter discusses the different neural architecture search methods that exist for automated NN architecture design.

Chapter 4

Neural Architecture Search

Methods for automated NN architecture design have been around for a number of years [3, 86]. More recently, in 2017, Zoph and Le [96] introduced the Neural Architecture Search (NAS) paradigm, wherein they proposed a novel reinforcement learning method to automatically find well-performing NN architectures for a provided dataset.

A number of different NAS methods have since been proposed, some of which have already been successful in finding NN architectures that outperform state-of-the-art NN architectures designed by human experts [24, 50, 88].

The purpose of this chapter is to provide an overview of NAS and review the evolutionary NAS methods that exist for automated NN architecture design. Section 4.1 discusses NAS in more detail. Section 4.2 provides an overview of existing evolutionary NAS methods. Section 4.3 concludes the chapter.

4.1 How Neural Architecture Search Works

Neural architecture search (NAS) aims to automatically find well-performing NN architectures for a provided dataset by casting NN architecture design as an optimisation problem [86]. NAS works by optimising $f(a)$ for a set of NN architectures A, where $f(a)$ represents an objective measure of performance for architecture $a \in A$ [86, 91].

Elsken *et al.* [25] suggested that any given NAS method should define at least three equally important components that describe how the particular method approaches NN

architecture design as an optimisation problem. The three components are: search space, search strategy, and performance estimation strategy [25].

The search space component is responsible for defining the possible NN architectures that can exist within the particular problem context, which is based on the provided dataset [25, 88]. The search strategy explores the NN architecture search space by generating NN architectures from the defined search space [25]. The performances of the generated NN architectures are then evaluated by the performance estimation strategy [25]. Figure 4.1 illustrates how the three components of NAS are related.

The rest of this section discusses each of the three NAS components in more detail. Section 4.1.1 provides an overview of the NN architecture search space as a NAS component. The search strategy of NAS is discussed in Section 4.1.2. Section 4.1.3 discusses the performance estimation strategy component of NAS. An overview of the challenges associated with reporting on NAS research is given in Section 4.1.4.

4.1.1 Search Space

The NAS search space is responsible for defining the NN architectures that can exist in principle, thereby describing the structural paradigm that the search strategy can explore [25, 32, 88]. Therefore, the NAS search space represents a *subspace* of the general definition of a NN architecture, as it was given in Chapter 2.

Defining a good NAS search space can be challenging and should include prior knowledge about the provided dataset [32]. For example, if the provided dataset contains sequential data with significant time dependencies, an appropriate RNN architecture search space should be defined [88].

He *et al.* [32] identified a number of different categorical NAS search spaces, which include the global search space and the cell-based search space, amongst others. With global search space approaches, the entire structure of the NN architecture is searched for, which includes all units of the NN architecture, how these units are connected, and the activation functions for each individual unit [32]. Thus, global search spaces are very large, and exploring a global search space can be computationally expensive [25, 32].

The intuitive idea behind the cell-based search space is to generate cells from the defined search space and then stack multiple cells to create a NN architecture [88]. The

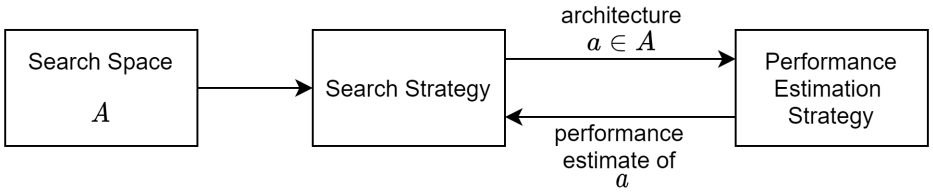


Figure 4.1: The three components of Neural Architecture Search as presented by Elsken et al. [25].

cell can be viewed as a miniature NN that accepts some inputs and contains a number of connected units with combination methods and activation functions that eventually produce an output [6]. A single cell can accept multiple inputs, which may include the input of the NN and the output of other cells. A template can be specified that prescribes the possible configurations or limitations of a cell, for example, restricting the number of units that a cell can contain [86]. Methods that employ the cell-based search space are considered more efficient compared to global search space methods, since cell-based methods only search for individual cell configurations from the defined search space [32]. Figure 4.2 shows an example of a NN architecture that contains a number of stacked cells and an example of a cell which contains a combination method and two activation functions.

Zoph and Le [96] defined a cell-based search space for RNN architectures wherein a single recurrent cell g is described by

$$\mathbf{h}_t = g_{\theta,\alpha}(\mathbf{x}_t, \mathbf{h}_{t-1}, \mathbf{c}_{t-1}),$$

where θ represents the architecture of the cell g, α is the trainable parameters of the architecture, $\mathbf{h}_t$ is the hidden state, $\mathbf{x}_t$ is the input, and $\mathbf{c}_t$ is the cell state at time step t. In their study, Zoph and Le [96] stacked two recurrent cells to make up the final RNN architecture. Combining multiple inputs to a cell was limited to the use of either addition or elementwise multiplication, and the activation functions were limited to the identity, tanh, sigmoid, and ReLU activation functions [96].

Each cell that is constructed from the defined cell-based search space is represented by some encoding scheme [86]. The encoding scheme is used to define the inputs of the cell, how the inputs are combined if multiple inputs are used, and what activation

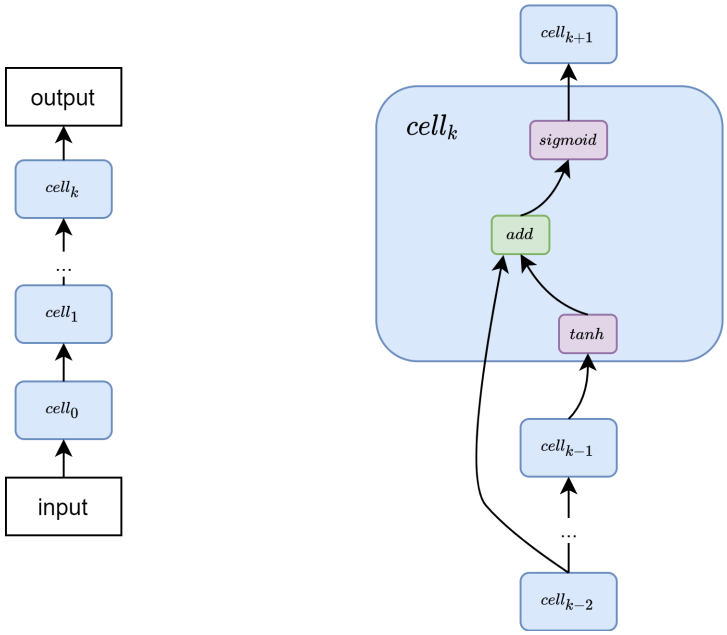


Figure 4.2: A NN architecture containing k cells (left) and an example cell decomposition (right).

functions are applied. Zoph and Le [96] opted for a string encoding scheme in their reinforcement learning NAS approach, where a RNN controller was used as the agent. The RNN controller explored the search space by generating a string of computation steps, which included combination methods and activation functions that were allowed according to the defined search space [96]. A cell was then created from the string encoding, which was subsequently used for constructing the RNN architecture [96].

NAS methods that employ EAs as the underlying search strategy for search space exploration should implement appropriate encoding schemes that can accommodate for structural changes to the NN architectures during the recombination stage of the evolutionary cycle. Previous EA-based NAS studies [52, 53] have been successful in using string encoding schemes along with performing crossover on multiple parent architectures to generate offspring architectures. However, those studies [52, 53] were specifically focused on CNN architecture search and not RNN architecture search. Angeline *et al.* [3] and Bayer *et al.* [6] conjectured that there is a significant amount of complexity involved with performing crossover on RNN architectures during evolutionary recombi-

nation, which is discussed in more detail in Section 4.2.

The next section discusses the search strategy component of NAS which is used for exploring the NN architecture search space.

4.1.2 Search Strategy

The search strategy is responsible for the efficient exploration of the NN architecture search space, and for the automation of the process of constructing different NN architectures, which would otherwise be done manually by human experts [32].

Different NAS methods are typically distinguished based on the particular NN architecture search space exploration strategy that is implemented [25, 88]. The NN architecture search space can be explored in many ways, which include reinforcement learning (RL) based searching [96], evolutionary algorithm (EA) based searching [25], gradient-based searching [79], and more.

RL based NAS methods often extend the approach from Zoph and Le [96], wherein an RNN controller was used to generate NN architectures from the defined search space [32]. With RL based methods, the validation accuracy of the trained models is typically used as the reward signal for the agent to promote the generation of better performing architectures [32].

EA-based NAS approaches search for NN architectures in a population-based paradigm and employ an EA for search space exploration [88]. The NN architectures are treated as individuals in the population and are evolved over a number of generations [50, 88]. This study deals with EA-based NAS methods only, which is discussed in more detail in Section 4.2. For a more extensive discussion on different NAS search strategies, refer to [32].

The NN architectures generated by the search strategy should be evaluated to determine their respective performances, such that the best performing architectures can be identified. The NAS performance estimation strategy component is responsible for reporting on NN architecture performance and is discussed in the following section.

4.1.3 Performance Estimation Strategy

The performance estimation strategy is responsible for reporting the objective measure of the performance of the NN architectures that are constructed by the search strategy [25, 72]. Thus, an accurate measure of architecture performance is important for achieving the goal of finding well-performing NN architectures [25, 72]. Model accuracy on some set of unseen data from the provided dataset is a good measure of architecture performance [25].

The simplest way to evaluate the performance of an architecture is to train and evaluate the architecture on training and validation subsets of the provided dataset [25]. However, training and evaluating multiple architectures can become computationally expensive. For example, Zoph and Le [96] trained and evaluated 12 800 architectures in their study. The training of the models were distributed among 800 graphics processing units (GPUs), which equates to a search cost of more than 22 000 GPU days [88, 96]. GPU days is a search cost measure used in NAS, which is calculated by multiplying the number of GPUs used by the number of consecutive days that the search was running for.

Several methods have been proposed to make the performance estimation component of NAS more efficient, such as parameter sharing [24, 70], performance prediction [35], and more. With parameter sharing, the parameters of the models that have already been trained are used to initialise the parameters of newly constructed models [24]. Therefore, the newly constructed models do not require training from scratch [25].

In general, performance prediction techniques attempt to predict the performance of an architecture with minimal or no training [25]. White *et al.* [87] identified 31 performance prediction techniques, some of which have shown promising results. This study did not implement any specific performance prediction techniques, and instead relied on methods such as parameter sharing, early stopping, and reduced population size techniques to improve the efficiency of the NAS method. The specific methods and techniques employed in this study are discussed in more detail in Chapter 5.

It is often the case in real world applications where a reasonable trade-off between architecture accuracy and architecture complexity is accepted [13, 24]. To account for this, the performance evaluation of NN architectures can consider multiple objectives,

such as the model's achieved accuracy, the number of trainable parameters, and the time it takes the model for a forward propagation of a single input, i.e., model inference time [17, 24]. A number of NAS methods have already been implemented that consider multiple objectives for architecture performance evaluation, instead of basing architecture performance purely on model accuracy [13, 24, 53, 84]. These studies often report reduced accuracy scores compared to other NAS methods, but the architectures found have lower computational resource demand and shorter model inference times [17, 24].

NAS methods are typically judged based on the accuracy achieved by the best performing architecture that the particular NAS method found [89, 42, 46, 48, 72]. The next section discusses some concerns related to associating a NAS method's quality with the accuracy achieved by the method's best performing architecture.

4.1.4 Neural Architecture Search Method Quality

One of the challenges in NAS research is the lack of an objective measure to represent the quality of any given NAS method [42, 46, 89, 91]. The results reported by most NAS publications are often found to be impossible to reproduce, due to the amount of computational resource allocation for the experiments, insufficient details provided regarding the experimental setup, and more [48, 89].

Given that NAS aims to automatically find well-performing architectures for a provided dataset, different NAS methods are being compared based on the accuracy achieved by the best performing architecture that the NAS methods have found [25, 32, 50, 88]. Lindauer and Hutter [48] suggested that although most NAS publications report their results on the same datasets, the NAS methods typically used different search spaces and different model hyperparameter optimisation techniques, which make the results incomparable.

Yang *et al.* [89] stated that the extensive hyperparameter optimisation performed by some NAS methods resulted in the models achieving very high accuracy scores, and therefore misrepresent the quality of the overall NAS method. Li and Talwalkar [46] implemented a random search method that constructed NN architectures from the search spaces defined by two separate NAS approaches, both of which had complex search strategies for constructing NN architectures. The random search method implemented

by Li and Talwalkar [46] was able to find NN architectures that outperformed both NAS approaches.

Following the aforementioned observations, Li and Talwalkar [46] suggested that the evaluation of NAS methods should be based on the robustness and consistency of the particular method across multiple independent runs. NAS methods that employ search strategies such as RL based searching or EA-based searching should include ablation studies that report on the results from performing a random search on the defined NN architecture search space [46].

It is worth noting that the above concerns relate to NAS methods that consider model accuracy as the single architecture performance objective [46]. It will be interesting to compare the results from a multi-objective NAS method with a random search implementation for the same search space.

Multi-objective EA-based NAS methods allow for generationally optimising the NN architectures, which could give the particular approach an advantage over a random search implementation. The use of multiple objectives in EA-based NAS methods are discussed in the next section.

4.2 Evolutionary NAS Methods

EA-based NAS methods refer to those NAS methods that employ EAs as their core search space exploration strategy, where NN architectures are searched for in a population-based paradigm [50]. EA-based NAS methods have been successful in finding well-performing NN architectures for a provided dataset [24, 51, 52, 53, 66].

Elsken *et al.* [24] proposed a Lamarckian Evolutionary algorithm for Multi-Objective Neural Architecture Design, which they named LEMONADE. The LEMONADE algorithm considered a cell-based CNN architecture search space, which was explored through an evolutionary approach that approximated a Pareto front (refer to Section 3.2.1) [24]. The LEMONADE algorithm did not apply any specific recombination operators such as crossover or mutation, and instead relied on the concept of network morphism for offspring generation [24]. Network morphism describes the process whereby some network transformations are performed to make structural changes to an architecture [24].

Network transformations refer to the specific changes that are made to the architecture, such as adding units or introducing new connections between units [10, 24].

Elsken *et al.* [24] noted that previous implementations of network morphism were limited to constructive network transformations, which results in an increased NN architecture complexity. In a multi-objective paradigm where some NN architecture complexity related objectives are considered, appropriate network transformations are required to optimise NN architecture complexity objective(s), such as the removal of units from the NN architecture.

Elsken *et al.* [24] proposed the concept of approximate network morphism to cater for destructive network transformations. Destructive network transformations in the LEMONADE algorithm allowed for the removal of units in the architecture and the removal of connections between units [24].

The network morphism approach employed by the LEMONADE algorithm for offspring generation started by duplicating a parent architecture to construct the offspring architecture, and then performing network transformations on the offspring architecture [24]. A parameter sharing mechanism was also implemented that used the parameters of the trained parent models to initialise the parameters of the corresponding offspring models [24].

The specific experiments conducted by Elsken *et al.* [24] were focused on finding CNN architectures, and the objectives they considered included the accuracy of the models on image classification tasks, the number of parameters of the models, and the inference time of the models. Elsken *et al.* [24] reported that the architectures found by the LEMONADE algorithm achieved comparable accuracy to architectures from other related NAS studies. In terms of parameters and inference time, the architectures found by LEMONADE dominated the architectures from other studies, which demonstrates the real-world benefits associated with multi-objective NAS approaches [24].

Lu *et al.* [52] proposed a multi-objective evolutionary NAS method called NSGA-Net. The NSGA-Net algorithm is based on the NSGA-II algorithm, and searches specifically for CNN architectures in a global search space [52]. Lu *et al.* [52] implemented both crossover and mutation operators for NSGA-Net to create offspring architectures.

During their experimentation, Lu *et al.* [52] considered two objectives for representing

architecture performance, which was based on the model's classification error on an image classification task and an architecture complexity objective [45]. Lu *et al.* [52] stated that they found the number of floating-point operations needed to execute the forward pass of a CNN to be the most reliable objective measure of network complexity. The results reported by Lu *et al.* [52] showed that the CNN architectures found by NSGA-Net significantly outperform CNN architectures that were designed by human experts.

The use of EA-based NAS methods to search for RNN architectures are scarce in the current literature, and Liu *et al.* stated that the majority of EA-based NAS approaches are focused on searching for CNN architectures.

Conceptually, the LEMONADE [24] and NSGA-Net [52] algorithms can be used to search for RNN architectures, but will require significant adaptation in terms of their respective search spaces and network transformations so that they can be used to search for RNN architectures. For example, the LEMONADE algorithm included appropriate network transformations that allowed for the optimisation of NN architecture complexity objectives, but was specifically developed for CNN architectures. Therefore, the transformations implemented by the LEMONADE algorithm can not be used for optimising RNN architecture complexity-related objectives due to the structural differences between CNN architectures and RNN architectures, as discussed in Chapter 2.

Angeline *et al.* [3] proposed an EA approach that simultaneously searched for RNN architectures and their weight values, which resulted in a representation of individuals that defines both the architecture of the RNN and its corresponding weight values. Angeline *et al.* [3] based the fitness of individuals in the population on a single objective that represented the individual's performance on a specific machine learning task. One of the tasks that they considered was based on a dataset that comprised a number of strings, which was randomly generated from seven different regular languages [3]. An individual's fitness represented the corresponding model's loss value after training and evaluation on the particular dataset [3].

Angeline *et al.* [3] stated that the individuals of the population were sorted based on their fitness values and the top 50% were selected as the parents [3]. For each parent, a single offspring was generated, after which parametric and structural mutations were applied to the individual [3]. Parametric mutations involved changes to the parameters

of the RNN, whereas structural mutations refer to changes that were applied to the RNN architecture [3].

The structural mutations applied to the RNN architectures were limited to the adding and removing of neurons in the hidden layer, and neurons in the input and output layers were considered immutable [3]. All neurons in the hidden layer used the sigmoid activation function by default, which could not be changed during structural mutation [3].

Angeline *et al.* [3] excluded the use of a crossover operator during the recombination phase, which they motivated by noting that performing crossover on RNN architecture topological representations is a nontrivial and complex task, since the architectures in the population could vary significantly [3]. Additionally, by not performing crossover, the individuality of the architectures are respected [3]. Therefore, Angeline *et al.* [3] indicated that they have specifically avoided a GA-based approach and instead employed a more general EA approach that does not involve explicit crossover.

The results reported by Angeline *et al.* [3] showed that their approach was able to find RNN architectures that outperformed the RNN architectures from other studies related to RNN architecture design at the time.

Chihi and Arous [14] developed a NSGA inspired multi-objective method to search for RNN architectures and parameters, similar to the method constructed by Angeline *et al.* [3]. The objectives they considered were related to model accuracy, and included a specialised metric developed to measure the average of mutual information between a pair of models [14].

Chihi and Arous [14] did not implement a crossover operator for RNN architecture evolution, and instead only allowed for the adding and removal of a hidden unit in the hidden layer during recombination. Although a mutation was included that could reduce the RNN architecture complexity, an architecture complexity objective was not considered during selection [14]. Chihi and Arous [14] reported that the best performing RNN architecture found by their method was able to outperform a manually designed RNN architecture on a specific speech corpus.

Bayer *et al.* [6] proposed an approach for finding RNN architectures that is based on the NSGA-II algorithm. In their study, Bayer *et al.* [6] used a cell structure representa-

tion of the individuals. The representation of an individual comprises a set of cells, where each cell represents a topology that contains multiple neuron units with different activation functions and different methods for combining multiple inputs from other units [6]. They have also incorporated the use of *flags* as part of the cell structure representation, where each flag represents a specific property such as whether the activation functions of the cell have trainable parameters, and whether the cell is an input or output to another cell in the RNN architecture [6].

