Supplementary-Architecture Weight-Optimization Neural Networks

Jared O'Reilly[1*] and Nelishia Pillay[1]

[1]Department of Computer Science, University of Pretoria, Lynnwood Road, Pretoria, 0002, Gauteng, South Africa.

*Corresponding author(s). E-mail(s): u17051429@tuks.co.za; Contributing authors: nelishia.pillay@up.ac.za;

Abstract

Research efforts in the improvement of artificial neural networks have provided significant enhancements in learning ability, either through manual improvement by researchers or through automated design by other artificial intelligence techniques, and largely focusing on the architecture of the neural networks or the weight update equations used to optimize these architectures. However, a promising unexplored area involves extending the traditional definition of neural networks to allow a single neural network model to consist of multiple architectures, where one is a primary architecture and the others supplementary architectures. In order to use the information from all these architectures to possibly improve learning, weight update equations are customized per set-of-weights, and can each use the error of either the primary architecture or a supplementary architecture to update the values of that set-of-weights, with some necessary constraints to ensure valid updates. This concept was implemented and investigated. Grammatical evolution was used to make the complex architecture choices for each weight update equation, which succeeded in finding optimal choice combinations for classification and regression benchmark datasets, the KDD Cup 1999 intrusion detection dataset, and the UCLA graduate admission dataset. These optimal combinations were compared to traditional single-architecture neural networks, which they reliably outperformed at high confidence levels across all datasets. These optimal combinations were analysed using data mining tools, and this identified clear patterns, with theoretical explanation provided as to how these patterns may be linked to optimality. The optimal combinations were shown to be competitive with state-of-the-art techniques on the same datasets.

Keywords: artificial neural networks, weight update equations, supplementary architectures, neuro-evolution, grammatical evolution

1 Introduction

Researchers have poured efforts into identifying weaknesses in artificial neural networks, to remedy them with new constructs and mechanisms, and this process has yielded great improvements in learning ability, or 'performance'. These improvements have been manual, where human experts have identified and patched weaknesses with mathematical specifications and greater processing power. They have also been automatic through neuro-evolution, where other artificial intelligence techniques are used to search within some original or expanded aspect of neural networks and identify optimal solutions.

The aspects of neural networks which truly enable learning are the weight update equations. In this research, the implementation of specialized per-weight-set weight update equations is combined with an expansion to the traditional definition of a neural network, specifically, a single neural network model can consist of multiple architectures. One of these architectures is the primary architecture, which produces the error of the neural network, and any other architectures are supplementary architectures.

This is done by allowing the weight update equations for any set-of-weights in the model to use the error of either the primary architecture or one of the supplementary architectures to update weight values, with some constraints to ensure valid weight update equations. This combination has not been explored in current research, and has the potential to improve the learning ability of the neural network model, which would be a contribution to the field.

This improvement is not guaranteed, as various choices are required: what primary architecture should be used, which supplementary architectures should be used, and which architecture should each weight update equation use. These choices are complex and interlinked: therefore, the search technique of grammatical evolution will be used to identify optimal combinations of the choices above. The investigative objectives of this research are therefore:

1. To ascertain the effectiveness of combinations of primary architecture, supplementary architecture and weight update equation choices on multiple classification and regression problems.
2. To compare the performance of the standard neural network model, with a single weight update equation and architecture, to the performance of neural networks which can use supplementary architectures in their per-weight-set update equations, on the mentioned problems.
3. To analyse why the standard or expanded neural network model performs better on these problems, in particular, why particular combinations of architecture choices in the expanded model affect performance.

This research is presented in the following sections: Section 2 covers any related work that is needed for context, Section 3 presents the definition of this new technique, Section 4 specifies the experimental setup and design in order to fulfill the research objectives, Section 5 provides the results of these experiments, Section 6 analyses these results, Section 7 presents a comparison to state-of-the-art techniques, and Section 8 concludes with a clear link to the research objectives and possible future work.

2 Related work

No work is directly related, as the primary research objective is to investigate if the novel mechanism can improve neural network learning performance. However, concepts needed for context and motivation are introduced.

2.1 Manual design of the weight update equation

The stochastic gradient descent weight update equation used for neural networks is the following, where w_{xy} is any particular set of weights in the neural network that connects layer x to layer y, α is the learning rate, and E is the error function calculated on the architecture of the neural network:

$$w'_{xy} = w_{xy} - \alpha * \frac{dE}{dw_{xy}} \qquad \alpha \approx 0.01 \tag{1}$$

Improvement by researchers has been done manually, where weaknesses were identified and new mathematical constructs implemented as remedies, centered around two major ideas: the introduction of momentum into the weight update equation, and the introduction of per-weight learning rates.

The re-application of the previous weight update to simulate momentum [1] as well as the look-ahead gradient calculation embedded in the Nesterov accelerated gradient method [2] were both improvements with the primary objective of using momentum and its convergence-stabilising properties.

The idea of adaptive, per-weight learning rates which decreased for weights with frequent updates and increased for weights with infrequent updates was pioneered by Adagrad [3]. Issues with vanishing learning rates were remedied by both RMSprop [4] and Adadelta [5], which both use a decaying average of past squared gradients instead of maintaining and using a store of all past squared gradients to adapt learning rates.

The use of both momentum and adaptive per-weight learning rates was combined in Adam [6], which uses a decaying average of past squared gradients to adapt learning rates and a decaying average of past gradients to simulate momentum. Various improvements of Adam such as AdaMax [6], Nadam [7] and AMSGrad [8] were also derived, designed to remedy some flaw of Adam.

2.2 Automated design of the weight update equation

Automated improvement and/or reformulation of the weight update equation has also been investigated. Neuro-evolution is the primary sub-field of artificial intelligence devoted to this task, where neuro-evolution is simply defined as the use of evolutionary algorithms to optimize aspects of neural networks [9, 10].

The major groups of work within neuro-evolution include: the search for optimal weight values for fixed architectures [11–16]; the search for optimal architectures trained with gradient methods, also known as neural architecture search or NAS [17, 18]; the simultaneous search for both optimal architectures and optimal weights for these architectures [19, 20]; and the concept of synaptic-plasticity [21–28] and neuromodularity [29–31] to alter not only weight values but also architectures over the course of training.

The major group of work within neuro-evolution most relevant to this research is the evolution of learning rules [9, 10] i.e. weight update equations. These approaches automatically design the learning rules either directly in the learning rule space [32–34] or in another space used to generate learning rules [35, 36]. Common themes in this group of related work include: the use of Hebbian learning rules [37] as an alternative to gradient descent in various contexts [35, 38] as well as other synaptic-plasticity mimicking techniques [39, 40] which alter the architecture over time [41], and the evolution of local learning rules for neurons with various encoding schemes [38, 42].

The evolution of local learning rules is related to the concept of per-weight local learning rates utilized by previously mentioned optimizers [3–6], which led to significantly increased learning capability. Local learning rules can be applied at various levels of granularity e.g. for each neuron, for each weight, or perhaps for each set-of-weights between layers in the architecture, which should provide a good balance between search space size and optimality potential.

A possibility not yet explored in the field is that of using multiple distinct architectures in one model. NAS involves searching the space of possible architectures for one that is optimal for further gradient-based learning, but this assumes that a neural network's learning rules can only consider and use one architecture. Further, the concept of local learning rules for each set-of-weights in an architecture provides a surface onto which the use of multiple architectures can be applied. The weight update equations for each set-of-weights could be separately specified, and each weight update equation could somehow incorporate different architectures.

2.3 Grammatical evolution

Just as NAS searches the space of architectures, this new concept also needs a search, but in the space of multiple architectures and their combined use. Evolutionary algorithms like genetic algorithms or other grammar-based search techniques such as grammatical evolution (GE) [43] can be used. GE evolves programs defined by a target grammar using a binary string of variable length, utilizing a mapping process which converts binary strings maintained in the population, genotypes/chromosomes, into valid programs from the target grammar, phenotypes. These programs are evaluated by a fitness function, relating a fitness value to the chromosome used to generate the program [43].

Grammatical evolution has the following significant advantage over other evolutionary algorithms [43]: other algorithms search directly in the program space, so their genetic operators must be designed to produce valid programs. GE searches within a binary string space which is mapped onto the program space, and this mapping always produces valid programs. Therefore, standard bitstring genetic operators can be used to move the search, which require less manual design than validity-preserving genetic operators working directly in the program space, which enables an unconstrained search of the genotype [43].

3 Supplementary Architecture Weight Optimization

Based on current research, a new optimization algorithm for neural networks is proposed: Supplementary Architecture Weight Optimization, or SAWO. This algorithm extends traditional gradient descent weight optimization by introducing supplementary architectures alongside the primary neural architecture. The errors that the supplementary architectures produce can be used to update particular weight sets in the primary architecture, rather than the error of the primary architecture. The choices of which weight sets to do this for, and which supplementary architectures to use, are decided by grammatical evolution.

3.1 Architectures: definition and construction

Neural network architectures are made up of layers of neurons, which are connected sequentially using sets of weights. Definitions given below:

Definition 1 (Layer set L) In the architectures of this research, 5 different layers can be used in an architecture, and they are defined in the layer set L:

$$L = \{i, a, b, c, o\} \tag{2}$$

Definition 2 (Layer size and types) Each layer contains a number of neurons, known as the size of the layer, and this can be retrieved using the cardinality operator i.e. the size of $x \in L$ is $|x|$, with $|x| \in \mathbb{N}$. Each of these layers is one of 3 types, defined by how they can be connected to other layers. i is an input layer, and can only connect to other layers. a, b and c are hidden layers, which can connect to other layers and be connected to by other layers, but not to themselves. o is an output layer, which can only be connected to by other layers.

Definition 3 (Set-of-weights set W) Every possible connection between layers will involve a set-of-weights. Therefore, the set of all possible sets-of-weights W is defined, bearing in mind the connective possibilities of layers from Definition 2:

$$W = \{ \quad w_{xy} \quad | \quad x \in \{i, a, b, c\},\ y \in \{a, b, c, o\},\ x \neq y \quad \} \tag{3}$$

Definition 4 (Set-of-weights dimensions) The dimensions of a set of weights are the sizes of the layers it connects i.e. w_{xy} between layers x and y has dimensions $|x|$ x $|y|$.

Definition 5 (Architecture) An architecture is defined as an n-tuple, where every element of the n-tuple is a set that contains 1 or more layers from L. For an architecture d, the k-th element of d is denoted as d_k. The following constraints apply:

- d_1 can only contain i, i.e. $d_1 = \{i\}$ and d_n can only contain o, i.e. $d_n = \{o\}$
- The other elements of d can contain input (i) and hidden (a, b, c) layers, but each hidden layer can only appear once in an architecture.

Definition 6 (Architecture set A) The architecture set A contains all possible valid architectures according to Definition 5, and is given in Appendix A.

Some examples from the full architecture set A found in Appendix A are given in Table 1 in the correct representation of an n-tuple of sets, as well as shorthand representations, used in the appendix and throughout this research.

