Neural Processing Letters (2024) 56:200
https://doi.org/10.1007/s11063-024-11659-0

# Multi-objective Evolutionary Neural Architecture Search for Recurrent Neural Networks

**Reinhard Booysen**[1] · **Anna Sergeevna Bosman**[1]

Accepted: 26 May 2024 / Published online: 18 June 2024


**Abstract**
Artificial neural network (NN) 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 NN architectures and has proven to be successful in automatically finding NN architectures that outperform those manually designed by human experts. NN architecture performance can be quantified based on multiple objectives, which include model accuracy and some NN architecture complexity objectives, among others. The majority of modern NAS methods that consider multiple objectives for NN architecture performance evaluation are concerned with automated feed forward NN architecture design, which leaves multi-objective automated recurrent neural network (RNN) architecture design unexplored. RNNs are important for modeling sequential datasets, and prominent within the natural language processing domain. It is often the case in real world implementations of machine learning and NNs that a reasonable trade-off is accepted for marginally reduced model accuracy in favour of lower computational resources demanded by the model. This paper proposes a multi-objective evolutionary algorithm-based RNN architecture search method. The proposed method relies on approximate network morphisms for RNN architecture complexity optimisation during evolution. The results show that the proposed method is capable of finding novel RNN architectures with comparable performance to state-of-the-art manually designed RNN architectures, but with reduced computational demand.

**Keywords** Recurrent neural networks · Neural architecture search · Evolutionary algorithms

## 1 Introduction

Recurrent neural networks (RNNs) are a set of specialised neural network (NN) architectures that are designed specifically to learn from data with sequential or prominent temporal structures by simulating a discrete-time dynamical system [1–3]. RNNs have been successfully used in solving problems across multiple domains such as forecasting, natural language processing (NLP), load prediction, and more [4–7].

✉ Reinhard Booysen
r.booysen@tuks.co.za

Anna Sergeevna Bosman
anna.bosman@up.ac.za

[1] Department of Computer Science, University of Pretoria, Pretoria, South Africa

Designing a NN architecture for a specific problem is a nontrivial task and often requires a high level of human expertise [8–10]. A number of ways have been proposed to automate the task of NN architecture design, collectively referred to as neural architecture search (NAS) methods [10, 11]. 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 perform comparably to state-of-the-art NN architectures designed by human experts [11–13].

Different approaches to NAS exist for finding well-performing NN architectures, such as reinforcement learning (RL) methods [10], evolutionary algorithm (EA) methods [14], gradient-based methods [15], and more. Evaluating the performance of a particular NN architecture in NAS is typically based on the corresponding model accuracy [10, 15, 16]. However, a number of methods have been proposed that consider multiple objectives for evaluating NN architecture performance, which includes NN architecture complexity and the corresponding model computational resource demand [8, 17, 18], amongst others. NAS methods that consider multiple objectives for NN architecture performance evaluation typically employ some multi-objective optimisation techniques for finding the best performing NN architectures.

Multi-objective optimisation aims to solve a problem where a solution's effectiveness in reference to a number of objectives determines the solution's quality, where individual objectives may be competing with each other [19]. In general, for multi-objective problems, assuming the goal is to minimise the respective problems, a vector-valued objective function $F : R^d \rightarrow R^n$ is defined for $n$ objectives, where $n > 1$, and $d$ is the dimension of the decision vector $\mathbf{x}$ [20–22]. The aim is then to minimise the $\mathbf{y}$ objective vector such that:

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

When dealing with the optimisation of multiple objectives, the goal is to find the best possible compromise between the objectives [23]. Therefore, the advantage of using multi-objective optimisation over single-objective optimisation is that the multi-objective optimisation approach may produce a solution that is more favourable when a reasonable trade-off between the respective objectives is accepted, whereas a single-objective optimisation approach may produce a better quality solution for a single objective.

A multi-objective optimisation approach in NAS aims to optimise multiple objectives that relate to NN architectures, such as the model accuracy, the number of parameters of the model, model inference time, and more. Since multi-objective optimisation techniques aim to find a compromise between the defined objectives, the model accuracy achieved by a multi-objective NAS approach is likely to be worse compared to a NAS approach that considers a single model accuracy objective exclusively. However, NAS approaches that only consider model accuracy objectives disregard the computational resource demand of the models during the search for the optimal NN architecture. Therefore, the trade-off solutions found by the multi-objective NAS approach should produce models with some reasonable model accuracy (potentially worse compared to single-objective approaches), but with reduced computational resource demand.

The practical advantage of a multi-objective NAS approach is that compared to single-objective approaches, models with fewer parameters will be produced if the trade-off between model accuracy and model computational resource demand is acceptable. With RNN architecture search, the majority of the single-objective approaches [10, 24, 25] produced models with more than 23 M parameters, but none of these approaches were able to produce a model that could outperform a manually designed RNN architecture in terms of model accuracy [26, 27].

Network morphisms were shown to be a useful tool for evolutionary NAS approaches [11, 28]. Generating offspring NN architectures with network morphism is done through the use of network transformation operations, which make structural changes to a cloned parent NN architecture to generate an offspring NN architecture [28, 29]. These network transformations typically involve the addition of new units to the architecture and the addition of new connections between units in the NN architecture, which are referred to as constructive network transformations [11, 28]. Destructive network transformation operations, which were introduced by Elsken et al. [11], allow for the removal of units and removal of connections between units in the NN architecture, which effectively reduces the overall complexity of the NN architecture.

In this work, we propose 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 MOE/RNAS algorithm is specifically designed to be capable of optimising some model accuracy-related objectives, along with some RNN architecture complexity-related objectives. The MOE/RNAS algorithm differs from [30] in that the MOE/RNAS algorithm is capable of optimising an RNN architecture complexity-related objective with the use of approximate network morphisms.

Novel contributions of this study are summarised as follows:

- MOE/RNAS: a multi-objective EA-based NAS algorithm specifically designed for RNN architecture search is proposed.
- Approximate network morphism is implemented to optimise an RNN architecture complexity objective.
- A modular RNN architecture block encoding scheme is proposed that is fully capable of catering for destructive RNN network transformations.
- An empirical analysis of the MOE/RNAS algorithm's effectiveness to find RNN architectures for three different 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.

The rest of this paper is structured as follows: Sect. 2 provides an overview of the relevant background and related work. Section 3 presents the MOE/RNAS algorithm. Section 4 discusses the experiments and results. Section 5 concludes the paper.

## 2 Background and Related Work

This section provides an overview of the background and related work and is structured as follows: Sect. 2.1 provides a brief discussion of RNN architectures. An overview of EA-based multi-objective optimisation is provided in Sect. 2.2. Section 2.3 discusses the use of multi-objective EAs in NAS. Finally, existing RNN architecture search methods are discussed in Sect. 2.4.

### 2.1 Recurrent Neural Network Architecture

Recurrent neural networks (RNNs) are a set of specialised NN architectures that are designed specifically to learn from data with sequential or prominent temporal structures by simulating a discrete-time dynamical system [1–3]. 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 [2, 3], thereby allowing the RNN to have a form of memory [31].

RNNs face challenges with gradient-based training where input sequences with longer-term dependencies are used. In this case, during backward propagation, the gradient values will either grow exponentially, or go exponentially fast to zero (vanish), such that they become insignificant [32, 33]. In an attempt to address the RNN's vanishing gradient problem, Hochreiter and Schmidhuber [34] introduced a novel RNN architecture dubbed Long Short-Term Memory (LSTM). The LSTM deals with the vanishing gradient problem by employing memory cells and gate units [34], 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 [6].

The various gate units of the LSTM are defined by:

$$
\begin{aligned}
\mathbf{f}_t &= \sigma(\mathbf{W}_{xf}\mathbf{x}_t + \mathbf{W}_{hf}\mathbf{h}_{t-1} + \mathbf{b}_f),\\
\mathbf{i}_t &= \sigma(\mathbf{W}_{xi}\mathbf{x}_t + \mathbf{W}_{hi}\mathbf{h}_{t-1} + \mathbf{b}_i),\\
\mathbf{o}_t &= \sigma(\mathbf{W}_{xo}\mathbf{x}_t + \mathbf{W}_{ho}\mathbf{h}_{t-1} + \mathbf{b}_o),\\
\mathbf{g}_t &= tanh(\mathbf{W}_{xg}\mathbf{x}_t + \mathbf{W}_{hg}\mathbf{h}_{t-1} + \mathbf{b}_g),\\
\mathbf{c}_t &= \mathbf{f}_t \cdot \mathbf{c}_{t-1} + \mathbf{i}_t \cdot \mathbf{g}_t,\\
\mathbf{h}_t &= \mathbf{o}_t \cdot tanh(\mathbf{c}_t),
\end{aligned}
$$

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 [35]. $\mathbf{c}_t$ 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 [36, 37], thereby supplementing the hidden state $h_t$ memory that is implicit in the RNN architecture [34].

