




Article

Meta-Heuristic Optimization Methods for Quaternion-Valued Neural Networks

Jeremiah Bill ¹, Lance Champagne ^{1,*}, Bruce Cox ¹ and Trevor Bihl ²

¹ Air Force Institute of Technology, Department of Operational Sciences, WPAFB, OH 45433, USA; jeremiah.bill@afit.edu (J.B.); bruceacox1@gmail.com (B.C.)

² Air Force Research Laboratory, Sensors Directorate, WPAFB, OH 45433, USA; trevor.bihl.2@us.af.mil

* Correspondence: lance.champagne@afit.edu

Abstract: In recent years, real-valued neural networks have demonstrated promising, and often striking, results across a broad range of domains. This has driven a surge of applications utilizing high-dimensional datasets. While many techniques exist to alleviate issues of high-dimensionality, they all induce a cost in terms of network size or computational runtime. This work examines the use of quaternions, a form of hypercomplex numbers, in neural networks. The constructed networks demonstrate the ability of quaternions to encode high-dimensional data in an efficient neural network structure, showing that hypercomplex neural networks reduce the number of total trainable parameters compared to their real-valued equivalents. Finally, this work introduces a novel training algorithm using a meta-heuristic approach that bypasses the need for analytic quaternion loss or activation functions. This algorithm allows for a broader range of activation functions over current quaternion networks and presents a proof-of-concept for future work.

Keywords: multilayer perceptrons; quaternion neural networks; metaheuristic optimization; genetic algorithms



Citation: Bill, J.; Champagne, L.; Cox, B.; Bihl, T. Meta-Heuristic Optimization Methods for Quaternion-Valued Neural Networks. *Mathematics* **2021**, *9*, 938. <https://doi.org/10.3390/math9090938>

Academic Editor: Alessandro Nicolai

Received: 31 March 2021

Accepted: 17 April 2021

Published: 23 April 2021

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Over the last several decades, the explosive growth in artificial intelligence and machine learning (AI/ML) research has driven a need for more efficient data representations and machine learning training methods. As machine learning applications have expanded into new and exciting domains, the scale of data processed through enterprise systems has grown to an almost incomprehensible level. While computational resources have grown commensurately with this increase in data, inefficiencies in current neural network architectures continue to hamper progress on difficult optimization problems.

This work examines the use of hypercomplex numbers in neural networks, with a particular emphasis on the use of quaternions in neural network architectures. This work demonstrates that quaternion data representations can reduce the total number of trainable neural network parameters by a factor of four, resulting in improvements in both computer memory allocations and computational runtime. Additionally, this work presents a novel, gradient-free, quaternion genetic algorithm that enables the use of several loss and activation functions previously unavailable due to differentiability requirements.

The remainder of this article is organized as follows: Section 2 provides a review of neural networks, the quaternion number system, quaternion neural networks, and metaheuristic optimization techniques. Section 3 describes the methodology used to develop a quaternion neural network and a novel quaternion genetic training algorithm. Section 4 presents the network results, comparing the quaternion genetic algorithm performance to two analogous real-valued networks. Additionally, a multidimensional input/multidimensional output network is presented for predicting the Lorenz attractor chaotic dynamical system. Finally, Section 5 provides conclusions, recommendations, and proposals for future work.

2. Background and Related Work

2.1. Neural Networks and Multi-Layer Perceptrons

Statistical learning processes have received increasing attention in recent years with the proliferation of large datasets, ever-increasing computing power, and simplified data exploration tools. In 1957, Frank Rosenblatt proposed a neural structure called the perceptron [1]. A perceptron is composed of several threshold logic units (TLUs), each of which takes a weighted sum of input values and uses the resulting sum as the input to a non-linear activation function. While each TLU computes a linear combination of the inputs based on the network weights, the use of a non-linear activation function allows the perceptron to estimate a number of non-linear functions by adjusting the weights of each input.

Stacking multiple layers of perceptrons together so that the output of one perceptron forms the input to a subsequent perceptron allows for the estimation of a vast set of linear and non-linear problems. In fact, two contemporaries, Cybenko [2] and Hornik et al. [3] both independently showed that a network with a single hidden layer and sigmoidal activation functions is able to approximate any nonlinear function to an arbitrary degree of accuracy. This network structure is called the multilayer perceptron (MLP) and it forms the most basic deep neural network (DNN). This result (called the Universal Approximation Theorem) has provided the theoretical justification that has driven neural network research to the present day. A representation of an MLP is shown in Figure 1, and [4] provides an overview of MLPs and other common neural network structures.

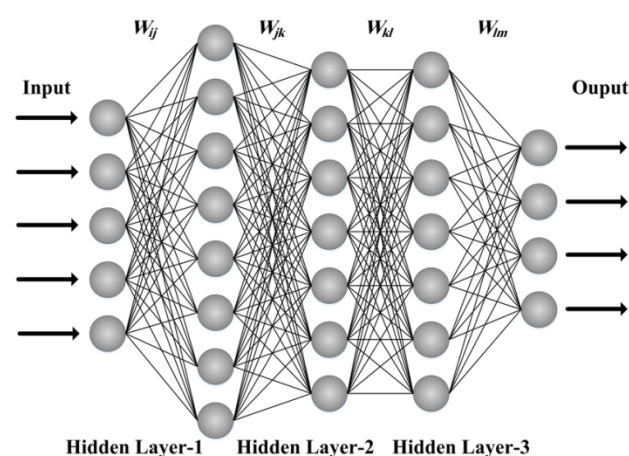


Figure 1. Representation of a basic MLP [5].

2.1.1. The Backpropagation Algorithm

Although artificial neural networks have existed since the mid-20th century, researchers found them to be computationally expensive to train and impractical for most applications. As a result, neural network research was largely stagnant until 1986, when Rumelhart et al. [6] introduced the backpropagation algorithm for training a neural network. The algorithm developed by Rumelhart et al. extended several key ideas that Werbos [7] presented in his unpublished doctoral dissertation.

The backpropagation algorithm has proven to be a straightforward, easy-to-understand, and easy-to-implement algorithm that has enabled efficient implementations of neural networks across a wide-range of problem sets. Examples of custom architectures include convolutional neural networks (CNNs) for processing image data, recurrent neural networks (RNNs) for processing sequence data, and generative adversarial networks (GANs) which have been used in recent years to create deep fakes and very convincing counterfeit data [8].

2.1.2. Shortfalls

Despite artificial neural networks achieving state-of-the-art results in a breathtaking array of problem domains, ANNs are not without their shortfalls. For example, ANNs

often require a vast amount of training data. Standard machine learning datasets such as the ImageNet dataset for computer vision often contain several million datapoints [9]. Consequently, training an ANN requires a large amount of computer resources, in terms of both RAM and processing time. Additionally, the backpropagation algorithm requires a significant amount of low-level computational power in order to perform the matrix multiplications for each forward and backward pass. While GPUs have proven to be particularly well-suited for this task [10], many of the current large-scale ANN research applications require prohibitive amounts of computer memory and GPU hours.

Finally, MLPs can struggle to maintain any sort of spatial relationships that are present within the training data. A simple example of this is seen in color image processing. In general, each of the three color channels of an RGB image are processed separately in an MLP since the 3-dimensional matrix representation of the image must first be flattened into a vector for the network forward pass step. This results in the loss of the spatial relationship between the red, green, and blue pixel intensities at each pixel.

Many spatial dependency issues can be alleviated using more advanced ANN architectures such as convolutional neural networks, which preserve spatial relationships within the data using successive convolutional layers to transform the input data [11]. However, every CNN must contain at least one fully-connected layer prior to the output layer which flattens the output of the final convolution into a 1-dimensional real-valued vector. Thus, even with a CNN, there is some spatial information that is lost when the output of the final convolution is flattened.

On the other hand, Yin et al. [12] highlight the fact that this spatial hierarchy between pixel intensity values can be maintained when using higher-dimensional number systems such as quaternions as opposed to real numbers, and their result is a significant motivation for this paper. Matsui et al. [13] demonstrated similar experimental results on a 3-dimensional affine transformation problem, showing that quaternion-valued deep neural networks were able to recover the spatial relationships between 3-dimensional coordinates. Section 2.2 provides a brief summary of hypercomplex number systems, along with a review of their use and success in advanced neural network applications.