Table 1 Examples from architecture set A

Example	Shorthand
$(\{i\}, \{o\})$	$i \cdot o$
$(\{i\}, \{a\}, \{c\}, \{o\})$	$i \cdot a \cdot c \cdot o$
$(\{i\}, \{b\}, \{i, a\}, \{o\})$	$i \cdot b \cdot \|ia\| \cdot o$
$(\{i\}, \{i, c\}, \{a, b\}, \{o\})$	$i \cdot \|ic\| \cdot \|ab\| \cdot o$

The use of parallel layers, like in $i \cdot |ic| \cdot |ab| \cdot o$, brings up the question of how these layers are connected together by sets-of-weights. The rules used to construct an architecture d are presented below:

Definition 7 (Architecture construction) The sets-of-weights connecting d_i and d_{i+1} with $i < n$ is given by the following rules:

1. If d_i contains a single layer x
and d_{i+1} contains a single layer y,
then x and y will simply be connected using w_{xy}

2. If d_i contains a single layer x
and d_{i+1} contains J non-input layers $y_1, ..., y_j, ..., y_J$ with $J > 1$,
then x will be connected to each y_j with w_{xy_j}

3. If d_i contains J layers $x_1, ..., x_j, ..., x_J$ with $J > 1$
and d_{i+1} contains K non-input layers $y_1, ..., y_k, ..., y_K$ with $K >= 1$,
we perform the following:
Consider x_j. We need to connect x_j to every layer in $y_1, ..., y_k, ..., y_K$.
Therefore, we will create layers $z_{j1}, ..., z_{jk}, ..., z_{jK}$, which are temporary copies of $y_1, ..., y_k, ..., y_K$ for x_j to use. As they are copies, $z_{jk} = y_k$.
Each x_j is then connected to each z_{jk} using the set of weights $w_{x_j y_k}$.
Now that the x layers are connected to the z layers, the z layers must be connected to the y layers. An Average operation is used, which averages values from multiple layers of the same size in an element-wise fashion and inputs them to the next layer. Specifically, we use the Average operation to average element-wise the values of

$z_{1k}, ..., z_{jk}, ..., z_{Jk}$ and feed this into layer y_k. All of $z_{1k}, ..., z_{jk}, ..., z_{Jk}$ are temporary copies of y_k, and so will be the same size.

For the architectures of this research, a **linear activation function** is used for all neuron outputs. This is the "canonical" neural network activation function, so is suitable for assessing the potential of this novel approach. More recent non-linear activation functions exist, but choosing a single one for this work is complex, and a wrong choice could bias results. The use of different activation functions is discussed in Section 8. To illustrate the use of the rules in Definition 7, three examples architectures and a diagrammatic representation of their connections between layers are given in Figures 1, 2 and 3.

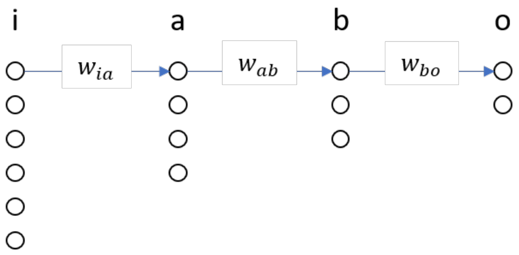

Fig. 1 Diagrammatic representation of the architecture $i \cdot a \cdot b \cdot o$

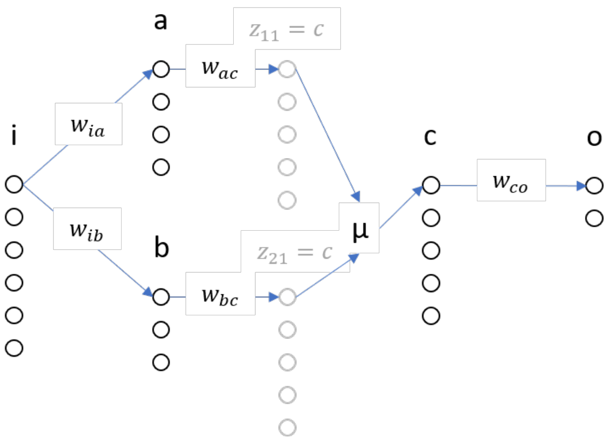

Fig. 2 Diagrammatic representation of the architecture $i \cdot |ab| \cdot c \cdot o$

The use of linear activation functions influences the choice of which element-wise arithmetic operation to use in the third rule of Definition 7, as this activation function is predisposed to forward-propagation saturation. Viable candidates for this operation are Add, Multiply, Min, Max and Average.

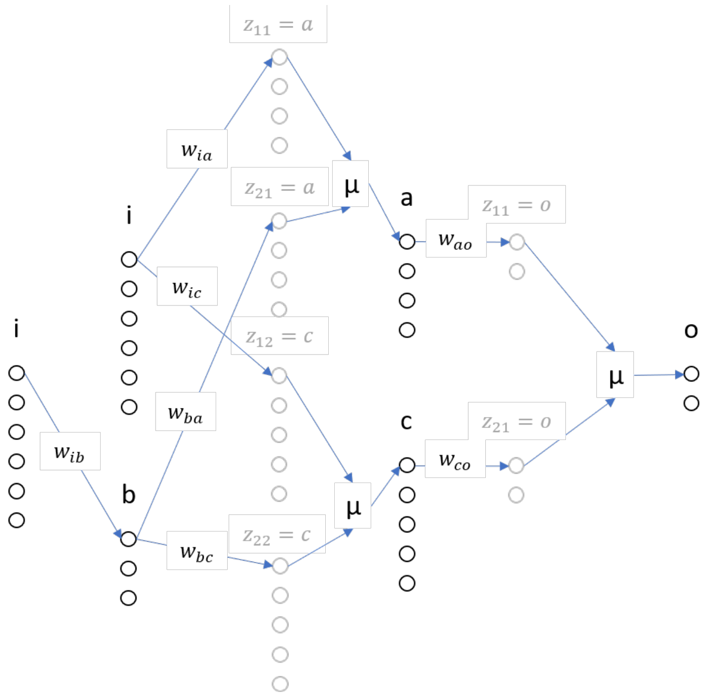

Fig. 3 Diagrammatic representation of the architecture $i \cdot |ib| \cdot |ac| \cdot o$

Add, and especially Multiply, would certainly cause saturation, as forward-propagated values would rapidly grow in magnitude, and this was seen in trial runs. Min and Max are less likely to cause saturation, but these operations only consider one of the element-wise values, meaning entire streams of encoded knowledge from previous layers are discarded.

Average is unlikely to cause saturation, as the average of element-wise values will always be smaller than the largest element-wise value. Every element-wise value is used in this calculation, so all streams of encoded knowledge are used to some extent and none are discarded. These two attributes of the Average operation make it suitable for use with a linear activation function.

Definitions to extract information from a particular architecture or set-of-weights are given, which simplify further definitions and algorithms:

Definition 8 (Set-of-weights appearing in an architecture) For a particular architecture $d \in A$, we have the following condition relating to whether the set of weights $w_{xy} \in W$ appears in d:

$$\exists j : x \in d_j \land y \in d_{j+1} \quad \Rightarrow \quad w_{xy} \quad \text{appears in} \quad d \tag{4}$$

For example, w_{co} appears in $i \cdot a \cdot c \cdot o$ and w_{ab} appears in $i \cdot |ia| \cdot |bc| \cdot o$, whereas w_{ib} does not appear in $i \cdot a \cdot o$ and w_{bo} does not appear in $i \cdot |ib| \cdot a \cdot o$.

Definition 9 (Set-of-weights architecture subset A_{xy}) We can create subsets of the architecture set A for every set of weights in W. Each set-of-weights subset contains all the architectures from A that that set of weights appears in. We therefore define the set-of-weights architecture subset A_{xy} like so:

$$\forall w_{xy} \in W \quad : \quad A_{xy} = \{ \quad d \quad | \quad d \in A,\ w_{xy} \text{ appears in } d \quad \} \tag{5}$$

For example, let us find $A_{ac} = \{ \quad d \quad | \quad d \in A,\ w_{ac} \text{ appears in } d \quad \}$

$$= \{ \ i \cdot a \cdot c \cdot o,\ \ i \cdot |ia| \cdot c \cdot o,\ \ i \cdot a \cdot |ic| \cdot o,\ \ i \cdot |ia| \cdot |ic| \cdot o,\ \ i \cdot |ab| \cdot c \cdot o,\ \ i \cdot |iab| \cdot c \cdot o,$$
$$i \cdot |ab| \cdot |ic| \cdot o,\ \ i \cdot a \cdot |bc| \cdot o,\ \ i \cdot |ia| \cdot |bc| \cdot o,\ \ i \cdot a \cdot |ibc| \cdot o,\ \ i \cdot a \cdot c \cdot b \cdot o,\ \ i \cdot |ia| \cdot c \cdot b \cdot o,$$
$$i \cdot a \cdot |ic| \cdot b \cdot o,\ \ i \cdot a \cdot c \cdot |ib| \cdot o,\ \ i \cdot |ia| \cdot |ic| \cdot b \cdot o,\ \ i \cdot |ia| \cdot c \cdot |ib| \cdot o,\ \ i \cdot a \cdot |ic| \cdot |ib| \cdot o,\ \ i \cdot b \cdot a \cdot c \cdot o,$$
$$i \cdot |ib| \cdot a \cdot c \cdot o,\ \ i \cdot b \cdot |ia| \cdot c \cdot o,\ \ i \cdot b \cdot a \cdot |ic| \cdot o,\ \ i \cdot |ib| \cdot |ia| \cdot c \cdot o,\ \ i \cdot |ib| \cdot a \cdot |ic| \cdot o,\ \ i \cdot b \cdot |ia| \cdot |ic| \cdot o \ \}$$

Definition 10 (Architecture set-of-weights subset W_d) We can also create subsets of the set-of-weights set W for every architecture in A. Each architecture subset contains all the sets-of-weights from W that appear in that architecture. We therefore define the architecture set-of-weights subset W_d like so:

$$\forall d \in A \quad : \quad W_d = \{ \quad w_{xy} \quad | \quad w_{xy} \in W,\ w_{xy} \text{ appears in } d \quad \} \tag{6}$$

For example, let us find

$$W_{i \cdot |ia| \cdot |bc| \cdot o} = \{ \quad w_{xy} \quad | \quad w_{xy} \in W,\ w_{xy} \text{ appears in } i \cdot |ia| \cdot |bc| \cdot o \quad \}$$
$$= \{ \ w_{ia},\ \ w_{ib},\ \ w_{ic},\ \ w_{ab},\ \ w_{bc},\ \ w_{bo},\ \ w_{co} \ \}$$

3.2 Extending the weight update equation

A neural network has two basic aspects: its architecture with weights, and the learning rules used to update the weights i.e. the weight update equations (WUEs). For a set-of-weights w_{xy} between layer x and y in an architecture d, the stochastic gradient descent WUE for that set-of-weights is the following:

$$w'_{xy} = w_{xy} - \alpha * \frac{dE}{dw_{xy}} \tag{7}$$

α represents the learning rate, and E is the error function, which calculates the differences between actual target values and the target values that the architecture d's weights produce by forward-propagation. Stochastic gradient descent is the "canonical" gradient-descent optimizer for neural networks, so is used in this research to assess the potential of this novel approach and establish proof-of-concept. The use of more recent optimizers is discussed in Section 8.

Let us consider architecture $i \cdot a \cdot b \cdot o$, which is drawn in Figure 1. The sets-of-weights used in this architecture is given by equation 6: $W_{i \cdot a \cdot b \cdot o} = \{w_{ia}, w_{ab}, w_{bo}\}$. Therefore, the set of WUEs needed for this architecture is:

$$w'_{ia} = w_{ia} - \alpha * \frac{dE}{dw_{ia}}$$
$$w'_{ab} = w_{ab} - \alpha * \frac{dE}{dw_{ab}}$$

$$w'_{bo} = w_{bo} - \alpha * \frac{dE}{dw_{bo}} \tag{8}$$

These weight update equations are now further specified. The error function, E, is implicitly referring to the error produced by a particular architecture. Let us more explicitly denote this with a definition:

Definition 11 (Error of an architecture E_d) The error function E, such as mean-squared error or cross-entropy error, formulated according to the structure of architecture $d \in A$ is explicitly referred to as E_d, rather than just E.
For example, $E_{i \cdot c \cdot o}$ is the error function E formulated for architecture $i \cdot c \cdot o$.