One notable alternative to the LSTM is the Gated Recurrent Unit (GRU) introduced by Cho et al. [38] in 2014. The premise of the GRU is that it allows the recurrent unit to capture the dependencies of different time scales [39]. The GRU employs the same gate-unit philosophy of the LSTM, and the GRU's gate units are defined by:

$$
\begin{aligned}
\mathbf{z}_t &= \sigma(\mathbf{W}_{xz}\mathbf{x}_t + \mathbf{W}_{hz}\mathbf{h}_{t-1} + \mathbf{b}_z),\\
\mathbf{r}_t &= \sigma(\mathbf{W}_{xr}\mathbf{x}_t + \mathbf{W}_{hr}\mathbf{h}_{t-1} + \mathbf{b}_r),\\
\mathbf{n}_t &= tanh(\mathbf{W}_{xn}\mathbf{x}_t + \mathbf{W}_n(\mathbf{r}_t \cdot \mathbf{h}_{t-1})),\\
\mathbf{h}_t &= \mathbf{z}_t \cdot \mathbf{h}_{t-1} + (1 - \mathbf{z}_t) \cdot \mathbf{n}_t.
\end{aligned}
$$

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 [36, 39].

### 2.2 Evolutionary Multi-Objective Optimisation

Multi-objective EAs aim to optimise multiple objectives, which are typically conflicting with each other [19, 40]. The optimisation of the objectives is done generationally in a survival-of-the-fittest fashion within the population-based paradigm [41, 42]. Candidate solutions are selected from the population based on their quality, i.e., fitness, and are then combined to produce new offspring solutions for the following generation [19].

Deb et al. [43] proposed a fast and elitist nondominated sorting algorithm, called NSGA-II, for evolutionary multi-objective optimisation. The NSGA-II sorts individuals based on their nondomination with respect to the multiple objectives by using a dominance operator [43]. 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 [19, 44]. Additionally, the NSGA-II uses a mechanism called crowding distance assignment to increase the diversity of the population [43].

The crowding distance of the NSGA-II represents the distance between candidate solutions, and is used as a density estimator to guide the algorithm towards a uniform population distribution [43, 45]. After the fittest individuals are selected, offspring are generated by the recombination process of the algorithm with crossover and mutation operators.

### 2.3 Multi-Objective Evolutionary NAS

Following the introduction of the RL-based NAS method by Zoph and Le [10] in 2017, a number of different approaches to NAS have since been proposed, which include EA-based NAS methods [11, 14, 46–48], amongst others. EA-based NAS methods refer to those NAS methods that employ EAs as their core search space exploration strategy, where the NN architecture search space is traversed in a population-based paradigm [12].

The majority of modern EA-based NAS studies focused on convolutional neural network (CNN) architecture search [11, 13, 49]. Aside from Bayer et al. [30], 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 [11].

Existing multi-objective EA-based NAS approaches that focus on CNN architecture search is not directly transferable to RNN architecture search, since they lack the ability to represent recurrent connections in the architectures that are explored during the evolutionary search. Additionally, the hidden state of the RNN architecture can not be adequately captured by feed-forward NN architecture representations. Therefore, a specialised NAS method is required for representing RNN architectures.

Wei et al. [29] proposed network morphism as an approach to creating a child network from a parent network such that the function and outputs of the parent NN architecture are preserved in the newly created child network. Therefore, the offspring model's parameters are initialised with the parameters of the corresponding parent model, which have already been optimised during the training of the parent model. The offspring model can then be trained for fewer epochs, making the NN architecture performance evaluation of the NAS method more efficient [11, 28].

Elsken et al. [11] proposed a Lamarckian Evolutionary algorithm for Multi-Objective Neural Architecture Design (LEMONADE). The LEMONADE algorithm uses a cell-based CNN architecture search space, which is explored through an evolutionary approach that

approximates a Pareto front [11]. The LEMONADE algorithm does not apply any specific recombination operators such as crossover or mutation, and relies on the concept of network morphism for offspring generation instead [11].

Elsken et al. [11] 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, *destructive* network transformations are required to optimise NN architecture complexity objective(s).

Elsken et al. [11] proposed the concept of approximate network morphism to cater for destructive network transformations. Destructive network transformations in the LEMONADE algorithm allow for the removal of units in the architecture and the removal of connections between units [11].

The MOE/RNAS algorithm proposed in this paper is similar to the LEMONADE algorithm, but designed specifically for RNN architecture evolution. Therefore, a novel RNN architecture encoding scheme, as well as a set of network morphisms applicable to RNN architectures, are proposed. Furthermore, the MOE/RNAS algorithm builds on top of the NSGA-II based approach from Bayer et al. [30] for RNN architecture evolution. Unlike the approach from Bayer et al. [30], the MOE/RNAS algorithm also considers RNN architecture complexity objectives, along with appropriate destructive network transformations that allow for the optimisation of RNN architecture complexity-related objectives.

### 2.4 Recurrent Neural Network NAS

Liu et al. [24] proposed a differentiable architecture search method, called DARTS. The DARTS algorithm works by searching for NN architectures in a continuous search space and optimising the NN architectures with respect to their validation set performance by gradient descent [24]. Liu et al. [24] reported that the DARTS algorithm was able to find CNN and RNN architectures with comparable performance to those found by state-of-the-art NAS methods, but with significantly reduced computational costs. However, the architectures generated by the DARTS algorithm optimise a single model accuracy objective. Therefore, the reduction in computational costs is a by-product as opposed to a design goal, which makes the reduced computational cost potentially unreliable compared to a multi-objective approach, where a reduction in computational cost is one of the objectives considered during optimisation.

Zoph and Le [10] proposed a reinforcement learning (RL) based NAS approach for RNN architecture search wherein they used an RNN controller as the RL agent. The RNN controller explores the RNN cell-based search space by generating a string of computation steps, which includes combination methods and activation functions that are allowed according to the defined search space [10]. A cell is then created from the string encoding, which is subsequently used for constructing the RNN architecture [10]. Zoph and Le [10] defined a cell-based search space for RNN architectures, wherein a single recurrent cell $g$ is described by

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

where $\theta$ represents the architecture of the cell $g$, $\alpha$ 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 [10] 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 [10].

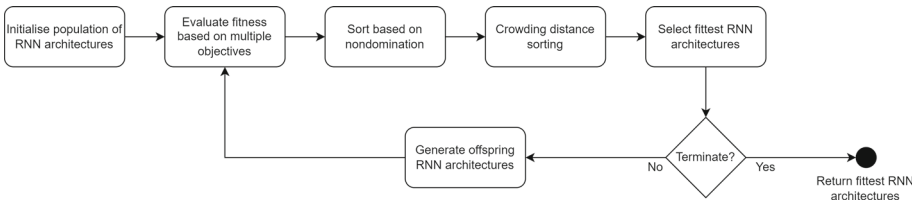


**Fig. 1** Overview of the MOE/RNAS algorithm

The evaluation of RNN architecture performance considered in [10, 16, 24] was based on a single objective that relates to the model accuracy. Thus, RNN architecture complexity was not explicitly considered during exploration of the respective RNN architecture search spaces.

Bayer et al. [30] proposed a method for multi-objective RNN architecture search that was based on a multi-objective EA. However, the multiple objectives considered during their search were based on model accuracy across multiple datasets and did not include any RNN architecture complexity-related objectives, such as the number of parameters that the models have or model inference time [30]. Additionally, the particular mutations considered in their approach were limited to constructive network transformations that only allowed for increasing the size of the RNN architectures [30].

Furthermore, a number of existing NAS studies rely on the high-level building blocks of RNN architectures to explore varying connections between existing NN architecture structures [50]. On the contrary, the method proposed in this work focuses on the optimisation of RNN architectures on a lower level.

To the best of the authors' knowledge, no dedicated studies of a multi-objective EA-based NAS method for novel RNN architecture search exist that also consider an architecture complexity objective. 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.

## 3 Multi-objective Evolutionary algorithm for Recurrent Neural Architecture Search

In this section, we present the MOE/RNAS algorithm: a **M**ulti-**O**bjective **E**volutionary algorithm for **R**ecurrent **N**eural **A**rchitecture **S**earch, to automatically construct RNN architectures for a provided dataset. The MOE/R-NAS algorithm relies on a multi-objective EA that is based on the NSGA-II algorithm for the exploration of the cell-based RNN architecture search space. An overview of the MOE/RNAS algorithm is presented in Fig. 1. The rest of this section is structured as follows: Sect. 3.1 discusses the search space of the MOE/RNAS algorithm. The search method employed by the MOE/RNAS algorithm for exploration of the search space is discussed in Sect. 3.2.