2.2. The Quaternions

The quaternion numbers (denoted by \mathbb{H}) are a four-dimensional extension of the complex numbers. Complex numbers have the form $x + iy$, consisting of a real part x and an imaginary part y , and can be thought of as an isomorphism of \mathbb{R}^2 . That is, the complex numbers contain two copies of the real number line, allowing a single complex number to encode twice as much information as a single real number. Complex numbers are particularly useful for describing motion in 2-dimensional space, since there is a very succinct analogue between complex multiplication and rotations in the plane [14].

Quaternions are referred to as hypercomplex numbers. Each quaternion \mathbf{q} consists of a real part and three imaginary parts, so that the quaternions form an isomorphism with \mathbb{R}^4 with basis elements 1, \mathbf{i} , \mathbf{j} , and \mathbf{k} :

$$\mathbf{q} = r + x\mathbf{i} + y\mathbf{j} + z\mathbf{k}. \quad (1)$$

Quaternions form a generalization of the complex numbers, where the three imaginary components \mathbf{i} , \mathbf{j} , and \mathbf{k} follow the same construct as \mathbf{i} in \mathbb{C} :

$$\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = -1. \quad (2)$$

However, the three imaginary basis components must also satisfy the following rules:

$$\mathbf{jk} = -\mathbf{kj} = \mathbf{i} \quad (3)$$

$$\mathbf{ki} = -\mathbf{ik} = \mathbf{j} \quad (4)$$

$$\mathbf{ij} = -\mathbf{ji} = \mathbf{k}. \quad (5)$$

These rules clearly demonstrate that quaternion multiplication is non-commutative. However, since the multiplication of any two basis elements is plus or minus another basis element, the quaternions under these rules form a non-abelian group, denoted Q_8 . The group Q_8 , along with the operations of addition and multiplication form a division algebra, which is an algebraic structure similar to a field where multiplication is non-commutative.

The 4-dimensional structure of each quaternion number indicates that quaternions are capable of encoding four copies of the real number line into a single quaternion number, analogous to the two copies of \mathbb{R} encoded in the complex numbers. Quaternions were discovered by the Irish mathematician Sir William Rowan Hamilton in 1843 [15], hence why the set of quaternions is referred to as \mathbb{H} and the quaternion notion of multiplication, described below, is referred to as the *Hamilton Product*. For an in-depth review of quaternions and their applications, see [16].

2.2.1. Quaternion Algebra

The quaternions form a division algebra, meaning that the set of quaternions along with the operations of addition and multiplication follow 8 of the 9 field axioms (all but commutativity). Quaternion addition is defined using the element-wise addition operation. For two quaternions $\mathbf{q}_1, \mathbf{q}_2 \in \mathbb{H}$, where:

$$\mathbf{q}_1 = r_1 + x_1\mathbf{i} + y_1\mathbf{j} + z_1\mathbf{k}$$

and

$$\mathbf{q}_2 = r_2 + x_2\mathbf{i} + y_2\mathbf{j} + z_2\mathbf{k}.$$

The sum $\mathbf{q}_1 + \mathbf{q}_2$ is defined as,

$$\mathbf{q}_1 + \mathbf{q}_2 := (r_1 + r_2) + (x_1 + x_2)\mathbf{i} + (y_1 + y_2)\mathbf{j} + (z_1 + z_2)\mathbf{k}. \quad (6)$$

Quaternion multiplication, referred to as the Hamilton Product, can easily be derived using the basis multiplication rules in Equations (3)–(5) and the distributive property. In reduced form, the Hamilton Product of two quaternions \mathbf{q}_1 and \mathbf{q}_2 is defined as:

$$\begin{aligned} \mathbf{q}_1 * \mathbf{q}_2 := & (r_1r_2 - x_1x_2 - y_1y_2 - z_1z_2) \\ & + (r_1x_2 + x_1r_2 + y_1z_2 - z_1y_2)\mathbf{i} \\ & + (r_1y_2 - x_1z_2 + y_1r_2 + z_1x_2)\mathbf{j} \\ & + (r_1z_2 + x_1y_2 - y_1x_2 + z_1r_2)\mathbf{k}. \end{aligned} \quad (7)$$

2.2.2. Quaternion Conjugates, Norms, and Distance

The notion of a quaternion conjugate is analogous to that of complex conjugates in \mathbb{C} . The conjugate of a quaternion $\mathbf{q} = r + x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$ is given by $\mathbf{q}^* = r - x\mathbf{i} - y\mathbf{j} - z\mathbf{k}$. The norm of a quaternion is equivalent to the Euclidean norm in \mathbb{R} and is given by:

$$\|\mathbf{q}\| := \sqrt{\mathbf{q}\mathbf{q}^*} = \sqrt{r^2 + x^2 + y^2 + z^2}. \quad (8)$$

With this quaternion norm, one can also define a notion of distance $d(\mathbf{q}, \mathbf{p})$ between two quaternions \mathbf{q} and \mathbf{p} as:

$$d(\mathbf{q}, \mathbf{p}) := \|\mathbf{q} - \mathbf{p}\|. \quad (9)$$

2.2.3. Quaternionic Matrices

Since the set of quaternions \mathbb{H} form a division algebra under addition and the Hamilton product, they also form a non-commutative ring under the same operations. Hence,

quaternionic matrix operations can be defined as for matrices over an arbitrary ring. Given any two quaternionic matrices $A, B \in \mathbb{H}^{M \times N}$, the sum $A + B$ is defined element-wise:

$$(A + B)_{ij} := A_{ij} + B_{ij}. \quad (10)$$

Similarly, for any quaternionic matrix $A \in \mathbb{H}^{M \times N}$ and $B \in \mathbb{H}^{N \times P}$, the product $AB \in \mathbb{H}^{M \times P}$ is defined as:

$$(AB)(m, p) := \sum_{n=1}^N A(m, n)B(n, p), \quad \forall m = 1, \dots, M, p = 1, \dots, P. \quad (11)$$

As with matrix multiplication over an arbitrary ring, quaternionic matrix multiplication is non-commutative. Additionally, great care must be taken to ensure the proper execution of the Hamilton product when multiplying each row of A with each column of B , since the Hamilton product itself is non-commutative.

2.3. Quaternion-Valued Neural Networks (QNNs)

Many practical applications of machine learning techniques involve data that are multidimensional. With the mathematical machinery described in Section 2.2, the quaternions provide a succinct and efficient way of representing multidimensional data. Additionally, when applied to neural network architectures, quaternions have been shown to preserve spatial hierarchies and interrelated data components that are often separated and distorted in real-valued MLP architectures. This section provides a brief review of QNN research, starting with a brief note on some of the issues in QNN construction stemming from quaternionic analysis and quaternion calculus. Then, the development of QNNs is traced chronologically from early works to the state of the art.

2.3.1. A Note on Quaternion Calculus and Quaternionic Analysis

There are very few analytic functions of a quaternion variable. To account for this, quaternion networks generally utilize “split” activation functions, where a real-valued activation function is applied to each quaternion coefficient. For example, the split quaternion sigmoid function [17] for a quaternion $\mathbf{q} = r + xi + yj + zk$ is given by:

$$\sigma(\mathbf{q}) = \sigma(r) + \sigma(x)\mathbf{i} + \sigma(y)\mathbf{j} + \sigma(z)\mathbf{k}, \quad (12)$$

where $\sigma(\cdot)$ is the real-valued sigmoid function. Similar definitions hold for any real-valued activation function, and many QNNs utilize these split activation functions even when quaternionic functions, such as the quaternion-valued hyperbolic tangent function, are available. Research has indicated that true quaternionic activation functions can improve performance over split activation functions [18], but they require special considerations since their analyticity can only be defined over a localized domain, and the composition of two locally analytic quaternion functions is generally not locally analytic [19], providing limited utility in deep neural networks. Additionally, many complex and quaternion-valued elementary transcendental functions, including the hyperbolic tangent, are unbounded and contain singularities [20] that make neural network training difficult.

These issues, along with the non-commutativity of quaternions, also affect the gradient descent algorithm employed in many quaternion networks. Generally speaking, the non-commutativity of quaternions precludes the development of a general product rule and a quaternion chain rule to compute quaternion derivatives and partial derivatives. Thus, quaternion networks must employ split loss functions and the partial derivatives used in the backpropagation algorithm are calculated using a similar “split” definition. The split

partial derivative used in training a Quaternion Multilayer Perceptron (QMLP) network, first defined by [17], is given by:

$$\frac{\partial E}{\partial W^l} = \frac{\partial E}{\partial W_r^l} + \frac{\partial E}{\partial W_x^l} \mathbf{i} + \frac{\partial E}{\partial W_y^l} \mathbf{j} + \frac{\partial E}{\partial W_z^l} \mathbf{k}, \quad (13)$$

where E is the loss function and W^l is the weight matrix at layer l . Some researchers refer to this as a “channelwise” [18] or vectorized implementation.