The fitness of the individuals were based on multiple objectives that represented an individual's ability to recognize context-free and context-sensitive languages [6]. They also iteratively increased the complexity of the regular languages based on the achieved performance of the individuals in the population as the GA progressed [6]. Bayer *et al.* [6] did not implement a specific selection strategy, and instead performed recombination on the entire population of individuals to produce individuals for the following generation.

It is worth noting that Bayer *et al.* [6] decided to exclude the implementation of a crossover operator in their algorithm for the sake of simplicity, a sentiment shared by the previously discussed study that was done by Angeline *et al.* [3]. Thus, recombination consists of applying some mutations to each of the individuals in the population [6].

Bayer *et al.* [6] allowed their algorithm to select a mutation from a list of available mutations, whereby each of the mutations had a predefined probability of being selected. The list of allowable mutations included the addition of new units to the architecture, the addition of connections between two existing units in the architecture, and the modification of the combination and activation functions of randomly selected units in the architecture [6].

The lack of survival selection along with the exclusion of mutation operators that could reduce the number of units in a RNN architecture, resulted in an approach whereby RNN architectures can only grow in size [6]. Smaller RNN architectures from previous generations can therefore not be revisited.

Bayer *et al.* [6] reported that its NSGA-II based approach for RNN architecture design was able to find RNN architectures that outperform the LSTM architecture after being trained on datasets that were generated from context-free and context-sensitive languages such as $a^n b^n, n \in [1, 5]$ and $a^n b^n c^n, n \in [1, 5]$, respectively. However, their

approach was unable to find a RNN architecture that could outperform the LSTM on the more complicated language $a^n b^m c^n, (m, n) \in [1, 4]$ [6].

Ortego *et al.* [65] implemented a genetic algorithm (GA) based NAS method to search for the hyperparameters related to connecting a CNN model and an LSTM model. The hyperparameters that their NAS method searched for were only used to define how the output of the CNN model is fed into the LSTM model as input, and therefore did not search for novel RNN architectures [65]. A similar implementation was done by Almalaq and Zhang [2], wherein they used a GA to optimise the number of hidden neurons and input sequence length for an LSTM model.

To the best of the author's knowledge, no dedicated studies of a multi-objective EA-based NAS method for novel RNN architecture search exist, that also considers an architecture complexity objective. Existing EA-based RNN architecture search methods have also not been implemented to search for well-performing RNN architectures on the Penn Treebank dataset [55]. Furthermore, since RNN architecture complexities have not been considered in existing EA-based NAS methods, the use of destructive network transformations has not been studied for RNN architecture evolution.

4.3 Summary

This chapter discussed NAS in detail and the EA-based NAS methods that exist. Although multi-objective EA-based RNN architecture search methods exist in the current literature, the use of multi-objective EAs for RNN architecture search specifically within the NAS paradigm has not been investigated along with relevant RNN architectural complexity objectives. The next chapter proposes a multi-objective EA-based NAS approach for RNN architectures, in the form of a framework.

Chapter 5

Framework

In Chapter 4 it was observed that a multi-objective EA-based RNN architecture search method that considers architecture complexity objectives and relevant transformations to optimise architecture complexity objectives, has not been studied before. The network morphism approach implemented by Elsken *et al.* [24] for evolutionary CNN architecture search can be adapted to the RNN architecture domain, which would allow for an RNN architecture complexity objective to be optimised along with relevant model accuracy objectives. A template-driven RNN architecture cell-based search space can be defined that would allow for sufficient modularity to support destructive network transformations.

This chapter proposes a **M**ulti-**O**bjective **E**volutionary algorithm for **R**ecurrent **N**eural **A**rchitecture **S**earch, dubbed MOE/RNAS, to automatically construct RNN architectures for a provided dataset. The proposed method is presented in the form of a framework that searches for RNN architectures in a modular template-driven cell-based search space with an efficient architecture performance estimation strategy. The remainder of this chapter is organized as follows: Section 5.1 provides an overview of the RNN architecture search space and the encoding structure that is used by the framework. The EA-based search space exploration strategy is discussed in Section 5.2. Finally, the chapter is summarised in Section 5.3.

5.1 Search Space

The cell-based RNN architecture search space considered by the MOE/RNAS algorithm draws inspiration from the recurrent cell defined by Zoph and Le [96]:

$$\mathbf{h}_t = g_{\theta,\alpha}(\mathbf{x}_t, \mathbf{h}_{t-1}, \mathbf{c}_{t-1}), \tag{5.1}$$

where θ represents the architecture of the cell g, α the trainable parameters of the architecture, $\mathbf{h}_t$ the hidden state, $\mathbf{x}_t$ the input, and $\mathbf{c}_t$ the cell state at time step t. The particular search space defined by Zoph and Le [96] considered the addition and elementwise multiplication combination methods. Activation functions were limited to the identity, tanh, sigmoid, and ReLU activation functions [96].

The search space of the MOE/RNAS algorithm built in this study considers the addition, subtraction, and elementwise multiplication combination methods. The activation functions that the MOE/RNAS algorithm allows for are the linear, identity, tanh, sigmoid, ReLU, and leaky ReLU activation functions.

The encoding scheme employed by the MOE/RNAS algorithm works as follows. A directed acyclic graph (DAG) representation of a RNN architecture is assumed, where each node of the DAG is encoded by a block encoding structure. The block encoding structure developed for the MOE/RNAS algorithm represents a simplified version of the cell that is typically used in cell-based search space implementations, as previously discussed in Chapter 4. Therefore, a RNN architecture can be encoded using a number of blocks. A block is smaller compared to a cell and is constrained in terms of the operations that it supports, which is discussed in more detail below.

An input to a block is the output of a previous block in the architecture. A single block must have at least one input and can accept a maximum number of two inputs. If a block accepts two inputs, a combination method is required to specify how the inputs should be combined. The result from the combined inputs is then returned as the output of the block. When a block combines two inputs, the use of an activation function is optional. If an activation function is applied to the result of the block's combined inputs, the output of the activation function is returned as the output of the block. If the block accepts a single input value, an activation function must be specified and the output of the activation function is then returned as the output of the block. Figure 5.1 illustrates

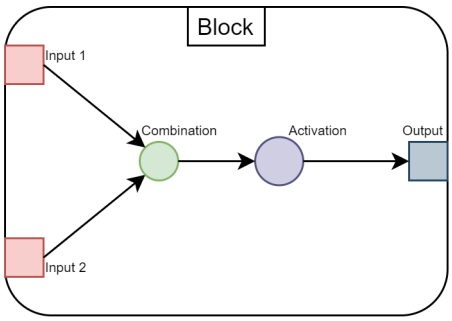


Figure 5.1: Block encoding.

the block encoding that is used by the MOE/RNAS algorithm.

The MOE/RNAS algorithm is developed such that it is capable of handling blocks that simultaneously combine two inputs and then apply an activation function to the result of the combined inputs. However, in this study, the combination and activation responsibilities are decoupled, which results in two separate blocks instead. A block that combines two inputs is referred to as a combination block, whereas a block that applies an activation function to the single input of the block is referred to as an activation block. This convention is adopted to allow for a more detailed discussion of the transformations that can be performed on an architecture, which is further discussed in Section 5.2. The distinction between combination and activation blocks also allows for a more detailed analysis of architectures during the empirical analysis process, which is discussed in Chapter 6.

The MOE/RNAS algorithm's search space is driven by a template that prescribes the following properties of the RNN architecture, which is assumed to contain an input layer, a hidden layer, and an output layer. The units found at each of the three layers will be encoded by the block encoding structure. The input layer has three immutable blocks, which represent the input, the previous hidden state, and the previous memory state, respectively. The three input blocks are activation blocks that apply the linear activation function to their respective inputs, and these blocks may only be used as inputs to subsequent blocks in the hidden layer of the architecture. The block that represents the input $\mathbf{x_t}$ is referred to as the x_t block. The blocks that represent the previous hidden state $\mathbf{h_{t-1}}$ and the previous memory state $\mathbf{c_{t-1}}$ are referred to as the

h_{t-1} block and the c_{t-1} block, respectively.

The output layer of the RNN architecture has two blocks, h_t and c_t, which represent the hidden state and memory state at time step t, respectively. The output layer blocks do not have any combination or activation functions, and can therefore only accept a single input, which is then used directly as the particular block's output value. The block that serves as the input to these output blocks can be changed during evolution. The output values of the h_t and c_t blocks are used as the inputs to the h_{t-1} and c_{t-1} input layer blocks at the following time step.

For a RNN architecture to be considered valid, both the x_t and h_{t-1} blocks must be used as inputs to at least one block in the hidden layer. The use of the c_{t-1} block is optional and has no impact on the validity of an architecture. The MOE/RNAS algorithm defines appropriate transformations that can be performed on the architecture so that the use of the c_{t-1} block can be altered during evolution; an architecture can therefore initially include the c_{t-1} block and after evolution exclude the c_{t-1} block as an input to blocks in the hidden layer.

An additional requirement specified by the template is that the h_t output layer block must have a valid input, which should be a block from the hidden layer. The c_t output layer block does not affect the validity of an architecture, and the architecture's use of the c_t output layer block depends on whether the c_{t-1} input layer block is used by a block in the hidden layer. The transformations responsible for altering the use of the c_{t-1} and c_t blocks are discussed in more detail along with the other transformations in Section 5.2.

An example of a block encoding representation of the basic RNN architecture,

$$\mathbf{h}_t = f_h(\mathbf{W}_h\mathbf{x}_t + \mathbf{b}_x + \mathbf{U}_h\mathbf{h}_{t-1} + \mathbf{b}_h),$$

with a tanh activation function can be seen in Figure 5.2. For the $\mathbf{x}_t$ input to the RNN, an x_t input layer block is created. Similarly, an h_{t-1} input layer block was created for the $\mathbf{h}_{t-1}$ input to the RNN. The *linear_b* block that accepts the x_t input layer block as its input, represents the weighted linear activation and bias of the $\mathbf{W}_h\mathbf{x}_t + \mathbf{b}_x$ inputs to the basic RNN architecture. A separate weighted linear activation and bias is used for the h_{t-1} input layer block. The outputs of the two linear activation blocks are then combined using an addition combination block. A *tanh* activation function is then applied to the

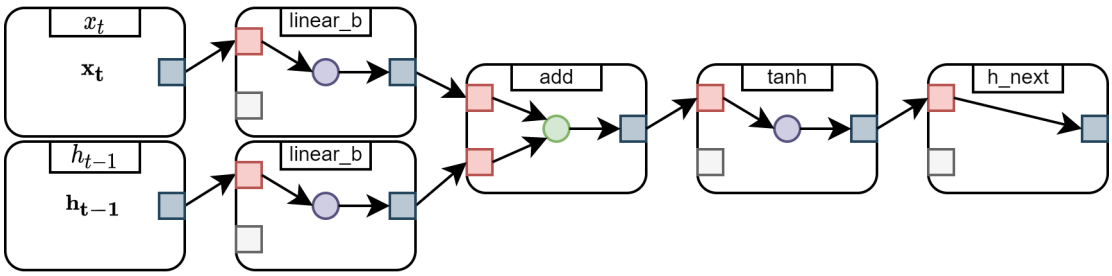


Figure 5.2: Basic RNN architecture block encoding structure.

output of the combination block, which represents the f_h activation function of the basic RNN architecture. The $\mathbf{h}_t$ output of the RNN architecture at time step t is represented by the h_{next} output layer block, which simply returns the output of the preceding $tanh$ activation block. Note that since the basic RNN architecture does not use the c_{t-1} input layer block, it is simply ignored.

A simplified illustration of the architecture from Figure 5.2 can be seen in Figure 5.3, which is how RNN architectures will be presented for the remainder of this dissertation. Table 5.1 lists the node values and their respective relationship to the RNN architecture encoding structure.

The next section discusses the evolutionary search strategy used in the MOE/RNAS algorithm for exploring the RNN architecture search space, and how the search strategy works to optimise RNN architecture-related objectives.

5.2 Search Strategy

Some NAS methods that consider a single architecture performance related objective have been criticised for their unfounded complexity and insignificant contribution towards the final architecture performance [46, 89]. Multi-objective EA-based NAS methods generationally evolve architectures with the goal of optimising more than a single architecture performance-related objective, which justifies a search strategy component that can be considered at least more complicated compared to a random architecture search approach. Multi-objective driven architecture evolution should therefore have a more significant contribution towards the final architecture performance than what can

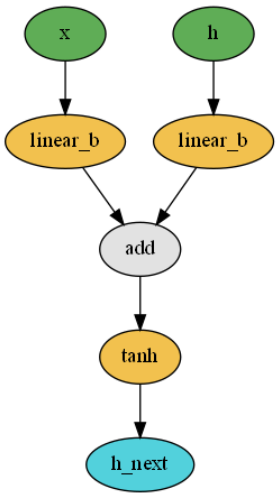


Figure 5.3: Basic RNN architecture.

be achieved with a random architecture search approach.

The underlying search strategy that is implemented by the MOE/RNAS algorithm is based on the NSGA-II algorithm. A population of RNN architecture candidate solutions are generationally evolved for a predefined number of generations to find the best performing architecture for the provided dataset, where architecture performances are based on multiple objectives. The MOE/RNAS algorithm maintains a Pareto-optimal front and employs the rank-based selection operator from the NSGA-II algorithm. Unlike the NSGA-II algorithm, the MOE/RNAS algorithm relies on a network morphism approach to generating offspring as opposed to a multi-parent recombination component. Pseudocode for the MOE/RNAS algorithm can be seen in Algorithm 4.

The rest of this section discusses the multi-objective EA-based search strategy that is implemented by the MOE/RNAS algorithm. Section 5.2.1 discusses the MOE/RNAS algorithm's network morphism approach for generating offspring. The initial population generation procedure is described in Section 5.2.2. Section 5.2.3 discusses the fitness evaluation of architectures in more detail. The MOE/RNAS algorithm's architecture selection strategy is discussed in Section 5.2.4.

Node value	Description	Inputs	Outputs
x	The $\mathbf{x}_t$ input at step t.	0	1
h	The $\mathbf{h}_{t-1}$ input at step t.	0	1
c	The $\mathbf{c}_{t-1}$ input at step t.	0	1
h_next	The new $\mathbf{h}_t$ hidden state for step t.	1	-
c_next	The new $\mathbf{c}_t$ memory state for step t.	1	-
add	Addition combination function.	2	1
sub	Subtraction combination function.	2	1
elem_mul	Elementwise multiplication combination function.	2	1
linear_b	Weighted linear activation function that includes a bias unit.	1	1
linear	Weighted linear activation function without a bias unit.	1	1
identity	Identity activation function.	1	1
sigmoid	Unipolar sigmoid (logistic) activation function.	1	1
tanh	Hyperbolic tangent activation function.	1	1
relu	Rectified Linear Unit (ReLU) activation function.	1	1
leaky_relu	Leaky ReLU activation function.	1	1
1	Integer value.		

Table 5.1: Descriptions of node values in architecture representation.

5.2.1 Recurrent Neural Network Morphism

The NSGA-II based search space exploration strategy implemented by the MOE/RNAS algorithm employs a network morphism approach instead of a recombination stage with crossover and mutation operators. With network morphism, a single offspring architecture is generated from a single parent architecture, which avoids the complexities associated with performing crossover on multi-parent RNN architectures, as observed in [3, 6].

Elsken *et al.* [24] postulated that the difference in performance between parent and offspring architectures should be low when a maximum number of three network transfor-

Algorithm 4 MOE/RNAS Algorithm

Inputs: N, ϕ, termination condition, objectives, seeds;
$i \leftarrow 0$; ▷ initialise generation counter
$\Upsilon \leftarrow \emptyset$; ▷ initialise empty archive for keeping track of previously evaluated architectures
$P \leftarrow initialisePopulation(N, seeds)$ ▷ initialise parent population of size N, include seed architectures
$Q \leftarrow$ initialise offspring population to $\emptyset$;
$f \leftarrow evaluateFitness(P)$;
$[F_1, F_2, ...] \leftarrow nondominatedSort(f, \hat{f}(P))$ ▷ calculate and construct Pareto-fronts based on nondomination
$\Upsilon \leftarrow \Upsilon \cup P$;
$p \leftarrow tournamentSelection(P, [F_1, F_2, ...])$
$Q \leftarrow generateOffspring(p, \phi)$ ▷ generate ϕ number of offspring architectures
while termination condition not met **do**
  $f' \leftarrow evaluateFitness(Q)$;
  $[F_1, F_2, ...] \leftarrow nondominatedSort(f \cup f', \hat{f}(P) \cup \hat{f}(Q))$
  $dist \leftarrow crowdingDistanceAssignment(F_1, F_2, ...)$
  $P \leftarrow survivorSelection(P \cup Q, [F_1, F_2, ...], dist, N)$
  $i \leftarrow i + 1$; ▷ update generation counter
  $\Upsilon \leftarrow \Upsilon \cup Q$;
  $Q \leftarrow generateOffspring(P, \phi)$
end while

mations are performed on the offspring architecture. This would allow for a more efficient performance evaluation strategy, since offspring models that share trained parameters with their parent models can be trained for fewer epochs [24].

The aforementioned parent-offspring performance difference postulation has only been tested in the case where up to three network transformations are performed on the offspring architectures [24]. Although the assumption allows for an efficient performance evaluation strategy, it may lead to a slower search space exploration when

architectures are only changed with a low number of network transformations after each generation. Therefore, the MOE/RNAS algorithm includes an input parameter that can be used to specify the maximum number of consecutive network transformations that are allowed to be performed on an individual architecture. If the input parameter value is greater than one, the number of consecutive network transformations that are performed on an architecture is randomly selected from the range between one and the specified number; a random number in this range is selected for each individual architecture. The network transformations included in the MOE/RNAS algorithm are described in detail below.

Recurrent Neural Network Transformations

1. *add_unit*: inserts a new activation block between two existing blocks in the architecture. A new block is created and assigned an activation function, which is randomly chosen from: $[linear\_b, linear, identity, sigmoid, tanh, relu, leaky\_relu]$ (see Table 5.1 for descriptions). An existing block b_r is randomly selected from the hidden layer. The newly created block is then inserted between block b_r and one of its inputs; if block b_r has two inputs, one is randomly selected. The effect of this transformation is that an activation will now be applied to selected input from block b_r before the input is passed into block b_r. This transformation is an adaptation of the add unit mutation developed by Bayer *et al.* [6]. Bayer *et al.* [6] restricted the activation function to the linear activation function whereas the MOE/RNAS algorithm randomly selects an activation function that is applied to the newly created block.

2. *remove_unit*: removes a randomly selected activation block from the hidden layer. The *remove_unit* transformation is effectively the inverse of the *add_unit* transformation. The single input of the activation block to be removed is set as the input to the subsequently connected blocks that expected the removed block as one of their inputs; this procedure ensures that there are no dangling blocks in the architecture. The *remove_unit* transformation is a destructive network transformation that allows for the optimisation of an architecture complexity objective.

3. *add_connection*: two randomly selected hidden layer blocks are combined. A constraint is enforced to ensure that the two blocks are not already combined or directly connected to each other. A new combination block is then created that accepts both the selected blocks as its inputs; the addition combination method is used for combining the two inputs. All the blocks in the architecture that expect the first of the two randomly selected blocks as their input are identified, and the newly created combination block is set as the replacement input to the identified blocks instead. This transformation is an adaptation of the add connection mutation developed by Bayer *et al.* [6]. Bayer *et al.* [6] stated that they connected the two units with an identity connection, whereas the MOE/RNAS algorithm introduces the new connection by using the elementwise addition combination method.