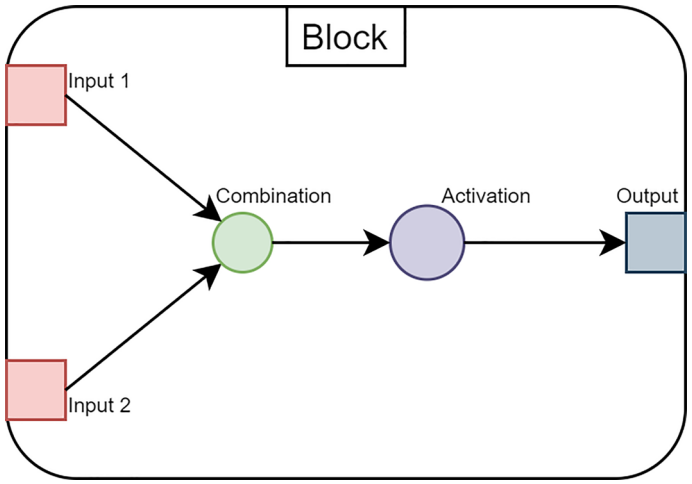


**Fig. 2** MOE/RNAS algorithm block encoding scheme

### 3.1 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 [10], as given in Eq. 1 in Sect. 2.4.

The MOE/RNAS algorithm's search space comprises the addition, subtraction, and elementwise multiplication combination methods. The activation functions allowed are the linear, identity, tanh, sigmoid, ReLU, and leaky ReLU activation functions.

The MOE/RNAS algorithm's approach to encoding RNN architectures is discussed below.

#### 3.1.1 Encoding

White et al. [51] identified different categorical encoding schemes that are employed throughout existing NAS methods. The encoding scheme developed for the MOE/RNAS algorithm can be placed within the categorical path encoding scheme that was identified by White et al. [51].

When a directed acyclic graph (DAG) representation of the RNN cell is assumed, each node of the DAG can be encoded by a block encoding structure. In the MOE/RNAS algorithm, specifically, an individual block is responsible for performing some operation on one or two inputs. Thus, each block can accept a minimum of one input and a maximum of two inputs. If the block accepts a single input, an activation function must be specified, and the output of the activation function is then used as the output of the particular block. If a block accepts two inputs, a combination method must be specified to indicate how the two inputs must be combined. When a block combines two inputs, the output of the combined inputs can be used as the output of the block, or an optional activation function can be applied to the combined inputs, which is then returned as the output of the block. The MOE/RNAS algorithm's block encoding scheme is illustrated in Fig. 2.

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

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

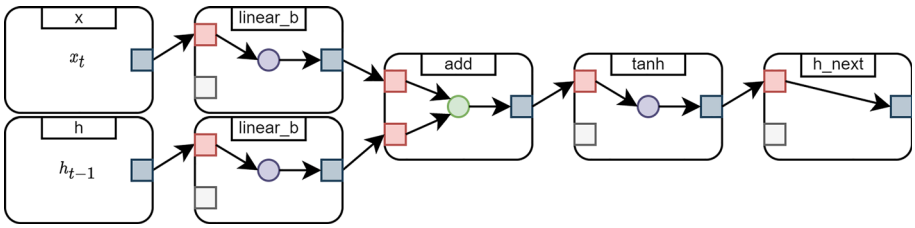


**Fig. 3** Basic RNN cell block encoding

with a tanh activation function can be seen in Fig. 3. 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 is 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 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, $c_{t-1}$ is simply ignored.

### 3.2 Search Method

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 evolved for a predefined number of generations to find the best performing architecture for the provided dataset, where architecture performance is based on multiple objectives. The MOE/RNAS algorithm maintains a Pareto-optimal front and employs the nondominated 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 generate offspring as opposed to a multi-parent recombination component.

Whilst similarities may exist between the proposed MOE/RNAS algorithm and genetic programming, it should be noted that the block-based representation method presented in Sect. 3.1 is only used as a representation for the RNN architecture individuals in the EA's population. During the fitness evaluation stage, an RNN model is constructed based on the blocks of a particular individual; and the blocks themselves have no ability to perform any kind of function, since they merely represent the connections that can exist between the nodes in some RNN architecture.

The rest of this section discusses the multi-objective EA-based search strategy that is implemented by the MOE/RNAS algorithm. Section 3.2.1 discusses the MOE/RNAS algorithm's network morphism approach for generating offspring. The initial population generation procedure is described in Sect. 3.2.2. Section 3.2.3 discusses the fitness evaluation of architectures in more detail. The MOE/RNAS algorithm's architecture selection strategy is discussed in Sect. 3.2.4.

**Table 1** Descriptions of block values used by the MOE/RNAS block encoding representation

| Block value | Description | Inputs | Outputs |
|---|---|---|---|
| x | The $x_t$ input at step $t$ | 0 | 1 |
| h | The $h_{t-1}$ input at step $t$ | 0 | 1 |
| c | The $c_{t-1}$ input at step $t$ | 0 | 1 |
| h_next | The new $h_t$ hidden state for step $t$ | 1 | – |
| c_next | The new $c_t$ 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 | – | – |

### 3.2.1 Approximate 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 [30, 52].

Elsken et al. [11] postulated that the difference in performance between parent and offspring architectures should be low when a maximum number of three network transformations 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 [11].

For each offspring architecture, the MOE/RNAS algorithm randomly selects a number from the range [1, 3], which is then used as the total number of consecutive network transformations that will be applied to that architecture. The network transformations that will be applied are randomly selected with uniform probability. The network transformations implemented by the MOE/RNAS algorithm for offspring generation are described in detail below.

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 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.

[30]. Bayer et al. [30] restricted the activation function to the linear activation function whereas the MOE/RNAS algorithm randomly selects an activation function from the allowable activation functions listed in Table 1. The output of the newly created block is the result of applying the randomly selected activation to the block's input.

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 the 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. [30]. Bayer et al. [30] 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.
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 and summarised in Table 1. The particular block's original activation function is excluded from the list of activation functions to choose from.

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 and listed in Table 1. The particular block's original combination method is excluded from the list of combination methods to choose from.

### 3.2.2 Initial Population

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 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 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.

### 3.2.3 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 is calculated based on the objectives provided. It is expected for one of the objectives to represent the 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 a parameter sharing technique, where offspring models are initialised with the parameters of their respective parent models. 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. The performance difference between parents and offspring is based on the observations reported by Elsken et al. [11], when a network morphism approach is used for generating offspring.

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

### 3.2.4 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.

Survivor selection is performed in the same way as it is done by the NSGA-II algorithm. 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 $N$.

Pseudocode for the MOE/RNAS algorithm is given in Algorithm 1.

## 4 Empirical Results

This section presents the results of the MOE/RNAS algorithm's ability to find and evolve novel RNN architectures. The MOE/RNAS algorithm was set to optimise the following

**Algorithm 1** MOE/RNAS Algorithm

Inputs: $N$, $\phi$, 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 $\phi$ 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**

two objectives: (i) the model accuracy objective, and (ii) an RNN architecture complexity objective, which was based on the number of blocks that the architecture contained. The following tasks were considered:

1. A standard word-level NLP task based on the Penn Treebank dataset. The Penn Treebank dataset is often used as a benchmark in RNN NAS research [10, 16, 24, 26]. Although it is unlikely for any current NAS method to find a novel RNN architecture that outperforms state-of-the-art RNN architectures that were designed by human experts [16, 49], an EA-based RNN architecture search method has not been implemented on the Penn Treebank dataset.
   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.
2. A sequence learning task based on artificially generated strings from a context-sensitive language, which was previously used in the study by Bayer et al. [30]. The training and testing datasets consist of strings that are generated from the $a^n b^n c^n$ context-sensitive language, where the value of $n$ is randomly selected from the range {1..10} for each string.
   By artificially generating the sequence learning 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.
3. A sentiment analysis task that is based on the ACL-IMBD [53] dataset. This dataset contains 50 000 sentences, each of which has either a positive or a negative sentiment.

For all three tasks, the RNN architecture complexity objective was based on the number of blocks that an architecture comprised. The model accuracy objective that was used is discussed in more detail under the relevant sections below.

Technical implementation details for the experiments were as follows:

1. All the source code implementations[1] of this study were developed using the Python programming language and the PyTorch [54] framework.
2. Experiments were run on a system with a single Nvidia V100 16GB GPU.

The rest of this section is structured as follows: Sect. 4.1 discusses the results for the word-level NLP task that is based on the Penn Treebank dataset. The sequence learning task results are discussed in Sect. 4.2. Section 4.3 discusses the results for the sentiment analysis task. The observations from the experiments are summarised in Sect. 4.4.

### 4.1 Word-Level Natural Language Processing Task

A total of three experimental runs were performed for the word-level NLP task. One experimental run included the LSTM and GRU architectures in the initial population. Two experimental runs were performed where the LSTM and GRU architectures were excluded from the initial population. The limited number of experimental runs were due to the inherently high computational resource demand of NAS.

The performance of a model implemented for the standard word-level NLP task is calculated based on how well the model is able to predict the next word, which is commonly represented by a metric called *perplexity* [55, 56]. Perplexity measures how accurately a model can predict the next word, such that for a given test set $D_G = d_1 d_2 ... d_Q$, the perplexity is calculated by:

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

normalised by the number of words [56]. As noted by Jurafsky and Martin [56], the chain rule can be used to expand the probability of $D_G$ such that:

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

Therefore, the model accuracy objective considered during this experiment is based on the perplexities achieved by the respective models on the Penn Treebank dataset.