Researchers have made several advances in quaternion calculus, dubbed the generalized Hamilton-Real (GHR) calculus [21], with novel product and chain rules. However, as of this writing, the GHR calculus and the associated learning algorithms implementing the GHR product and chain rules have yet to be applied to any real-world machine learning dataset with a deep quaternion network.

This work proposes a genetic algorithm to train a quaternion-valued neural network with fully quaternion activation functions at each layer of the network. The genetic algorithm circumvents the need for the convoluted calculus rules that one must employ in traditional QNNs due to the non-commutativity of quaternions and the locally analytic nature of the activation functions, allowing for a broader range of available activation functions. While not yet proven in the quaternion domain, this approach has a strong theoretical basis that is supported in both the complex- and real-valued domains ([2,3,20]).

2.3.2. Quaternion Neural Networks

The QMLP was first introduced by Arena et al. [17] in 1994, as noted in Section 2.3.1. The initial QMLP used split sigmoid activation functions and a version of the mean square error (MSE) loss function E , formed by substituting quaternions into the real-valued MSE equation. For a network with $l = 1, \dots, M$ layers and $1 < n < N_l$ nodes per layer, the output of each node n in each layer l is computed as:

$$y_n^l = \sigma(S_n^l), \quad (14)$$

where σ is any split sigmoidal activation function and S_n^l is the linear combination of network weights, biases, and the output of the $l - 1$ layer computed as in a normal MLP:

$$S_n^l = \sum_{m=0}^{N_{l-1}} w_{nm}^l * y_m^{l-1} + b_n^l. \quad (15)$$

For each S_n^l , the weights, biases, and y -values are all quaternions. Thus, $*$ represents the Hamilton Product. The loss function E is given by:

$$E = \frac{1}{N} \sum_{n=1}^N (\mathbf{t}_n - \mathbf{y}_n^{(M)})^2, \quad (16)$$

where \mathbf{t} represents the target (truth) data and $\mathbf{y}^{(M)}$ represents the neural network output at the M th layer.

The authors also introduced a simple learning algorithm using the split or “channel-wise” partial derivatives discussed in Section 2.3.1, where the gradient Δ_n^l at the output layer is simply the output error of the network $(\mathbf{t}_n - \mathbf{y}_n^{(M)})$ and the error at each prior layer l is calculated using the formula:

$$\Delta_n^l = \sum_{h=1}^{N^{l+1}} w_{hn}^{*l+1} * (\Delta_n^{l+1} \cdot \sigma'(S_n^{l+1})), \quad (17)$$

where w_{hn}^{*l+1} represents the quaternion conjugate of the weight connecting node h in the l th layer to node n in the $l + 1$ st layer. Additionally, (\cdot) represents the componentwise product,

not the Hamilton Product between the gradient at the $l + 1$ st layer and the channelwise partial derivative of $\sigma(\cdot)$. Using this gradient rule, the biases at each layer are updated according to the normal backpropagation process:

$$b_n^l = b_n^l + \epsilon \Delta_n^l, \quad (18)$$

where ϵ is the learning rate. Note, however, that the weights are updated using the rule:

$$w_{nm}^l = w_{nm}^l + \epsilon \Delta_n^l * S_m^{*l-1}, \quad (19)$$

where S_m^{*l-1} represents the conjugate of the input to the l th layer S_m^{l-1} .

Although the quaternion backpropagation algorithm bears similarities to the real-valued backpropagation algorithm, it is unique in several ways. The first is the use of split derivatives in the weight and bias update step. Although the use of split derivatives may seem like a trick to bypass a true quaternion derivative definition, it builds on [22], which proved that split activation functions and derivatives in the complex domain could universally approximate complex-valued functions. While unproven in the quaternion domain, Arena et al. demonstrated the effectiveness of this network on a small function approximation problem, where a quaternion network was used to approximate a quaternion-valued function. Additionally, the weight update and the gradients leverage the quaternion conjugate, which improves training performance.

Since the introduction of the QMLP and its associated training algorithm, researchers have used QMLPs for a variety of tasks. In particular, QMLPs have been used as autoencoders [23], for color image processing [24], text processing [25], and polarized signal processing [26]. Another natural application of quaternions is in robotic control [27], since quaternions can compactly represent 3-dimensional rotation and motion through space. Parcollet et al. [28] note that in every scenario, QMLPs always outperform real-valued MLPs when processing 3- or 4-dimensional signals. These simple networks have driven further research in more advanced network architectures such as convolutional neural networks and recurrent neural networks, both of which have shown promise in the quaternion domain for advanced image processing [29], speech recognition [30], and other tasks.

2.4. Metaheuristic Optimization Techniques

Whereas the backpropagation algorithm discussed in Section 2.1.1 has dominated nearly all neural network research since it was first introduced, recent work has shown that heuristic search methods can also effectively train neural networks at a scale comparable to gradient descent and backpropagation. Metaheuristic optimization encompasses a broad range of optimization techniques that do not provide guarantees of algorithmic closure or convergence, but have shown empirically to perform well in a variety of complex optimization tasks. In contrast to gradient-based methods such as the backpropagation algorithm, many metaheuristics do not require any gradient information.

Perhaps the most famous application of a metaheuristic approach in training neural networks is the NeuroEvolution through Augmenting Topologies (NEAT) [31] process, which uses a genetic algorithm to simultaneously train and grow neural networks through an evolutionary process. NEAT has proven to be a very effective neural network training tool, and subsequent variants of NEAT have successfully evolved neural networks with millions of weight and bias parameters [32]. More recently, researchers with Uber's OpenAI Labs have shown that even basic Genetic Algorithms can compete with backpropagation in training large networks with up to four million parameters [33]. Several other metaheuristic implementations have shown promise in training neural networks and optimizing the hyperparameters of neural networks. See [34] for a full review of metaheuristic optimization in neural network design.

Metaheuristic optimization methods have also been applied to a limited number of search problems in the quaternion domain. A quaternion variant of the Firefly Algorithm [35] demonstrated comparable performance to the real-valued Firefly Algorithm

in optimizing nonlinear test functions. In addition, [36] introduced a quaternion-based Harmony Search algorithm, demonstrating the algorithm's performance on a similar range of nonlinear test functions. The hypothesis of both approaches is that the search space in the hypercomplex domain is smoother than the search space in \mathbb{R} . While not proven, [37] summarizes the approach. Additionally, Khuat et al. [38] introduced a quaternion genetic algorithm with multi-parent crossover that was used to optimize a similar set of nonlinear test functions. Finally, [39] used the Harmony Search algorithm introduced in [36] to fine-tune the hyperparameters of a neural network. However, as of this writing, quaternion metaheuristic search methods have yet to be applied to more complex tasks, such as optimizing a large number of weights and biases in a quaternion neural network.

Given the difficulties in defining globally analytic quaternion loss functions, activation functions, and quaternion partial derivatives, metaheuristic optimization provides an ideal method of training quaternion neural networks. Section 3 outlines a novel quaternion genetic algorithm for training the weights and biases of quaternion neural networks. The algorithm does not require gradient information and makes no assumptions on the analyticity of the activation functions of the network at each layer, allowing for a broader range of quaternion activation functions than have been available in prior works.

3. Methodology

This section describes the test methodology employed in comparing the performance of real-valued MLPs to quaternion-valued MLPs in several multidimensional function approximation tasks. First, Section 3.1 describes the test functions selected for use in the study. Section 3.2 outlines the structure of the neural networks, including an overview of the neurons, layers, and total trainable parameters of each network. Section 3.3 details the genetic algorithm used to train the real- and quaternion-valued networks. Finally, Section 3.4 presents a description of the evaluation strategy and key comparison metrics.

3.1. Test Functions

Demonstrating the ability of a neural network to approximate an arbitrary nonlinear function is a crucial step in the development of any ANN structure. Cybenko's Universal Approximation Theorem [2], discussed in Section 2.1, provides the theoretical underpinning for all modern ANN research and has legitimized many of the ANN applications to date. While still unproven for the quaternion domain, this research demonstrates that quaternion neural networks with elementary transcendental activation functions and a genetic training algorithm can effectively approximate arbitrary nonlinear functions, using the Ackley function and the Lorenz attractor chaotic system as test cases.