4. *remove_connection*: removes a randomly selected combination block from the hidden layer; only combination blocks with an addition combination method are considered. When a combination block is removed, it is possible that both of its inputs will be left unused.

   To deal with this, a procedure is implemented that inspects the architecture to identify the consequences of removing the selected combination block. If it is found that both of the selected combination block's inputs are used by other blocks in the architecture, then the combination block is a good candidate for the *remove_connection* transformation, and the transformation may therefore proceed without leaving unused blocks in the architecture. If no blocks can be found in the architecture that are good candidates for the *remove_connection* transformation, then the transformation is simply ignored.

5. *add_recurrent_connection*: introduces a connection between a randomly selected block b_r and either one of the h_t or c_t output layer blocks. This transformation is similar to the *add_connection* transformation, but aims to specifically add a connection between the randomly selected block and one of the output layer blocks.

   A newly created combination block with the addition combination method is set as the input to one of the output layer blocks, which is randomly selected. The input from the randomly selected output layer block is assigned as one of the inputs to

the newly created combination block. The randomly selected block b_r is then set as the second input to the newly created combination block.

This transformation provides for the ability to change an architecture so that it can start using the c_{t-1} input layer block if it has not done so previously, as discussed in Section 5.1.

6. *change_activation*: this transformation consists of randomly selecting an activation block from the hidden layer and then simply changing the block's specific activation function to a different activation function, which is randomly selected from the list of allowable activation functions as defined by the search space. The particular block's original activation function is excluded from the list of activation functions to choose from. This transformation was inspired by the change connection mutation developed by Bayer *et al.* [6]. However, Bayer *et al.* [6] considered the identity and linear activation functions exclusively, whereas the MOE/RNAS algorithm considers a wider range of activation functions.

7. *change_combination*: this transformation consists of randomly selecting a combination block in the hidden layer and then simply changing the block's specific combination method to a different combination method, which is randomly selected from the list of allowable combination methods as defined by the search space. The particular block's original combination method is excluded from the list of combination methods to choose from.

During generation of the initial population, the MOE/RNAS algorithm relies on network transformations for uniformly sampling architectures from within the defined search space. The MOE/RNAS algorithm's procedure for generating the initial population is discussed in the next section.

5.2.2 Initial Population

The MOE/RNAS algorithm employs a procedure that randomly generates architectures to make up the initial population. Existing architectures, such as the LSTM or GRU,

can be supplied to the MOE/RNAS algorithm, which are then included in the initial population.

If existing architectures are supplied, they are added to the initial population, after which new architectures are randomly generated until the size of the initial population equals the specified population size N. The MOE/RNAS algorithm's procedure for randomly generating an architecture starts with a base RNN architecture and then performs a number of consecutive network transformations on the architecture. The number of consecutive network transformations that are performed on the architecture is randomly selected from the range $[1, 10]$. The base RNN architecture includes the following blocks:

- b_1, the x_t input layer block;
- b_2, the h_{t-1} input layer block;
- b_3, the c_{t-1} input layer block;
- b_4, a linear activation block that receives b_1 as input;
- b_5, a linear activation block that receives b_2 as input;
- b_6, a linear activation block that receives b_3 as input;
- b_7, a block that receives blocks b_4 and b_5 as inputs and combines these inputs, the combination function is randomly chosen from $[add, sub, elem\_mul]$ (see Table 5.1);
- b_8, an activation block that receives b_7 as input, the activation function is randomly chosen from $[linear\_b, linear, identity, sigmoid, tanh, relu, leaky\_relu]$ (see Table 5.1);
- b_9, the h_t output layer block that receives b_8 as input;
- b_{10}, the c_t output layer block that receives b_6 as input.

The *remove_unit* and *remove_connection* network transformations are excluded when randomly generating architectures for the initial population. This is done so that only constructive network transformations are allowed, which will effectively promote a more diverse initial population.

Each architecture in the population is assigned a unique identifier. The unique identifier is generated using the template X_c, where X is a short string that is assigned to the initial architecture, and c is an integer to represent the count of the particular architecture. The initial architectures will start with a c value of 0, and subsequently generated offspring architectures will have increased values for c. Randomly generated architectures are assigned an X value of $rdmY$, where Y represents a unique integer assigned to that particular architecture. Therefore, the first randomly generated architecture in the initial population will be assigned the identifier $rdm0\_0$, the second randomly generated architecture in the initial population $rdm1\_0$, and so on. If existing architectures are supplied to be included in the initial population, they will be assigned appropriate identifiers. For example, if the LSTM architecture is supplied to be included in the initial population, the architecture's identifier will be $LSTM\_0$.

After the initial population generation procedure has concluded, the fitness values for each of the individual architectures are evaluated based on the provided objectives. The MOE/RNAS algorithm's fitness evaluation process is described in detail in the following section.

5.2.3 Fitness Evaluation

The fitness evaluation procedure implemented by the MOE/RNAS algorithm assumes the responsibility of the NAS performance estimation component. Thus, the performances of the architectures are based on the fitness values calculated by the MOE/RNAS algorithm's EA fitness evaluation method.

The fitness of architectures in the population are calculated based on the objectives provided. It is expected for one of the objectives to represent an architecture's achieved accuracy after being trained and validated on relevant subsets of the provided dataset. Furthermore, at least one objective should be included that relates to architecture complexity. The MOE/RNAS algorithm supports the following architecture complexity related objectives:

- the number of blocks that the architecture contains;
- the number of parameters of the model;

- the model inference time, i.e., how long the model takes for a forward propagation of a single input pattern.

The MOE/RNAS algorithm does not implement any specific techniques that predict model accuracy. Instead, the MOE/RNAS algorithm relies on existing methods to make the training and testing of models more efficient. As a result of the network morphism approach for generating offspring architectures along with parameter sharing between parents and offspring, the offspring models can be trained for fewer epochs, due to the difference between parent and offspring model performance previously discussed in Section 5.2.1. The performance difference between parents and offspring is based on the observations reported by Elsken *et al.* [24].

The MOE/RNAS algorithm allows for a threshold value to be specified, which is used to decide whether offspring models should be trained for a reduced number of epochs. After the offspring model has been trained for the reduced number of epochs, the performance of the offspring model is compared to the performance of the parent model. If the performance difference between the parent and offspring model is more than the specified threshold value, training of the offspring model will resume, and the offspring model will be trained for the same number of epochs that the parent model was trained for.

During the training of any given model, the performance of the particular model can be analysed after each epoch. If it is found that the performance of a particular model does not improve after each epoch, for a specified number of consecutive epochs, an early stopping procedure can be executed that will terminate the training of the model. Furthermore, if an offspring model is being trained for the same number of epochs than its parent, the performance of the offspring model can be compared to the performance of its parent after each epoch. If the offspring model does not exhibit improved performance compared to its parent for a specified number of consecutive epochs, an early stopping procedure can be executed, which will terminate the training of the offspring model. Both aforementioned early stopping mechanisms are optional and only provided for improved efficiency during model accuracy evaluation.

The MOE/RNAS algorithm performs the selection of architectures based on their respective fitness values and ranking during the evolutionary cycle, which is done ac-

cording to the selection operators of the NSGA-II algorithm. The next section provides an overview of how architecture selection is performed by the MOE/RNAS algorithm.

5.2.4 Selection

After the fitness values for each of the individuals in the population have been evaluated, the individuals are sorted based on their nondomination and placed into appropriate Pareto fronts. The nondominated sorting of individuals in the population based on their objective values is done according to the NSGA-II nondominated sorting method, without any adaptation. Similarly, the crowding distance assignment method from the NSGA-II is used verbatim. However, the MOE/RNAS algorithm includes an optional parameter to specify whether the original NSGA-II distance metric calculation should be used or whether the distance metric calculation

$$dis_j = dis_j + \frac{f^k_{n+1} - f^k_n}{f^k_{max} - f^k_{min}}, \tag{5.2}$$

which was proposed by Chu and Yu [16], should be used instead. The inclusion of the alternative distance metric calculation is motivated by the results published by Chu and Yu [16], wherein they have provided empirical evidence that the alternative distance metric calculation leads to a faster convergence of the Pareto-front, which was tested on nine different benchmark problems.

Survivor selection is performed in the same way as it is done by the NSGA-II algorithm, as previously discussed in Chapter 3. The NSGA-II algorithm generates N offspring, which results in a $2N$ sized combined population from which survivor selection is performed. With larger values of N, a significant number of models need to be trained and validated. The MOE/RNAS algorithm has an input parameter that can be used to specify the maximum number of parents to select for offspring generation. The top performing architectures are selected as parents if the aforementioned input parameter is smaller than N.

5.3 Summary

This chapter proposed the MOE/RNAS algorithm for multi-objective evolutionary RNN architecture search. The MOE/RNAS algorithm considers a template-driven RNN architecture search space and employs a modular block-based architecture encoding method. Destructive network transformations were defined that allow for the optimisation of RNN architecture complexity objective(s).

The effectiveness of the MOE/RNAS algorithm is evaluated in the next chapter by performing an empirical analysis on the RNN architectures found by the MOE/RNAS algorithm for natural language processing datasets.

Chapter 6

Empirical Analysis

This chapter discusses the experimental procedure followed and the experimental results obtained for this study. The MOE/RNAS algorithm proposed in Chapter 5 was implemented to find RNN architectures for two natural language processing (NLP) tasks. The purpose of the experimental work was to evaluate the effectiveness of the MOE/RNAS algorithm to find RNN architectures for the NLP tasks. The ability of the proposed MOE/RNAS algorithm to optimise RNN architecture complexity related objectives during evolution was also investigated.

The rest of this chapter is structured as follows. Section 6.1 provides an overview of the experimental method and Section 6.2 discusses the results of the experiments. Lastly, the chapter is concluded in Section 6.3.

6.1 Empirical Procedure

Designing an RNN architecture for a particular problem is a complex task, as discussed in Chapter 2. Current methods [6, 42, 46, 49, 96] for automated RNN architecture consider the accuracy of the model as the single objective to evaluate the performance of the RNN architecture. Since current methods for automated RNN architecture design disregard any objectives related to RNN architecture complexity, RNN architectures only grow in size, resulting in large models with many parameters [46]. Therefore, finding a reasonable trade-off between model performance and model computational resource demand is not

possible with the existing methods for automated RNN architecture design.

This section describes the experimental procedure followed in this study to evaluate the effectiveness of the MOE/RNAS algorithm proposed in Chapter 5. The effectiveness of the MOE/RNAS algorithm is judged on the basis of the MOE/RNAS algorithm's ability to automatically design well-performing RNN architectures for a particular problem while maintaining a reasonable trade-off between model accuracy and model computational resource demand.

The experimental results report on the proposed MOE/RNAS algorithm's ability to find and optimise RNN architectures for two separate NLP datasets. The following NLP tasks were considered:

1. A standard word-level language modeling task based on the Penn Treebank dataset. The Penn Treebank dataset is often used as a benchmark in RNN NAS research [42, 46, 49, 96]. Although it is unlikely for any current NAS method to find a novel RNN architecture that outperform state-of-the-art RNN architectures that were designed by human experts [32, 42], an EA-based RNN architecture search method has not been implemented on the Penn Treebank dataset.

2. A standard character-level language modeling task based on artificially generated strings from a context-sensitive language, which was previously used in the study published by Bayer *et al.* [6]. The training and testing datasets consisted of strings that were generated from the $a^n b^n c^n$ context-sensitive language, where the value of n was randomly selected from the range 1..10 for each string. The training and testing datasets are discussed in more detail in Section 6.2.2.

The Penn Treebank dataset contains 10 000 unique words, and is therefore a good candidate for testing whether the RNN architectures evolved by the MOE/RNAS algorithm can learn from the provided dataset. Since the models are expected to predict the next word in the sequence, model accuracy highly depends on what the model has learned from the data during training.

By artificially generating the character-level language modeling task's dataset from a context-sensitive language, the MOE/RNAS algorithm is inadvertently presented with a challenge to evolve RNN architectures with sufficient memory capabilities, such that

they can learn the significance of the determinism of the particular context-sensitive language. Therefore, this dataset is useful for gaining a better understanding of the relationship between multi-objective RNN architecture evolution and model accuracy.

Technical implementation details for this study are as follows:

1. All the source code implementations of this study were developed using the Python programming language. The PyTorch [69] framework was used for machine learning model training only.

2. The MOE/RNAS algorithm was built entirely from scratch, which includes the block encoding scheme and how a model is constructed from an encoding, the transformations and how these transformations were applied to the individuals, and the multi-objective EA (see Algorithm 4). Source code implementation of the MOE/RNAS algorithm is available at `https://github.com/reinn-cs/rnn-nas`.

3. Experiments were run on a single Nvidia V100 16GB GPU cluster at the Centre for High Performance Computing (CHPC).

6.2 Empirical Study

This section presents the empirical analysis of the results obtained after implementing the proposed method to automatically find and optimise RNN architectures. Section 6.2.1 discusses the word-level NLP task results. The character-level NLP task results are discussed in Section 6.2.2.

6.2.1 Word-Level Language Modeling Task

This section discusses the results obtained after implementing the MOE/RNAS algorithm to search for and optimise RNN architectures for a standard word-level language modeling task based on the Penn Treebank dataset. The model accuracy objective that is relevant to the Penn Treebank dataset is discussed below.

Evaluation of the Model Accuracy Objective

Existing NAS studies that use the Penn Treebank dataset for RNN architecture search based the model accuracy objective on the test perplexity achieved by the model after being trained on the training dataset [38, 42, 49, 96]. The test perplexity achieved by the model for the test set $D_G = d_1 d_2 ... d_Q$ is calculated by

$$PP(D_G) = \sqrt[Q]{\prod_{i=1}^{Q} \frac{1}{P(d_i|d_1...d_{i-1})}}, \tag{6.1}$$

which is described in more detail in Section 2.2.3.

Klyuchnikov *et al.* [42] adopted the RNN model training settings from Merity *et al.* [59], and found that a model comprised of three stacked cells (RNN architectures), a hidden layer unit dimension of 600, and a batch size of 20, yielded the best test perplexity on the Penn Treebank dataset. An LSTM model with the aforementioned hyperparameter setup has more than 10 million trainable parameters, which can take multiple hours to train, as observed in [49]. Furthermore, Klyuchnikov *et al.* [42] performed the same extensive hyperparameter optimisation as observed in [49, 59].

In this experiment, the RNN architectures created by the MOE/RNAS algorithm were not stacked, and each model contained a single instance of the corresponding RNN architecture. The models were implemented with a hidden layer dimension of 650, and a batch size of 20 was used during model training. Models were trained by a backpropagation training algorithm, and hyperparameter optimisation was done using optimisation techniques as provided by the PyTorch [69] framework; no further extensive hyperparameter optimisation was performed during the model accuracy evaluation stage. RNN models were unrolled for 35 time steps during backpropagation training.

To determine the best model training configuration, an LSTM model was trained for 50 epochs on the Penn Treebank dataset. Four different optimisation techniques were used to determine which yields the lowest test perplexity. The four different optimisation techniques and the corresponding LSTM model test perplexities achieved are listed in Table 6.1. From the results listed in Table 6.1, it can be seen that the standard stochastic gradient descent optimisation technique resulted in the LSTM model achieving the lowest test perplexity of 82.82.

Figure 6.1 illustrates the validation perplexity of the LSTM model during training, for each of the respective optimisation techniques considered. It can be seen in Figure 6.1a that the stochastic gradient descent optimisation technique converged soon after the 20^{th} epoch.

In consideration of the aforementioned observations, all models in this experiment were trained using the PyTorch provided stochastic gradient descent optimisation technique. All models in the initial population were trained for 30 epochs, and offspring models were trained for 5 epochs if a reduced number of epochs could be justified (see Section 5.2.3). The Penn Treebank dataset is already split into a training dataset which consists of 912 344 words, a validation dataset with 131 768 words, and a testing dataset that consists of 129 654 words.

Four different experiments were run to test the MOE/RNAS algorithm's ability to find RNN architectures for the Penn Treebank dataset. The differences between the four experiments relate to the parameters of the multi-objective EA implemented by the MOE/RNAS algorithm, such as the population size, termination condition, initial population seeds, and more. With four different experiments, a sufficient number of permutations of the MOE/RNAS algorithm's configurable parameters can be studied in relation to the MOE/RNAS algorithm's ability to find RNN architectures, while also taking the inherently high computational resource demand of NAS into account.

Each of the four different experiments are discussed below, under scenarios A1-A4, respectively. In summary, each of the scenarios report on the results of the MOE/RNAS algorithm's ability to find and optimise RNN architectures for the Penn Treebank dataset, but with different configurations; each of the scenarios provide a more detailed discussion on their respective implementation details and configuration.

Scenario A1

This scenario considered a population size of 100 architectures. The basic RNN, LSTM, and GRU architectures were included in the initial population, which resulted in 97 RNN architectures being randomly generated from the defined search space. 100 offspring architectures were generated for each generation. A maximum number of three consecutive network transformations were allowed during offspring generation. Objec-

Optimisation technique	Test perplexity
Stochastic gradient descent	**82.82**
Averaged stochastic gradient descent	89.35
Adam	108.66
Stochastic gradient descent with momentum	138.05

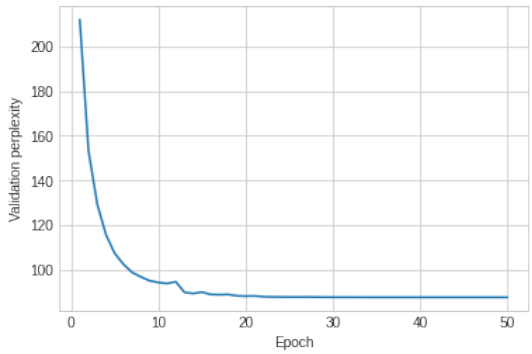
Table 6.1: Comparison of LSTM model test perplexities achieved after being trained with four different PyTorch optimisation techniques.

tives considered for this scenario were the model's test perplexity and the architecture's number of blocks. The rest of the MOE/RNAS algorithm implementation configuration that was used for this scenario is listed in Table 6.2.

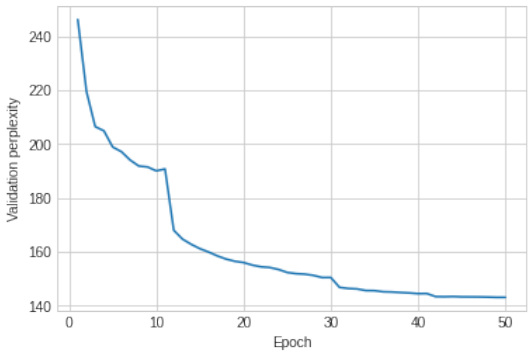
During the run of this scenario, the MOE/RNAS algorithm succeeded in optimising the architecture complexity objective by maintaining a consistent decrease in the average number of blocks across the population of architectures per generation, which can be seen in Figure 6.2a. However, the average test perplexity per generation does not exhibit a similar trend, but did not worsen across the generations, as illustrated in Figure 6.2b. Therefore, the MOE/RNAS algorithm was able to optimise the architecture complexity objective without negatively influencing the model accuracy objective across the 30 generations.

From the Pareto front illustrated in Figure 6.3, it can be seen that the LSTM and basic RNN architectures were dominated by other architectures, which includes the GRU architecture. The performances of the Pareto front architectures are listed in Table 6.3, which includes the performance of the LSTM_0 architecture for reference. After 30 generations, the MOE/RNAS algorithm was only able to optimise a single architecture that outperformed the original LSTM in terms of test perplexity. However, this particular architecture was an offspring architecture generated from the original LSTM_0 architecture.