The RNN architectures created by the MOE/RNAS algorithm were not stacked (i.e. repeated) during model creation, and instead, each model contained a single instance of the corresponding RNN architecture, i.e., a single cell. The models were implemented with an embedding layer dimension of 650 and a hidden layer dimension of 650, which was adopted from [57]. A batch size of 20 was used during model training, and the RNN models were unrolled for 35 time steps during backpropagation training. For each model, a dropout layer was included to randomly zero some of the elements of the input with a probability of 0.5. Models were trained for 30 epochs using a stochastic gradient descent training method. Training of the models started with a learning rate of 20, and the learning rate was reduced

[1] Source code implementation of the MOE/RNAS algorithm is available at https://github.com/reinn-cs/rnn-nas.

**Table 2** The word-level NLP task's Pareto-optimal architecture performances

| Architecture | Test perplexity | Number of blocks |
|---|---|---|
| LSTM_58 | **83.782** | 25 |
| LSTM_0 | 83.945 | 26 |
| GRU_0 | 89.766 | 23 |
| rdm68_45 | 92.704 | 11 |
| rdm8_0 | 99.625 | 10 |
| rdm8_3 | 169.047 | 9 |
| rdm8_190 | 172.487 | **8** |

Bold values represent the best performing results for the respective metrics presented in each column

when the model performance started stagnating; the specific learning rate reduction was adopted from [5]. Offspring model parameters were initialised with their respective parent model parameters. If the initial test perplexity difference between an offspring model and its parent was more than 5 perplexity points, the offspring model was trained for 30 epochs. Alternatively, offspring models were only trained for 5 epochs.

The initial population included the basic RNN, LSTM, and GRU architectures. 97 RNN architectures were uniformly sampled from the search space, which resulted in a total population size of 100. The search was terminated after 30 generations and the total search cost was 8.25 GPU days for one experimental run.

The Pareto-optimal RNN architectures found by the MOE/RNAS algorithm are listed in Table 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; the rdm68_45 architecture is illustrated in Fig. 4 (refer to Sect. 3.2.2 for RNN architecture identifier notation).

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 2.5M 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.

The results show that 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 Fig. 5. The average test perplexity per generation did not exhibit a similar trend, but neither did it worsen across the generations, as illustrated in Fig. 6. Therefore, the MOE/RNAS algorithm was able to optimise the architecture complexity objective without negatively influencing the model accuracy objective across the 30 generations. The best performing RNN architectures that were evolved by the MOE/RNAS algorithm dominated the manually designed LSTM architecture in terms of Pareto-optimality, while the GRU architecture remained non-dominated across the 30 generations; the final Pareto-front for this experiment can be seen in Fig. 7.

**Control Experiment - Exclude LSTM and GRU From Initial Population:** In this experiment, the LSTM and GRU architectures were not included in the initial population. Therefore, the initial population comprised the basic RNN architecture and 99 randomly generated architectures. The results show that the MOE/RNAS algorithm was able to consistently optimise the RNN architecture complexity objective as the EA progressed, which

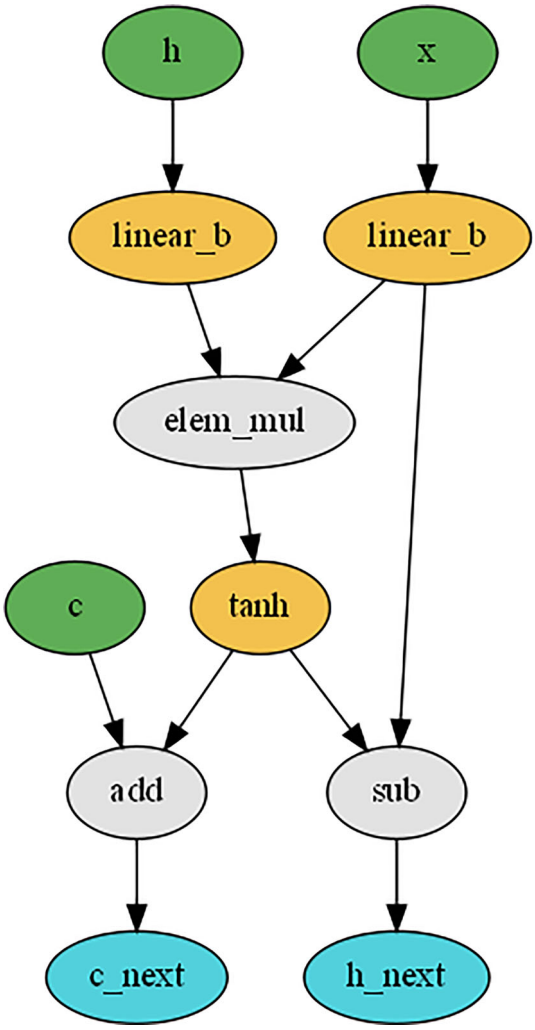


**Fig. 4** The rdm68_45 architecture evolved by the MOE/RNAS algorithm

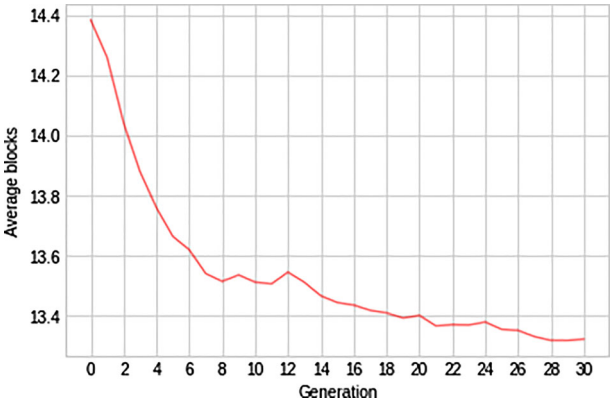


**Fig. 5** Word-level NLP task average number of blocks per generation for a single run

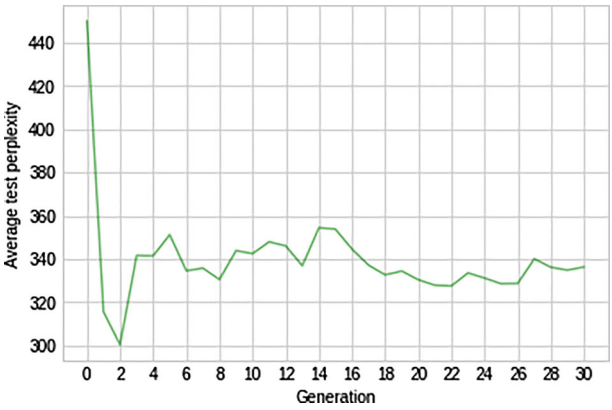


**Fig. 6** Word-level NLP task average test perplexity per generation for a single run

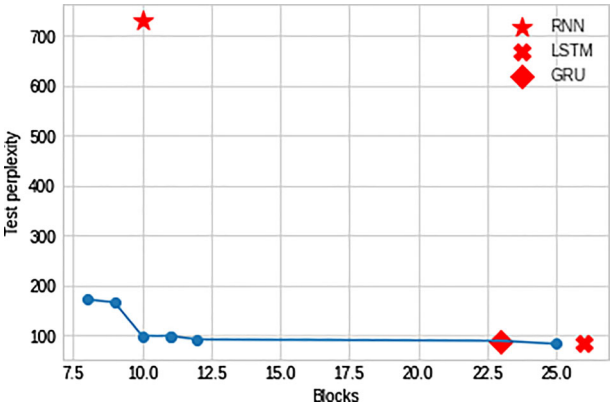


**Fig. 7** The Pareto-front of the word-level NLP task experiment. RNN and LSTM architectures are included for reference

can be seen in Fig. 8. Despite the average test perplexity per generation exhibiting some improvement as the EA progressed, the average test perplexity per generation is higher compared to the average test perplexity per generation from the previous experiment. From the Pareto-optimal architectures listed in Table 3, it can be seen that the best performing RNN architecture evolved during this experiment achieved a test perplexity of 94.318, which is 1.6 perplexity points worse compared to the best performing architecture evolved by the MOE/RNAS algorithm in the previous experiment.

After 30 generations, the total search cost of this experiment was 6.25 GPU days, which is better compared to the 8.25 GPU days search cost of the previous experiment. By including the LSTM and GRU architectures in the previous experiment's initial population, a number of offspring architectures were generated from the LSTM and GRU architectures, which contributed towards longer model training times and inevitably led to a higher search cost.