3.1.1. The Ackley Function

The Ackley function is a non-convex test function that is often used to test global optimization algorithms. It was first introduced by David Ackley [40] and has since been included in a standard library of optimization test functions. In three dimensions, the function is characterized by an elevated eggcrate-like surface, with a global minimum in the center of the function that sinks down to zero. The Ackley function is a good test case for quaternion networks since it can easily be defined in any number of dimensions. A vector representation of the function is given in Equation (20), where a , b , and c are constants and n represents the dimensionality of the vector \mathbf{x} . Additionally, a three-dimensional plot of the Ackley function is shown in Figure 2.

$$f(\mathbf{x}) = -a \exp \left(-b \sqrt{\frac{1}{n} \sum_{i=1}^n x_i^2} \right) - \exp \left(\frac{1}{n} \sum_{i=1}^n \cos(c \cdot x_i) \right) + a + \exp(1) \quad (20)$$

This research uses a 4-dimensional Ackley function, with the a , b , and c coefficient values set to 20.0, -0.2 , and 2π , respectively. The function's x , y , and z values are generated over the range $[-5, 5]$, using a meshgrid with a spacing of 0.5 between each point. With

three-dimensional input, and this results in 9261 data points. The coordinate values are then translated from \mathbb{R} into \mathbb{H} by taking the coordinates of each point and casting them into the three imaginary parts of a quaternion. For example, the point $(-5, -5, -5) \Rightarrow \mathbf{q}_1 = 0r - 5\mathbf{i} - 5\mathbf{j} - 5\mathbf{k}$.

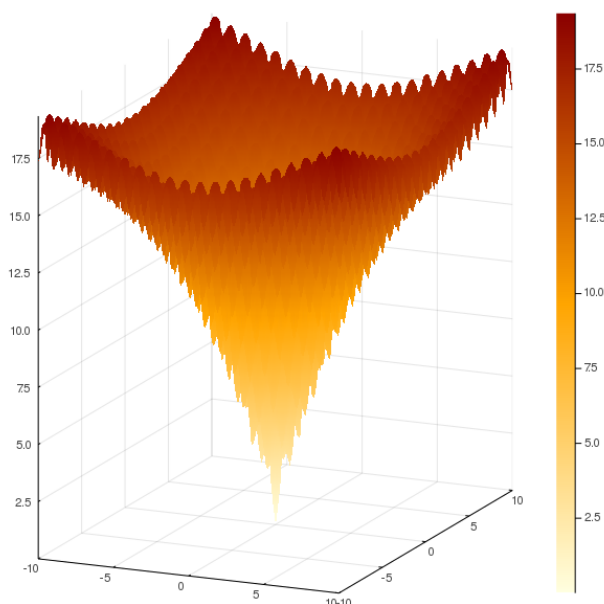


Figure 2. 3D Ackley function.

Finally, the data is split into a training set and a test set. The purpose of this split is to ensure that the neural networks are producing functions with good generalization capabilities. The data points are randomly shuffled and 80% of the data points are retained as training data while 20% of the data points are split into the test set.

3.1.2. The Lorenz Attractor Chaotic System

The Lorenz attractor is a deterministic system of differential equations that was first presented by Edward Lorenz [41]. The Lorenz attractor is a chaotic system, meaning that while it is deterministic, the system never cycles and never reaches a steady state. Additionally, the system is very sensitive to initial conditions. When represented as a set of 3-dimensional coordinates, the Lorenz attractor produces a mesmerizing graph often referred to as the Lorenz butterfly. A static representation of this is shown in Figure 3.

The Lorenz attractor is governed by the following system of differential equations:

$$\frac{dx}{dt} = \sigma(y - x) \quad (21)$$

$$\frac{dy}{dt} = \rho x - y - xz \quad (22)$$

$$\frac{dz}{dt} = xy - \beta z \quad (23)$$

where σ , ρ , and β are constants. For this experiment (and in Figure 3), $\sigma = 10$, $\rho = 28$, and $\beta = \frac{8}{3}$. Quaternions are naturally well-suited to predicting chaotic time series, including the Lorenz attractor, since the problem involves both a multidimensional input and a multidimensional output. Split quaternion neural networks have proven quite successful at chaotic time series prediction based on small training datasets ([42–45]).

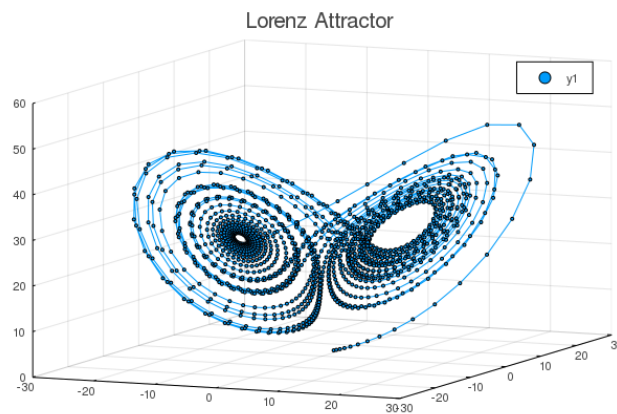


Figure 3. Lorenz Attractor.

The data for the Lorenz attractor was again split 80%/20% between training and test datasets. Additionally, both the inputs and the outputs were cast into the quaternion domain. This allowed for a direct output error calculation using the quaternion distance metric defined in Section 2.2.2. The full details of the loss function, the activation functions, the neurons, and the layers of the networks used in both experiments are discussed in Section 3.2.

3.2. MLP Network Topologies

3.2.1. Function Approximation

The function approximation experiment focused on the relative performance of real-valued network architectures to quaternion networks with pure quaternion activation functions. The comparison experiment operated on three distinct network architecture and training algorithm combinations. The first is the quaternion multilayer perceptron trained with a genetic algorithm (from here on referred to as QMLP+GA). This network consists of an input layer, two hidden layers, and an output layer.

Between each layer of the network, a “normalization” step was added, where the output of each layer is individually normalized. Since the training data-points were encoded into quaternion values, the input and output layer require a single node each. The two hidden layers of the network contain 3 nodes each, resulting in a total of 22 trainable weights and biases for the network. The pure-quaternion hyperbolic tangent (\tanh) function was selected as the nonlinear activation function for the input layer and both hidden layers. The \tanh function in the quaternion domain is defined as:

$$\tanh(\mathbf{q}) = \frac{e^{2\mathbf{q}} - 1}{e^{2\mathbf{q}} + 1}, \quad \mathbf{q} \in \mathbb{H}. \quad (24)$$

To determine the loss at the output layer, the final output is first mapped from \mathbb{H} into \mathbb{R} using the norm defined in Section 2.2.2. This mapping allows for the use of any real-valued loss function, and the mean absolute error (MAE) loss function was selected due to its simplicity. The MAE is given by:

$$\frac{1}{N} \sum_{i=1}^N |\hat{y} - y|, \quad (25)$$

where N is the number of data-points, \hat{y} is the predicted value, and y is the truth or target value.

To provide a baseline comparison for the QMLP+GA network, an equivalent real-valued network is constructed and trained using the same genetic algorithm as the QMLP+GA. Finally, an identical MLP is constructed and trained using the gradient descent (GD) algorithm. These two variants are referred to as the MLP+GA network and the

MLP+GD network, respectively. The layers, neurons per layer, and total parameters of each of the three networks are summarized in Table 1.

Table 1. Neural network topologies for the Ackley Function approximation.

| Network | Input | Hidden 1 | Hidden 2 | Output | Parameters |
|---------|-------|----------|----------|--------|------------|
| QMLP+GA | 1 | 3 | 3 | 1 | 22 |
| MLP+GA | 3 | 9 | 9 | 3 | 136 |
| MLP+GD | 3 | 9 | 9 | 3 | 136 |

The real-valued hyperbolic tangent was used as the activation function on the input layer and both hidden layers, with a MAE loss function. However, since the hyperbolic tangent is globally analytic in \mathbb{R} , the normalization layers from the QMLP were removed. The learning rate η for the gradient descent algorithm was set to $\eta = 0.03$. The real-valued MLPs contained a total of 136 trainable weight and bias parameters, a six-fold increase over the QMLP.

3.2.2. Chaotic Time Series Prediction

Chaotic time series prediction of the Lorenz attractor requires multidimensional input data as well as multidimensional output data. It is a notoriously difficult problem, especially considering the system's sensitivity to initial conditions. In contrast with the function approximation experiment, the time series prediction experiment focused on the ability of quaternion networks to learn complex multidimensional nonlinearities. To that end, the time series prediction experiment centered on optimizing a set of quaternion network hyperparameters and did not consider any equivalent real-valued networks.