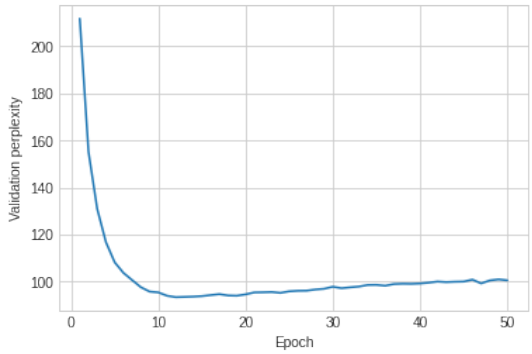
As observed in Table 6.3, the LSTM_58 architecture has one block less compared to the original LSTM_0. This was due to a *remove_connection* transformation that was performed on the architecture, which led to the removal of the memory gate's combination block that combined the x_t and h_{t-1} input layer blocks. Since a combination block



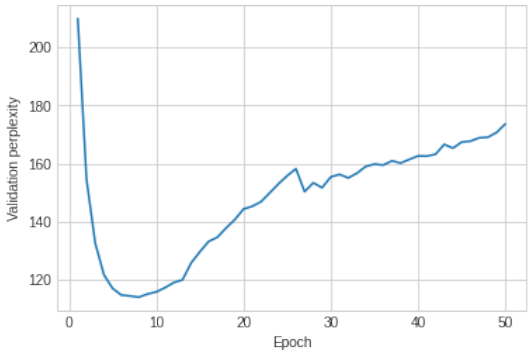
(a) Stochastic gradient descent



(b) Stochastic gradient descent with momentum



(c) Averaged stochastic gradient descent



(d) Adam

Figure 6.1: LSTM model optimiser validation perplexity results.

Parameter	Value
Population size	100
Offspring generation	100
Generations	30
Maximum transformations	3
Alternative crowding distance	True
Seed architectures	RNN, GRU, LSTM
Objectives	Number of blocks and test perplexity

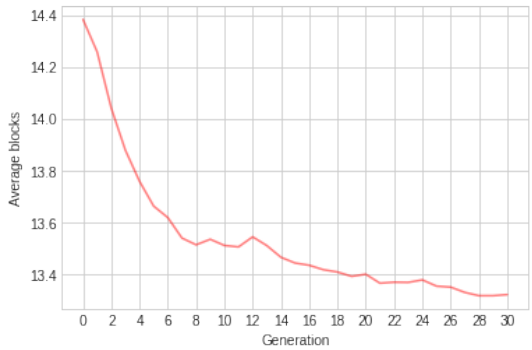
Table 6.2: Scenario A1 MOE/RNAS algorithm implementation configuration.

Architecture	Test perplexity	Blocks	Parameters	Training time (seconds)
LSTM_58	**83.782**	25	3 385 200	2 761.47
LSTM_0	83.945	26	3 385 200	3 948.45
GRU_0	89.766	23	2 538 900	3 089.10
rdm68_45	92.704	12	846 300	1 024.99
rdm8_0	99.625	10	846 300	1 103.25
rdm8_3	169.047	9	846 300	818.49
rdm8_190	172.487	**8**	846 300	914.77

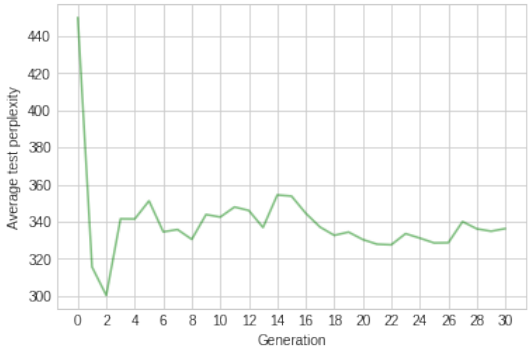
Table 6.3: Scenario A1 Pareto front architecture performances.

was removed from the architecture, the total number of parameters remained unchanged. An additional *change_combination* transformation was performed on the LSTM_58 architecture, which changed the combination method of the forget gate's combination block from an addition combination to an elementwise multiplication combination method. The aforementioned transformations led to an initial perplexity difference between the LSTM_0 parent architecture and the LSTM_58 offspring architecture of more than 5 perplexity points, which resulted in the LSTM_58 offspring architecture being trained for a total number of 30 epochs. The LSTM_58 offspring architecture's corresponding model parameters were initialised using the parent model's parameters, and the training time of the LSTM_58 offspring model was reduced by nearly 20 minutes. The LSTM_0 and LSTM_58 architectures can be seen in Figure 6.4 and Figure 6.5, respectively.

The rdm8_190 architecture had a total of eight blocks, and no other architectures were found with fewer blocks. The rdm8_190 architecture achieved a test perplexity of 172.48 as seen in Table 6.3, which is close to 100 perplexity points worse than the LSTM_0 architecture. However, the rdm8_190 architecture was only able to achieve a test perplexity of 172.48 due to the parameter sharing that is implemented with the network morphism approach of the MOE/RNAS algorithm. Illustrated in Figure 6.6, it can be seen that the rdm8_190 architecture merely combines the two linear activation nodes from the x_t and h_{t-1} input layer nodes. The basic RNN architecture, which can be seen in Figure 6.7, has an additional activation node after combining the x_t



(a) Average number of blocks per generation for A1.



(b) Average test perplexity per generation for A1.

Figure 6.2: Average number of blocks and average test perplexity per generation for scenario A1.

and h_{t-1} input layer nodes, yet it was only able to achieve a test perplexity of 730.94. Therefore, the rdm8_190 architecture was able to survive selection with its low perplexity score, which it was able to achieve due to the knowledge that was transferred from the parent models during the evolutionary cycle. This presents an interesting application of the MOE/RNAS algorithm to leverage the knowledge from large pre-trained models to evolve smaller models that can achieve marginally good accuracy, but with reduced computational demand.

During the experimental run of scenario A1, a total number of 2 836 constructive network transformations were performed and 1 619 destructive network transformations were performed, which is illustrated in Figure 6.8. It is expected for the number of constructive network transformations to be higher compared to the number of destructive network transformations, which is due to the exclusion of the destructive network transformations during the generation of the initial population, as previously discussed in Section 5.2.2.

The rdm68_45 architecture achieved the best test perplexity of 92.704 across all architectures that were generated and evolved by the MOE/RNAS algorithm in A1; the rdm68_45 architecture is illustrated in Figure 6.9. The LSTM outperformed the rdm68_45 architecture by 8.76 perplexity points, however, the rdm68_45 architecture has 14 blocks less compared to the LSTM. Furthermore, the rdm68_45 architecture has 2 538 900

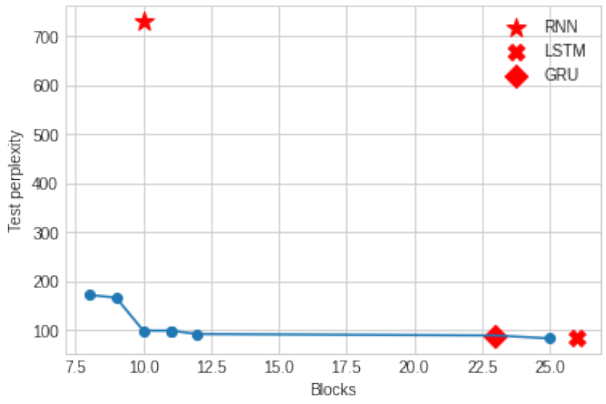


Figure 6.3: Scenario A1 Pareto front.

fewer parameters compared to the LSTM, which makes the rdm68_45 architecture significantly more efficient compared to the LSTM. The reduced computational demand of the rdm68_45 architecture justifies the reasonable 8.76 perplexity point trade-off compared to the better performing LSTM architecture.

Thus, in scenario A1, the MOE/RNAS algorithm succeeded in constructing novel RNN architectures capable of learning from the provided dataset, and was able to evolve the architectures such that the complexities of the architectures were optimised during the evolutionary cycle.

The total 30 generation search for A1 took 198 hours, which equates to a search cost of 8.25 GPU days (see Section 4.1.3).

Scenario A2

As observed in scenario A1, the LSTM architecture dominated the MOE/RNAS algorithm's generated RNN architectures in terms of test perplexity. A2 implemented the same MOE/RNAS algorithm implementation configuration as A1, except for including the LSTM and GRU architectures in the initial population. The rest of the MOE/RNAS algorithm implementation configuration that was used for this scenario is listed in Table 6.4.

The performances of the Pareto front architectures for A2 are listed in Table 6.5. It can be seen from Table 6.5 that the rdm35_108 architecture achieved the best test

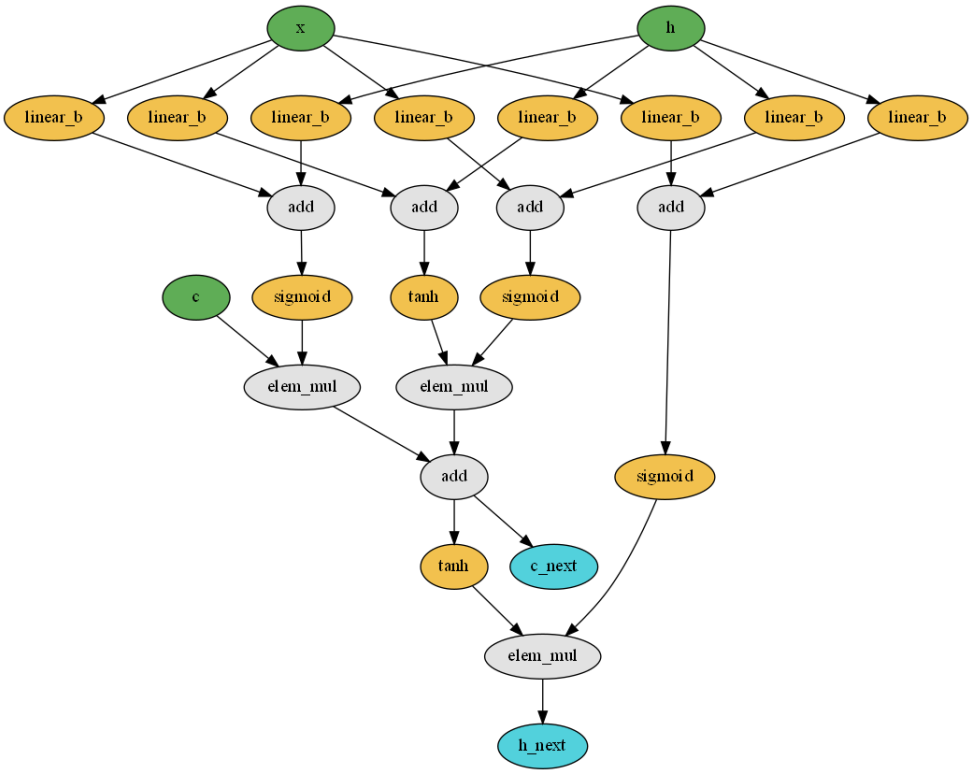


Figure 6.4: LSTM_0 architecture.

perplexity of 94.318 across all architectures found during the run for A2.

Similar to scenario A1, scenario A2 was able to find an architecture with 8 blocks, as shown in Table 6.5. It was observed that the rdm28_26 architecture was able to achieve a low perplexity due to the parameter sharing mechanism implemented by the MOE/RNAS algorithm, which is the same reason for the rdm8_190 architecture's achieved test perplexity as previously discussed in A1.

During this experiment, a similar consistency to that of A1 was observed in terms of the average number of blocks per architecture, which can be seen in Figure 6.10a. Optimisation of the architecture complexity objective is reflected in the network transformations that were performed in A2. A total number of 2 809 constructive network transformations were performed, whereas a total number of 1 852 destructive network transformations were performed, which can be seen in Figure 6.11. The average number of blocks for the final generation of A2 was 13.0 across all 100 architectures, whereas A1

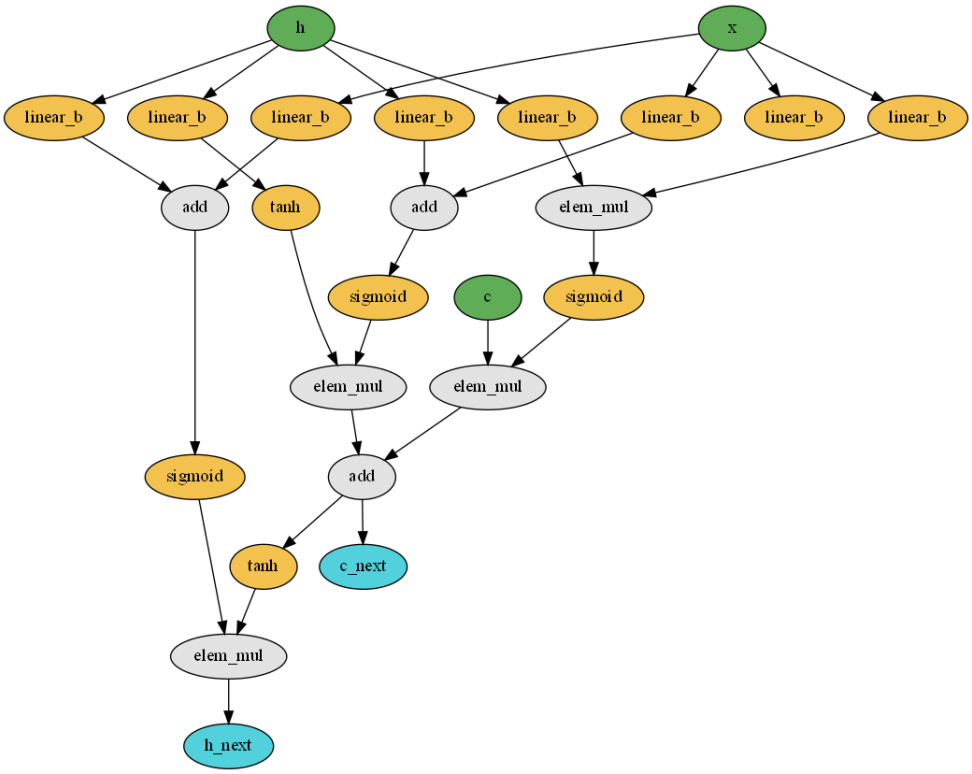


Figure 6.5: The LSTM_58 architecture evolved in scenario A1.

had an average number of 13.4 blocks after the final generation. The total number of destructive network transformations performed in A2 was higher compared to the 1 619 destructive network transformations performed in A1.

Although the average test perplexity per generation of scenario A2 exhibits some improvement, as shown in Figure 6.10b, the average test perplexity across all 30 generations did not go below 400 perplexity points. The average test perplexity per generation observed in A1 was better compared to what was observed in A2.

The rdm35_108 architecture achieved the best test perplexity across all architectures in A2. However, the best performing architecture generated by the MOE/RNAS algorithm in A1 outperformed the rdm35_108 architecture by 2 perplexity points.

An interesting observation can be made by analysing the rdm35_108 architecture, which is shown in Figure 6.12. The MOE/RNAS algorithm evolved a gate-like unit, similar to the gate units typically found in the LSTM and GRU architectures. This

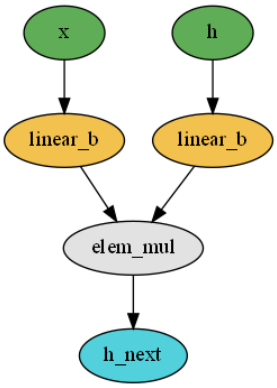


Figure 6.6: The rdm8_190 architecture evolved in scenario A1.

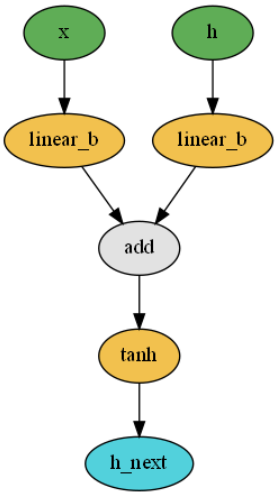


Figure 6.7: BASIC_0 architecture.

Parameter	Value
Population size	100
Offspring generation	100
Generations	30
Maximum transformations	3
Alternative crowding distance	True
Seed architectures	RNN
Objectives	Number of blocks, test perplexity

Table 6.4: Scenario A2 MOE/RNAS algorithm implementation configuration.

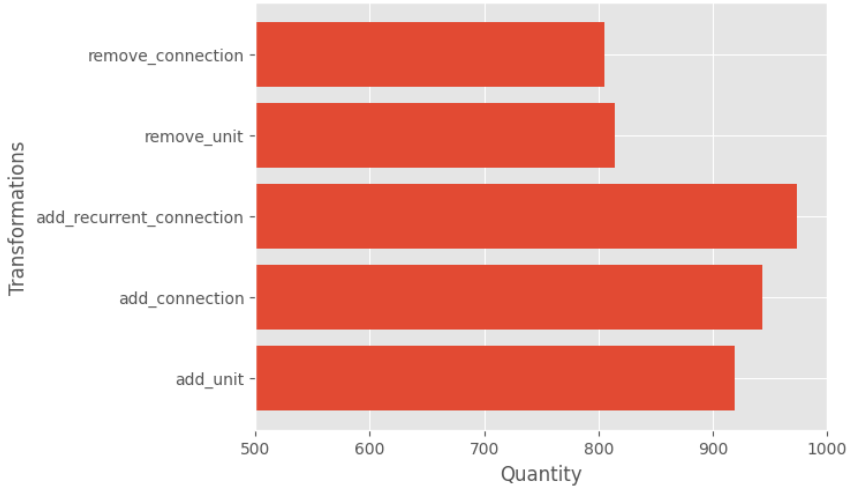


Figure 6.8: Total number of constructive and destructive network transformations that were performed during scenario A1.

gate-like unit is formed from the x_t and h_{t-1} input layer linear activation nodes that are combined by an elementwise multiplication combination node. The output of the combination node is then used as the input to a sigmoid activation node, and is also introduced into the path of the $\mathbf{c}_{t-1}$ memory state. Unfortunately, the c_{t-1} input layer node does not have any impact on the output of the architecture, since there are no paths from the c_{t-1} input layer node to the h_t output layer node. Therefore, the addition combination node that combines the output of the elementwise multiplication combination node and the c_{t-1} input layer node introduces unnecessary computation for calculating the output of the architecture.

Another noteworthy observation from the rdm35_108 architecture is the output of the two subsequently connected sigmoid activation nodes that are subtracted from the x_{t-1} input layer node. This is an unconventional configuration for a RNN architecture, when compared to hand-crafted architectures such as the LSTM and GRU, which highlights the stochastic nature of the EA driven RNN architecture offspring generation.

A2 had a total search cost of 6.25 GPU days, which is better compared to the 8.25 GPU days search cost of scenario A1. The search costs observed for scenarios A1 and A2 are relatively high, considering the efficiency techniques that are implemented, as previously discussed in Chapter 5. The stochastic nature of the EA contributes towards

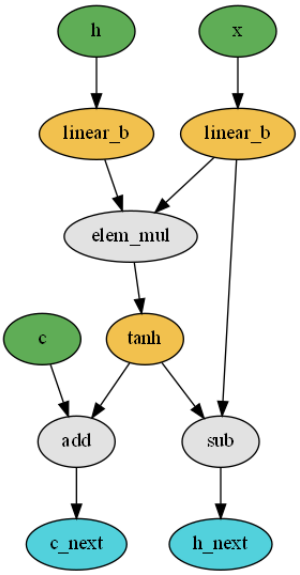


Figure 6.9: The rdm68_45 architecture evolved in scenario A1.

the higher search costs observed. This is because the recombination stage, which is implemented through network morphism, randomly selects the network transformations to perform, as previously discussed in Chapter 5. With the random network transformations that are performed on the architectures, it is more likely for the difference between offspring and parent model test perplexities to be higher than the threshold of 5 perplexity points. Thus, offspring models are trained for 30 epochs, since training the offspring models for a reduced number of epochs can no longer be justified.