This control experiment was then repeated with the same configuration. The best architecture found during this experimental run was generated after 20 generations and achieved a test perplexity of 91.304. From the Pareto-optimal architectures listed in Table 4, it is observed

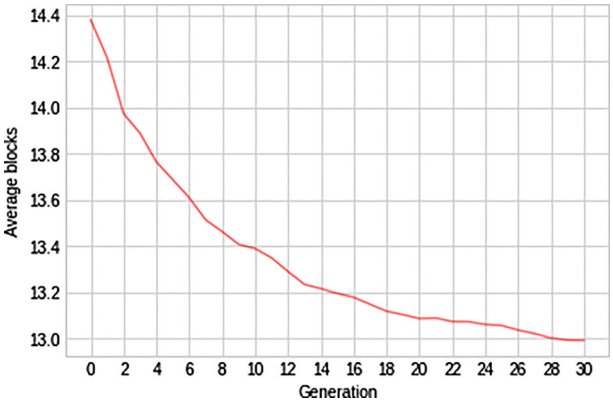


**Fig. 8** Average number of blocks per generation for control experiment

**Table 3** The word-level NLP task's control experiment Pareto-optimal architecture performances

| Architecture | Test perplexity | Number of blocks |
|---|---|---|
| rdm35_108 | **94.318** | 14 |
| rdm35_116 | 96.588 | 15 |
| rdm21_62 | 97.318 | 19 |
| rdm21_40 | 99.476 | 17 |
| rdm5_0 | 101.313 | 10 |
| BASIC_18 | 165.862 | 9 |
| rdm28_26 | 170.537 | **8** |

Bold values represent the best performing results for the respective metrics presented in each column

**Table 4** The second run of the word-level NLP task's control experiment Pareto-optimal architecture performances

| Architecture | Test perplexity | Number of blocks |
|---|---|---|
| rdm6_104 | **91.304** | 14 |
| rdm6_23 | 91.379 | 13 |
| rdm6_90 | 92.088 | 12 |
| rdm6_76 | 93.582 | 11 |
| rdm46_388 | 101.820 | 10 |
| rdm46_86 | 103.486 | 9 |
| rdm79_79 | 167.251 | 8 |
| rdm46_25 | 171.414 | **7** |

Bold values represent the best performing results for the respective metrics presented in each column

that the MOE/RNAS algorithm was able to consistently optimise the complexity objective, with 14 blocks being the highest number of blocks amongst the Pareto-optimal architectures.

### 4.2 Sequence Learning Task Based on Artificially Generated Data

This section discusses the experimental results obtained after implementing the MOE/RNAS algorithm to search for and optimise RNN architectures for a sequence learning task. The dataset used for this 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. [30] in their multi-objective EA-based RNN architecture search method.

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 $n$ were randomly chosen from the range $\{1..10\}$. For example, $n = 3$ results in the 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.

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 the backpropagation through time training algorithm with a stochastic gradient descent optimisation technique and a learning rate of 0.01.

Parent selection was limited to the top 25 architectures of the Pareto front (see Sect. 3.2.4). Thus, only 25 offspring architectures were produced for each generation. During offspring generation, up to ten consecutive network transformations were allowed per architecture (see Sect. 3.2.1). Therefore, fewer offspring architectures were generated and a higher number of consecutive network transformations were allowed compared to the previous word-level NLP experiments. This was done specifically to gain some insight into the multi-objective RNN morphism approach employed by the MOE/RNAS algorithm.

The initial population included the basic RNN, LSTM, and GRU architectures. 97 RNN architectures were uniformly sampled from the search space, which resulted in a total population size of 100. The search was terminated after 15 generations, which resulted in a total search cost of 0.33 GPU days.

According to Fig. 9, 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.

Although the average MSE per generation shown in Fig. 10 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 Pareto-optimal architectures listed in Table 5, 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.

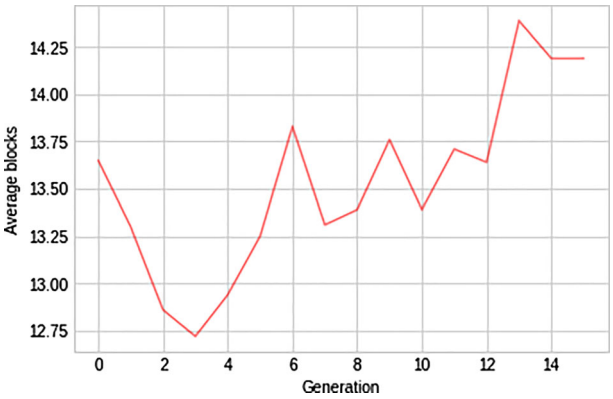


**Fig. 9** Average number of blocks per generation observed for the sequence learning task

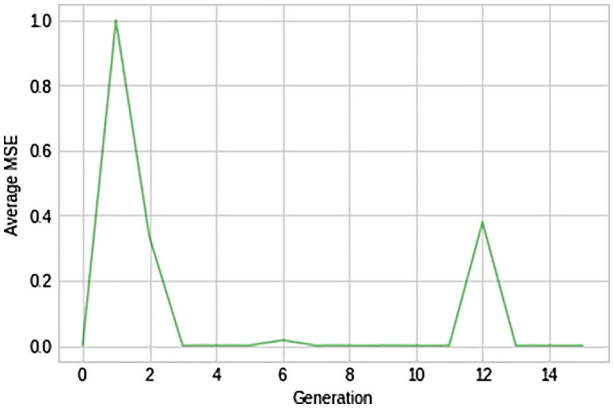


**Fig. 10** Average MSE loss per generation observed for the sequence learning task

**Table 5** Pareto-optimal architecture performances of the sequence learning task, which includes the performance of the LSTM architecture for reference

| Architecture | MSE loss | Number of blocks |
|---|---|---|
| rdm82_21 | **0.000014** | 15 |
| LSTM_0 | 0.000157 | 26 |
| rdm82_18 | 0.000939 | 14 |
| rdm1_21 | 0.001806 | 11 |
| rdm43_0 | 0.017431 | 10 |
| BASIC_29 | 0.064309 | **9** |

Bold values represent the best performing results for the respective metrics presented in each column

The rdm82_21 architecture shown in Fig. 11 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 Sect. 3.2.1. 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 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

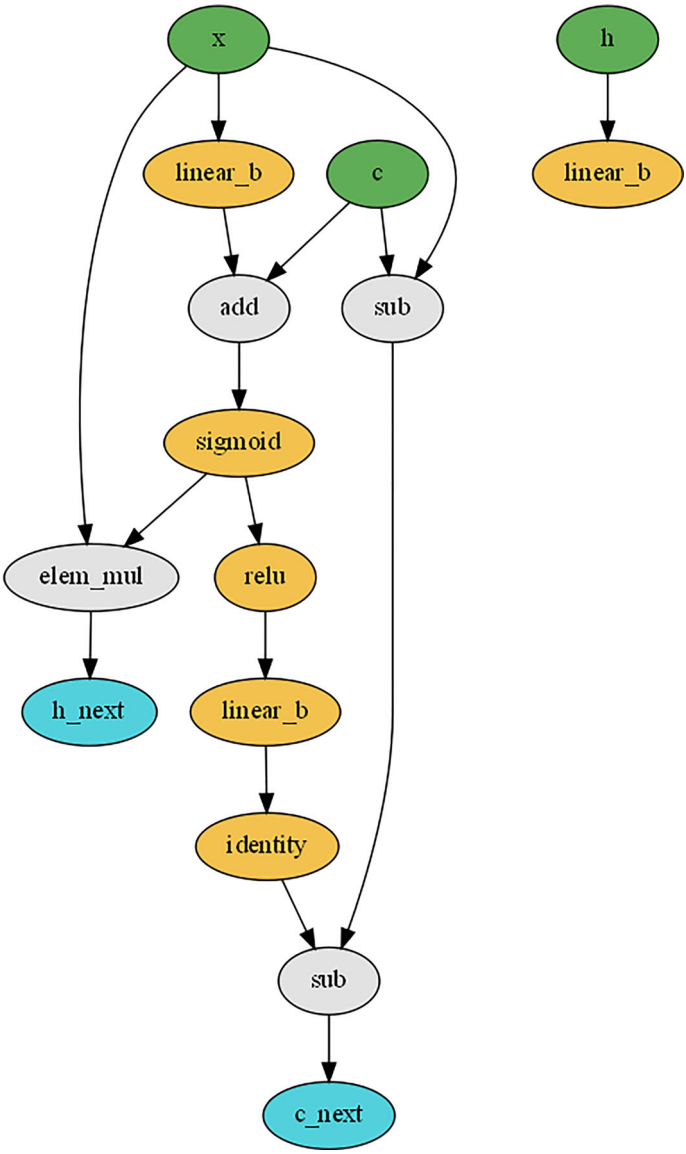


**Fig. 11** The rdm82_21 architecture evolved by the MOE/RNAS algorithm

$h_{t-1}$ 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. Therefore, the architecture effectively used the $c_t$ output layer block as a substitute for the hidden state.