To test the predictive capabilities of a simple QMLP+GA network, a set of 500 time series inputs were generated using a fixed-timestep 4th-order Runge–Kutta Ordinary Differential Equation (ODE) solver. The first 400 time series formed the training dataset, while the last 100 were held out for the test set. The starting point for each time series was randomly generated using a uniform $U[-10.0, 10.0]$ distribution for the x - and y -coordinates and a uniform $U[0.0, 10.0]$ distribution for the z -coordinates. Initial tests focused on relatively short time series inputs. Each series was generated over a range of 20 timesteps, and the first 10 values of each series formed the input training data, while the last 10 values formed the target values for training.

Figure 4 illustrates the sensitivity of the Lorenz system to initial starting conditions. Several initial starting points were generated using the distributions defined above for the x -, y -, and z -coordinates. Each system was then solved for 500 timesteps, starting at the initial position in 3-space. While each curve exhibits the characteristic “butterfly” shape, the individual coordinates of each series at each time step are drastically different.

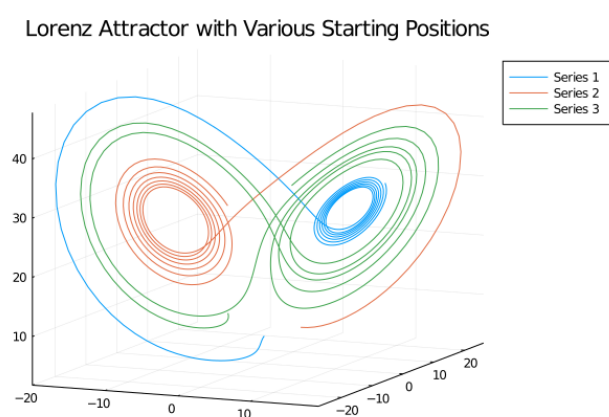


Figure 4. Impact of initial conditions on the Lorenz system.

Initial experiments showed that simple, smaller networks performed better with the genetic algorithm than larger networks. A 4-layer network was constructed for the time series prediction experiment. The structure of the network closely resembles an autoencoder network, where large input layers are scaled down throughout the network before being scaled back up for the output layer. This structure proved successful over several rounds of experimentation in predicting the 10-step ahead x , y , and z coordinates for the test set data. As a final experiment, a QMLP was created to predict the Lorenz coordinates 50 steps ahead based on an input time series of 25 steps. The layers, neurons per layer, and total parameters of each network are summarized in Table 2.

Table 2. Neural network topology for chaotic prediction.

| Network | Input | Hidden 1 | Hidden 2 | Output | Parameters |
|---------|-------|----------|----------|--------|------------|
| QMLP+GA | 10 | 3 | 3 | 10 | 85 |
| QMLP+GA | 25 | 5 | 10 | 50 | 740 |

Before processing through the network, the training and test datasets were cast into the quaternion domain using a vectorized approach. For an input vector τ_i , the corresponding quaternion input vector was constructed using the following approach:

$$\tau_i = \begin{bmatrix} \vec{x}_1 \\ \vec{x}_2 \\ \vdots \\ \vec{x}_{10} \end{bmatrix} \Rightarrow \tau_{qi} = \begin{bmatrix} 0.0 + x_1\mathbf{i} + y_1\mathbf{j} + z_1\mathbf{k} \\ 0.0 + x_2\mathbf{i} + y_2\mathbf{j} + z_2\mathbf{k} \\ \vdots \\ 0.0 + x_{10}\mathbf{i} + y_{10}\mathbf{j} + z_{10}\mathbf{k} \end{bmatrix}. \quad (26)$$

Additionally, the target values were cast into quaternions. At each iteration, a quaternionic form of the MAE measured the fitness of each solution. Only the imaginary components of each input and target vector contained coordinate information, so this experiment introduced a $QMAE_{imag}$ calculation, defined in Equation (27) below.

$$\begin{aligned} QMAE_{imag} &:= \frac{1}{N} \sum_{i=1}^N \|\hat{y}_{\mathbf{q}i} - y_{\mathbf{q}i}\|_{imag} \\ &= \frac{1}{N} \sum_{i=1}^N \|(\hat{x}_i\mathbf{i} + \hat{y}_i\mathbf{j} + \hat{z}_i\mathbf{k}) - (x_i\mathbf{i} + y_i\mathbf{j} + z_i\mathbf{k})\| \\ &= \frac{1}{N} \sum_{i=1}^N \left(\sqrt{(\hat{x}_i - x_i)^2 + (\hat{y}_i - y_i)^2 + (\hat{z}_i - z_i)^2} \right)^2. \end{aligned} \quad (27)$$

Since this experiment did not consider any real-valued networks, several quaternion activation functions were utilized during testing that are not available as activation functions in the real-domain. In particular, Ref. [46] notes that quaternionic functions with local analytic conditions are isomorphic to analytic complex functions. Additionally, Ref. [20] demonstrate that hyperbolic and inverse hyperbolic trigonometric functions are universal approximators in the complex domain. This experiment explored the use of several quaternionic elementary transcendental functions and found the inverse hyperbolic tangent, defined in [47], to provide the best performance:

$$\operatorname{arctanh}(p) := \frac{\ln(1+p) - \ln(1-p)}{2}. \quad (28)$$

Whereas the Lorenz prediction QMLP+GA networks required a slightly different network structure than the Ackley function approximation networks, both networks employed an identical genetic algorithm in the training phase. This approach eliminated the need for differentiability of both the loss function and the activation functions of the network. Additionally, it eliminated the need for a quaternion partial derivative calculation, which

is a notoriously difficult problem. Section 3.3 describes the details of the algorithm, while Section 4 discusses the results and performance of the algorithm in both experiments.

3.3. Quaternion Genetic Algorithm

This section describes the quaternion genetic algorithm that was developed to train the QMLP-GA. A simple change of the underlying data type from quaternions to real-valued inputs, weights, and biases enabled the training of the MLP-GA with an identical algorithm. This research took a similar approach to Uber’s OpenAI Labs genetic algorithm training process [33], opting for a very basic algorithm with minimal enhancements to demonstrate the proof-of-concept. Based on the success of this approach in Uber’s experiments as well as in the quaternion domain presented here, a more advanced algorithm incorporating any of the many algorithmic improvements would likely improve on the baseline results discussed in Section 4.

A general diagram of the genetic algorithm process flow is shown in Figure 5. A genetic algorithm is a population-based search method, operating on a population of solutions to iteratively find improving solutions. In this case, an individual neural network, defined by its weights and biases, represents a single solution. To initialize the algorithm, a population of $N = 20$ distinct neural networks was instantiated, with all weights and biases randomly generated following a uniform distribution over $[-1, 1]$.

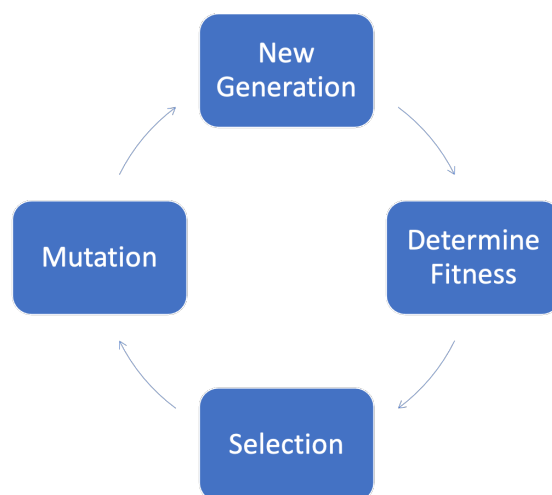


Figure 5. Genetic algorithm/genetic programming process.

After instantiation, the algorithm measures the fitness of each solution. For each neural network, the entire training dataset is processed through the network, capturing the total MAE for each network. The networks are then rank-ordered based on the lowest MAE value.

In the selection step, the n best solutions are retained as the “parents” for the next generation of the algorithm. In this research, $n = 5$ networks were retained as the parent generation in each iteration of the algorithm. While many advanced selection techniques exist, this work employed a simple rank selection, which selected the five best networks from each generation.

Finally, to generate a new population of solutions, the genetic algorithm performs a random mutation step, where a parent solution is randomly selected from the $n = 5$ best parent solutions. Then, the algorithm creates a “child” solution by mutating roughly half of the weights and biases of the parent solution with random noise. In this case, the generating distribution for the random noise was the standard normal distribution, $\mathcal{N}(0, 1)$. This process repeats for $N - n = 20 - 5 = 15$ times to create a new generation of solutions.