Scenario A3

High search costs were observed in scenarios A1 and A2, where the searches were run for 30 generations and with population sizes of 100. In scenario A3, the search was run for 50 generations and with a population size of 30 instead. The LSTM and GRU architectures were included in the initial population. The rest of the MOE/RNAS algorithm implementation configuration that was used for this scenario is listed in Table 6.6.

After 50 generations and a total search cost of 7.5 GPU hours, no architectures were found that outperformed the original LSTM_0 architecture in terms of test perplexity achieved. Additionally, from the Pareto front illustrated in Figure 6.13, it can be seen

Architecture	Test perplexity	Blocks	Parameters	Training time (seconds)
rdm35_108	**94.318**	14	846 300	1 166.56
rdm35_116	96.588	15	846 300	1 160.75
rdm21_62	97.318	19	2 114 450	2 228.25
rdm21_40	99.476	17	2 114 450	2 016.04
rdm5_0	101.313	10	846 300	1 022.03
BASIC_18	165.862	9	846 300	370.34
rdm28_26	170.537	**8**	846 300	215.42

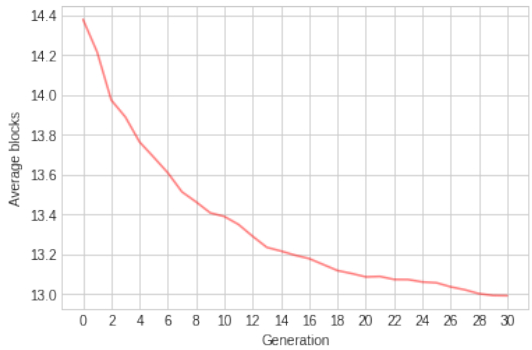
Table 6.5: Scenario A2 Pareto front architecture performances.

that the LSTM architecture was included in the non-dominated set after the run was terminated. Furthermore, from the performances of the architectures in the Pareto front listed in Table 6.14, it is observed that three architectures in the Pareto front are related to the LSTM architecture.

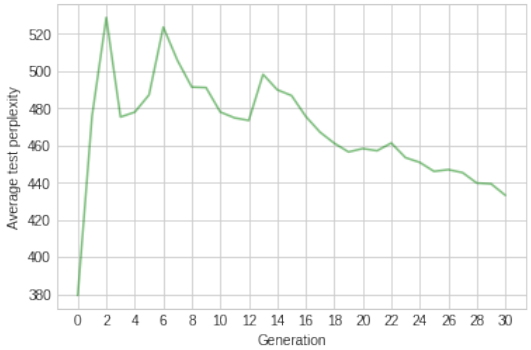
Contrary to what was observed in scenarios A1 and A2, the average number of blocks per generation increased across the 50 generations in scenario A3, which can be seen in Figure 6.14a. This is due to the smaller population size that was used, which resulted in a less diverse population. The LSTM_0 architecture was the better performing architecture in terms of test perplexity and eventually started dominating the population, which is also reflected in the Pareto front listed in Table 6.7. The LSTM_0 architecture has many blocks, and therefore the offspring generated from the LSTM_0 had similar numbers of blocks, which leads to the increasing number of average blocks per generation.

A total number of 1 255 constructive network transformations were performed and a total number of 789 destructive network transformations were performed, which can be seen in Figure 6.15. Although A3 was run for more generations compared to A1 and A2, the total number of transformations performed in A3 were expected to be lower compared to A1 and A2, since a population size of 30 was considered.

Although the average test perplexity per generation did not exhibit any consistent improvement over the 50 generations for scenario A3 as shown in Figure 6.14b, the average test perplexity for generation 8 was below 260 perplexity points, which is the



(a) Average number of blocks per generation for A2.



(b) Average test perplexity per generation for A2.

Figure 6.10: Average number of blocks and average test perplexity per generation for scenario A2.

lowest average test perplexity for a generation observed among scenarios A1-A3.

The rdm19_69 architecture was the best performing novel architecture created by the MOE/RNAS algorithm during the experiment run for A3; the rdm19_69 architecture can be seen in Figure 6.16. The rdm19_69 architecture does not present any interesting RNN architecture differences compared to the previous architectures encountered from A1 and A2. However, during evolution, the rdm19_69 architecture started using the c_{t-1} input layer node, which was subsequently added to the path of the x_t and h_{t-1} input layer nodes. Unfortunately, the architecture's use of the c_{t-1} node does not have any impact on the output of the architecture, since there are no connections from the c_{t-1} path that were introduced into the h_t output layer node's path.

Overall, A3 did not perform as well as A1 and A2, and a reduced population size meant that a smaller area of the search space was explored during A3. Although the experiment was run for 50 generations, the better performing LSTM and GRU architectures could not be outperformed in terms of test perplexity. However, the MOE/RNAS algorithm was still able to construct novel RNN architectures that are capable of learning from the provided dataset, and sufficient architecture complexity objective optimisation was observed, which was reflected by the evolution of the rdm19_69 architecture with 14 blocks that achieved a test perplexity of 94.088.

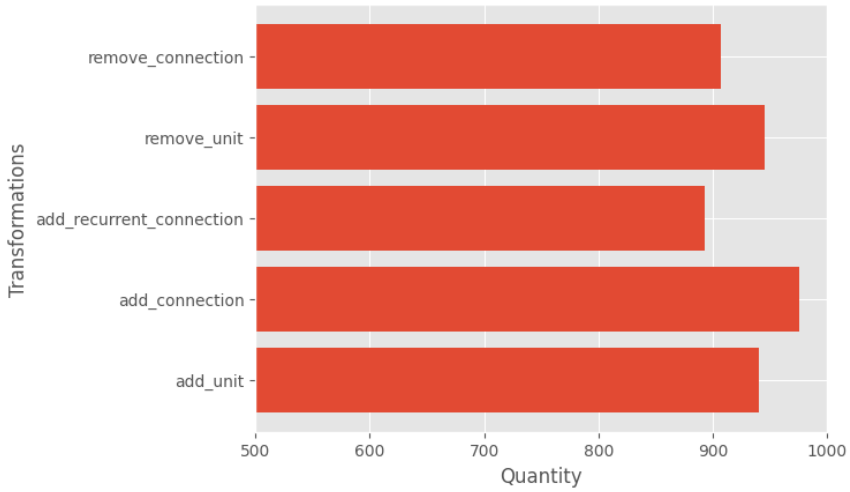


Figure 6.11: Total number of constructive and destructive network transformations that were performed during scenario A2.

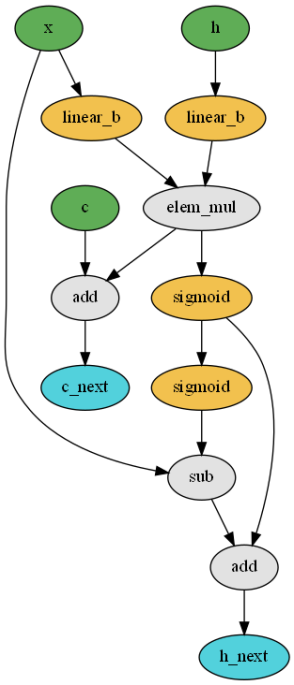


Figure 6.12: The rdm35_108 architecture evolved in scenario A2.

Parameter	Value
Population size	30
Offspring generation	30
Generations	50
Maximum transformations	3
Alternative crowding distance	True
Seed architectures	RNN, GRU, LSTM
Objectives	Number of blocks, test perplexity

Table 6.6: Scenario A3 MOE/RNAS algorithm implementation configuration.

Scenario A4

Scenario A4 implemented the same MOE/RNAS algorithm search configuration as A2 with the only exception being the original NSGA-II algorithm's crowding distance calculation was used instead. A2 resulted in a better average number of blocks per generation and exhibited a more consistent improvement of the average test perplexity per generation compared to A1 and A3. From Figure 6.17a it can be seen that A4 achieved a similar consistency regarding the average number of blocks per generation as observed in A2. However, the average number of blocks per generation in A4 were higher compared to the average number of blocks per generation in A2.

Although the average test perplexity per generation for A4 was lower than what was observed in A2, A2 exhibited a more consistent decrease in average test perplexity per generation as the evolutionary cycle progressed. Furthermore, the majority of the top performing architectures in terms of test perplexity observed in A2 were offspring architectures, whereas the best performing architecture in terms of test perplexity in A4 was an architecture generated during the initial population.

Table 6.9 lists the performances of the architectures of the Pareto front, which shows that after 30 generations, no architectures were found that outperformed the rdm76_0 architecture's achieved test perplexity. However, the rdm47_58 architecture had a much shorter training time compared to the rdm76_0 architecture, even though the rdm47_58 architecture had more trainable parameters. This was due to the rdm47_58 architecture

Architecture	Test perplexity	Blocks	Parameters	Training time (seconds)
LSTM_0	**83.945**	26	3 385 200	3 928.27
GRU_10	84.048	22	2 538 900	2 323.99
LSTM_176	84.352	26	3 385 200	3 012.34
LSTM_92	84.381	26	3 385 200	2 682.04
LSTM_105	84.653	26	3 385 200	2 319.65
rdm19_69	94.088	14	1 268 800	1 492.20
rdm19_0	95.310	11	846 300	1 107.66
rdm11_105	99.321	10	846 300	1 111.05
rdm11_55	172.809	**8**	846 300	727.38

Table 6.7: Scenario A3 Pareto front architecture performances.

Parameter	Value
Population size	100
Offspring generation	100
Generations	30
Maximum transformations	3
Alternative crowding distance	False
Seed architectures	RNN
Objectives	Number of blocks, test perplexity

Table 6.8: Scenario A4 MOE/RNAS algorithm implementation configuration.

Architecture	Test perplexity	Blocks	Parameters	Training time (seconds)
rdm76_0	**92.46**	13	846 300	1 380.75
rdm47_58	99.036	22	1 268 800	277.29
rdm34_24	99.448	16	846 300	1 561.85
BASIC_31	100.051	10	846 300	959.98
rdm4_46	169.556	**9**	846 300	480.02

Table 6.9: Scenario A4 Pareto front architecture performances.

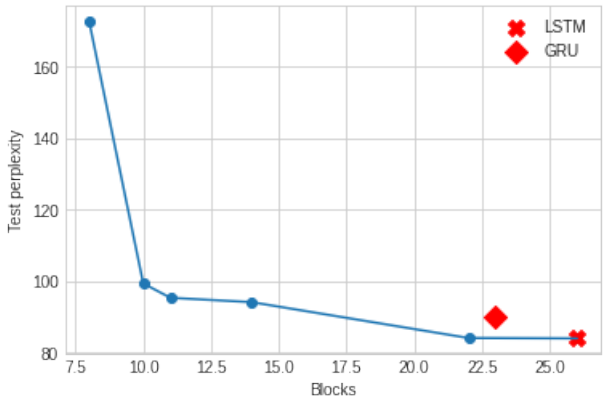


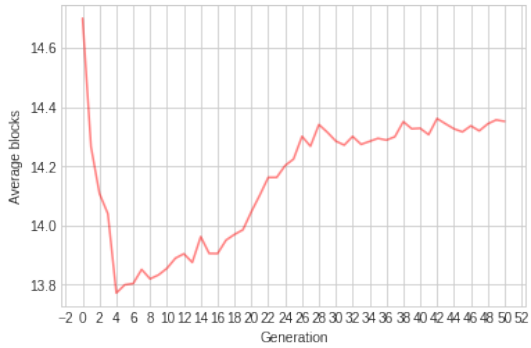
Figure 6.13: Scenario A3 Pareto front.

being trained for a reduced number of epochs, since the initial perplexity difference between the rdm47_58 architecture and its parent was low enough to justify training the offspring model for a reduced number of epochs (refer to Section 5.2.3).

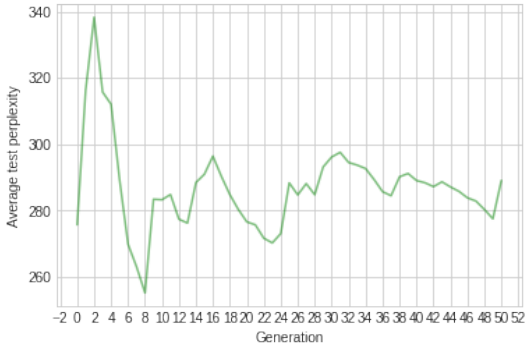
Overall, the use of the original NSGA-II algorithm's crowding distance calculation did not yield any promising results in A4. The best performing architecture in A4 was not evolved over a number of generations, and instead the best performing architecture was an architecture randomly generated during the initialisation of the initial population. Thus, based on the results obtained from A1-A3, the alternative crowding distance metric yielded better convergence of the Pareto front, and was able to evolve better performing architectures in terms of the model accuracy objective that was considered.

Summary

From scenarios A1-A4, it was observed that the MOE/RNAS algorithm can easily find architectures that dominate the LSTM when multiple objectives are considered. However, among the total of 10 830 architectures evaluated throughout scenarios A1-A4, the LSTM architecture remains superior regarding the model test perplexity achieved, which also reflects the results from [38, 42]. The destructive network transformations that were included during offspring generation were successful in optimising the architecture complexity objective. Furthermore, it was observed that the MOE/RNAS algorithm is capable of constructing RNN architectures that can learn from the provided dataset.



(a) Average number of blocks per generation for A3.



(b) Average test perplexity per generation for A3.

Figure 6.14: Average number of blocks and average test perplexity per generation for scenario A3.

Although the MOE/RNAS algorithm implements techniques that make the RNN architecture search method more efficient, such as parameter sharing and early stopping, the search costs observed across scenarios A1-A4 were still relatively high compared to the 1 GPU day search cost of the DARTS algorithm [49]. However, the best performing novel RNN architecture evolved by the MOE/RNAS algorithm achieved a test perplexity of 92.704 with 846 300 parameter, after being trained for less than 20 minutes (with parent parameter sharing). In comparison, the LSTM achieved a test perplexity of 83.945 with more than 3 million parameters, after being trained for more than one hour (see Table 6.3). Thus, the MOE/RNAS algorithm is clearly capable of catering for a reasonable trade-off between model accuracy and model computational resource demand.

The results observed from scenarios A1 and A2 were relatively similar, whereby both scenarios achieved a similar consistency in terms of the optimisation of the RNN architecture complexity-related objective. Scenario A1 achieved a better average test perplexity per generation compared to scenario A2, which is attributed to the difference in configuration between the two scenarios. The only difference between the configuration of scenarios A1 and A2 was that A1 included the LSTM and GRU architectures in the initial population. The LSTM and GRU architectures are expected to perform better with the model accuracy objective compared to the smaller RNN architectures generated by the MOE/RNAS algorithm. Therefore, the offspring architectures generated

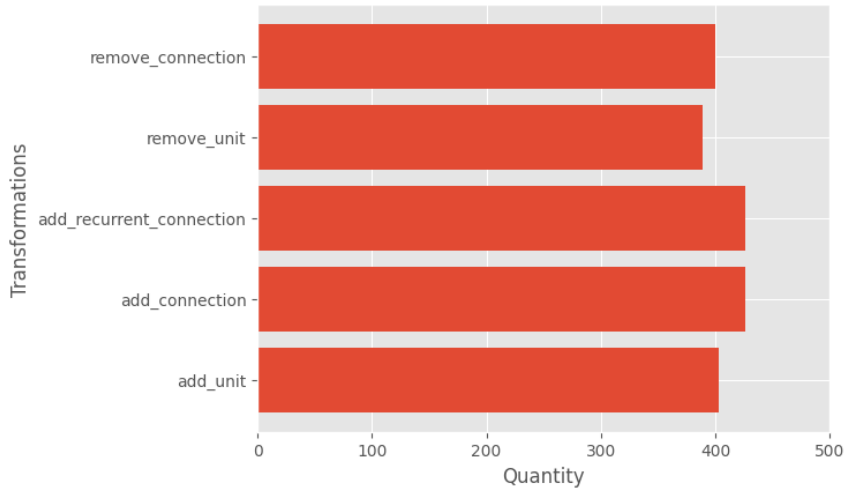


Figure 6.15: Total number of constructive and destructive network transformations that were performed during scenario A3.

from the LSTM and GRU architectures achieved comparable results to their respective parent architectures, resulting in a lower average test perplexity per generation than what was observed in A2 where the LSTM and GRU architectures were excluded from the population entirely.

Scenario A3 tested the MOE/RNAS algorithm with a smaller population size compared to the 100 population size that were used in scenarios A1 and A2. It was observed that the MOE/RNAS algorithm was still able to construct and evolve novel RNN architectures that are capable of learning from the provided dataset, but the smaller population size resulted in an expected decrease in diversity among the population. Scenario A4 implemented the same configuration for the MOE/RNAS algorithm that was used in scenario A2, except for the use of the crowding distance calculation. Scenario A4 considered the original NSGA-II crowding distance calculation as opposed to the alternative crowding distance calculation of Chu and Yu [16] that was used in scenarios A1-A3. Compared to scenario A2, scenario A4 performed worse and did not achieve a similar consistency in terms of the optimisation of either objectives. Therefore, based on the observations from scenarios A1-A3, the alternative crowding distance calculation of Chu and Yu [16] is preferred when implementing the MOE/RNAS algorithm for evolving RNN architectures.

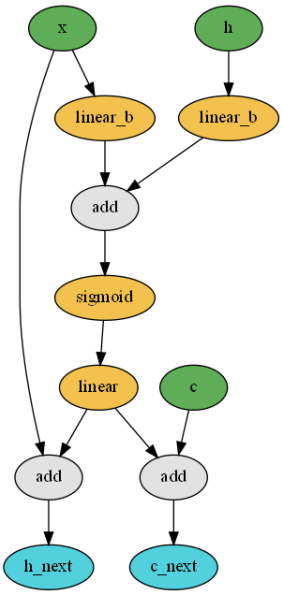
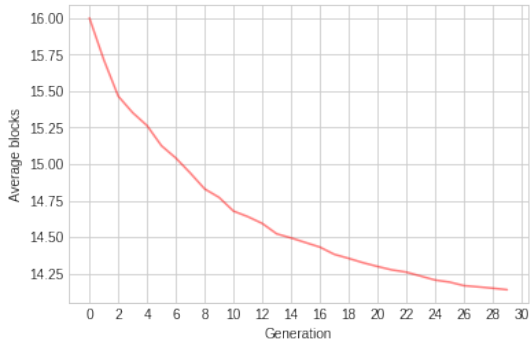


Figure 6.16: The rdm19_69 architecture evolved in scenario A3.

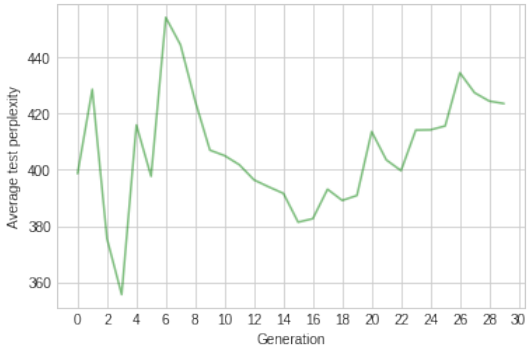
6.2.2 Character-Level Language Modeling Task

This section discusses the results obtained after implementing the proposed MOE/RNAS algorithm to search for and optimise RNN architectures for a character-level language modeling task. The dataset used for this character-level language modeling task was generated from the $a^n b^n c^n$ context-sensitive language, which is the same context-sensitive language used by Bayer *et al.* [6] in their multi-objective EA-based RNN architecture search method. The approach from Bayer *et al.* [6] did not consider any RNN architecture complexity-related objectives. Therefore, Bayer *et al.* [6] did not develop any transformations that are capable of optimising RNN architecture complexity-related objectives, whereas the MOE/RNAS algorithm is fully capable of optimising RNN architecture complexity-related objectives (see Chapter 5).