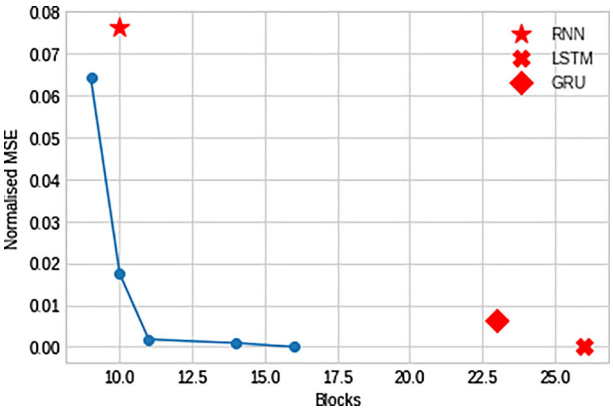


**Fig. 12** Pareto-front of the sequence learning task, which also includes the RNN, GRU, and LSTM architectures for reference

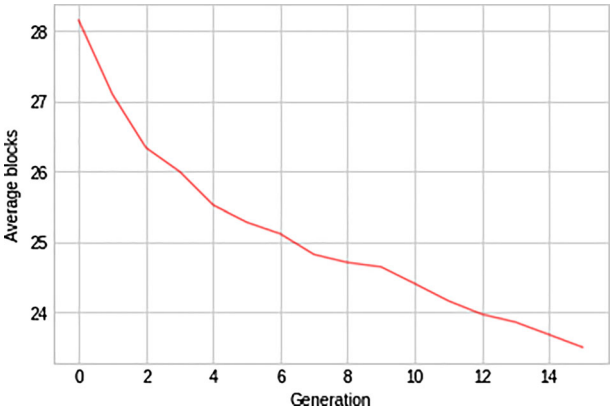


**Fig. 13** Average number of blocks per generation for the sequence learning control experiment

Thus, despite being unable to optimise the average number of blocks per generation, the MOE/RNAS algorithm was able to find and evolve novel RNN architectures that dominated the RNN, GRU, and LSTM architectures in terms of Pareto-optimality in less than 15 generations; the final Pareto-front for this experiment can be seen in Fig. 12.

**Control Experiment - Reduced Number of Consecutive Network Transformations Allowed:** In this experiment, a maximum number of three consecutive transformations were considered during network morphism. Additionally, 100 offspring architectures were created for each generation.

Figures 13 and 14 show the favourable trends in terms of the average number of blocks per generation and the average MSE per generation across the 15 generations, respectively. 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.

Table 6 lists the Pareto-optimal architecture performances. Apart from the BASIC_0 archi-

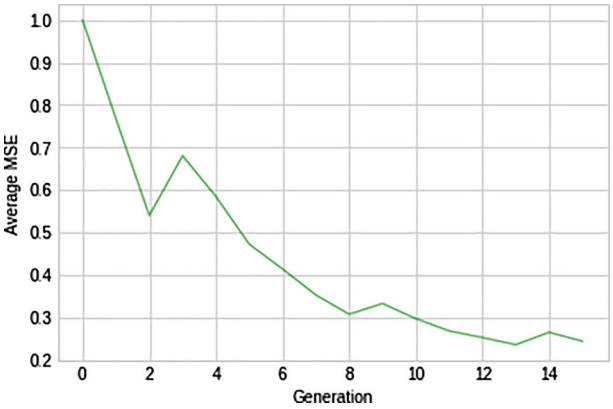


**Fig. 14** Average MSE loss per generation for the sequence learning control experiment

**Table 6** Pareto-optimal architecture performances for the sequence learning control experiment

| Architecture | MSE loss | Number of blocks |
|---|---|---|
| rdm32_45 | **0.00115** | 18 |
| rdm32_32 | 0.00272 | 17 |
| rdm32_26 | 0.03956 | 16 |
| rdm72_41 | 0.04896 | 14 |
| rdm54_29 | 0.06305 | 13 |
| BASIC_20 | 0.07508 | 12 |
| BASIC_0 | 0.13433 | **10** |

Bold values represent the best performing results for the respective metrics presented in each column

tecture, all other architectures in the Pareto front listed in Table 6 are offspring architectures that were optimised from the randomly generated architectures during the initialisation of the population.

The sequence learning task was repeated with the same configuration, but for 20 generations as opposed to 15 generations. From the results of this run it was observed that the average number of blocks per generation appeared to have reached an optimum after 14 generations. From the 15th generation onwards, the average number of blocks per generation started increasing, which can be seen in Fig. 15. Although the second run of the control experiment did not exhibit a similar trend in terms of the model accuracy objective to that of the initial run of the control experiment, the MOE/RNAS algorithm was still able to maintain a relatively low average MSE per generation throughout the run, which can be seen in Fig. 16.

Therefore, the MOE/RNAS algorithm is clearly capable of generationally optimising multiple RNN architecture objectives when the appropriate configuration is considered, such as the number of consecutive network transformations allowed during network morphism.

### 4.3 Sentiment Analysis

This section discusses the experimental results obtained after implementing the MOE/RNAS algorithm to search for and optimise RNN architectures for a sentiment analysis task based

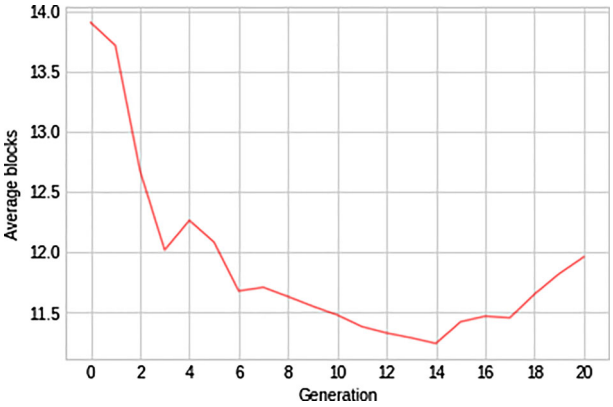


**Fig. 15** Average number of blocks per generation for the second run of the sequence learning control experiment

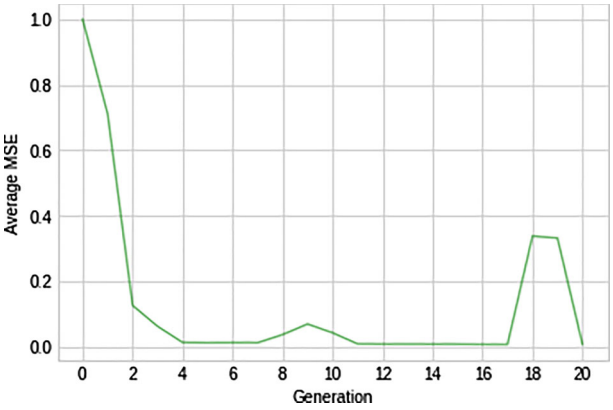


**Fig. 16** Average MSE loss per generation for second run of the sequence learning control experiment

on the the ACL-IMBD [53] dataset. The models were implemented with an embedding layer dimension of 1000 and a hidden layer dimension of 65, and a batch size of 50 was used during model training. Training and testing models on this sentiment analysis task consist of presenting the model with a sentence, and the model is then expected to predict whether the input sentence has a positive or a negative sentiment. Therefore, the model accuracy objective considered for this dataset is based on the number of predictions that were correct out of all the sentences from the testing data, which is then represented as a percentage value.

An LSTM model implemented for this task was able to achieve an accuracy of 83.10% after being trained for only 5 epochs. The GRU architecture achieved an accuracy of 83.26%, whereas the basic RNN architecture achieved an accuracy of 80%.

The MOE/RNAS algorithm was implemented to search for and optimise RNN architectures for this task over a maximum of 15 generations and a population size of 100. The initial population included the basic RNN architecture and 99 RNN architectures were uniformly sampled from the search space.

The total search cost for this task was 20 GPU days, which is relatively high compared to the search costs observed from the previous tasks. Despite the low number of epochs that

**Table 7** Pareto-optimal architecture performances for the sentiment analysis task

| Architecture | Accuracy | Number of blocks |
|---|---|---|
| rdm43_40 | **85.22** | **18** |
| rdm69_43 | 84.96 | 20 |
| rdm72_28 | 84.48 | 21 |
| rdm48_4 | 84.34 | 22 |
| rdm69_79 | 83.92 | 23 |

Bold values represent the best performing results for the respective metrics presented in each column

**Table 8** Pareto-optimal architecture performances for the second run of the sentiment analysis task

| Architecture | Accuracy | Number of blocks |
|---|---|---|
| rdm96_26 | **86.64** | **18** |
| rdm5_25 | 86.46 | 19 |
| rdm96_62 | 86.42 | 22 |
| rdm5_104 | 86.28 | 24 |
| rdm96_63 | 83.90 | 26 |

Bold values represent the best performing results for the respective metrics presented in each column

the models were trained for, the average training time was around 15 min per model, which played a significant role in the high search cost of this task.

The MOE/RNAS algorithm was able to successfully optimise the RNN architectures to outperform the LSTM and GRU architectures, with the best performing architecture found achieving an accuracy of 85.22%. Furthermore, the best performing architecture was also the architecture with the lowest number of blocks amongst the Pareto-optimal architectures, which can be seen in Table 7.

The experiment was then repeated using the same configuration, and after 15 generations and a search cost of 20 GPU days, better performing architectures were found compared to the first run of this experiment in terms of the model accuracy objective. The Pareto-optimal architectures found during this experimental run can be seen in Table 8.