This process is commonly referred to as a genetic program, where generations are created solely through the mutation process. Often, genetic algorithms will include an additional crossover step prior to mutation, where new child solutions are created using

a selection of features from separate parent solutions. Crossover was omitted from this algorithm, since mutation alone provided a good baseline performance, reiterating the fact that the most simple genetic algorithms are competitive to the popular backpropagation algorithm. A summary of the algorithm is shown in Algorithm 1. Additional details of the genetic algorithm along with a brief comparison of the computational effort required for the genetic algorithm versus classic gradient descent are provided in Appendix A.

Algorithm 1 Quaternion Genetic Algorithm

```

1: Instantiate  $\mathcal{P}_m$  parent networks,  $m \in N = \{1, \dots, 20\}$ , input mutation function  $\psi$ .
2: for  $i \in N$  do
3:   Evaluate population fitness  $F_i$ 
4: end for
5: for  $g = 1$  to  $G$  generations do
6:   Sort population  $\leftarrow F_i$ 
7:   Select best parents  $\mathcal{P}_n^{g-1}$ ,  $n = 1, \dots, 5$ 
8:   for  $j = n + 1$  to  $N$  do
9:     Generate  $k = \text{UniformInt}(1, n)$ 
10:     $\mathcal{P}_j^g = \psi(\mathcal{P}_k^{g-1})$ .
11:   end for
12: end for
13: Return final population  $\mathcal{P}_m^G$  for  $m \in N$ .
  
```

3.4. Evaluation and Analysis Strategy

Each of the networks described in Section 3.2 processed the training data from the Ackley function and the Lorenz attractor system. At each training epoch, the algorithms either recorded the MAE of the overall system in the case of the gradient descent network, or the MAE or (QMAE) of the best solution for the genetic algorithm networks. Additionally, several computational metrics were recorded including memory allocations and computational runtime. Finally, each of the trained models processed the test data, recording the test set percentage error for each instance. Section 4 contains a discussion of network performance in each problem instance for each network in regards to these metrics.

4. Results

All computations presented here were performed on a desktop workstation running Windows 10 Enterprise with 64 GB of RAM and dual Intel Xeon Silver 4108 CPUs. Each CPU contained eight physical cores running at 1.80 GHz. Coding was performed in Julia 1.5.3 using the Quaternion.jl package and Flux.jl [48] for the MLP+GD network.

4.1. Function Approximation Results

The focus of the function approximation test was twofold. First, the function approximation task served as a proof-of-concept for the QMLP-GA. While quaternion neural networks and metaheuristic neural network training algorithms both exist separately in the literature, this work demonstrates the first use of metaheuristics to effectively train quaternion neural networks. Second, this experiment demonstrated some of the computational benefits that quaternions provide.

In keeping with these two goals, the three neural networks employed default parameters and very basic training algorithm implementations. No attempt was made to tune the hyperparameters of any of the models; instead, the results speak for themselves. The training set error for each of the three networks versus epoch is shown in Figure 6.

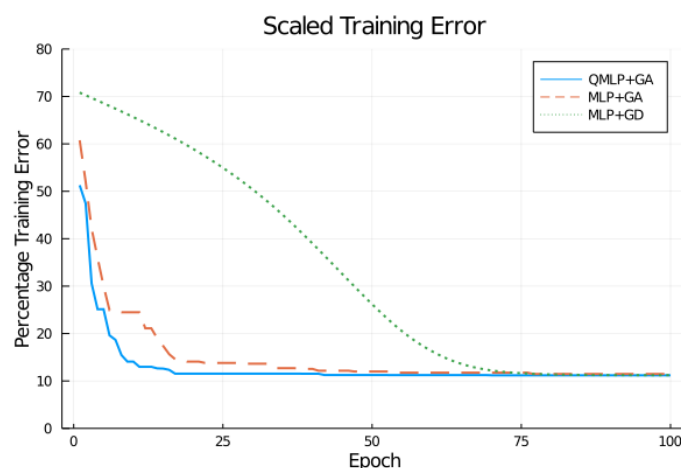


Figure 6. The training set mean absolute error for each network.

The QMLP+GA initialized using the random uniform weight initialization scheme described in Section 3 had the lowest initial prediction error, at roughly 50% in the first epoch. In contrast, the MLP+GA started with a nearly 60% initial error, while the MLP+GD was above 70%. The genetic algorithm improved rapidly, showing significantly faster initial algorithmic improvement versus the gradient descent algorithm. Both GA-trained networks showed rapid improvements over the first 25 training epochs, while the MLP+GD network searched for nearly 75 epochs before catching up to the GA-trained networks. The MLP+GD eventually caught up to the other two networks, but the prediction error remained slightly higher for the gradient descent network throughout the entire training process.

Table 3 shows the test set performance for each of the three networks across several measures of merit. In each column, the best results are highlighted in bold text. The quaternion network had the fastest overall runtime, resulting in the lowest test set error with the fewest number of trainable parameters. The real-valued MLP had a similar performance and required less overall system memory throughout the runtime of the algorithm, but required nearly six times the number of trainable parameters. Finally, the gradient descent-trained MLP had the worst performance in every category. While the test set error was comparable to the other two networks, the MLP+GD took more than 50 times as long to run with over 70 times as much memory allocated to store the gradient and error information for the backpropagation process.

These results, while cursory, clearly demonstrate the viability of quaternion networks trained with genetic algorithms. The quaternion network showed the fastest overall improvement, lowest final error, and lowest computational cost (in terms of runtime) when compared to two comparable networks. Additionally, the two GA-trained networks outperformed the gradient descent network across all measures of merit. These results validate the use of genetic algorithms in neural network training and show that quaternion networks can easily outperform equivalent real-valued networks involving multidimensional input data.

Table 3. Neural network comparison results.

| Network | Runtime (s) | Memory (GB) | Parameters | Test Error |
|---------|---------------|--------------|------------|---------------|
| QMLP+GA | 17.421 | 10.238 | 22 | 11.01% |
| MLP+GA | 18.069 | 9.497 | 136 | 11.15% |
| MLP+GD | 955.040 | 778.027 | 136 | 11.23% |

4.2. Time Series Prediction Results

While the function approximation results demonstrate a viable proof-of-concept for quaternion neural networks, the chaotic time series prediction task illustrates the power of QNNs in the difficult task of predicting noisy systems. Additionally, chaotic time series

prediction provides a natural multidimensional input + multidimensional output test that is almost tailor made for quaternion networks. In each of the figures displayed in this section, the orange graph represents the true chaotic time series, while the blue graph represents the predicted values. The final prediction results presented in Figure 7 are far from current state-of-the-art results using deep recurrent neural networks (RNNs) or long-short term memory (LSTM) networks, yet they illustrate the ability of simple QNNs to learn complex nonlinearities over time.

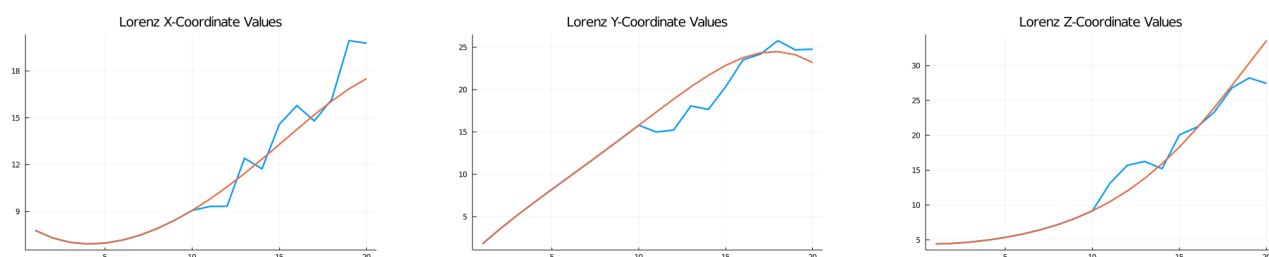


Figure 7. 10-step ahead predicted coordinate values.

This experiment utilized two distinct QNN network topologies. The first network predicted the Lorenz attractor for 10 timesteps in the future based on an input time series of 10 timesteps. The second network predicted the Lorenz attractor for 50 timesteps in the future based on an input of 25 observations. The structure of each network is listed in Table 2, while the results for both networks are listed in Table 4. The test error percentage listed in Table 4 was measured using the mean absolute percentage error (MAPE) for time series forecasting, defined in Equation (29), where e_t is the unscaled prediction error for observation t and y_t is the target value at t :

$$MAPE = \text{mean} \left(\left| 100 \frac{e_t}{y_t} \right| \right). \quad (29)$$

Early tests indicated that smaller networks performed better with the genetic algorithm. The final two networks contained comparatively few nodes in each layer and were structured as autoencoder networks, which perform a type of downsampling and subsequent upsampling as information passes through the network. Each network was trained for 50,000 epochs, which equated to roughly 28 min for the 10-step prediction network and around 4 h for the 50-step prediction network.