In this case, the architecture is $i \cdot a \cdot b \cdot o$, so the WUEs are changed to:

$$\begin{aligned} w'_{ia} &= w_{ia} - \alpha * \frac{dE_{i \cdot a \cdot b \cdot o}}{dw_{ia}} \\ w'_{ab} &= w_{ab} - \alpha * \frac{dE_{i \cdot a \cdot b \cdot o}}{dw_{ab}} \\ w'_{bo} &= w_{bo} - \alpha * \frac{dE_{i \cdot a \cdot b \cdot o}}{dw_{bo}} \end{aligned} \tag{9}$$

At this point, the core question of this research can be phrased: what would happen if some other "supplementary" architecture $g \in A$ was used to produce error instead of $i \cdot a \cdot b \cdot o$ for one or more of these equations? For example, what if the architecture $i \cdot a \cdot |ib| \cdot o$ was introduced and used instead of $i \cdot a \cdot b \cdot o$ for the WUE of w_{ab}? That would transform the WUEs to:

$$\begin{aligned} w'_{ia} &= w_{ia} - \alpha * \frac{dE_{i \cdot a \cdot b \cdot o}}{dw_{ia}} \\ w'_{ab} &= w_{ab} - \alpha * \frac{dE_{i \cdot a \cdot |ib| \cdot o}}{dw_{ab}} \\ w'_{bo} &= w_{bo} - \alpha * \frac{dE_{i \cdot a \cdot b \cdot o}}{dw_{bo}} \end{aligned} \tag{10}$$

There are two complications introduced by this change:

1. Previously, all needed sets-of-weights were known, through $W_{i \cdot a \cdot b \cdot o} = \{w_{ia}, w_{ab}, w_{bo}\}$, and they each had WUEs. But with the introduction of the new supplementary architecture $i \cdot a \cdot |ib| \cdot o$ with $W_{i \cdot a \cdot |ib| \cdot o} = \{w_{ia}, w_{ab}, w_{io}, w_{bo}\}$, an additional set-of-weights is introduced: w_{io}. This set-of-weights needs its own WUE in order to contribute to learning.
2. Not every supplementary architecture from A can be used in the error for a particular WUE.
   - The architecture chosen above, $i \cdot a \cdot |ib| \cdot o$, is suitable for use in the WUE for w_{ab} because w_{ab} appears in $i \cdot a \cdot |ib| \cdot o$ according to Definition 8. Therefore, the weight values in w_{ab} will be used in the calculation of $E_{i \cdot a \cdot |ib| \cdot o}$, and so the error derivative $\frac{dE_{i \cdot a \cdot |ib| \cdot o}}{dw_{ab}}$ will contain meaningful values and be of the same dimensionality as w_{ab}. This ensures meaningful and valid updates.

- An architecture such as $i \cdot a \cdot o$ could not be used for the WUE of w_{ab}, as w_{ab} does not appear in $i \cdot a \cdot o$ and so the weight values in w_{ab} are not used in the calculation of $E_{i \cdot a \cdot o}$. Therefore, the error derivative $\frac{dE_{i \cdot a \cdot o}}{dw_{ab}}$ will simply evaluate to 0, much like the derivative $\frac{d}{dz}(x^2 + 1)$ would evaluate to 0, and so the values in w_{ab} would never change from their initial values.

Considering these complications, the WUEs must be further transformed. Considering complication 1, a new set-of-weights w_{io} is introduced and given its own WUE. Considering complication 2, an appropriate supplementary architecture must be chosen for this new WUE. The criterion for this choice is that w_{io} must appear in the supplementary architecture. This criterion fits well with the definition of equation 5, and so we find A_{io} according to equation 5, and choose the architecture $i \cdot o \in A_{io}$. Any architecture in A_{io} could be chosen, but $i \cdot o$ ensures no further sets-of-weights are introduced, as $W_{i \cdot o} = \{w_{io}\}$, which is ideal to end this example. The WUEs are therefore transformed to:

$$\begin{aligned} w'_{ia} &= w_{ia} - \alpha * \frac{dE_{i \cdot a \cdot b \cdot o}}{dw_{ia}} \\ w'_{ab} &= w_{ab} - \alpha * \frac{dE_{i \cdot a \cdot |ib| \cdot o}}{dw_{ab}} \\ w'_{io} &= w_{io} - \alpha * \frac{dE_{i \cdot o}}{dw_{io}} \\ w'_{bo} &= w_{bo} - \alpha * \frac{dE_{i \cdot a \cdot b \cdot o}}{dw_{bo}} \end{aligned} \tag{11}$$

Now that all required WUEs are defined, and each WUE will evaluate meaningfully, the following question is pertinent: what will happen if the transformed WUEs from equation set 11 are used in the training of the neural network, rather than the original WUEs from equation set 9? If decreasing the error of the original (or "primary") architecture $i \cdot a \cdot b \cdot o$ is the goal of training, which set of WUEs will better achieve this goal? This question is difficult to answer theoretically, so we need to train using both sets of WUEs separately and compare their performance.

But regardless of the outcome of this comparison, another question arises: could better supplementary architectures be chosen than the ones above, which would better achieve this goal? And what if different sets-of-weights were chosen to use a supplementary architecture in their WUE? These choices are complex and numerous, and a human expert would struggle to consider them all. For this reason, a grammar is created to define all possible weight-and-architecture choices, and grammatical evolution is used to search this possible space of choices to find those which achieve the goal of training best.

Henceforth, a neural network with a primary architecture that can use supplementary architectures for weight updates is defined as a supplementary-architecture weight-optimization neural network, or **SAWO-NN**.

3.3 Grammar of possible SAWO-NNs

A grammar defining a WUE for any set-of-weights w_{xy} is specified, denoted G_{xy}. This grammar is instantiated for every set-of-weights in W from equation 3. An overall grammar is then specified, denoted as G, which uses these grammars to describe a neural network using supplementary architectures.

The grammar G_{xy} for a set-of-weights w_{xy} is given below:

$$\begin{aligned}
\langle start \rangle &::= w'_{xy} = w_{xy} \text{ - } \langle constant \rangle \text{ * dE } \langle architecture \rangle \text{ / d } w_{xy} \\
\langle constant \rangle &::= 0.01 \mid 0.1 \mid 0.25 \mid 0.33 \mid 0.5 \mid 0.66 \mid 0.75 \mid 0.9 \mid 0.99 \\
\langle architecture \rangle &::= g \in A_{xy}
\end{aligned}$$

This grammar encodes choices of which constant learning rate and which architecture from the A_{xy} set to use for the WUE of w_{xy}. This grammar is instantiated for every $w_{xy} \in W$, giving $G_{ia}, G_{ib}, ..., G_{co}$. Consider the grammar G_{ba}. A few example instances of G_{ba} are:

- $w'_{ba} = w_{ba}$ - 0.25 * dE $i \cdot b \cdot a \cdot o$ / d w_{ba}
- $w'_{ba} = w_{ba}$ - 0.66 * dE $i \cdot |ib| \cdot a \cdot o$ / d w_{ba}
- $w'_{ba} = w_{ba}$ - 0.01 * dE $i \cdot c \cdot b \cdot |ia| \cdot o$ / d w_{ba}

In this research, the error used for all weight update equations is **mean-squared error** or MSE, for simplicity sake. The grammar G is given below, with instances of this grammar encoding all information needed to instantiate a SAWO-NN. Layer sizes are restricted to 20 to keep training times short.

$$\begin{aligned}
\langle start \rangle &::= \langle layers \rangle\ \langle weightupdates \rangle\ \langle primary \rangle \\
\langle layers \rangle &::= a{:}\langle size \rangle,\ b{:}\langle size \rangle,\ c{:}\langle size \rangle \\
\langle size \rangle &::= n \in \{2...20\} \\
\langle weightupdates \rangle &::= e_{ia} \in L(G_{ia}),\ ...,\ e_{xy} \in L(G_{xy}),\ ...,\ e_{co} \in L(G_{co}) \\
\langle primary \rangle &::= d \in A
\end{aligned}$$

The sizes of the a, b and c layers are encoded using the *layers* rule. The primary architecture is encoded using the *primary* rule. A WUE for all sets-of-weights in W is chosen from the language of the corresponding grammar and encoded using the *weightupdates* rule, with the WUE chosen from grammar G_{xy} denoted as e_{xy}. Note that every set-of-weights in W has a WUE chosen, even if that set-of-weights is not needed to train the primary architecture. This choice will be elaborated when the training procedure is defined.

Here is an example of a valid instance of the G grammar:

$$
\begin{aligned}
& a:16, b:4, c:9 \\
& w'_{ia} = w_{ia} - 0.33 * \frac{dE\ i \cdot a \cdot o}{dw_{ia}} \\
& w'_{ib} = w_{ib} - 0.25 * \frac{dE\ i \cdot b \cdot a \cdot o}{dw_{ib}} \\
& w'_{ic} = w_{ic} - 0.75 * \frac{dE\ i \cdot |ib| \cdot c \cdot o}{dw_{ic}} \\
& w'_{io} = w_{io} - 0.01 * \frac{dE\ i \cdot |ic| \cdot o}{dw_{io}} \\
& w'_{ab} = w_{ab} - 0.1 * \frac{dE\ i \cdot |ac| \cdot |ib| \cdot o}{dw_{ab}} \\
& w'_{ac} = w_{ac} - 0.99 * \frac{dE\ i \cdot a \cdot c \cdot o}{dw_{ac}}
\end{aligned}
\qquad
\begin{aligned}
& w'_{ao} = w_{ao} - 0.66 * \frac{dE\ i \cdot |ibc| \cdot a \cdot o}{dw_{ao}} \\
& w'_{ba} = w_{ba} - 0.5 * \frac{dE\ i \cdot b \cdot |ia| \cdot o}{dw_{ba}} \\
& w'_{bc} = w_{bc} - 0.1 * \frac{dE\ i \cdot b \cdot c \cdot o}{dw_{bc}} \\
& w'_{bo} = w_{bo} - 0.9 * \frac{dE\ i \cdot |ia| \cdot c \cdot |ib| \cdot o}{dw_{bo}} \\
& w'_{ca} = w_{ca} - 0.25 * \frac{dE\ i \cdot |ibc| \cdot a \cdot o}{dw_{ca}} \\
& w'_{cb} = w_{cb} - 0.33 * \frac{dE\ i \cdot c \cdot b \cdot o}{dw_{cb}} \\
& w'_{co} = w_{co} - 0.66 * \frac{dE\ i \cdot |ab| \cdot |ic| \cdot o}{dw_{co}} \\
& i \cdot |ia| \cdot c \cdot |ib| \cdot o
\end{aligned}
$$

Note that the architectures chosen in each WUE can possibly be the same as the primary architecture, such as for w_{bo} above. If this is true for every set-of-weights that appears in the primary architecture, then we essentially have standard gradient descent, similar to the configuration of equation set 9.

3.4 Training and using a SAWO-NN

The instances of G fully describe a particular SAWO-NN, but not how they are trained, with the goal of decreasing the error produced by the primary architecture. This training process is described, for some instance of G.

Before the neural network can be trained, the needed sets-of-weights must be established, as well as the architectures used to update these sets-of-weights. As mentioned, not every set-of-weights WUE is needed for every instance of G, but every WUE is encoded by the grammar: it is easier to discard excess structure than to generate the exact amount of structure needed.