### 4.4 Results Discussion

He et al. [49] postulated that current RNN NAS methods have yet to find novel RNN architectures that outperform state-of-the-art manually designed RNN architectures, specifically within the NLP domain. According to He et al. [49], the best performing RNN architecture found by existing RNN NAS publications is the RNN cell discovered by the DARTS NAS method [24], which achieved a test perplexity of 56.1 on the Penn Treebank dataset.

The best performing architecture found by MOE/RNAS achieved a test perplexity of 92.7 on the Penn Treebank dataset with a total of 13.8M trainable parameters. In comparison, the DARTS cell has 33 M trainable parameters [24]. Therefore, the architecture found by the MOE/RNAS algorithm has a much lower computational resource demand, but at the cost of reduced model accuracy.

From the experiments performed on the sequence learning task, it was observed that the number of consecutive network transformations 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 optimised by the MOE/RNAS algorithm do not outperform those randomly created during the initialisation of the initial population.

Although the search for the sequence learning task was terminated after 15 generations, the total search cost was significantly lower compared to the more than 8 GPU days search cost of the experiments that used the word-level NLP task's dataset. This lower search cost compared to the search cost of the word-level NLP experiments 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. It was observed during the control experiment for the sequence learning task that the 15 generation search cost increased to 1.78 GPU days when 100 offspring architectures were created per generation.

However, the 8 GPU days search cost of the experiments that used the word-level NLP task's dataset is significant, since the 8 GPU days search cost shows that the MOE/RNAS algorithm has a higher overall computational demand compared to the DARTS NAS method [24]. This observation was further confirmed with the sentiment analysis experiments from Sect. 4.3. The higher search cost of the MOE/RNAS algorithm is attributed to the training of the RNNs at each generation, despite the implementation of network morphism and early stopping to make the MOE/RNAS algorithm more efficient. Training and testing 100 RNN models at each generation is expected to have a high computational demand, and the methods implemented to make model accuracy evaluation more efficient were limited such that multi-objective RNN architecture evolution can be studied in more detail instead.

## 5 Conclusion

In this paper, we proposed the MOE/RNAS algorithm as a multi-objective EA-based NAS method for automated RNN architecture search, which was specifically developed to optimise both the model accuracy objective along with the RNN architecture complexity objective. The MOE/RNAS algorithm relies on methods such as network morphism and early stopping to make the generational RNN architecture evolution more efficient.

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 current state-of-the-art RNN architectures in terms of test perplexity on the Penn Treebank dataset. However, RNN architectures were discovered that achieved comparable perplexity, but with significantly lower computational cost. Furthermore, the MOE/RNAS algorithm was able to find and evolve Pareto-optimal RNN architectures that dominated the manually designed RNN architectures, such as the LSTM.

It was observed that the approximate RNN morphism is sensitive to the maximum number of consecutive network transformations allowed during offspring generation. Lower numbers of consecutive network transformations result in a more consistent generational optimisation of the multiple objectives considered.

Overall, the MOE/RNAS algorithm is a good candidate for real-world machine learning applications where the model computational resource demand is of concern. Additionally, the MOE/RNAS method will be beneficial to use cases where the knowledge of existing pretrained models can be leveraged to search for models with reduced computational resource demand while maintaining an acceptable model accuracy objective.

An obvious avenue for future work would be to enhance the network morphism approach of the MOE/RNAS algorithm, which could include an RL agent to consider the impact of previous network transformations on the resulting RNN architecture fitness. Furthermore, performance prediction techniques, such as the density estimation technique implemented in [11], can be incorporated to improve the overall search cost of the MOE/RNAS algorithm.

**Acknowledgements** The authors would like to thank the Centre for High Performance Computing (CHPC) (https://www.chpc.ac.za/) for the use of their cluster to obtain the data for this study. The financial assistance of the National Research Foundation (NRF) of South Africa towards this research is hereby acknowledged. The research has been supported by the NRF Thuthuka grant number TTK210316590115. Opinions expressed and conclusions arrived at, are those of the author and are not necessarily to be attributed to the NRF.

**Funding** Open access funding provided by University of Pretoria.

**Data Availability** 1. The Penn Treebank dataset used for the word-level NLP task in Sect. 4.1 is available for download at: https://github.com/reinn-cs/rnn-nas/tree/master/example_datasets/ptb/data. 2. The dataset used for the sequence learning task in Sect. 4.2 is artificially generated as described in the relevant section. The source code for the generation of the dataset is included in the source code repository of the MOE/RNAS algorithm implementation, which is available at: https://github.com/reinn-cs/rnn-nas. 3. The data used for the analysis of the MOE/RNAS algorithm was based on the experimental results obtained after implementing the MOE/RNAS algorithm to search for and optimise RNN architectures for the respective datasets. The source code for the MOE/RNAS algorithm implementation is available at: https://github.com/reinn-cs/rnn-nas.

## Declarations

**Conflict of interest** The authors have no relevant financial or non-financial interests to disclose.



## References

1. Mandic DP, Chambers JA (2001) Recurrent Neural Networks for Prediction, Wiley Series in Adaptive and Learning Systems for Signal Processing, Communications, and Control, vol 4. John Wiley & Sons Ltd, Chichester, UK, p 297. https://doi.org/10.1002/047084535X
2. Medsker LR, Jain LC (2001) Recurrent Neural Networks: Design and Applications, 1st edn. CRC Press, Boca Raton
3. Pascanu R, Gulcehre C, Cho K, Bengio Y (2014) How to construct deep recurrent neural networks. In: Proceedings of the second international conference on learning representations (ICLR 2014)
4. Radford ISA, Wu J, Child R, Luan D, Amodei D (2020) Language models are unsupervised multitask learners. OpenAI Blog 1:1–7
5. Merity S, Keskar NS, Socher R (2018) Regularizing and optimizing LSTM language models. In: Proceedings of the 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC,

Canada, April 30–May 3, Conference Track Proceedings. Retrieved from https://openreview.net/forum?id=SyyGPP0TZ
6. Kong W, Dong ZY, Jia Y, Hill DJ, Xu Y, Zhang Y (2019) Short-term residential load forecasting based on LSTM recurrent neural network. IEEE Trans Smart Grid 10(1):841–851. https://doi.org/10.1109/TSG.2017.2753802
7. Suzgun M, Belinkov Y, Shieber SM (2019) On evaluating the generalization of LSTM models in formal languages. Proc Soc Comput Linguist 2:277–286. https://doi.org/10.7275/s02b-4d91
8. Wang C, Wang H, Feng G, Geng F (2020) Multi-objective neural architecture search based on diverse structures and adaptive recommendation. arXiv preprint arXiv:2007.02749
9. Yang Z, Wang Y, Chen X, Shi B, Xu C, Xu C, Tian Q, Xu C (2019) Cars: continuous evolution for efficient neural architecture search. In: Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, pp 1829–1838
10. Zoph B, Le QV (2017) Neural architecture search with reinforcement learning. In: Proceedings of the 5th international conference on learning representations, ICLR 2017, Toulon, France, April 24–26, Conference Track Proceedings. Retrieved from https://openreview.net/forum?id=r1Ue8Hcxg
11. Elsken T, Metzen JH, Hutter F (2019) Efficient multi-objective neural architecture search via lamarckian evolution. In: Proceedings of the 7th international conference on learning representations, ICLR 2019, New Orleans, LA, USA, May 6–9. Retrieved from https://openreview.net/forum?id=ByME42AqK7
12. Liu Y, Sun Y, Xue B, Zhang M, Yen GG, Tan KC (2021) A survey on evolutionary neural architecture search. IEEE Trans Neural Netw Learn Syst. https://doi.org/10.1109/TNNLS.2021.3100554
13. Wistuba M, Rawat A, Pedapati T (2019) A survey on neural architecture search. arXiv preprint arXiv:1905.01392
14. Lu Z, Whalen I, Dhebar Y, Deb K, Goodman E, Banzhaf W, Boddeti VN (2020) NSGA-Net: neural architecture search using multi-objective genetic algorithm. In: Proceedings of the twenty-ninth international joint conference on artificial intelligence, international joint conferences on artificial intelligence organization, pp 4750–4754. https://doi.org/10.24963/ijcai.2020/659
15. Chen Z, Zhou F, Trimponias G, Li Z (2020) Multi-objective neural architecture search via non-stationary policy gradient. arXiv preprint arXiv:2001.08437
16. Klyuchnikov N, Trofimov I, Artemova E, Salnikov M, Fedorov M, Filippov A, Burnaev E (2022) NAS-Bench-NLP: neural architecture search benchmark for natural language processing. IEEE Access 10:45736–45747. https://doi.org/10.1109/ACCESS.2022.3169897
17. Hu S, Cheng R, He C, Lu Z (2021) Multi-objective neural architecture search with almost no training. In: Proceedings of the 11th international conference on evolutionary multi-criterion optimization. Springer, Cham, pp 492–503. https://doi.org/10.1007/978-3-030-72062-9_39
18. Chu X, Zhang B, Xu R, Ma H (2020) Multi-objective reinforced evolution in mobile neural architecture search. In: Proceedings of the European conference on computer vision. Springer, Cham, pp 99–113. https://doi.org/10.1007/978-3-030-66823-5_6
19. Smith JE, Eiben A (2015) Introduction to Evolutionary Computing, 2nd edn. Springer Publishing Company Inc., Cham
20. Rudolph G (1998) On a multi-objective evolutionary algorithm and its convergence to the Pareto set. In: Proceedings of the IEEE conference on evolutionary computation, ICEC. IEEE, pp 511–516. https://doi.org/10.1109/icec.1998.700081
21. Zhou A, Qu BY, Li H, Zhao SZ, Suganthan PN, Zhangd Q (2011) Multiobjective evolutionary algorithms: a survey of the state of the art. Swarm Evolut Comput 1(1):32–49. https://doi.org/10.1016/j.swevo.2011.03.001
22. Zitzler E, Deb K, Thiele L (2000) Comparison of multiobjective evolutionary algorithms: empirical results. Evolut Comput 8(2):173–195. https://doi.org/10.1162/106365600568202
23. Coello Coello CA, Van Veldhuizen DA, Lamont GB (2007) Evolutionary algorithms for solving multi-objective problems. Genetic and evolutionary computation series. Springer, US, Boston, MA. https://doi.org/10.1007/978-0-387-36797-2
24. Liu H, Simonyan K, Yang Y (2018) Darts: differentiable architecture search. arXiv preprint arXiv:1806.09055
25. Pham H, Guan MY, Zoph B, Le QV, Dean J (2018) Efficient neural architecture search via parameters sharing. In: Proceedings of the 35th international conference on machine learning, vol. 80. PMLR, pp 4095–4104
26. Li L, Talwalkar A (2019) Random search and reproducibility for neural architecture search. In: Proceedings of the thirty-fifth conference on uncertainty in artificial intelligence, UAI 2019, Tel Aviv, Israel, July 22–25, pp 367–377. Retrieved from http://proceedings.mlr.press/v115/li20c.html
27. Yang Z, Dai Z, Salakhutdinov R, Cohen WW (2017) Breaking the softmax bottleneck: a high-rank RNN language model. arXiv preprint arXiv:1711.03953