Table 4. Lorenz prediction results.

| Prediction Steps | Runtime (s) | Memory (GB) | Params | Test Error |
|------------------|-------------|-------------|--------|------------|
| 10 | 1668.565 | 947.304 | 85 | 10.89% |
| 50 | 14769.069 | 2.815 (TB) | 740 | 9.59% |

The test set error listed in Table 4 indicates that on average, individual predicted values were off by about 11%. The actual versus predicted x -, y -, and z -coordinates for one of the test set time series are shown in Figure 7, while two 3-dimensional path predictions are shown in Figure 8. While the test error is relatively high, the QMLP+GA performs remarkably well on future predictions, especially in the long sweeping sections of the Lorenz attractor curves. The errors understandably grow and compound in the two “wings” of the curve, where the graph circles closely around each pole of the attractor.

The final experiment tested the ability of the QMLP to predict long sequences based on a relatively short input. The network was trained over 50,000 epochs to predict 50 observations based on an input sequence of length 25. Table 4 summarizes several measures of merit for the network, while the x -, y -, and z -coordinate results for a representative test set sequence are shown in Figure 9.

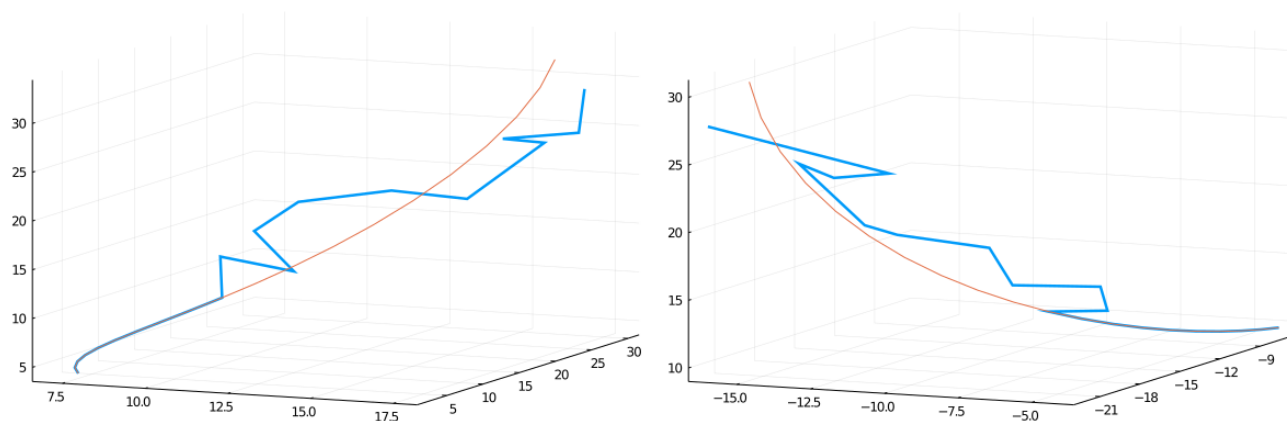


Figure 8. 10-step ahead path predictions.

In each coordinate direction, there is some clear noise at each prediction step, but the network accurately predicts the general motion of each variable. The motion of each prediction path is even more evident in the 3-dimensional plots shown in Figure 10, which shows two path predictions for two series from the test set data. As with the 10-step prediction model, the 50-step model makes the best predictions along the long sweeping arcs of the system, with errors compounding near the two “wings” of the attractor.

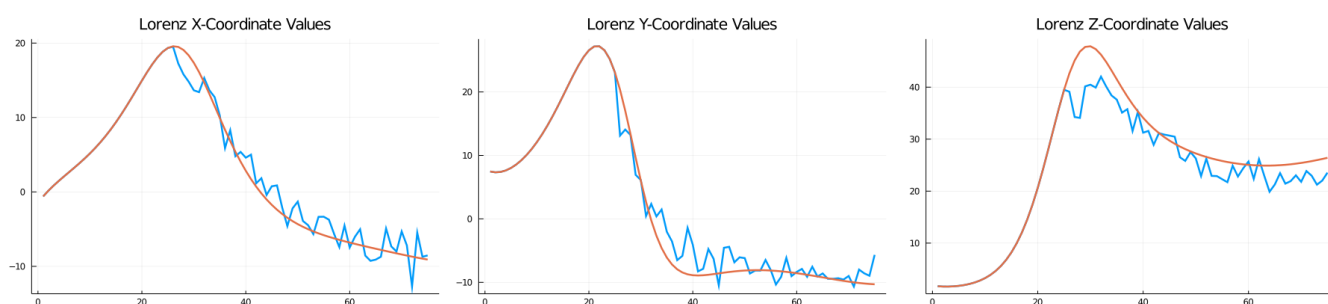


Figure 9. 50-step ahead predicted coordinate values.

Finally, the unscaled training error plots for both networks are shown in Figure 11. The genetic algorithm showed similar performance in both time series prediction tasks as it did in the function approximation task, with dramatic initial improvements and slow but consistent improvements as the iterations progressed. Surprisingly, the 50-step prediction experiment resulted in a lower test set prediction error than the 10-step prediction network, likely due to the scale of each predicted value.

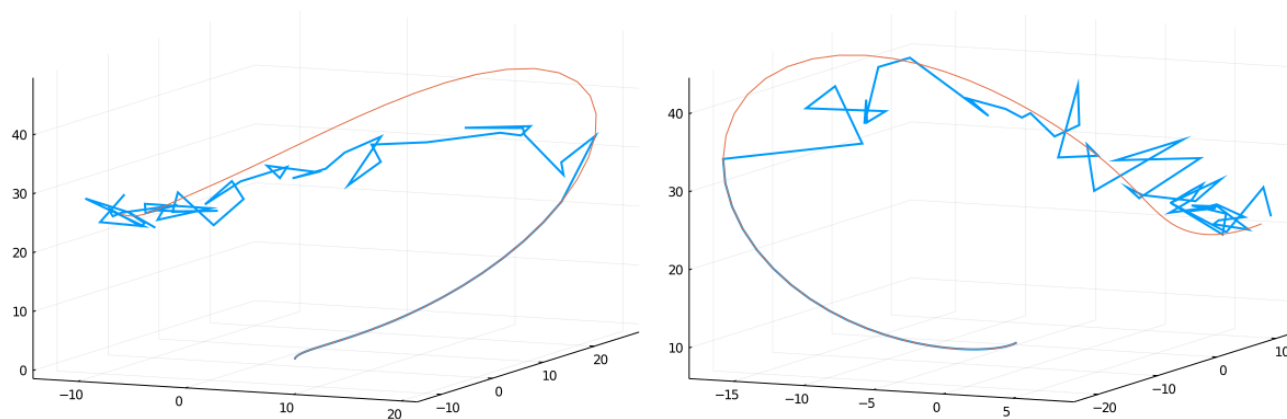


Figure 10. 50-step ahead path predictions.

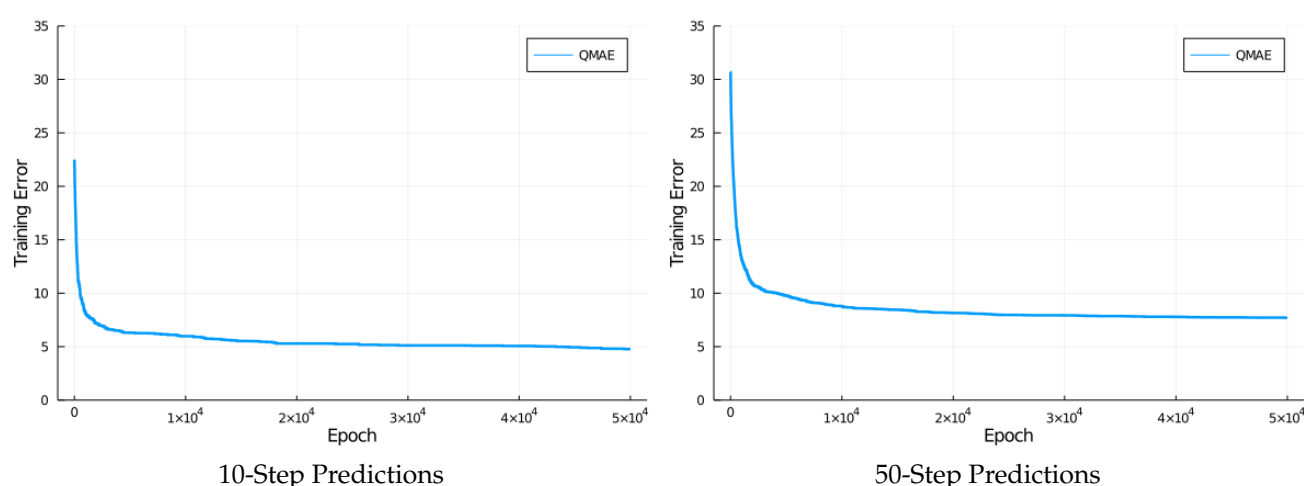


Figure 11. Unscaled QMAE training error.