Before presenting the algorithm to do this, a useful definition is given, which extracts the architecture used in a particular WUE:

Definition 12 (Architecture used in a WUE α_{xy}) If e_{xy} is the WUE for set-of-weights w_{xy}, then:

$$\alpha_{xy} = \{\ d \mid d \in A\ ,\ d \text{ is used in the error derivative term of } e_{xy}\ \} \qquad (12)$$

For example, for e_{ic} chosen as $w'_{ic} = w_{ic} - 0.75 * \frac{dE\ i \cdot c \cdot o}{dw_{ic}}$, we have $\alpha_{ic} = \{i \cdot c \cdot o\}$, and for e_{ao} chosen as $w'_{ao} = w_{ao} - 0.33 * \frac{dE\ i \cdot b \cdot a \cdot o}{dw_{ao}}$, we have $\alpha_{ao} = \{i \cdot b \cdot a \cdot o\}$.

The algorithm to find the needed sets-of-weights W' and needed architectures A' is given:

```
1. Initialise a directed graph. Create a node for every weight set in $W$.
2. Perform the following for each $w_{xy} \in W$:
   (a) Consider the WUE for $w_{xy}$ i.e. $e_{xy}$. Find the architecture used in
       this WUE, in other words, find $\alpha_{xy}$.
   (b) For the single architecture $d$ in $\alpha_{xy}$, establish which sets-of-weights
       appear in $d$. In other words, find $W_d$.
   (c) For every set-of-weights $w_{ij} \in W_d$, if one does not exist, draw a
       directed edge in the graph from the $w_{xy}$ node to the $w_{ij}$ node.
3. Create an empty set $W'$, the needed set of weights set.
4. Consider the primary architecture, denoted $p$. Establish which sets-of-
   weights appear in $p$ i.e. find $W_p$. Perform the following for every weight
   set $w_{xy} \in W_p$:
   (a) Consider the $w_{xy}$ node in the directed graph. Find all nodes that
       are connected to this $w_{xy}$ node by a directed walk i.e. they are
       reachable from the $w_{xy}$ node.
   (b) Add the set-of-weights for each of these nodes to the set $W'$.
5. Create an empty set $A'$, the needed architectures set. Add the primary
   architecture $p$ to $A'$.
6. Perform the following for each $w_{xy} \in W'$:
   (a) Consider the WUE for $w_{xy}$ i.e. $e_{xy}$. Find the architecture used in
       this WUE, in other words, find $\alpha_{xy}$.
   (b) Add the single architecture $d$ in $\alpha_{xy}$ to $A'$.
7. We now have our set of needed sets-of-weights $W'$ and our set of needed
   architectures $A'$. Randomly initialise all sets of weights in $W'$.
```

For the example instance of G grammar provided previously, the elements of W' and A' are derived and given below. In this research, all weights in W' are randomly initialized using a **normal distribution** with mean of 0 and standard deviation of 0.2 i.e. $\mathcal{N}(0, 0.2)$:

$$W' = \{\ w_{ia},\ w_{ib},\ w_{ic},\ w_{io},\ w_{ac},\ w_{ao},\ w_{ba},\ w_{bc},\ w_{bo},\ w_{ca},\ w_{cb},\ w_{co}\ \}$$

$$\begin{gathered} A' = \{\quad i \cdot |ia| \cdot c \cdot |ib| \cdot o, \quad i \cdot a \cdot o, \quad i \cdot b \cdot a \cdot o, \quad i \cdot |ib| \cdot c \cdot o, \\ i \cdot |ic| \cdot o, \quad i \cdot a \cdot c \cdot o, \quad i \cdot |ibc| \cdot a \cdot o, \quad i \cdot b \cdot |ia| \cdot o, \\ i \cdot b \cdot c \cdot o, \quad i \cdot |ibc| \cdot a \cdot o, \quad i \cdot c \cdot b \cdot o, \quad i \cdot |ab| \cdot |ic| \cdot o \quad \} \end{gathered}$$

The algorithm for training a SAWO-NN is given. This algorithm takes as input: the W' and A' sets derived previously; all information encoded in the grammar instance such as the primary architecture p; all WUEs; and the dataset which the SAWO-NN is being optimized for in two parts: the training subset and validation subset. When error needs to be calculated, **mean-squared error** is again used. See the algorithm below:

1. Construct all architectures in A' according to Definition 7. Then, to initialise the values of the sets-of-weights in the architectures, make shallow copies of the corresponding sets-of-weights in W'.
2. Calculate either the accuracy (classification) or error (regression) of p on the validation subset. Store this value as *best-perf* and the current values of the sets-of-weights in W' as *best-perf-weights*.
3. Repeat the following for n epochs:
   - (a) Shuffle the training subset, and then split it evenly into a certain number of training mini-batches, denoted as T.
   - (b) Perform the following for each training mini-batch $t_i \in T$:
     - (i) Evaluate each WUE of the weight sets in W', thereby updating these weight sets in W'. When an error partial derivative is evaluated, use the required architecture from A' on the training mini-batch t_i to calculate error.
     - (ii) Remove all constructed architectures.
     - (iii) Re-construct all architectures in A', and initialise weight values with shallow copies of the newly updated sets-of-weights in W'.
   - (c) Calculate either the accuracy or error of p on the validation subset. If this is better than *best-perf*, store this value in *best-perf* and the current values in the sets-of-weights in W' in *best-perf-weights*.
4. Return *best-perf*, *best-perf-weights*, and p.

The returned *best-perf* represents the SAWO-NN at its best generalization ability throughout training. One can reproduce this using the returned primary architecture p and *best-perf-weights*, which contains the set-of-weights values to use in p to produce *best-perf*. The increasing (classification) or decreasing (regression) of *best-perf* is regarded as the goal of training.

Note that this training algorithm closely mimics the training algorithm of standard gradient-descent weight-optimization. Indeed, if every set-of-weights in the primary architecture p uses p in its WUE, then regardless of the other architectures choices, standard gradient-descent will occur, as W' will be equal to W_p and A' will only contain p. This algorithm would then just update the sets-of-weights in the primary architecture, using training mini-batches over a number of training epochs. This can occur by chance when constructing an instance from the grammar, for any of the primary architectures in A. This fact will be used to inform initial population generation.

At this stage, a SAWO-NN is well defined in terms of configurations and usage. However, the exact configurations which would optimize learning ability and subsequent usage are unknown. It is also unknown whether a SAWO-NN can learn more optimally than the standard gradient-descent approach which it extends. To establish these unknowns, which correspond to research objective 1 and 2 respectively, an experiment is conducted: find optimal configurations of SAWO-NN on different tasks, represented by datasets, and then compare these optimal SAWO-NNs to their standard gradient-descent counterparts on these tasks. This is the focus of the rest of this research.

4 Experimental setup

The specifics of this experiment are presented. The datasets which the techniques will be evaluated on are provided first. For each dataset, grammatical evolution (GE) is used to search the space of SAWO-NNs defined by the grammar G. SAWO-NNs are evaluated using their performance on these datasets after training. The general design and hyperparameters of GE is also provided, as well as how many times GE will be run and how these results are aggregated.

4.1 Datasets

SAWO-NNs are capable of classification and regression, so the datasets chosen represent both problem domains: well-known benchmark data sets from the binary classification, multi-class classification and regression fields, and lesser-known real-world datasets tied to specific application domains. This will give a good picture of the capabilities of SAWO-NNs. See Appendix B for more information on the source and pre-processing of these datasets.

4.1.1 Benchmark problems

The datasets in this section are extensively used in machine learning literature, and are considered benchmark datasets which new techniques should be tested against. Table 2 presents the chosen binary classification datasets, Table 3 presents the chosen multi-class classification datasets, and Table 4 presents the chosen regression datasets.

Table 2 Binary-class classification datasets

ID	Dataset	Rows	Features	Class Dist %	Domain
1	Breast Cancer Wisconsin (Diagnostic)	569	32	B=63 M=37	healthcare, cancer
2	Mushroom Classification	8124	23	e=52 p=48	nature, public safety
3	Heart Attack Analysis and Prediction	303	14	0=46 1=54	healthcare, heart conditions

4.1.2 Real-world application problems

The datasets in this section represent real-world problems, which have been encoded into dataset format for machine learning. The inclusion of these datasets is for establishing the suitability of SAWO-NN for difficult real-world problems. Table 5 presents the chosen real-world application datasets.

Table 3 Multi-class classification datasets

ID	Dataset	Rows	Features	Class Dist %	Domain
4	Iris Species	150	5	0 to 2: 33, 33, 33	earth & nature, biology
5	Red Wine Quality	1599	12	3 to 8: 1, 3, 43, 40, 13, 1	chemistry, alcohol
6	Glass Classification	214	10	1 to 7: 33, 36, 8, 0, 6, 4, 14	industrial, chemistry
7	Wheat Seeds	210	8	0 to 2: 33, 33, 33	earth & nature, agriculture

Table 4 Regression datasets

ID	Dataset	Rows	Features	Mean Output Value	Domain
8	Boston House Price	505	11	22.53	real estate, social science
9	Abalone Rings	4177	9	9.93	earth & nature, biology
10	1985 Automobile Insurance	201	29	121.43	insurance, vehicles

Table 5 Real-world application datasets

ID	Dataset	Rows	Features	Problem Type	Domain
11	KDDCup 99 Intrusion Detection	25k	42	Binary classification	intrusion detection, networks
12	Graduate Admission	400	9	Regression	education, universities & colleges

4.2 GE design

A generational model is used, where a new generation is entirely produced using genetic operators. The stopping criteria is a set number of generations. This GE design was chosen to achieve good results across all datasets.

4.2.1 Initial population generation

Chromosome bitstrings are usually randomly generated, however, experimental runs struggled to find optimal SAWO-NNs. This is because a randomly generated chromosome will generate random supplementary architectures for all WUEs, which may not be the same as, or similar to, the primary architecture. It was observed that the use of all of these supplementary architectures at once was not effective at decreasing the error of the primary architecture.

Some optimal SAWO-NNs were found, but they were needles in a haystack, with one common property: they only had a few WUEs using supplementary architectures, with the majority of the WUEs using the primary architecture. This optimality pattern makes sense: single-architecture gradient-descent is known to be effective for learning, and so using different architectures for almost all WUEs would deviate from this known point of optimality.

Therefore, an informed initial population is used. This means that the initial population is not just randomly generated, rather, a known heuristic is used to guide the random generation. Members in the initial population will already have some optimality, and further evolution of this initial population should produce more optimal members.

Instead of randomly generating bitstrings, a set of template bitstrings is defined, one for each possible primary architecture in A. Each template bitstring has some bits filled, which is determined by the corresponding primary architecture. The primary architecture choice is filled to map to the primary architecture. The sets-of-weights in the primary architecture are established, and the supplementary architecture choices for each of those WUEs are also filled to map to the primary architecture. Any other bits are left as ”empty”.

This filling strategy ensures that if any template bitstring is chosen, and ”empty” bits randomly filled, the chromosome would map to standard single-architecture gradient-descent in G. This filling strategy works because the number of codons needed can be exactly determined, which is detailed later. However, members of the population should deviate slightly from this configuration, using a few supplementary architectures rather than none.

Therefore, when a new member of the population must be created, a template is randomly chosen. 1 or 2 bits are randomly flipped in this template, and then the ”empty” bits are randomly filled. This ensures that the bitstrings do not map to single-architecture gradient-descent on the corresponding primary architecture, rather, they will slightly differ and use about 1 or 2 supplementary architectures. These generated chromosomes therefore display the same optimality pattern as the needles in the haystack mentioned previously, which is the heuristic being used for this informed initial population.