The training and testing datasets consisted of strings that were generated from the $a^n b^n c^n$ context-sensitive language. The training dataset consisted of 500 strings generated from the language $a^n b^n c^n$, where the value of n was randomly selected from the range 1..10 for each string. The testing dataset was limited to 100 strings, and the values for n randomly chosen from the range 1..10. For example, $n = 3$ results in the



(a) Average number of blocks per generation for A4.



(b) Average test perplexity per generation for A4.

Figure 6.17: Average number of blocks and average test perplexity per generation for scenario A4.

string *aaabbbccc* being generated. One single input sequence from either the training or testing datasets consisted of a string where each character of that particular string was considered an input in the input sequence. For each of the input sequences, the model was presented with an arbitrary sub-string of the particular input sequence and the model was then expected to predict the remaining characters of the string from that particular input sequence.

Evaluation of the Model Accuracy Objective

The model accuracy objective considered throughout this experiment was based on the mean squared error (MSE) loss obtained by the model on the generated testing dataset, after the model was trained on the training dataset. In this experiment, the RNN architectures created by the MOE/RNAS algorithm were not stacked, and each model contained a single instance of the corresponding RNN architecture. The models were implemented with a hidden layer dimension of 128, and since the dataset is relatively small, batching was not implemented during training. The models were unrolled for the full length of the input sequence, which was up to a maximum of 10 steps. Training of the models was done using a backpropagation training algorithm that used a PyTorch stochastic gradient descent optimisation technique.

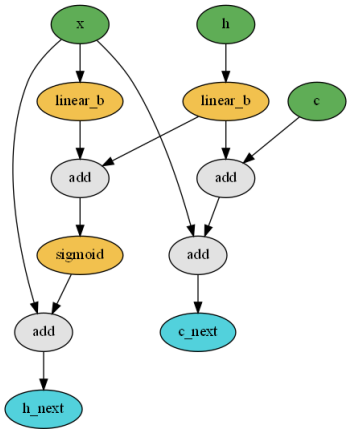


Figure 6.18: The rdm76_0 architecture evolved in scenario A4.

Similar to Section 6.2.1, four different experiments were run to test the MOE/RNAS algorithm's ability to find and optimise RNN architectures for the character-level NLP task. Each of the four different experiments are discussed below, under scenarios B1-B4, respectively. The differences between the four scenarios relate to the configuration of the multi-objective EA implemented by the MOE/RNAS algorithm. More specifically, the differences between the four scenarios were implemented such that the impact of the population size and the number of offspring architectures generated for each generation can be studied. Additionally, the different scenarios considered more aggressive RNN architecture evolution by allowing more consecutive network transformations during network morphism compared to what was previously considered in Section 6.2.1. Each of the scenarios provide a more detailed discussion on their respective implementation details and configuration.

Scenario B1

For this scenario, a population size of 100 architectures were considered. Parent selection were limited to the top 25 architectures of the Pareto front (see Section 5.2.4). Thus, only 25 offspring architectures were generated for each generation. During offspring generation, up to ten network transformations were allowed per architecture (see Section 5.2.1). Bayer *et al.* [6] stated that they found the best performing RNN architectures within 10 generations. For this scenario, the run was terminated after 15 generations. The rest of

Parameter	Value
Population size	100
Offspring generation	25
Generations	15
Maximum transformations	10
Alternative crowding distance	True
Seed architectures	RNN, GRU, LSTM
Objectives	Number of blocks, MSE loss

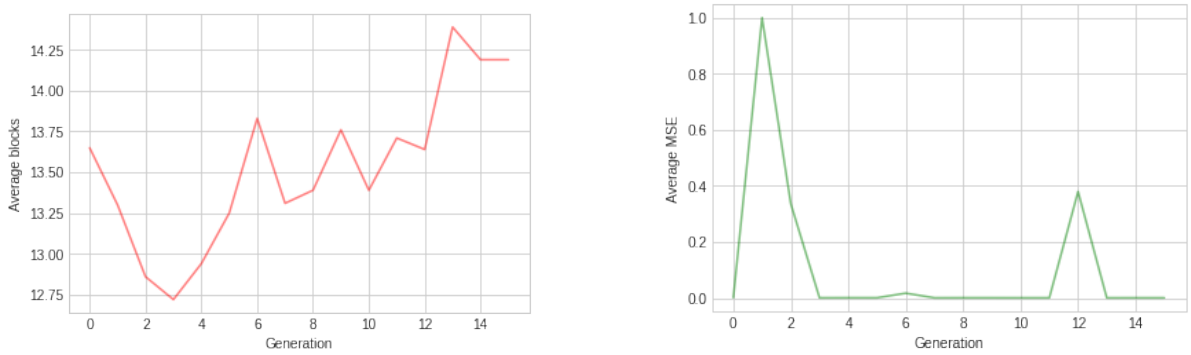
Table 6.10: Scenario B1 MOE/RNAS algorithm implementation configuration.

the MOE/RNAS algorithm implementation configuration used for this scenario is listed in Table 6.10.

According to Figure 6.19a, the MOE/RNAS algorithm struggled to maintain an optimised RNN architecture complexity objective across the population of RNN architectures, since the average number of blocks per generation increased as the evolutionary cycle progressed. This was a result of the increased number of consecutive network transformations allowed during network morphism. The increased number of consecutive network transformations led to a total number of 1 346 constructive network transformations performed, whereas a total number of 570 destructive network transformations were performed, which can be seen in Figure 6.20.

Although the average MSE per generation shown in Figure 6.19b does not exhibit a noticeable trend, the MOE/RNAS algorithm was able to successfully optimise the model accuracy objective. According to the performances of the architectures of the Pareto front listed in Table 6.11, it is observed that the MOE/RNAS algorithm was able to find and evolve a novel RNN architecture that outperformed the LSTM in both the model accuracy and architecture complexity objectives.

The rdm82_21 architecture shown in Figure 6.21 is particularly interesting. During the network morphism, the validity of an architecture is determined based on its use of the hidden state blocks, as previously discussed in Section 5.2.1. There is no verification performed to verify that a path exists exactly from the h_{t-1} input layer block to the h_t output layer block. The evolutionary algorithm exploited this during the evolution of



(a) Average number of blocks per generation for B1.

(b) Average MSE loss per generation for B1.

Figure 6.19: Average number of blocks and average MSE loss per generation for scenario B1.

the architecture rdm82_21. The h_t output layer block has at least one input, and there is at least one other block that uses the h_{t-1} block as its input. Thus, the generation of this particular architecture did not violate any of the predefined constraints.

The interesting observation from the rdm82_21 architecture is that it still maintains a recursive structure through the path of the c_{t-1} input layer block, which eventually reaches the h_t output layer block. The output of the h_t output layer block at the last input of the input sequence is used as the output of the architecture. Thus, the architecture effectively used the c_t output layer block as a substitute for the hidden state.

The total search time for scenario B1 was eight hours. This shorter search time compared to the search time of the experiments in Section 6.2.1 was due to a significantly smaller training dataset. Additionally, since only 25 offspring architectures were created per generation, fewer models had to be trained per generation.

Overall, scenario B1 yielded interesting results in terms of the architectures found during the search. Despite being unable to optimise the average number of blocks per generation, the MOE/RNAS algorithm was able to find and evolve a novel RNN architecture that dominated the LSTM in less than 15 generations.

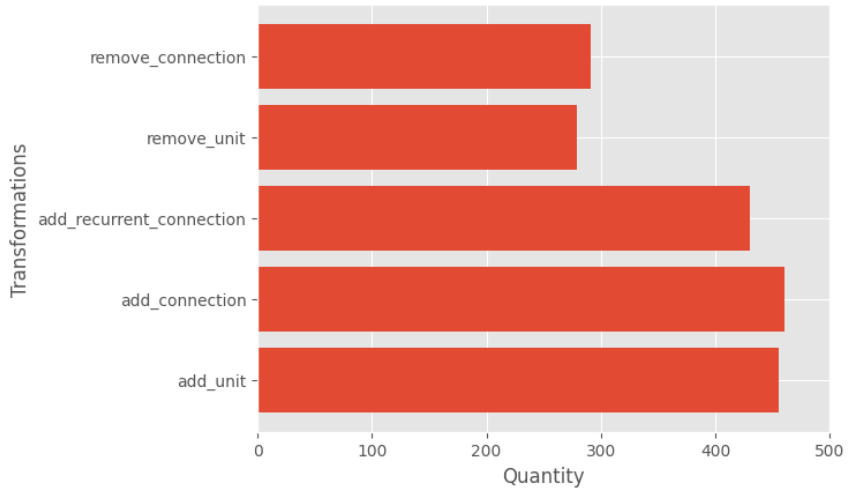


Figure 6.20: Total number of constructive and destructive network transformations that were performed during scenario B1.

Scenario B2

Scenario B2 used the same MOE/RNAS algorithm implementation configuration that was used for scenario B1, which is listed in Table 6.10. This was done in an attempt to reproduce the observations from B1, which included the MOE/RNAS algorithm's ability to find and evolve a novel RNN architecture that dominates the LSTM.

From Figure 6.23a it can be seen that the average number of blocks across the population increased as the evolutionary cycle progressed, similar to what was observed in B1. Additionally, a total number of 1 345 constructive network transformations were performed and a total number of 622 destructive network transformations were performed, as shown in Figure 6.24.

According to Figure 6.23b, the average MSE per generation observed in B2 was more inconsistent than what was observed in B1. This observation is reflected in the performances of the architectures in the Pareto front, which are listed in Table 6.12. As shown in Table 6.12, the MOE/RNAS algorithm was unable to find and evolve a novel RNN that outperformed the LSTM in terms of MSE achieved.

Although the average number of blocks per generation increased in scenario B2, the MOE/RNAS algorithm was able to optimise the architecture complexity objective. As

Architecture	MSE loss	Blocks	Parameters	Training Time (seconds)
rdm82_21	**0.000014**	16	50 692	108.200
LSTM_0	0.000157	26	133 252	700.270
rdm82_18	0.000939	14	50 692	71.739
rdm1_21	0.001806	11	34 180	43.140
rdm43_0	0.017431	10	34 180	221.099
BASIC_29	0.064309	**9**	34 180	42.937

Table 6.11: Scenario B1 Pareto front architecture performances. The performance of the LSTM architecture is included for reference.

shown in Table 6.12, the RNN architectures generated by the MOE/RNAS algorithm have fewer blocks compared to the other top performing architectures, and the corresponding models of the RNN architectures generated by the MOE/RNAS algorithm were able to achieve marginally good accuracies.

The rdm44_6 architecture is the only noteworthy RNN architecture constructed by the MOE/RNAS algorithm in B2. Although the rdm44_6 architecture did not make it to the Pareto front, it was able to achieve an MSE of 0.007755. For the rdm44_6 architecture, the MOE/RNAS algorithm evolved an interesting path for the c_{t-1} input node to the h_t output layer node, which can be seen in Figure 6.25. Although the c_{t-1} input layer node has no impact on the output of the architecture, evolution of the architecture happened such that the x_t input layer node was introduced into the c_{t-1} input layer node's path to the h_t output layer node, which ensured that the x_t input layer node maintains some contribution towards the final output of the architecture.

After 15 generations and a total search time of 8.5 hours, the MOE/RNAS algorithm was unable to produce a novel RNN architecture that outperformed the LSTM in terms of both objectives considered. However, the RNN architectures found during B2 had significantly fewer trainable parameters compared to the LSTM and GRU architectures, which resulted in reduced computational resources demanded by the models and quicker training times, as shown in Table 6.12.

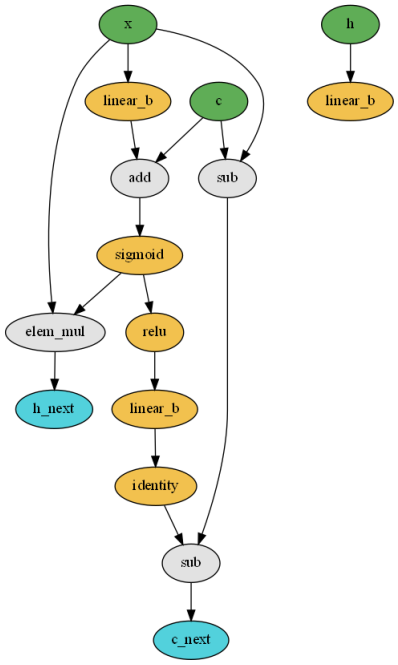


Figure 6.21: The rdm82_21 architecture evolved in scenario B1.

Scenario B3

For scenario B3, a population size of 100 was considered and parent selection was limited to the top 40 architectures of the Pareto front. Additionally, the experiment for B3 was run for 50 generations, and the LSTM and GRU architectures were excluded from the initial population. The rest of the MOE/RNAS algorithm implementation configuration for scenario B3 is listed in Table 6.13.

During B3, the MOE/RNAS algorithm faced a similar difficulty observed in scenarios B1 and B2 regarding the optimisation of the average number of blocks across the population, which is shown in Figure 6.26a. After 50 generations, a total number of 5 244 constructive network transformations were performed and a total number of 3 241 destructive network transformations were performed, which are illustrated in Figure 6.27.

According to Figure 6.26b, B3 exhibited a similar inconsistency regarding the average MSE per generation across the population compared to scenarios B1 and B2. Therefore, it is clear that when a higher number of consecutive network transformations are considered, the network morphism phase is too aggressive, which leads to an inconsistent

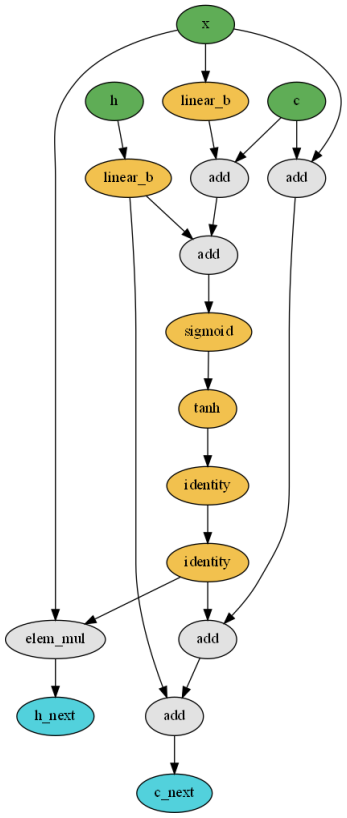
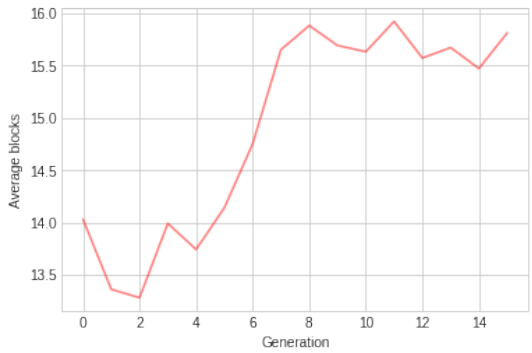
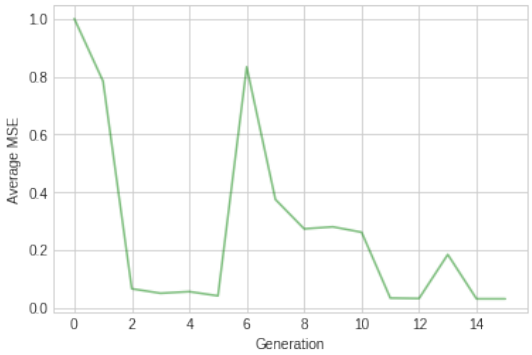


Figure 6.22: The rdm82_28 architecture evolved in scenario B1.



(a) Average number of blocks per generation for B2.



(b) Average MSE loss per generation for B2.

Figure 6.23: Average number of blocks and average MSE loss per generation for scenario B2.

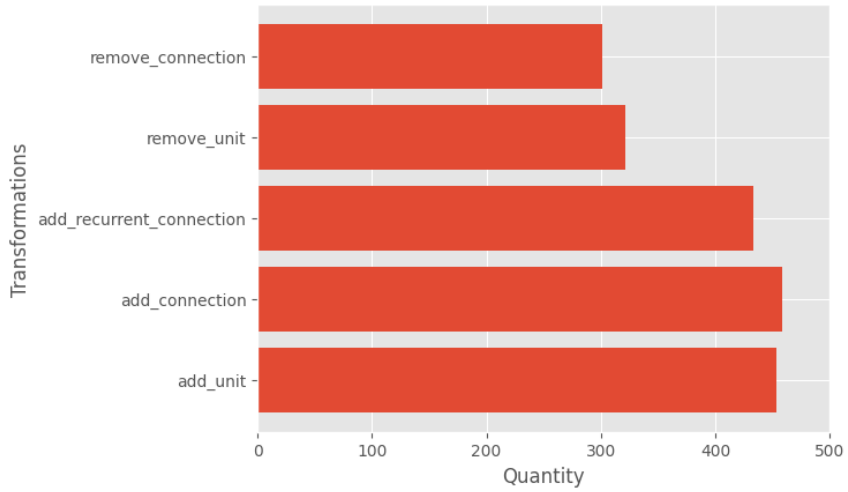


Figure 6.24: Total number of constructive and destructive network transformations that were performed during scenario B2.

multi-objective optimisation of the RNN architectures. In Section 6.2.1, RNN architecture optimisation exhibited a more favorable trend when a maximum of three consecutive network transformations were considered.

Table 6.14 lists the performances of the architectures in the Pareto front. The aforementioned observations regarding the effect of the high number of consecutive network transformations on the MOE/RNAS algorithm's multi-objective optimisation capabilities are confirmed in Table 6.14. The rdm28_0 architecture, which was randomly generated during the initialisation of the initial population, achieved the best MSE among all RNN architectures in the population. The other architectures in the Pareto front listed in Table 6.14 performed much worse in terms of the model accuracy objective, despite being offspring architectures. Therefore, when the evolution of the RNN architectures are too aggressive, the NAS method does not have any more contribution towards RNN architecture performance than what can be achieved with a random search in the RNN architecture search space.

Architecture	MSE loss	Blocks	Parameter	Training time (seconds)
LSTM_32	**0.000067**	27	133 252	261.053
GRU_45	0.000124	24	100 228	118.556
LSTM_0	0.000157	26	133 252	700.270
rdm77_0	0.001957	11	34 180	96.643
rdm46_4	0.019962	10	34 180	80.566
rdm12_15	0.064543	**9**	34 180	52.817

Table 6.12: Scenario B2 Pareto front architecture performances.

Scenario B4

For scenario B4, a population size of 100 was considered, and 100 offspring architectures were created for each generation. A maximum number of three consecutive transformations were considered during network morphism. The rest of the MOE/RNAS algorithm implementation configuration for scenario B4 is listed in Table 6.15.

The first observation that can immediately be made from Figure 6.28 is the favourable trend in terms of the average number of blocks per generation, as well as the average MSE per generation across the 15 generations. Furthermore, the MOE/RNAS algorithm was able to maintain a consistent decrease in the average number of blocks per generation while simultaneously optimising the model accuracy objective. Thus, the number of consecutive network transformations considered during network morphism has a clear contribution towards the multi-objective optimisation of the RNN architectures.

After 15 generations, a total number of 3 844 constructive network transformations were performed and a total number of 1 059 destructive network transformations were performed, as seen in Figure 6.29. The total search time for B4 was 42.67 hours, or 1.78 GPU days, which is higher compared to the search costs observed in scenarios B1-B3 due to the 100 offspring architectures that were created for each generation in B4.