28. Cai H, Chen T, Zhang W, Yu Y, Wang J (2018) Efficient architecture search by network transformation. Proc AAAI Conf Artif Intell 32(1):2787–2794 arXiv:1707.04873
29. Wei T, Wang C, Rui Y, Chen CW (2016) Network morphism. In: Proceedings of the 33rd international conference on machine learning, vol. 48. PMLR, New York, New York, USA, pp 564–572
30. Bayer J, Wierstra D, Togelius J, Schmidhuber J (2009) Evolving memory cell structures for sequence learning. Artif Neural Netw ICANN 2009:755–764. https://doi.org/10.1007/978-3-642-04277-5_76
31. Chen G (2016) A gentle tutorial of recurrent neural network with error backpropagation. arXiv preprint arXiv:1610.02583
32. Bengio Y, Simard P, Frasconi P (1994) Learning long-term dependencies with gradient descent is difficult. IEEE Trans Neural Netw 5(2):157–166. https://doi.org/10.1109/72.279181
33. Pascanu R, Mikolov T, Bengio Y (2013) On the difficulty of training recurrent neural networks. In: Proceedings of the international conference on machine learning. PMLR, pp 1310–1318
34. Hochreiter S, Schmidhuber J (1997) Long short-term memory. Neural Comput 9(8):1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735
35. Karpathy A, Johnson J, Fei-Fei L (2015) Visualizing and understanding recurrent networks. arXiv preprint arXiv:1506.02078
36. Chung J, Gulcehre C, Cho K, Bengio Y (2015) Gated feedback recurrent neural networks. In: Proceedings of the 32nd international conference on machine learning, vol. 37. PMLR, pp 2067–2075
37. Jozefowicz R, Zaremba W, Sutskever I (2015) An empirical exploration of recurrent network architectures. In: Proceedings of the 32nd international conference on machine learning, vol. 37. JMLR.org, Lille, France, pp 2332–2340
38. Cho K, Van Merriënboer B, Gulcehre C, Bahdanau D, Bougares F, Schwenk H, Bengio Y (2014) Learning phrase representations using RNN encoder-decoder for statistical machine translation. In: Proceedings of the empirical methods in natural language processing. ACL, pp 1724–1734. https://doi.org/10.3115/v1/d14-1179
39. Chung J, Gulcehre C, Cho K, Bengio Y (2014) Empirical evaluation of gated recurrent neural networks on sequence modeling. In: Proceedings of the NIPS 2014 workshop on deep learning, December 2014, pp 1–9. arXiv:1412.3555
40. Engelbrecht AP (2007) Computational intelligence: an introduction, 2nd edn. Wiley Publishing, New Jersey
41. Ma X, Li X, Zhang Q, Tang K, Liang Z, Xie W, Zhu Z (2019) A survey on cooperative co-evolutionary algorithms. IEEE Trans Evolut Comput 23(3):421–441. https://doi.org/10.1109/TEVC.2018.2868770
42. Eiben A, Schoenauer M (2002) Evolutionary computing. Inf Process Lett 82(1):1–6. https://doi.org/10.1007/978-3-662-43693-6_3
43. Deb K, Pratap A, Agarwal S, Meyarivan T (2002) A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans Evolut Comput 6(2):182–197. https://doi.org/10.1109/4235.996017
44. Zeng SY, Ding LX, Kang LS, Chen Y (2003) A new multiobjective evolutionary algorithm: OMOEA. Congress on Evolutionary Computation, 2003. CEC '03 2, 898–905. https://doi.org/10.1109/CEC.2003.1299762. http://ieeexplore.ieee.org/document/1299762/
45. Chu X, Yu X (2018) Improved crowding distance for NSGA-II. arXiv preprint arXiv:1811.12667
46. Lu Z, Deb K, Goodman E, Banzhaf W, Boddeti VN (2020) NSGANetV2: evolutionary multi-objective surrogate-assisted neural architecture search. In: Proceedings of the European conference on computer vision. Springer, Cham, pp 35–51. https://doi.org/10.1007/978-3-030-58452-8_3
47. Lu Z, Whalen I, Dhebar Y, Deb K, Goodman ED, Banzhaf W, Boddeti VN (2021) Multiobjective evolutionary design of deep convolutional neural networks for image classification. IEEE Trans Evolut Comput 25(2):277–291. https://doi.org/10.1109/TEVC.2020.3024708
48. Park Km, Shin D, Yoo Y (2020) Evolutionary neural architecture search (NAS) using chromosome nondisjunction for Korean grammaticality tasks. Appl Sci 10(10):3457. https://doi.org/10.3390/app10103457
49. He X, Zhao K, Chu X (2021) AutoML: a survey of the state-of-the-art. Knowl Based Syst. https://doi.org/10.1016/j.knosys.2020.106622
50. Mo H, Custode LL, Iacca G (2021) Evolutionary neural architecture search for remaining useful life prediction. Appl Soft Comput 108:107474. https://doi.org/10.1016/j.asoc.2021.107474
51. White C, Neiswanger W, Nolen S, Savani Y (2020) A study on encodings for neural architecture search. In: Proceedings of the advances in neural information processing systems 33: annual conference on neural information processing systems 2020, NeurIPS 2020, December 6–12, virtual. Retrieved from https://proceedings.neurips.cc/paper/2020/hash/ea4eb49329550caaa1d2044105223721-Abstract.html
52. Angeline P, Saunders G, Pollack J (1994) An evolutionary algorithm that constructs recurrent neural networks. IEEE Trans Neural Netw 5(1):54–65. https://doi.org/10.1109/72.265960

53. Maas A, Daly R, Pham P, Huang D, Ng A, Potts C (2011) Learning word vectors for sentiment analysis. In: Proceedings of the 49th annual meeting of the association for computational linguistics: human language technologies. ACL, pp 142–150
54. Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin Z, Gimelshein N, Antiga L, Desmaison A, Köpf A, Yang E, DeVito Z, Raison M, Tejani A, Chilamkurthy S, Steiner B, Fang L, Bai J, Chintala S (2019) PyTorch: an imperative style, highperformance deep learning library. Advances in Neural Information Processing Systems. arXiv:1912.01703
55. Jozefowicz R, Vinyals O, Schuster M, Shazeer N, Wu Y (2016) Exploring the limits of language modeling. arXiv preprint arXiv:1602.02410
56. Jurafsky D, Martin J (2008) Speech and language processing. Prentice-Hall Inc, New Jersey
57. Nugaliyadde A, Sohel F, Wong KW, Xie H (2019) Language modeling through long-term memory network. In: Proceedings of the 2019 international joint conference on neural networks (IJCNN), Budapest, Hungary, pp 1–6. https://doi.org/10.1109/IJCNN.2019.8851909