Lastly, because the performance of a multi-point search like GE is largely determined by the starting points in the search space, a larger initial population is used. The best members selected from this initial population are placed into the smaller second generation, which is therefore more likely to contain optimal members, providing a better starting point.

4.2.2 Fitness function

The optimality of a SAWO-NN should be based on its performance on testing data after training i.e. generalization ability. The returned value of *best-perf* from the training algorithm represents this measure, however, the value is stochastic in nature, as the weight values for the SAWO-NN are initialized randomly. Therefore, multiple trainings will take place, and the average value of *best-perf* used to represent fitness.

For classification, a higher *best-perf* indicates better fitness, as this represents validation accuracy. For regression, a lower *best-perf* indicates better fitness, as this represents validation error. The number of trainings per fitness evaluation must be chosen carefully: a larger number of trainings gives a better picture of fitness, but takes longer to execute. An appropriate number of trainings will be chosen and specified in the hyperparameters section.

4.2.3 Selection methods and genetic operators

Two selections methods are used: m-elitism, which selects the best m members from the current generation, and k-way tournament selection, with the value of k offering control over the rate of convergence. As GE maintains a population of bitstrings, standard bitstring genetic operators are used:

1. Reproduction - the input bitstring is exactly copied and returned.
2. Creation - a new bitstring is generated as per the initial population generation template method and returned.
3. Mutation - a random bit is chosen and flipped in the input bitstring and that bitstring is returned.

Members chosen using m-elitism undergo reproduction, and members chosen using k-way tournament selection undergo mutation. Crossover was not used, as experimental runs found that crossover was having a destructive effect on population members, rarely producing offspring with better fitness values. This may be due to the fact that a sudden change in the primary architecture, which crossover is likely to cause, can have severely debilitating effects.

4.2.4 Hyperparameters

There are two sets of hyperparameters: one set specific to the functioning of SAWO-NN, shown in Table 6, and another set for tweaking the behaviour of GE, shown in Table 7. All hyperparameter choices were decided by trial and error from experimental runs, except a few which were explicitly chosen.

Table 6 Hyperparameters for SAWO-NN

Number of epochs per training	20
Number of mini-batches	5
Training/validation split	75/25

The most important GE choice is the number and size of codons. Due to G_{xy}'s *architecture* rule and G's *primary* rule, which both encode a large number of choices, the codon lengths must be large enough to choose every option, with 7 bits sufficient. The number of codons needed can be exactly

Table 7 Hyperparameters for GE

Number of generations	10
Initial population size	120
Subsequent population size	60
Tournament selection k	2
Reproduction rate	20%
Creation rate	30%
Mutation rate	50%
Number of codons	31
Codon length	7 bits
Number of trainings per evaluation	2

determined: 1 codon is needed for the start rule, 3 codons are needed for layer size choices, 2 codons are needed to make a constant and architecture choice for each of the 13 WUEs, and 1 codon is needed for a primary architecture choice. Therefore, exactly 31 codons are needed.

4.3 Experimental runs

Because GE is stochastic by nature, multiple runs should be done on each dataset. Exactly **10 runs** are done on each of the 12 datasets. The best SAWO-NN from each run is extracted, which gives 10 optimal models for each of the 12 datasets. Note that the best model for classification datasets achieves the highest *best-perf* values, and for regression, the lowest *best-perf* values.

Statistical measures are then calculated across these 10 models per dataset, summarising the performance of this GE approach in finding optimal SAWO-NNs. The single best model for each of the 12 datasets is selected, and these models are used for further comparisons to the baseline technique. These steps will help fulfill research objective 1.

4.4 Comparing techniques

The optimal SAWO-NNs need to be compared to the baseline technique they are attempting to improve. This baseline technique is standard single-architecture gradient-descent, where a single primary architecture is used in all WUEs. This comparison must be done to assess if the novel mechanisms used in the SAWO-NNs are able to provide a learning advantage over the baseline approach, to fulfill research objective 2.

Below are the specifics of the baseline model to be compared against:

- The single architecture used is $i \cdot a \cdot b \cdot c \cdot o$ with hidden layer sizes of 15. This is a fair architecture to compare against, as all hidden layers are used and the sizes are relatively large considering the range in G is from 2 to 20.
- The linear activation function will be used on every neuron in this architecture, and all weights are randomly initialized using a normal distribution

with mean of 0 and standard deviation of 0.2 i.e. $\mathcal{N}(0, 0.2)$, to ensure a fair comparison with SAWO-NNs.
- Optimal learning rates are dependent on the problem, so will be systematically fine-tuned and presented for every dataset. The G_{xy} grammar permits some learning rate tuning, so this is fair. Other training hyperparameters are the same as those used by SAWO-NNs, as described in Table 6.

To compare the performance of two techniques on a dataset, statistical hypothesis testing is used. This requires performance samples to be generated. A performance sample for a technique will contain either maximum validation accuracy values (classification) or minimum validation mean-squared error values (regression) i.e. *best-perf* values achieved across multiple trainings. Each training uses a randomly shuffled training/validation split of the dataset of interest. The performance samples will be of **size 30**.

One random seed is chosen for each dataset, and whenever a performance sample must be generated for that dataset, that random seed is used to generate all training/validation splits, sequentially. This ensures that the same shuffled split is used element-wise for any performance samples on a dataset, enabling the use of paired-sample hypothesis testing.

The performance sample of the best SAWO-NN found for a dataset is denoted as *sawo* with average $sawo_\mu$. The performance sample of the baseline with optimized parameters for the dataset is denoted as *base* with average $base_\mu$. The comparison between *sawo* and *base* for each dataset is done like so:

1. Perform two tests, each a one-sided paired two-sample t-test:
   (a) $H_0 : sawo_\mu = base_\mu$ and $H_a : sawo_\mu < base_\mu$
   (b) $H_0 : sawo_\mu = base_\mu$ and $H_a : sawo_\mu > base_\mu$
2. For classification datasets:
   (a) If H_0 is rejected in test i with confidence c, *base* wins with c confidence
   (b) If H_0 is rejected in test ii with confidence c, *sawo* wins with c confidence
   (c) If no H_0 is rejected or both are rejected, there is no clear winner
3. For regression datasets:
   (a) If H_0 is rejected in test i with confidence c, *sawo* wins with c confidence
   (b) If H_0 is rejected in test ii with confidence c, *base* wins with c confidence
   (c) If no H_0 is rejected or both are rejected, there is no clear winner

5 Results

5.1 Experimental runs

To calculate statistical measures across the 10 models for each dataset, 10 performance samples of size 30 for every dataset need to be aggregated. Each performance sample will be reduced to the average of that performance sample, which gives 10 averages for every dataset, which is easier to aggregate. The average of a model's performance sample will simply be referred to as the "Perf" value for the model. The design time taken to find these models using

GE, as well as the time taken to perform one training using the parameters of Table 6, are provided in average ± standard deviation format.

The results for the benchmark datasets are presented in Table 8, and for the real-world application applications in Table 9. The top section of each table represents classification datasets and validation accuracies, whereas the bottom section of each table represents regression datasets and validation mean-squared errors. This is also the case for Tables 10 and 11.

Table 8 Statistical summaries of best members across benchmark datasets

Dataset	Min-Perf	Avg-Perf	Std-Perf	Max-Perf	Design (mins)	Training (s)
1	0.959441	0.967459	0.004079	0.972028	204.00 ± 15.77	45.43 ± 5.82
2	0.978549	0.982633	0.001815	0.984917	204.90 ± 13.41	49.83 ± 9.61
3	0.841667	0.846447	0.002699	0.851316	198.72 ± 16.04	46.55 ± 4.02
4	0.873684	0.880144	0.003889	0.885088	206.37 ± 11.42	52.28 ± 7.57
5	0.545250	0.563408	0.010762	0.578417	215.21 ± 15.51	54.24 ± 8.47
6	0.553086	0.565802	0.010716	0.583333	228.70 ± 27.47	49.32 ± 8.00
7	0.922013	0.950063	0.011410	0.957233	216.61 ± 21.32	53.90 ± 3.69
8	0.013769	0.014365	0.000315	0.014849	203.72 ± 16.28	49.69 ± 7.03
9	0.007979	0.008167	0.000104	0.008354	205.00 ± 15.57	48.63 ± 4.81
10	0.017991	0.019459	0.001422	0.021770	181.09 ± 11.26	48.04 ± 5.57

Table 9 Statistical summaries of best members across real-world applications

Dataset	Min-Perf	Avg-Perf	Std-Perf	Max-Perf	Design (mins)	Training (s)
11	0.951995	0.954438	0.001124	0.956102	232.32 ± 14.56	56.36 ± 8.05
12	0.004265	0.004367	0.000102	0.004561	206.26 ± 12.60	48.65 ± 7.02

The baseline neural networks had an average training time of approximately 3 seconds, which is far quicker than the SAWO-NNs. However, the baselines use a single architecture with the Keras API in Python, which is known to be extremely efficient and runs mostly as C code. SAWO-NN maintains multiple architectures, and was created for correctness, so increasing the efficiency of the implementation can certainly be investigated.

For each dataset and its 10 selected members, the one with the best performance sample average is chosen as the champion i.e. members who produced the values in the Max-Perf column for classification and Min-Perf column for regression. These selections fulfill research objective 1. These champions are used for further comparisons with the baseline technique.

5.2 Comparisons

The comparison between the best SAWO-NN for each dataset and the baseline technique for the benchmark datasets is presented in Table 10, and in Table 11 for the real-world applications. The average of each technique's performance sample ("Perf") and the fine-tuned learning rate chosen for the baseline is given. The hypothesis testing comparison result between their performance samples is also provided, stating the winner and confidence - note that a higher classification accuracy is better, and a lower regression error is better.

Table 10 SAWO-NN versus baseline across benchmark datasets

Dataset	SAWO-NN Perf	BASE Perf		Comparison
1	0.972028	0.965035	α=0.23	*sawo* wins with 99.884% confidence
2	0.984917	0.978040	α=0.18	*sawo* wins with 99.999% confidence
3	0.851316	0.838596	α=0.22	*sawo* wins with 99.970% confidence
4	0.885088	0.871930	α=0.50	*sawo* wins with 99.665% confidence
5	0.578417	0.561917	α=0.48	*sawo* wins with 99.922% confidence
6	0.583333	0.568519	α=0.48	*sawo* wins with 87.652% confidence
7	0.957233	0.942138	α=0.39	*sawo* wins with 99.980% confidence
8	0.013769	0.015989	α=0.22	*sawo* wins with 99.956% confidence
9	0.007979	0.008437	α=0.34	*sawo* wins with 99.999% confidence
10	0.017991	0.020182	α=0.14	*sawo* wins with 99.616% confidence

Table 11 SAWO-NN versus baseline across real-world applications

Dataset	SAWO-NN Perf	BASE Perf		Comparison
11	0.956102	0.950381	α=0.18	*sawo* wins with 99.998% confidence
12	0.004265	0.004358	α=0.26	*sawo* wins with 96.551% confidence

Both the baseline method and optimal SAWO-NNs worked better with higher learning rates than typically used. This is likely due to the relatively small architectures used in this research compared to the huge deep learning models which tend to work better with minute learning rates.

The SAWO-NNs are resounding winners in this comparison, winning at very high confidence for all datasets. Not only do the SAWO-NNs win in terms of a hypothesis test, but the performance sample averages for the SAWO-NNs are convincingly better than the baseline across the 12 datasets. This thorough victory by the SAWO-NNs indicates that the use of supplementary architectures in particular WUEs was undoubtedly beneficial to the learning process. These results fulfill research objective 2, and the theoretical explanation for this boost in learning performance is presented in the next section.