Table 6.16 lists the performances of the architectures in the Pareto front. Apart from the BASIC_0 architecture, all other architectures in the Pareto front listed in Table 6.16 are offspring architectures. Thus, the MOE/RNAS algorithm is clearly capable of outperforming a random search method when the appropriate configuration is consid-

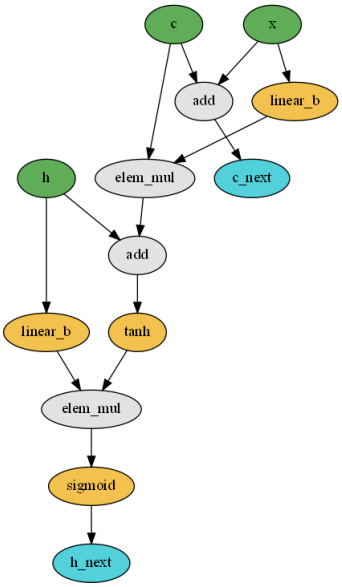


Figure 6.25: The rdm44_6 architecture evolved in scenario B2.

ered, such as the number of consecutive network transformations allowed during network morphism.

Although the model accuracies obtained during B4 were not as good as those observed in scenarios B1-B3, B4 has provided insight into the MOE/RNAS algorithm's multi-objective RNN architecture optimisation capabilities.

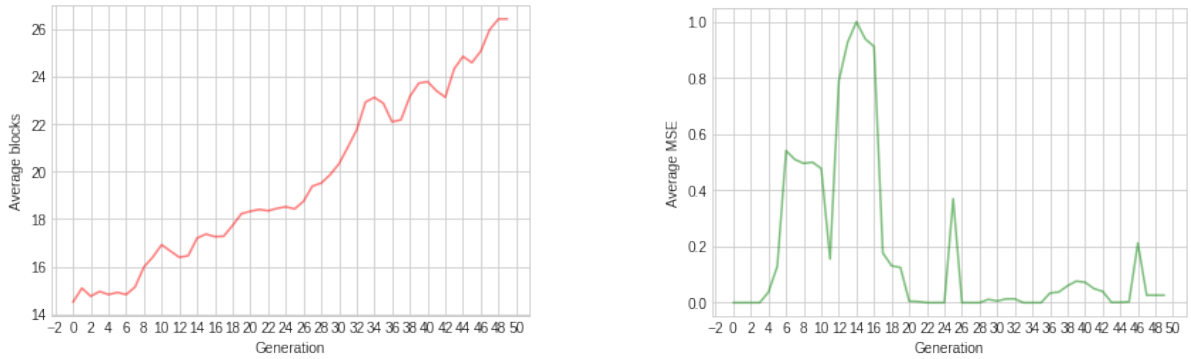
Summary

The MOE/RNAS algorithm was able to find and evolve a novel RNN architecture that outperformed the LSTM architecture in terms of both model accuracy and architecture complexity related objectives. The MOE/RNAS algorithm performed better compared to the approach presented by Bayer *et al.* [6], since the MOE/RNAS algorithm was able to optimise an RNN architecture complexity-related objective while maintaining a good model accuracy objective for the particular dataset. Additionally, some interesting evolutionary behaviour was observed, where the MOE/RNAS algorithm effectively evolved substitutes for the hidden state component of the RNN architecture to maintain some form of memory in the architecture.

From scenarios B1-B3 it was observed that the number of consecutive network trans-

Parameter	Value
Population size	100
Offspring generation	40
Generations	50
Maximum transformations	10
Alternative crowding distance	True
Seed architectures	RNN
Objectives	Number of blocks, MSE loss

Table 6.13: Scenario B3 MOE/RNAS algorithm implementation configuration.



(a) Average number of blocks per generation for B3.

(b) Average MSE loss per generation for B3.

Figure 6.26: Average number of blocks and average MSE loss per generation for scenario B3.

Architecture	MSE loss	Blocks	Parameter	Training time (seconds)
rdm28_0	**0.000049**	**13**	34 180	77.483
rdm47_264	0.056667	29	83 460	126.185
rdm50_360	0.064212	22	83 716	105.520
rdm50_348	0.068592	20	99 972	86.510
rdm47_266	0.093367	16	67 076	46.015

Table 6.14: Scenario B3 Pareto front architecture performances.

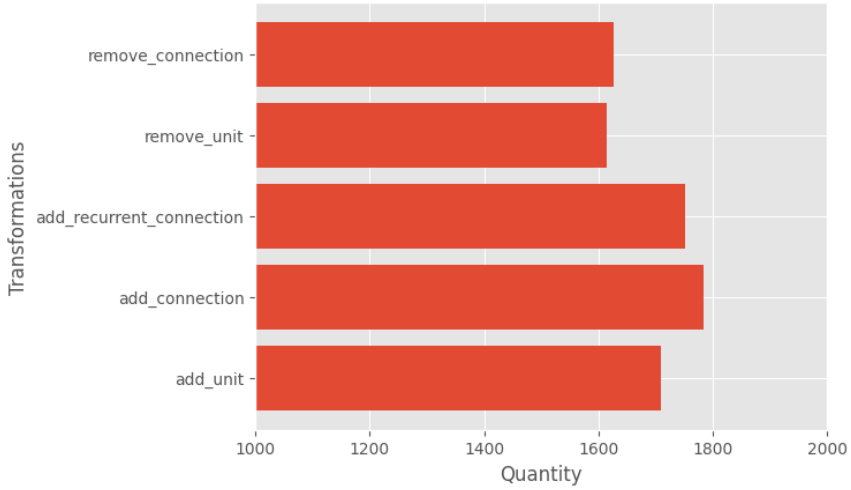


Figure 6.27: Total number of constructive and destructive network transformations that were performed during scenario B3.

formations considered during network morphism has a significant contribution towards the MOE/RNAS algorithm's ability to optimise multiple RNN architecture objectives. When the maximum number of consecutive network transformations considered are too high, the RNN architectures created by the MOE/RNAS algorithm do not outperform those created through random search of the RNN architecture search space.

In scenario B4, the MOE/RNAS algorithm was implemented with a configuration similar to that used in scenario A2 for the word-level language modeling task. Compared to the multi-objective optimisation results from A2, B4 achieved a similar consistency in terms of the optimisation of both the RNN architecture complexity-related objective and the model accuracy objective.

6.3 Summary

This chapter presented the empirical study done to evaluate the effectiveness of the proposed MOE/RNAS algorithm for multi-objective evolutionary algorithm based RNN architecture search. The experimental procedure was described, and the results obtained from implementing the proposed MOE/RNAS algorithm to search for RNN architectures were discussed.

Parameter	Value
Population size	100
Offspring generation	100
Generations	15
Maximum transformations	3
Alternative crowding distance	True
Seed architectures	RNN
Objectives	Number of blocks, MSE loss

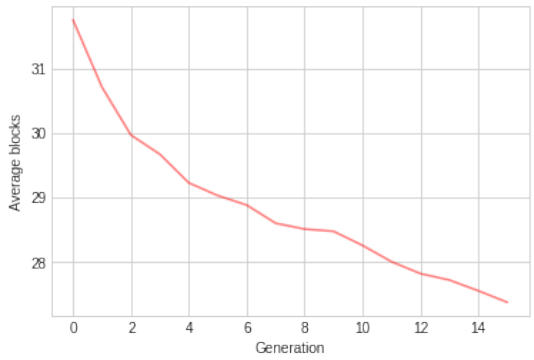
Table 6.15: Scenario B4 MOE/RNAS algorithm implementation configuration.

The experimental results obtained showed that the MOE/RNAS algorithm was able to automatically construct novel RNN architectures that can learn from the provided dataset. Additionally, it was observed that the MOE/RNAS algorithm is fully capable of optimising RNN architecture complexity related objectives, and when a reasonable trade-off is accepted between model accuracy and the computational resources demanded by the model, the MOE/RNAS algorithm can evolve computationally efficient RNN architectures that achieve reasonably good model accuracy.

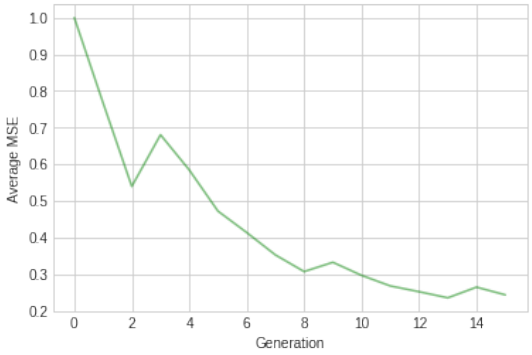
The MOE/RNAS algorithm was unable to find and evolve a novel RNN architecture that outperformed the LSTM architecture in terms of test perplexity on the Penn Treebank dataset, which reflects the observations from [32, 38, 42]. However, RNN architectures that achieved comparable perplexity, but with lower computational cost, were discovered.

For the character-level NLP task considered in Section 6.2.2, the MOE/RNAS algorithm was able to find and evolve RNN architectures that outperformed the LSTM in terms of model accuracy and architecture complexity objectives.

In Section 6.2.2 it was observed that higher numbers of consecutive network transformations during network morphism negatively impact the MOE/RNAS algorithm's ability to optimise RNN architectures. This was confirmed by scenario B4 in Section 6.2.2 along with the multi-objective optimisation trends observed in Section 6.2.1, where a maximum number of three consecutive network transformations was considered. Therefore, lower numbers of consecutive network transformations result in a more consistent



(a) Average number of blocks per generation for B4.



(b) Average MSE loss per generation for B4.

Figure 6.28: Average number of blocks and average MSE loss per generation for scenario B4.

generational optimisation of the multiple objectives considered.

Although the MOE/RNAS algorithm implements techniques that make the RNN architecture search method more efficient, the search costs observed during experimentation were still relatively high compared to other RNN NAS studies.

The next chapter presents the conclusions of this study and provides potential topics for future research.

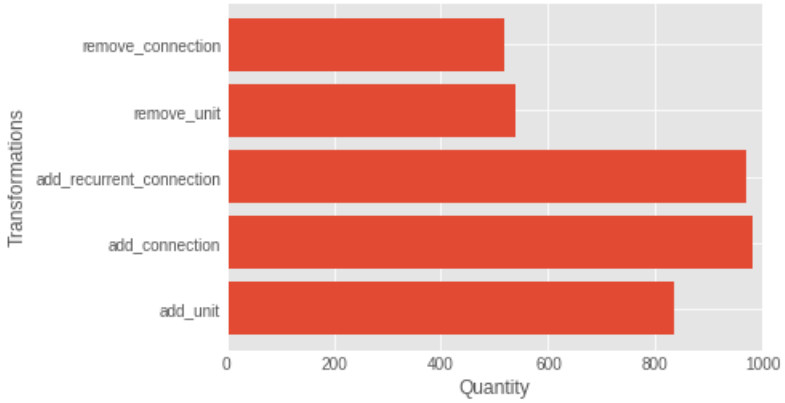


Figure 6.29: Total number of constructive and destructive network transformations that were performed during scenario B4.

Architecture	MSE loss	Blocks	Parameter	Training time (seconds)
rdm32_45	**0.00115**	18	50 692	73.775
rdm32_32	0.00272	17	50 692	68.944
rdm32_26	0.03956	16	34 180	63.643
rdm72_41	0.04896	14	50 564	77.642
rdm54_29	0.06305	13	50 692	53.498
BASIC_20	0.07508	12	50 564	36.200
BASIC_0	0.13433	**10**	34 180	71.210

Table 6.16: Scenario B4 Pareto front architecture performances.

Chapter 7

Conclusions

This chapter concludes the main part of the dissertation. The findings of the dissertation are summarised in Section 7.1, and directions for future work are discussed in Section 7.2.

7.1 Summary of Conclusions

The main objective of this work was to develop a novel multi-objective EA-based method for automated RNN architecture search. This was accomplished as follows:

The first sub-objective of the dissertation was to provide an overview of NNs and EAs that were used in this study. Chapter 2 discussed the fundamentals of NNs and the importance of NN architecture design. RNN architectures were discussed, and it was shown that the recursive nature of the RNN architecture makes them more complicated and difficult to train compared to feed-forward NN architectures. Chapter 3 discussed the EAs that were used in this study, and how EAs can be used for multi-objective optimisation.

The second sub-objective of the dissertation was to provide an overview of existing NAS methods. NAS for automated NN architecture design was discussed in Chapter 4, which included a review of existing EA-based NAS methods. It was found that the use of multi-objective EAs has not been extensively studied for RNN architecture search, specifically within the NAS paradigm where architecture complexity objectives were considered. Some complexities related to performing crossover on RNN architecture in-

dividual representations in GA based recombination were observed from existing studies, and network morphism was identified as an alternative method to generate offspring RNN architectures. Network morphism approaches that exclusively consider constructive network transformations are unable to optimise architecture complexity related objectives, since the architectures can only grow in size. Destructive network transformations were successfully implemented in existing EA-based NAS methods to optimise CNN architecture complexity objectives. Furthermore, Chapter 4 discussed existing NAS methods for RNN architecture search as well as the techniques that exist to make NAS more efficient, which addressed the third and fourth sub-objectives, respectively.

The fifth sub-objective of this study was to propose a modular RNN architecture search space and encoding scheme. In Chapter 5, a template-driven RNN architecture cell-based search space was proposed, along with a modular block-based RNN architecture encoding scheme. The sixth sub-objective of the dissertation was to propose a multi-objective EA-based search method to explore the proposed RNN architecture search space. Chapter 5 proposed the MOE/RNAS algorithm: a multi-objective EA-based NAS method for RNN architecture search. The fundamental search strategy that is employed by the proposed MOE/RNAS algorithm is largely based on the NSGA-II algorithm. Furthermore, the proposed MOE/RNAS algorithm includes appropriate network transformations, which allow for the optimisation of RNN architecture complexity related objectives during RNN architecture evolution. The seventh sub-objective of this work was to investigate techniques that can be implemented to make the MOE/RNAS algorithm more efficient. The MOE/RNAS algorithm employs techniques such as parameter sharing and early stopping to make the RNN model accuracy evaluation stage more efficient.

The eighth sub-objective of the dissertation was to implement the MOE/RNAS algorithm to search for RNN architectures for NLP datasets. An empirical study was conducted in Chapter 6 to evaluate the effectiveness of the proposed MOE/RNAS algorithm for RNN architecture search, and the findings can be summarised as follows:

1. The MOE/RNAS algorithm is able to find RNN architectures that outperform hand-crafted RNN architectures such as the LSTM and GRU, when architecture performance is based on multiple objectives.

2. The MOE/RNAS algorithm was unable to find novel RNN architectures that outperform the LSTM in terms of test perplexity achieved after being trained on a standard word-level language modeling task that is based on the Penn Treebank dataset.

3. For a character-level language modeling task, the MOE/RNAS algorithm was able to find novel RNN architectures that outperform the LSTM in terms of architecture complexity and model accuracy objectives.

4. Destructive network transformations are successful in optimising RNN architecture complexity related objectives, and a consistent improvement was observed during generational RNN architecture evolution.

5. The maximum number of consecutive network transformations allowed during network morphism has a clear impact on the MOE/RNAS algorithm's ability to optimise RNN architectures. When the maximum number of consecutive network transformations allowed during network morphism is too high, RNN architecture evolution is too aggressive, which reduces the model accuracies achieved by offspring architectures.

6. Parameter sharing and early stopping techniques improved the efficiency of the model accuracy evaluation stage.

7. The computational resource demand of the proposed MOE/RNAS algorithm is relatively high compared to existing NAS methods. This is attributed to the randomness of the network transformations performed on the offspring architectures, which leads to a significant model accuracy difference between the offspring and parent models. When the difference between the offspring and parent model accuracy is too high, training the offspring model for a reduced number of epochs can no longer be justified.

Overall, the MOE/RNAS algorithm developed in this study deepens the understanding of evolving RNN architectures using network morphism, and contributes towards making progress on fundamental questions related to RNN architecture complexity optimisation in an architecture search paradigm.

7.2 Future Work

This study considered a method for RNN architecture search that is exclusively driven by a multi-objective EA. The use of EAs for RNN architecture search in existing studies is scarce, and there are multiple research directions for future work that can build on the research done in this dissertation.

The obvious avenue to pursue that relates to the work done in this dissertation is to research a hybrid approach that employs an EA-based search strategy with a more complex network morphism stage, where the effects of previous network transformations are considered when selecting network transformations for future offspring generation. This can be done by adapting existing reinforcement learning NAS methods such that the RL agent is used for generating a sequence of network transformations instead, using the EA for maintaining a population of fit architectures based on the defined multiple objectives.

The problems considered in this study were limited to language modeling tasks. It would be interesting to perform an analysis on the effectiveness of the proposed NAS method on other machine learning tasks with sequential datasets that are unrelated to the natural language processing domain.

This study avoided model accuracy prediction techniques and instead relied on existing techniques to make model accuracy evaluation more efficient. Future work may examine the use of performance prediction techniques, such as learning curve extrapolation [4], to further improve the efficiency of the model accuracy evaluation stages.

The selection strategy implemented by the EA of the proposed NAS method was taken from the NSGA-II algorithm's selection strategy. Future work may examine the use of a density estimator to reduce the number of architectures selected for training, as observed in [24].

The network morphism approach implemented by the proposed NAS method in this study considered network transformations that make low-level changes to the RNN architecture, such as adding and removing units in the architecture. Future work may consider network transformations that could introduce entire gate units instead, such as the gate units of the LSTM and GRU architectures. This can be done by defining a gate unit template and considering the particular gate unit as a cell. The search space can

then be explored on a cell-based gate unit level as opposed to the low-level search that was done in this study.

In general, the research presented in this dissertation indicated that multi-objective EA-based RNN architecture search is able to optimise RNN architectures for multiple objectives. However, the proposed method was unable to find and optimise a novel RNN architecture that outperforms the LSTM architecture in terms of test perplexity on the Penn Treebank dataset. Future work that attempts multi-objective EA-based RNN architecture search can therefore address the shortcomings related to word-level language modeling tasks, such that the particular method can be compared to current state-of-the-art RNN architecture search methods.

Bibliography

[1] Harith Al-Sahaf, Ying Bi, Qi Chen, Andrew Lensen, Yi Mei, Yanan Sun, Binh Tran, Bing Xue, and Mengjie Zhang. A survey on evolutionary machine learning. In *Journal of the Royal Society of New Zealand*, volume 49, pages 205–228. Taylor & Francis, 2019.

[2] Abdulaziz Almalaq and Jun Jason Zhang. Evolutionary Deep Learning-Based Energy Consumption Prediction for Buildings. *IEEE Access*, 7:1520–1531, 2019.

[3] P.J. Angeline, G.M. Saunders, and J.B. Pollack. An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, 5(1):54–65, 1994.

[4] Bowen Baker, Otkrist Gupta, Ramesh Raskar, and Nikhil Naik. Accelerating neural architecture search using performance prediction. *arXiv preprint arXiv:1705.10823*, 2018.

[5] Imon Banerjee, Yuan Ling, Matthew C. Chen, Sadid A. Hasan, Curtis P. Langlotz, Nathaniel Moradzadeh, Brian Chapman, Timothy Amrhein, David Mong, Daniel L. Rubin, Oladimeji Farri, and Matthew P. Lungren. Comparative effectiveness of convolutional neural network (CNN) and recurrent neural network (RNN) architectures for radiology text report classification. *Artificial Intelligence in Medicine*, 97(August):79–88, 2019.

[6] Justin Bayer, Daan Wierstra, Julian Togelius, and Jürgen Schmidhuber. Evolving Memory Cell Structures for Sequence Learning. In *Artificial Neural Networks - ICANN 2009*, pages 755–764. Springer, Berlin, Heidelberg, 2009.

[7] Yoshua Bengio. Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, 2(1):1–127, 2009.

[8] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

[9] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Advanced Texts in Econometrics. Oxford University Press, Inc., USA, 1995.