6 Analysis

6.1 Identification of patterns

With the established improvements of SAWO-NN on standard single-architecture gradient-descent, the question of interest is *how* they caused an improvement. Why did particular architectures work as supplementary update architectures for particular weight sets, compared to other architectures? What aspects of these architectures made them suitable for supplementary use, and are they tied to aspects of the primary architecture and the weight set chosen?

To answer this, and fulfill research objective 3, the top SAWO-NNs need to be analysed for common patterns. For each of the 12 datasets, one model was chosen for comparisons, but this only gives 12 models, which is not sufficient to find confident patterns. Therefore, all 10 optimal SAWO-NNs from each dataset are considered, which gives 120 models that can be analysed.

These 10 models per dataset can be sorted by their performance sample average, to rank them by optimality. Patterns can then be extracted at different stages of optimality. All 10 models across all 12 datasets can be analysed, then the top 9 models, then the top 8, etc, until only the top 3 models for each dataset are analysed. This not only extracts strong patterns and trends for the top models, but gives an indication of whether these patterns and trends grow stronger or fade away as we become more exclusive in terms of optimality.

Identifying the most important patterns within these optimal models using the naked eye is difficult, so data mining tools are used to extract patterns. These tools are designed to find patterns in numerical features, and so the array-like architectures and sets-of-weights that use them must be described using numerical features. A list of features to extract was derived, with some features directly extracted from the architecture and others performing scaling on those directly extracted. Some of these features are presented in Table 12.

Table 12 Examples of extracted features

Feature	Meaning
len	the length of an architecture i.e. the value of n of the n-tuple
cnt	the number of layers in an architecture
n_i	the number of i layers in an architecture
w_i	whether the weight set starts with an i (1) or not (0)
strt	the distance from the front of the architecture to the front of the weight set
n_1s	the number of sets in the n-tuple of the architecture with one element
dnsty	defined as *cnt* / *len*
p_i	defined as *n_i* / *cnt*, percentage of i layers
p_1s	defined as *n_1s* / *len*, percentage of 1-sets

To illustrate how these features can be extracted from an architecture and a weight set, Table 13 gives some architecture and weight combinations, and what the values of these example features would be.

The pattern extraction process was done on pairs of numerical features. Each pair is derived from a model, and consists of a mapping of <features of primary arch and weight, features of supplementary arch and weight>. For some models, multiple weight sets use supplementary architectures, so multiple pairs are produced for these models. These pairs share the same values for the primary element, but have different values for the supplementary element.

Table 13 A few examples of architectures and extracted features

Arch & Weight	w_i	len	cnt	dnsty	n_i	p_i	strt	n_1s	p_1s
$i \cdot b \cdot o$ & io	1	3	3	1.00	1	0.33	0	3	1.00
$i \cdot \lvert ia\rvert \cdot b \cdot o$ & ib	1	4	5	1.25	2	0.40	1	3	0.60
$i \cdot \lvert ic\rvert \cdot \lvert ia\rvert \cdot o$ & ao	0	4	6	1.50	3	0.50	2	2	0.50
$i \cdot \lvert iabc\rvert \cdot o$ & ic	1	3	6	2.00	2	0.33	0	2	0.66
$i \cdot a \cdot \lvert ib\rvert \cdot c \cdot o$ & bc	0	5	6	1.20	2	0.33	2	4	0.80

These mapping pairs represent a separate input-produces-output problem, where features of the primary architecture and weight determine features of the supplementary architecture and weight. This "dataset" of mapping pairs was analysed for patterns. Multiple methods of analysis and pattern extraction were performed, including: statistical summaries, correlation analysis, decision tree analysis to predict supplementary features using primary features, and hypothesis testing to determine significant changes between corresponding features in primary and supplementary architectures.

The strongest patterns found are presented below. Supplementary architecture is abbreviated as *supp*, and primary architecture is abbreviated as *prim*. Closely linked patterns are grouped together into rough categories.

Patterns concerning density:

- Strongest pattern: the proportion of sets with 2 layers in the *supp*s was higher than in their *prim*s. This was extremely prevalent for all degrees of exclusivity. The number of sets containing 2 layers in the *supp*s was also noticeably higher.
- Very strong pattern: *supp*s were less dense than their *prim*s. This phenomenon visibly strengthened as the pool became more exclusive. This means that the *supp*s were longer, or had less layers, or both.
- Strong pattern: *supp*s contained less weight sets than their *prim*s. This pattern was more frequent as the pool became more exclusive. This is tied to a lower architecture density, as less dense architectures are less likely to contain parallel layers, which need more weight sets.
- Moderately strong pattern: *supp*s were longer than their *prim*s i.e. had more sets in the n-tuple. This pattern was highly prevalent for the entire optimal pool. This is tied to a lower architecture density.
- Moderate pattern: less layers in *supp*s compared to their *prim*s. This pattern emerged for more exclusive pools, and is tied to a lower architecture density.

Patterns concerning i-layer importance:

- Strong pattern: *supp*s contained less a, b and c layers than their *prim*s. This was frequent but became less so as the pool became more exclusive. This feature is not scaled per architecture, so is likely to be influenced by other features e.g. shorter architectures will have less a, b and c layers.

- Moderately strong pattern: the proportion of i layers in the *supp*s was higher than their *prim*s, and the proportion of a, b and c layers lower. This was frequent but faded as the pool became more exclusive. This is linked to the above pattern, but is scaled to architecture size, so is more reliable.

Patterns concerning weight position:

- Strong pattern: the relative position of the weight set in the architecture was closer to the front for *supp*s than for their *prim*s. This was not strong across the entire optimal pool, but as the pool became more exclusive, this pattern clearly started to emerge and was extremely evident for the most exclusive pools. This is a relative position factoring in the architecture length, so is more reliable than the distance to the weight set in the architecture.

The patterns presented above were derived from paired-sample hypothesis testing between related features in the *prim*s and *supp*s. These test for greater-than or less-than relationships, which reveal useful high-level trends.

Decision tree analysis created rules to predict supplementary features using primary features, which are more specific than high-level trends. However, trees were made for every supplementary feature, so interpretation of every tree is difficult. Therefore, a summation was done on the primary features' presence in the tree rules, to establish which primary features are used most often to predict supplementary features. Primary features used closer to the root of the trees were weighted higher than features used further down the trees.

The most important primary feature for supplementary prediction was the relative position of the weight set in *prim*, which was used most frequently in the tree rules. Second was the number of weight sets in *prim*, which was half as important. The third and fourth most important primary features were the distance from the weight set to the end of *prim*, and the density of *prim*, and these were just less than half as important as the number of weight sets. These results indicate that where the weight set of interest is in the primary architecture, as well as the density of the primary architecture, are most important for predicting what supplementary architecture will be used.

6.2 Theoretical explanation of patterns

The most certain patterns to emerge from analysis of the optimal members are theoretically examined, to assess their contribution to increased optimality.

The patterns of greatest interest are those which strengthened in certainty as the pool of members became more exclusive. These patterns include: a decreased density and related patterns, which visibly strengthened with exclusivity; and a closer-to-the-front relative position of the weight set in the architectures, which showed obvious strengthening with exclusivity.

Why are there learning benefits with less dense *supp*s? One explanation is that less dense architectures generally contain less weight sets, and so the role of each weight set is more important. A weight set in a less dense architecture is more isolated and forced to take responsibility, adapting more than a weight set in a dense architecture, which can rely on other weight sets to adapt instead. Therefore, updating a weight set using a less dense *supp* may cause more significant changes in the weight set than if it was updated by the *prim*, and strengthen its role in transforming input to output.

Why is it beneficial to have the weight set of interest closer to the front in the *supp* than in the *prim*? Weight sets near the end of an architecture are decisive, as they make final transformative steps to produce an output. They are more likely to be 'overfit' to a specific architecture and its error if they are updated using that error, and may not be suitable for use in other architectures. However, weight sets near the front of an architecture are less likely to be 'overfit' to the architecture, and so could be more suitable.

There are patterns left which require theoretical explanation, but it is likely they are all part of one larger pattern. The proportion of i layers in the *supp*s was higher than their *prim*s, and the number and proportion of a, b and c layers was lower. These two patterns indicate that i layers being in *supp*s is more important for optimality than a, b and c layers being in *supp*s.

This relationship is echoed in the strongest pattern found, which was that the proportion of sets containing 2 layers in the *supp*s was higher than in their *prim*s. There can only be one i layer in a hidden set, but it cannot be the only layer in the hidden set, meaning an extra i layer can only be in a set with size 2 or larger. Therefore, an increase in i layer proportion should increase the sizes of sets in the architecture n-tuple, leading to an increase of sets with 2 layers.

How could more i layers affect optimality? When an i layer is only found at the start, important information from the inputs must be preserved throughout all transformations by subsequent weight sets. This gives the information the best chance of reaching the output layer and strengthening prediction. However, when i layers are present later, less preservation of information is needed by earlier weight sets, as the information is fed into the architecture again.

Therefore, earlier weight sets can shift their focus from preserving information, to transforming the information into something useful to benefit prediction. The 'booster signal' that these additional i layers provide can help weight sets be more decisive in transformation, and not as reluctant to transform values and possibly lose encoded information. These weight sets may therefore provide more use to their *prim* through updates using these *supp*s, as these stronger transformations could strengthen predictions.

This identification and explanation of patterns fulfills research objective 3.

7 Comparison to state-of-the-art

The contribution of this research is focused on highlighting and understanding the effectiveness of supplementary architecture usage, and the opportunity it provides for boosting the learning abilities of neural networks. However, a comparison with state-of-the-art techniques on the selected datasets should be done, to give an indication of where SAWO-NNs fits in the "hierarchy" of machine learning techniques. The best SAWO-NNs found for each dataset are used again for this comparison.

Five survey works which cover state-of-the-art techniques on the datasets are identified: [44][45] covers the majority of the benchmark datasets, [46][47] covers dataset 11, the KDDCup 99 dataset, and [48] covers dataset 12, the Graduate Admission dataset. This comparison is not exhaustive, but certainly gives an indication of the performance of SAWO-NNs compared to other effective techniques.

For each survey paper, the top 4 techniques with the highest average-ranking across their chosen datasets are identified. These represent the state-of-the-art techniques which generally performed the best, bearing in mind that it is highly unlikely that any single technique will perform the best for every single dataset i.e. the No-Free-Lunch theorem [49]. The accuracies or mean squared errors these techniques achieved on the overlapping datasets which are chosen for this research are extracted.

The best state-of-the-art techniques from each survey paper [44–48] are labelled from the papers alphabetically as follows:

From [44]: (1) parRF_caret, (2) rf_caret, (3) svm_C, (4) svmPoly_caret.
From [45]: (1) ELM, (2) GBDT, (3) RF, (4) SVM.
From [46]: (1) DNN, (2) Light GBM, (3) Stacked classifier, (4) XG-Boost.
From [47]: (1) J48, (2) NB Tree, (3) Random Forest, (4) Random Tree.
From [48]: (1) 3rd-degree polynomial kernel support vector regression, (2) Linear regression, (3) Random forest regression, (4) RBF kernel support vector regression.

Using this labelling scheme, Table 14 presents the performance of these state-of-the-art techniques on the datasets used in this research. The averages of the best SAWO-NN performance samples are given in the SAWO column. The Techniques column presents the 4 techniques from each survey paper on each dataset, with the technique label in brackets followed by the performance value the technique achieved on the dataset. These techniques are ordered from best to worst performance, from left to right. The Rank column gives the ranking of SAWO-NN amongst these techniques, with rank 1 indicating that SAWO-NN achieved the best results on the dataset.

The rankings illustrate that some survey papers presented better results with similar techniques for the same datasets, likely due to different parameter tuning and different methods for measuring validation performance, which may also give them an unfair advantage over SAWO-NN.

Regardless, the average ranking of SAWO-NN amongst these state-of-the-art techniques is **2.92**, and SAWO-NN produces the best results for **5 of the 13** comparisons. This indicates that the best SAWO-NNs produced, which boost the performance of stochastic gradient descent with effective supplementary architecture choices, are competitive with the state-of-the-art, even though this is not the primary contribution of this work.

Table 14 Comparison to some state-of-the-art techniques

Dataset	SAWO	Techniques from [44]				Rank
1	0.9720	(4) 0.9790	(3) 0.9700	(2) 0.9700	(1) 0.9680	2
2	0.9849	(3) 1.0000	(4) 1.0000	(2) 1.0000	(1) 1.0000	5
3	0.8513	(3) 0.8600	(4) 0.8370	(2) 0.8300	(1) 0.8230	2
4	0.8851	(4) 0.9800	(1) 0.9670	(3) 0.9660	(2) 0.9600	5
5	0.5784	(2) 0.6900	(1) 0.6900	(3) 0.6400	(4) 0.6110	5
6	0.5833	(1) 0.7800	(2) 0.7570	(4) 0.7020	(3) 0.6890	5
7	0.9572	(3) 0.9570	(1) 0.9430	(4) 0.9380	(2) 0.9340	1

Dataset	SAWO	Techniques from [45]				Rank
2	0.9849	(4) 1.0000	(2) 1.0000	(3) 0.9951	(1) 0.9779	4
5	0.5784	(3) 0.5688	(4) 0.4938	(2) 0.4813	(1) 0.4500	1
7	0.9572	(1) 0.9048	(4) 0.8571	(2) 0.8571	(3) 0.8571	1

Dataset	SAWO	Techniques from [46]				Rank
11	0.9561	(3) 0.9835	(4) 0.9835	(2) 0.9826	(1) 0.9756	5

Dataset	SAWO	Techniques from [47]				Rank
11	0.9561	(1) 0.9382	(2) 0.9351	(3) 0.9279	(4) 0.9253	1

Dataset	SAWO	Techniques from [48]				Rank
12	0.004265	(2) 0.004801	(3) 0.005821	(1) 0.006248	(4) 0.007242	1

8 Conclusion

The primary objective of this research was to establish if the use of supplementary architecture errors for updating certain weight sets in a primary architecture could benefit the performance of these primary architectures, compared to if the weight sets in the primary architecture were only updated using the error of the primary architecture.

The answer to this question is yes: effective combinations of primary architecture, weight sets and supplementary architectures were found through a grammatical evolution search, fulfilling research objective 1. The use of these supplementary architecture errors improved the learning ability of the primary architecture on all the chosen datasets, fulfilling research objective 2.

These supplementary architectures had particular traits compared to the primary architecture they assisted: they were less dense, contained more i layers, and had the weight set of interest closer to the front. Therefore, research objective 3 is fulfilled, noting that it would have been difficult to theorize these patterns without performing a search to see what optimal trends appear. The top SAWO-NNs were also shown to be competitive with state-of-the-art techniques on the same datasets.

This results of this work poses further questions. Seeing as stochastic gradient descent was expanded to allow supplementary architecture use, could the same be done for more advanced gradient-descent optimizers, such as [1–8]? These weight update equations all use the "primary" architecture's error, so could the same expansion be applied to these optimizers, and would it be effective for boosting learning ability?

The canonical linear activation function was used to assess the potential of supplementary architecture usage, but is predisposed to saturation, hence the invention of non-linear activation functions. Future work can investigate the use of these modern activation functions: global use in all hidden layers, or local use by encoding and evolving activation function choices in the chromosome.

Gradient-descent is not the only weight optimization method for neural networks. Algorithms such as Newton's method [50], the conjugate gradient method [51], the quasi-Newton method [52] and the Levenberg–Marquardt algorithm [53] adjust weight values differently to gradient descent - but they are not always scalable in terms of speed or memory for large neural networks and/or datasets. Implementation of supplementary architecture usage for these optimization methods and subsequent benefits will be the topic of future work.

Could the information learnt from the optimal models be encoded into another machine learning technique, which could suggest a suitable supplementary architecture when given a primary architecture and weight set in the architecture? The same mechanisms used to extract numerical features could be used on new combinations, for which optimality is unknown. If the extracted features exhibit similar patterns to those identified previously, this could indicate an effective combination.

Lastly, are there better ways of searching for optimal combinations than GE? For example, would a single-point search in the bitstring space be more effective than the multi-point search used in GE? This could reduce design times, but would it find combinations of comparable optimality? Other grammar-based evolutionary algorithms could also be used instead of GE, or the grammar could be eliminated and a genetic algorithm used instead.

All these questions can be investigated in future research, as well as any other improvements that contributors to this field could theorize. However, this novel technique already has benefits and intricacies of its own, and warrants a deeper scientific investigation into how the learning ability of neural networks can be improved by providing additional information to the learning process.

Declarations

Funding

This work is based on the research supported wholly by the National Research Foundation of South Africa (Grant Number MND200804549990).

Conflict of interest

The authors declare that they have no conflict of interest.

Availability of data and material
Not applicable.

Code availability
The implementation code is available at the following link:
https://github.com/jared-oreilly/sawo-nn.

Compliance with ethical standards

References

[1] Qian, N.: On the momentum term in gradient descent learning algorithms. Neural networks **12**(1), 145–151 (1999)

[2] Nesterov, Y.: A method for unconstrained convex minimization problem with the rate of convergence o (1/kˆ 2). In: Doklady an Ussr, vol. 269, pp. 543–547 (1983)

[3] Duchi, J., Hazan, E., Singer, Y.: Adaptive subgradient methods for online learning and stochastic optimization. Journal of machine learning research **12**(7) (2011)

[4] Hinton, G., Srivastava, N., Swersky, K.: Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. Cited on **14**(8) (2012)

[5] Zeiler, M.D.: Adadelta: an adaptive learning rate method. arXiv preprint arXiv:1212.5701 (2012)

[6] Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)

[7] Dozat, T.: Incorporating nesterov momentum into adam (2016)

[8] Reddi, S.J., Kale, S., Kumar, S.: On the convergence of adam and beyond. arXiv preprint arXiv:1904.09237 (2019)

[9] Floreano, D., Dürr, P., Mattiussi, C.: Neuroevolution: from architectures to learning. Evolutionary intelligence **1**(1), 47–62 (2008)

[10] Stanley, K.O., Clune, J., Lehman, J., Miikkulainen, R.: Designing neural networks through neuroevolution. Nature Machine Intelligence **1**(1), 24–35 (2019)

[11] Schraudolph, N.N., Belew, R.K.: Dynamic parameter encoding for genetic algorithms. Machine learning **9**(1), 9–21 (1992)

[12] Mattiussi, C., Dürr, P., Floreano, D.: Center of mass encoding: a self-adaptive representation with adjustable redundancy for real-valued parameters. In: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation, pp. 1304–1311 (2007)

[13] Such, F.P., Madhavan, V., Conti, E., Lehman, J., Stanley, K.O., Clune, J.: Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. arXiv preprint arXiv:1712.06567 (2017)

[14] Igel, C.: Neuroevolution for reinforcement learning using evolution strategies. In: The 2003 Congress on Evolutionary Computation, 2003. CEC'03., vol. 4, pp. 2588–2595 (2003). IEEE

[15] Salimans, T., Ho, J., Chen, X., Sidor, S., Sutskever, I.: Evolution strategies as a scalable alternative to reinforcement learning. arXiv preprint arXiv:1703.03864 (2017)

[16] Mania, H., Guy, A., Recht, B.: Simple random search provides a competitive approach to reinforcement learning. arXiv preprint arXiv:1803.07055 (2018)

[17] Elsken, T., Metzen, J.H., Hutter, F.: Neural architecture search: A survey. The Journal of Machine Learning Research **20**(1), 1997–2017 (2019)

[18] Liu, Y., Sun, Y., Xue, B., Zhang, M., Yen, G.G., Tan, K.C.: A survey on evolutionary neural architecture search. arXiv preprint arXiv:2008.10937 (2020)

[19] Stanley, K.O., Miikkulainen, R.: Evolving neural networks through augmenting topologies. Evolutionary computation **10**(2), 99–127 (2002)

[20] Gruau, F.: Automatic definition of modular neural networks. Adaptive behavior **3**(2), 151–183 (1994)

[21] Nolfi, S., Miglino, O., Parisi, D.: Phenotypic plasticity in evolving neural networks. In: Proceedings of PerAc'94. From Perception to Action, pp. 146–157 (1994). IEEE

[22] Husbands, P., Harvey, I., Cliff, D., Miller, G.: The use of genetic algorithms for the development of sensorimotor control systems. In: Proceedings of PerAc'94. From Perception to Action, pp. 110–121 (1994). IEEE

[23] Soltoggio, A., Bullinaria, J.A., Mattiussi, C., Dürr, P., Floreano, D.: Evolutionary advantages of neuromodulated plasticity in dynamic, reward-based scenarios. In: Proceedings of the 11th International Conference on

Artificial Life (Alife XI), pp. 569–576 (2008). MIT Press

[24] Risi, S., Stanley, K.O.: A unified approach to evolving plasticity and neural geometry. In: The 2012 International Joint Conference on Neural Networks (IJCNN), pp. 1–8 (2012). IEEE

[25] Tonelli, P., Mouret, J.-B.: On the relationships between generative encodings, regularity, and learning abilities when evolving plastic artificial neural networks. PloS one **8**(11), 79138 (2013)

[26] Soltoggio, A., Durr, P., Mattiussi, C., Floreano, D.: Evolving neuromodulatory topologies for reinforcement learning-like problems. In: 2007 IEEE Congress on Evolutionary Computation, pp. 2471–2478 (2007). IEEE

[27] Velez, R., Clune, J.: Diffusion-based neuromodulation can eliminate catastrophic forgetting in simple neural networks. PloS one **12**(11), 0187736 (2017)

[28] Husbands, P., Smith, T., Jakobi, N., O'Shea, M.: Better living through chemistry: Evolving gasnets for robot control. Connection Science **10**(3-4), 185–210 (1998)

[29] Ellefsen, K.O., Mouret, J.-B., Clune, J.: Neural modularity helps organisms evolve to learn new skills without forgetting old skills. PLoS computational biology **11**(4), 1004128 (2015)

[30] Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y.L., Tan, J., Le, Q.V., Kurakin, A.: Large-scale evolution of image classifiers. In: International Conference on Machine Learning, pp. 2902–2911 (2017). PMLR

[31] Real, E., Aggarwal, A., Huang, Y., Le, Q.V.: Regularized evolution for image classifier architecture search. In: Proceedings of the Aaai Conference on Artificial Intelligence, vol. 33, pp. 4780–4789 (2019)

[32] Chalmers, D.J.: The evolution of learning: An experiment in genetic connectionism. In: Connectionist Models, pp. 81–90. Elsevier, ??? (1991)

[33] Fontanari, J., Meir, R.: Evolving a learning algorithm for the binary perceptron. Network: Computation in Neural Systems **2**(4), 353 (1991)

[34] DAN, A.D., Oflazer, K.: Genetic synthesis of unsupervised learning algorithms (1993)

[35] Baxter, J.: The evolution of learning algorithms for artificial neural networks. Complex systems, 313–326 (1992)

[36] Risi, S., Stanley, K.O.: Indirectly encoding neural plasticity as a pattern

of local rules. In: International Conference on Simulation of Adaptive Behavior, pp. 533–543 (2010). Springer