[10] Han Cai, Tianyao Chen, Weinan Zhang, Yong Yu, and Jun Wang. Efficient Architecture Search by Network Transformation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, pages 2787–2794. AAAI Press, 2018.

[11] Guillermo Campos Ciro, Frédéric Dugardin, Farouk Yalaoui, and Russell Kelly. A NSGA-II and NSGA-III comparison for solving an open shop scheduling problem with resource constraints. *IFAC-PapersOnLine*, 49(12):1272–1277, 2016.

[12] Gang Chen. A Gentle Tutorial of Recurrent Neural Network with Error Backpropagation. *arXiv preprint arXiv:1610.02583*, 2016.

[13] Zewei Chen, Fengwei Zhou, George Trimponias, and Zhenguo Li. Multi-objective Neural Architecture Search via Non-stationary Policy Gradient. *arXiv preprint arXiv:2001.08437*, 2020.

[14] Hanen Chihi and Najet Arous. Recurrent neural networks design by means of multi-objective genetic algorithm. *IJCSI International Journal of Computer Science Issues*, 8(3):296–302, 2011.

[15] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Proceedings of the Empirical Methods in Natural Language Processing*, pages 1724–1734. ACL, 2014.

[16] Xiangxiang Chu and Xinjie Yu. Improved Crowding Distance for NSGA-II. *arXiv preprint arXiv:1811.12667*, 2018.

[17] Xiangxiang Chu, Bo Zhang, Ruijun Xu, and Hailong Ma. Multi-objective reinforced evolution in mobile neural architecture search. In *Proceedings of the European Conference on Computer Vision*, pages 99–113, Cham, 2020. Springer.

[18] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. In *Proceedings of the NIPS 2014 Workshop on Deep Learning, December 2014*, pages 1–9, 2014.

[19] Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. Gated feedback recurrent neural networks. In *Proceedings of the 32nd International Conference on Machine Learning*, volume 37, pages 2067–2075. PMLR, 2015.

[20] Kalyanmoy Deb and Himanshu Jain. An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point-Based Nondominated Sorting Approach, Part I: Solving Problems With Box Constraints. *IEEE Transactions on Evolutionary Computation*, 18(4):577–601, 2014.

[21] Kalyanmoy Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[22] Aguston Eiben and Marc Schoenauer. Evolutionary computing. *Information Processing Letters*, 82(1):1–6, 2002.

[23] Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990.

[24] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Efficient multi-objective neural architecture search via lamarckian evolution. *arXiv preprint arXiv:1804.09081*, 2018.

[25] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *The Journal of Machine Learning Research*, 20(1):1997–2017, 2019.

[26] Andries P. Engelbrecht. *Computational Intelligence: An Introduction*. Wiley Publishing, 2 edition, 2007.

[27] Stuart Geman, Elie Bienenstock, and René Doursat. Neural Networks and the Bias/Variance Dilemma. *Neural Computation*, 4(1):1–58, 1992.

[28] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT press, 2018.

[29] Alex Graves, Abdel Rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *Proceedings of the 2013 IEEE international conference on acoustics, speech and signal processing*, number 3, pages 6645–6649. IEEE, 2013.

[30] Frauke Günther and Stefan Fritsch. Neuralnet: Training of neural networks. *R Journal*, 2(1):30–38, 2010.

[31] Eldad Haber and Lars Ruthotto. Stable architectures for deep neural networks. *Inverse Problems*, 34(1):1–23, 2018.

[32] Xin He, Kaiyong Zhao, and Xiaowen Chu. AutoML: A survey of the state-of-the-art. *Knowledge-Based Systems*, 212(Dl), 2021.

[33] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 1997.

[34] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[35] Shengran Hu, Ran Cheng, Cheng He, and Zhichao Lu. Multi-objective neural architecture search with almost no training. *Evolutionary Multi-Criterion Optimization*, 2020.

[36] Michael I. Jordan. Serial order: a parallel distributed processing approach. *Advances in Psychology*, 121(C):471–495, 1986.

[37] Rafal Jozefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the Limits of Language Modeling. *arXiv preprint arXiv:1602.02410*, 2016.

[38] Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. An Empirical Exploration of Recurrent Network Architectures. In *Proceedings of the 32nd International Conference on Machine Learning*, volume 37, pages 2332–2340, Lille, France, 2015. JMLR.org.

[39] Daniel Jurafsky and James Martin. *Speech and Language Processing*. Prentice-Hall, Inc., 2008.

[40] Andrej Karpathy, Justin Johnson, and Li Fei-Fei. Visualizing and Understanding Recurrent Networks. *arXiv preprint arXiv:1506.02078*, 2015.

[41] Youngkee Kim, Won Joon Yun, Youn Kyu Lee, Soyi Jung, and Joongheon Kim. Trends in Neural Architecture Search: Towards the Acceleration of Search. In *Proceedings of the 2021 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 421–424. IEEE, oct 2021.

[42] Nikita Klyuchnikov, Ilya Trofimov, Ekaterina Artemova, Mikhail Salnikov, Maxim Fedorov, Alexander Filippov, and Evgeny Burnaev. NAS-Bench-NLP: Neural Architecture Search Benchmark for Natural Language Processing. *IEEE Access*, 10:45736–45747, 2022.

[43] Weicong Kong, Zhao Yang Dong, Youwei Jia, David J. Hill, Yan Xu, and Yuan Zhang. Short-Term Residential Load Forecasting Based on LSTM Recurrent Neural Network. *IEEE Transactions on Smart Grid*, 10(1):841–851, 2019.

[44] Richard E Korf. Artificial Intelligence Search Algorithms. *Algorithms and Theory of Computation Handbook*, pages 1–40, 1996.

[45] Tao Lei, Yu Zhang, Sida I. Wang, Hui Dai, and Yoav Artzi. Simple recurrent units for highly parallelizable recurrence. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 4470–4481, Brussels, Belgium, 2020. Association for Computational Linguistics.

[46] Liam Li and Ameet Talwalkar. Random search and reproducibility for neural architecture search. *Uncertainty in artificial intelligence*, pages 367–377, 2020.

[47] You Li, Yingxin Kou, and Zhanwu Li. An Improved Nondominated Sorting Genetic Algorithm III Method for Solving Multiobjective Weapon-Target Assignment Part I: The Value of Fighter Combat. *International Journal of Aerospace Engineering*, 2018:1–23, 2018.

[48] Marius Lindauer and Frank Hutter. Best practices for scientific research on neural architecture search. *Journal of Machine Learning Research*, 21:1–18, 2020.

[49] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

[50] Yuqiao Liu, Yanan Sun, Bing Xue, Mengjie Zhang, Gary G. Yen, and Kay Chen Tan. A Survey on Evolutionary Neural Architecture Search. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–21, 2021.

[51] Zhichao Lu, Kalyanmoy Deb, Erik Goodman, Wolfgang Banzhaf, and Vishnu Naresh Boddeti. NSGANetV2: Evolutionary Multi-objective Surrogate-Assisted Neural Architecture Search. In *Proceedings of the European Conference on Computer Vision*, pages 35–51, Cham, 2020. Springer.

[52] Zhichao Lu, Ian Whalen, Yashesh Dhebar, Kalyanmoy Deb, Erik Goodman, Wolfgang Banzhaf, and Vishnu Naresh Boddeti. NSGA-Net: Neural architecture search using multi-objective genetic algorithm. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, pages 4750–4754. International Joint Conferences on Artificial Intelligence Organization, 2020.

[53] Zhichao Lu, Ian Whalen, Yashesh Dhebar, Kalyanmoy Deb, Erik D. Goodman, Wolfgang Banzhaf, and Vishnu Naresh Boddeti. Multiobjective Evolutionary Design of Deep Convolutional Neural Networks for Image Classification. *IEEE Transactions on Evolutionary Computation*, 25(2):277–291, 2021.

[54] Danilo P Mandic and Jonathon A Chambers. *Recurrent Neural Networks for Prediction*, volume 4 of *Wiley Series in Adaptive and Learning Systems for Signal*

Processing, Communications, and Control. John Wiley & Sons, Ltd, Chichester, UK, 2001.

[55] Mitchell Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of English: The Penn Treebank. *Computational linguistics*, 19:313, 1993.

[56] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, 1943.

[57] L. R. Medsker and L. C. Jain. *Recurrent Neural Networks: Design and Applications*. CRC Press, 1st edition, 2001.

[58] Oren Melamud, Jacob Goldberger, and Ido Dagan. context2vec: Learning generic context embedding with bidirectional LSTM. In *Proceedings of the 20th SIGNLL conference on computational natural language learning*, pages 51–61, 2016.

[59] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. Regularizing and optimizing LSTM language models. *arXiv preprint arXiv:1708.02182*, 2018.

[60] Risto Miikkulainen, Jason Liang, Elliot Meyerson, Aditya Rawal, Daniel Fink, Olivier Francon, Bala Raju, Hormoz Shahrzad, Arshak Navruzyan, Nigel Duffy, and Babak Hodjat. Evolving deep neural networks. In *Artificial Intelligence in the Age of Neural Networks and Brain Computing*, pages 293–312. Elsevier, Amsterdam, 2018.

[61] Tomaš Mikolov, Martin Karafiát, Lukaš Burget, Cernocky Jan, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Proceedings of the 11th Annual Conference of the International Speech Communication Association, Interspeech 2010*, number September, pages 1045–1048, 2010.

[62] Melanie Mitchell. *An introduction to genetic algorithms*. MIT press, 1998.

[63] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[64] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation Functions: Comparison of trends in Practice and Research for Deep Learning. *arXiv preprint arXiv:1811.03378*, 2018.

[65] Patxi Ortego, Alberto Diez-Olivan, Javier Del Ser, Fernando Veiga, Mariluz Penalva, and Basilio Sierra. Evolutionary LSTM-FCN networks for pattern classification in industrial processes. *Swarm and Evolutionary Computation*, 54:100650, 2020.

[66] Kang-moon Park, Donghoon Shin, and Yongsuk Yoo. Evolutionary Neural Architecture Search (NAS) Using Chromosome Non-Disjunction for Korean Grammaticality Tasks. *Applied Sciences*, 10(10):3457, 2020.

[67] Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. How to construct deep recurrent neural networks. In *Proceedings of the Second International Conference on Learning Representations (ICLR 2014)*, 2014.

[68] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *Proceedings of the International conference on machine learning*, pages 1310–1318. PMLR, 2013.

[69] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 2019.

[70] Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. Efficient Neural Architecture Search via Parameters Sharing. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80, pages 4095–4104. PMLR, 2018.

[71] Qingfu Zhang and Hui Li. MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition. *IEEE Transactions on Evolutionary Computation*, 11(6):712–731, 2007.

[72] Pengzhen Ren, Yun Xiao, Xiaojun Chang, Po-yao Huang, Zhihui Li, Xiaojiang Chen, and Xin Wang. A Comprehensive Survey of Neural Architecture Search. *ACM Computing Surveys*, 54(4):1–34, 2022.

[73] F Rosenblatt. *Principles of neurodynamics: perceptrons and the theory of brain mechanisms*. Spartan Books, Washington DC, 1962.

[74] Gunter Rudolph. On a multi-objective evolutionary algorithm and its convergence to the Pareto set. In *Proceedings of the IEEE Conference on Evolutionary Computation, ICEC*, pages 511–516. IEEE, 1998.

[75] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. *Learning Internal Representations by Error Propagation*. 1985.

[76] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, USA, 3rd edition, 2009.

[77] SanYou Zeng, LiXin Ding, LiShan Kang, and Yuping Chen. A new multiobjective evolutionary algorithm: OMOEA. *Congress on Evolutionary Computation, 2003. CEC '03*, 2:898–905, 2003.

[78] Siddharth Sharma, Simone Sharma, and Athaiya Anidhya. Activation Functions in Neural Networks. *International Journal of Engineering Applied Sciences and Technology*, 4(12):310–316, 2020.

[79] Xian Shi, Pan Zhou, Wei Chen, and Lei Xie. Darts-Conformer: Towards Efficient Gradient-Based Neural Architecture Search For End-to-End ASR. *arXiv preprint arXiv:2104.02868*, 2021.

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

[81] J E Smith and A.E. Eiben. *Introduction to Evolutionary Computing*. Springer Publishing Company Inc., 2nd edition, 2015.

[82] N. Srinivas and Kalyanmoy Deb. Muiltiobjective optimization using nondominated sorting in genetic algorithms. *Evolutionary Computation*, 2(3):221–248, 1995.

[83] Mirac Suzgun, Yonatan Belinkov, and Stuart M. Shieber. On evaluating the generalization of LSTM models in formal languages. In *Proceedings of the Society for Computation in Linguistics*, volume 2, pages 277–286, 2019.

[84] Chunnan Wang, Hongzhi Wang, Guosheng Feng, and Fei Geng. Multi-Objective Neural Architecture Search Based on Diverse Structures and Adaptive Recommendation. *arXiv preprint arXiv:2007.02749*, 2020.

[85] Zi Wang, Aws Albarghouthi, Gautam Prakriya, and Somesh Jha. Interval Universal Approximation for Neural Networks. In *Proceedings of the ACM on Programming Languages*, volume 6, pages 1–29. ACM New York, NY, USA, 2022.

[86] Colin White, Willie Neiswanger, Sam Nolen, and Yash Savani. A Study on Encodings for Neural Architecture Search. *arXiv preprint arXiv:2007.04965*, 2020.

[87] Colin White, Arber Zela, Binxin Ru, Yang Liu, and Frank Hutter. How Powerful are Performance Predictors in Neural Architecture Search? *arXiv preprint arXiv:2104.01177*, 2021.

[88] Martin Wistuba, Ambrish Rawat, and Tejaswini Pedapati. A Survey on Neural Architecture Search. *arXiv preprint arXiv:1905.01392*, 2019.

[89] Antoine Yang, Pedro M Esperanca, and Fabio Maria Carlucci. NAS evaluation is frustratingly hard. *arXiv preprint arXiv:1912.12522*, 2019.

[90] Zhaohui Yang, Yunhe Wang, Xinghao Chen, Boxin Shi, Chao Xu, Chunjing Xu, Qi Tian, and Chang Xu. Cars: Continuous evolution for efficient neural architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1829–1838, 2019.

[91] Chris Ying, Aaron Klein, Esteban Real, Eric Christiansen, Kevin Murphy, and Frank Hutter. NAS-BENCH-101: Towards reproducible neural architecture search. In *Proceedings of the International Conference on Machine Learning*, pages 7105–7114. PMLR, 2019.

[92] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent Neural Network Regularization. *arXiv preprint arXiv:1409.2329*, 2014.

[93] Aimin Zhou, Bo Yang Qu, Hui Li, Shi Zheng Zhao, Ponnuthurai Nagaratnam Suganthan, and Qingfu Zhangd. Multiobjective evolutionary algorithms: A survey of the state of the art. *Swarm and Evolutionary Computation*, 1(1):32–49, 2011.

[94] Yanqi Zhou and Gregory Diamos. Neural Architect: A Multi-objective Neural Architecture Search with Performance Prediction. *MLSys 2019*, pages 1–3, 2019.

[95] Eckart Zitzler, Kalyanmoy Deb, and Lothar Thiele. Comparison of Multiobjective Evolutionary Algorithms: Empirical Results. *Evolutionary Computation*, 8(2):173–195, 2000.

[96] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

Appendix A

Acronyms

BPTT	Back Propagation Through Time
CNN	Convolutional Neural Network
EA	Evolutionary Algorithm
EC	Evolutionary Computation
ELU	Exponential Linear Unit
GA	Genetic Algorithm
GRU	Gated Recurrent Unit
LReLU	Leaky Rectified Linear Unit
LSTM	Long Short-Term Memory
NAS	Neural Architecture Search
NLP	Natural Language Processing
NN	Artificial Neural Network
ReLU	Rectified Linear Unit
RL	Reinforcement Learning
RNN	Recurrent Neural Network
TanH	Hyperbolic Tangent
SOP	Scalar Objective Optimisation Problems
SRN	Simple Recurrent Neural Network

Appendix B

Symbols

This appendix lists the important symbols used throughout the thesis, as well as their corresponding meanings. The symbols are divided according to the chapter in which the symbols were first introduced.

B.1 Chapter 2: Artificial Neural Networks

$\mathbf{w}$	An artificial neuron's weight vector
x_n	The n^{th} input
n	Index of an input unit
N	Total number of examples in a labeled training dataset
net	Weighted sum of outputs
θ	The bias threshold
f	Activation function of a neuron
$\hat{y}$	The output of a neuron
M	Total number of hidden units
j	Index of a hidden unit
a_j	The weighted sum of the j^{th} hidden unit's inputs
g	Activation function of a hidden unit
z_j	Output of the j^{th} hidden unit

K	Total number of output units
k	Index of an output unit
$\widetilde{g}$	Activation function of an output unit
m	number of input patterns to consider for error value aggregation
E^m	NN error per m number of input patterns
w_{kj}	Weight between the k^{th} and j^{th} unit
Δw_{kj}	Update of the weight value between the k^{th} and j^{th} unit
υ	Learning rate
T	Total time steps in an input sequence
t	Time step
$\mathbf{h}_t$	RNN hidden state at time step t
$\mathbf{W}$	RNN input weight matrix
$\mathbf{U}$	RNN hidden state weight matrix
$\mathbf{b}$	RNN bias threshold
$\cdot$	Element-wise multiplication of matrices
D_G	Testing dataset
Q	Index of an input pattern in the testing dataset
d_Q	The Q^{th} input pattern in the testing dataset

B.2 Chapter 3: Evolutionary Algorithms

n_x	The dimensional search space of individuals in a population
$\mathbf{b}$	Binary-valued individual in the population
f	Fitness function
Γ	Data type of the elements of an individual's representation
$\mathbb{R}$	Set of all real numbers
n_s	Population size
n_{ts}	Population size of individuals selected through tournament selection
F	Vector-valued objective function

R^o	Parameter space
R^n	Objective space
$\mathbf{x}$	Decision vector
$\mathbf{y}$	Objective vector
$\mathbf{a}$	Individual in population
$\mathbf{b}$	Individual in population
$\prec$	Domination symbol
$\preceq$	Non-domination symbol
m	Total number of objectives used to represent individual fitness
k	An objective
j	An individual's rank
dis_j	Crowding distance of the j^{th} ranked individual
f_n^k	The k^{th} objective value for the n^{th} individual.
P	Population of candidate solutions
n_p	The number of solutions which dominate the solution p
S_p	A set of solutions that the p solution dominates
F_i	The i^{th} Pareto-front

B.3 Chapter 4: Neural Architecture Search

f	NN architecture performance measure
A	Set of NN architectures
$f(a)$	The performance measure of architecture a
t	Time step
g	A recurrent cell
h_t	RNN hidden state at time step t
θ	RNN architecture
α	Trainable parameters of the RNN architecture
x_t	The input at time step t

c_t	The cell state at time step t
k	The number of cells that a final RNN architecture comprise

B.4 Chapter 5: Framework

t	Time step
g	A recurrent cell
h_t	RNN hidden state at time step t
θ	RNN architecture
α	Trainable parameters of the RNN architecture
x_t	The input at time step t
c_t	The cell state at time step t
f_h	RNN activation function
W	RNN input weight matrix
U	RNN hidden state weight matrix
b	RNN bias threshold
b_r	Randomly selected block from an RNN architecture
N	Population size
i	Generation counter
ϕ	Number of offspring to generate
Υ	Archive that contains previously evaluated RNN architectures
P	Parent population
Q	Offspring population
f	Multi-objective fitness values for provided candidate solutions
F_n	The n^{th} Pareto front
p	Selected parent solutions

B.5 Chapter 6: Empirical Analysis

t	Time step
h_t	RNN hidden state at time step t
x_t	The input at time step t
c_t	The cell state at time step t