[37] Hebb, D.O.: The Organisation of Behaviour: a Neuropsychological Theory. Science Editions New York, ??? (1949)

[38] Floreano, D., Mondada, F.: Evolution of plastic neurocontrollers for situated agents. In: Proc. of The Fourth International Conference on Simulation of Adaptive Behavior (SAB), From Animals to Animats (1996). ETH Zürich

[39] Floreano, D., Urzelai, J.: Evolution of plastic control networks. Autonomous robots **11**(3), 311–317 (2001)

[40] Di Paolo, E.A.: Evolving spike-timing-dependent plasticity for single-trial learning in robots. Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences **361**(1811), 2299–2319 (2003)

[41] Nolfi, S., Parisi, D.: Learning to adapt to changing environments in evolving neural networks. Adaptive behavior **5**(1), 75–98 (1996)

[42] Floreano, D., Urzelai, J.: Evolutionary robots with on-line self-organization and behavioral fitness. Neural Networks **13**(4-5), 431–443 (2000)

[43] O'Neill, M., Ryan, C.: Grammatical evolution. IEEE Transactions on Evolutionary Computation **5**(4), 349–358 (2001)

[44] Fernández-Delgado, M., Cernadas, E., Barro, S., Amorim, D.: Do we need hundreds of classifiers to solve real world classification problems? The journal of machine learning research **15**(1), 3133–3181 (2014)

[45] Zhang, C., Liu, C., Zhang, X., Almpanidis, G.: An up-to-date comparison of state-of-the-art classification algorithms. Expert Systems with Applications **82**, 128–150 (2017)

[46] Rai, M., Mandoria, H.L.: Network intrusion detection: A comparative study using state-of-the-art machine learning methods. In: 2019 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT), vol. 1, pp. 1–5 (2019). IEEE

[47] Tavallaee, M., Bagheri, E., Lu, W., Ghorbani, A.A.: A detailed analysis of the kdd cup 99 data set. In: 2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications, pp. 1–6 (2009). IEEE

[48] Acharya, M.S., Armaan, A., Antony, A.S.: A comparison of regression

models for prediction of graduate admissions. In: 2019 International Conference on Computational Intelligence in Data Science (ICCIDS), pp. 1–5 (2019). IEEE

[49] Wolpert, D.H.: The lack of a priori distinctions between learning algorithms. Neural computation **8**(7), 1341–1390 (1996)

[50] Battiti, R.: First-and second-order methods for learning: between steepest descent and newton's method. Neural computation **4**(2), 141–166 (1992)

[51] Johansson, E.M., Dowla, F.U., Goodman, D.M.: Backpropagation learning for multilayer feed-forward neural networks using the conjugate gradient method. International Journal of Neural Systems **2**(04), 291–301 (1991)

[52] Setiono, R., Hui, L.C.K.: Use of a quasi-newton method in a feedforward neural network construction algorithm. IEEE Transactions on Neural Networks **6**(1), 273–277 (1995)

[53] Marquardt, D.W.: An algorithm for least-squares estimation of nonlinear parameters. Journal of the society for Industrial and Applied Mathematics **11**(2), 431–441 (1963)

Please view the appendices on the following pages.

Appendix A Full architecture set A

See all architectures in the architecture set A below:

$i \cdot o$

$i \cdot a \cdot o$

$i \cdot |ia| \cdot o$

$i \cdot b \cdot o$

$i \cdot |ib| \cdot o$

$i \cdot c \cdot o$

$i \cdot |ic| \cdot o$

$i \cdot a \cdot b \cdot o$

$i \cdot |ia| \cdot b \cdot o$

$i \cdot a \cdot |ib| \cdot o$

$i \cdot |ia| \cdot |ib| \cdot o$

$i \cdot b \cdot a \cdot o$

$i \cdot |ib| \cdot a \cdot o$

$i \cdot b \cdot |ia| \cdot o$

$i \cdot |ib| \cdot |ia| \cdot o$

$i \cdot |ab| \cdot o$

$i \cdot |iab| \cdot o$

$i \cdot a \cdot c \cdot o$

$i \cdot |ia| \cdot c \cdot o$

$i \cdot a \cdot |ic| \cdot o$

$i \cdot |ia| \cdot |ic| \cdot o$

$i \cdot c \cdot a \cdot o$

$i \cdot |ic| \cdot a \cdot o$

$i \cdot c \cdot |ia| \cdot o$

$i \cdot |ic| \cdot |ia| \cdot o$

$i \cdot |ac| \cdot o$

$i \cdot |iac| \cdot o$

$i \cdot b \cdot c \cdot o$

$i \cdot |ib| \cdot c \cdot o$

$i \cdot b \cdot |ic| \cdot o$

$i \cdot |ib| \cdot |ic| \cdot o$

$i \cdot c \cdot b \cdot o$

$i \cdot |ic| \cdot b \cdot o$

$i \cdot c \cdot |ib| \cdot o$

$i \cdot |ic| \cdot |ib| \cdot o$

$i \cdot |bc| \cdot o$

$i \cdot |ibc| \cdot o$

$i \cdot a \cdot b \cdot c \cdot o$

$i \cdot |ia| \cdot b \cdot c \cdot o$

$i \cdot a \cdot |ib| \cdot c \cdot o$

$i \cdot a \cdot b \cdot |ic| \cdot o$

$i \cdot |ia| \cdot |ib| \cdot c \cdot o$

$i \cdot |ia| \cdot b \cdot |ic| \cdot o$

$i \cdot a \cdot |ib| \cdot |ic| \cdot o$

$i \cdot |ab| \cdot c \cdot o$

$i \cdot |iab| \cdot c \cdot o$

$i \cdot |ab| \cdot |ic| \cdot o$

$i \cdot a \cdot |bc| \cdot o$

$i \cdot |ia| \cdot |bc| \cdot o$

$i \cdot a \cdot |ibc| \cdot o$

$i \cdot |abc| \cdot o$

$i \cdot |iabc| \cdot o$

$i \cdot a \cdot c \cdot b \cdot o$

$i \cdot |ia| \cdot c \cdot b \cdot o$

$i \cdot a \cdot |ic| \cdot b \cdot o$

$i \cdot a \cdot c \cdot |ib| \cdot o$

$i \cdot |ia| \cdot |ic| \cdot b \cdot o$

$i \cdot |ia| \cdot c \cdot |ib| \cdot o$

$i \cdot a \cdot |ic| \cdot |ib| \cdot o$

$i \cdot |ac| \cdot b \cdot o$

$i \cdot |iac| \cdot b \cdot o$

$i \cdot |ac| \cdot |ib| \cdot o$

$i \cdot b \cdot a \cdot c \cdot o$

$i \cdot |ib| \cdot a \cdot c \cdot o$

$i \cdot b \cdot |ia| \cdot c \cdot o$

$i \cdot b \cdot a \cdot |ic| \cdot o$

$i \cdot |ib| \cdot |ia| \cdot c \cdot o$

$i \cdot |ib| \cdot a \cdot |ic| \cdot o$

$i \cdot b \cdot |ia| \cdot |ic| \cdot o$

$i \cdot b \cdot |ac| \cdot o$

$i \cdot b \cdot c \cdot a \cdot o$

$i \cdot |ib| \cdot c \cdot a \cdot o$

$i \cdot b \cdot |ic| \cdot a \cdot o$

$i \cdot b \cdot c \cdot |ia| \cdot o$

$i \cdot |ib| \cdot |ic| \cdot a \cdot o$

$i \cdot |ib| \cdot c \cdot |ia| \cdot o$

$i \cdot b \cdot |ic| \cdot |ia| \cdot o$

$i \cdot |bc| \cdot a \cdot o$

$i \cdot |ibc| \cdot a \cdot o$

$i \cdot |bc| \cdot |ia| \cdot o$

$i \cdot c \cdot a \cdot b \cdot o$

$i \cdot |ic| \cdot a \cdot b \cdot o$

$i \cdot c \cdot |ia| \cdot b \cdot o$

$i \cdot c \cdot a \cdot |ib| \cdot o$

$i \cdot |ic| \cdot |ia| \cdot b \cdot o$

$i \cdot |ic| \cdot a \cdot |ib| \cdot o$

$i \cdot c \cdot |ia| \cdot |ib| \cdot o$

$i \cdot c \cdot |ab| \cdot o$

$i \cdot c \cdot b \cdot a \cdot o$

$i \cdot |ic| \cdot b \cdot a \cdot o$

$i \cdot c \cdot |ib| \cdot a \cdot o$

$i \cdot c \cdot b \cdot |ia| \cdot o$

$i \cdot |ic| \cdot |ib| \cdot a \cdot o$

$i \cdot |ic| \cdot b \cdot |ia| \cdot o$

$i \cdot c \cdot |ib| \cdot |ia| \cdot o$

Appendix B Dataset sources and processing

See the source URLs, pre-processing information and additional notes for each dataset used in this research, below. For all datasets, any categorical features were one-hot encoded, and any continuous features were either standardized, if an underlying Gaussian distribution was detected in the feature, or normalized/min-max scaled, if not detected. This preparation is necessary for neural network processing, to minimise saturation and ensure all features have equal importance.

1 - Breast Cancer Wisconsin (Diagnostic)
Source: data.csv from
https://www.kaggle.com/uciml/breast-cancer-wisconsin-data
Pre-processing: Removed *id* field, target is *diagnosis* field.

2 - Mushroom Classification
Source: mushrooms.csv from
https://www.kaggle.com/uciml/mushroom-classification
Pre-processing: Target is *class* field.

3 - Heart Attack Analysis and Prediction
Source: heart.csv from
https://www.kaggle.com/rashikrahmanpritom/heart-attack-analysis-prediction-dataset
Pre-processing: Target is *output* field.

4 - Iris Species
Source: Iris.csv from
https://www.kaggle.com/uciml/iris
Pre-processing: Removed *Id* field, target is *Species* field.

5 - Red Wine Quality
Source: winequality-red.csv from
https://www.kaggle.com/uciml/red-wine-quality-cortez-et-al-2009
Pre-processing: Target is *quality* field.

6 - Glass Classification
Source: glass.csv from
https://www.kaggle.com/uciml/glass
Pre-processing: Target is *Type* field.

7 - Wheat Seeds
Source: Seed_Data.csv from
https://www.kaggle.com/dongeorge/seed-from-uci
Pre-processing: Target is *target* field.

8 - Boston House Price
Source: housing.csv from
https://www.kaggle.com/vikrishnan/boston-house-prices
Pre-processing: Target is *MEDV* field.
Additional notes: Data from website is separated by spaces and not commas, have to parse differently, and also have to add header line.

9 - Abalone Rings
Source: abalone.csv from
https://www.kaggle.com/rodolfomendes/abalone-dataset
Pre-processing: Target is *Rings* field.
Additional notes: Ring values are technically discrete, but represent age, so can be considered continuous.

10 - 1985 Automobile Insurance
Source: auto_clean.csv from
https://www.kaggle.com/fazilbtopal/auto85
Pre-processing: Target is *normalized-losses* field.

11 - KDDCup 99 Intrusion Detection
Source: Train_data.csv from
https://www.kaggle.com/sampadab17/network-intrusion-detection
Pre-processing: Removed *num_outbound_cmds* and *is_host_login* fields, target is *class* field.
Additional notes: The two removed fields were both equal to 0 across all rows in dataset, so provided no use.

12 - Graduate Admission
Source: Admission_Predict.csv from
https://www.kaggle.com/mohansacharya/graduate-admissions
Pre-processing: Removed *Serial No.* field, target is *Chance of Admit* field.