5. Discussion

In the Ackley function approximation experiment, all three networks utilized a random uniform weight initialization scheme. However, the quaternion network had between a 10–20% lower initial prediction error than the real-valued networks. This is likely due to the fact that the quaternion network employed six times fewer weight and bias parameters than the real-valued networks. The quaternion network maintained the lowest training set error across the entire 100-epoch training period, resulting in the best test set performance. The larger networks constructed in the second experiment demonstrated similar training characteristics and test set performance.

The genetic algorithm removes the need for expensive gradient calculations, resulting in better memory performance and more than 50x faster runtime in the first experiment versus the real-valued gradient descent algorithm. Given the difficulty of calculating quaternion gradients, the improvement over a quaternion gradient descent algorithm would likely be even greater. However, a genetic programming approach does come with some drawbacks. In the naive approach presented here, the algorithm would sometimes stall for several iterations while searching for an improving solution. There are many existing techniques designed to mitigate this stalling, but the literature on genetic algorithms is much less developed compared to comparable work on gradient descent optimization.

Despite this, the genetic algorithm opened the aperture on viable activation functions and loss functions for use with quaternion networks. This is perhaps the most significant contribution of this research. The results from [46] indicate that any locally analytic complex-valued activation function can be extended and used in the quaternion domain, but this work presents the first successful implementation of inverse hyperbolic trigonometric functions in quaternion networks. The success of the inverse hyperbolic tangent function in the chaotic time series prediction task demonstrates the value of using gradient free optimization methods in the quaternion domain.

The quaternions and quaternion neural networks are relatively unexplored compared to real analysis and real-valued neural networks. While certain applications in image processing and other domains have driven research in the quaternions and QNNs, there is still room for significant improvement in both the theoretical and practical aspects of quaternions. Going forward, the following lines of research will be crucial for continued innovation in the quaternion domain.

First, a solid foundation of quaternionic analysis is crucial to theoretically sound QNN research. While a handful of researchers have published works on quaternionic analysis, the corpus is quite thin. Research in novel quaternion activation functions, quaternion differentiability, quaternion analytic conditions, and novel quaternion training algorithms could significantly enhance both the current understanding of quaternion optimization as well as quaternion implementations of common machine learning models. Additionally,

the quaternion Universal Approximation Theorem for either split or pure quaternion activation functions is an outstanding problem that is vital for establishing the legitimacy of quaternion networks from a theoretical point of view. Proving either variant of the Universal Approximation Theorem would be a substantial contribution to the field.

Finally, this research simply provided a proof-of-concept for GA-trained quaternion neural networks. The two examples presented were limited in scope and future work should build on these results to demonstrate the viability of GA-trained networks in large-scale optimization problems. In particular, quaternions are particularly well suited to the fields of image processing and robotic control, both of which have a plethora of neural network-related application opportunities. The authors intend to build on this proof-of-concept in future work by examining the scalability of quaternion GAs to large machine learning datasets and an in-depth comparative analysis of real-valued versus quaternion-valued neural networks using a design of experiments (DoE) hyperparameter-tuning approach. Finally, the authors intend to apply GA-trained QNNs to problem domains for which quaternions are particularly well suited, including 3D optimal satellite control and reinforcement learning for autonomous flight models.

Author Contributions: Conceptualization, J.B. and L.C.; methodology, J.B. and L.C.; software, J.B.; validation, J.B., L.C. and B.C.; formal analysis, J.B. and L.C.; investigation, J.B.; resources, L.C.; data curation, J.B.; writing—original draft preparation, J.B. and L.C.; writing—review and editing, J.B., L.C., B.C. and T.B.; visualization, J.B.; supervision, L.C.; project administration, L.C.; funding acquisition, L.C., B.C. and T.B. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded in part by Air Force Research Laboratory, WPAFB, OH, USA.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Appendix A. MLP-GD and MLP-GA Pseudocode

This section provides a high-level overview of the main algorithmic steps for the genetic algorithm (GA) and gradient descent (GD) neural network training algorithms. In terms of computational effort, the main differences between the two algorithms are that the GA requires a population of different neural networks while the GD algorithm only requires a single network but instead computes the error gradients at each training iteration. The main computational burden of the GA stems from processing the training data through each network at every iteration of the algorithm. In contrast, the GD algorithm's main computational effort stems from the calculation of expensive partial derivatives to determine the error gradient at each layer of the network for every iteration. In practice, the computational cost of the backpropagation step in the GD algorithm outweighs the repeated processing of training data through each network in the GA. The results in Section 4.1 provide a good demonstration of this.

Appendix A.1. MLP-GA

```

# Network Structure contains weight/bias values
#   that define each layer of the neural network

struct network
    bias1
    Weight1
    bias2
    Weight2
    bias3
    Weight3
    bias4
    fitness
end

# initialization function
function init() do
    instantiate n random networks
    return list of networks
end

# main forward pass
function update_fitness(X, Y, Networks) do
    foreach network in Networks do
        foreach x in X do
            y_predicted = network output
            error = |y_predicted - Y|
        end
        network.fitness = sum(error)
    end
end

# mutation operator
function mutate_weights(weight_array) do
    foreach weight in weight_array do
        weight = weight + random_noise
    end
    return weight_array
end

# Genetic Algorithm
function GA(X, Y) do
    population = init()
    n_epochs = N

    for i = 1:N do
        update_fitness(X, Y, population)
        sort(population, on = fitness, order = ascending)

        # retain the best k networks as parents
        # mutate parent weights to create new children
        for j = k+1:length(population) do
            rand = rand(1:k)
            population[j] = mutate_weights(population[rand])
        end
    end
end

```

```

        end
    end

    update_fitness(population)
    sort(population, on = fitness, order = ascending)
    best_entity = population[1]

    return best_entity
end

Appendix A.2. MLP-GD

# Network Structure contains weight/bias values
#   that define each layer of the neural network

struct network
    bias1
    Weight1
    bias2
    Weight2
    bias3
    Weight3
    bias4
end

# initialization function
function init() do
    instantiate single random network
    return network
end

# main forward pass
function forward_pass(X, Y) do
    foreach x in X do
        y_predicted = network output
        error = |y_predicted - Y|
    end
    return error
end

# main backward pass
function backpropagation(network, error, eta)do
    foreach layer in network do
        gradient(error, layer) = partial derivative of error
        layer.weights = layer.weights - eta*gradient
    end
    return network
end

# Training Algorithm
function train(X, Y) do
    network = init()
    n_epochs = N
    eta = learning rate

    for i = 1:N do

```

```

        error = forward_pass(X, Y)
        network = backpropagation(network, error, eta)
    end

    return network
end

```

References

1. Rosenblatt, F. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychol. Rev.* **1958**, *65*, 386. [\[CrossRef\]](#) [\[PubMed\]](#)
2. Cybenko, G. Approximation by superpositions of a sigmoidal function. *Math. Control Signals Syst.* **1989**, *2*, 303–314. [\[CrossRef\]](#)
3. Hornik, K.; Stinchcombe, M.; White, H. Multilayer feedforward networks are universal approximators. *Neural Netw.* **1989**, *2*, 359–366. [\[CrossRef\]](#)
4. Géron, A. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*; O'Reilly Media: Sebastopol, CA, USA 2019.
5. Hosseini, S.H.; Samanipour, M. Prediction of final concentrate grade using artificial neural networks from Gol-E-Gohar iron ore plant. *Am. J. Min. Metall.* **2015**, *3*, 58–62.
6. Rumelhart, D.E.; Hinton, G.E.; Williams, R.J. *Learning Internal Representations by Error Propagation*; Technical Report; California University San Diego La Jolla Institute for Cognitive Science: San Diego, CA, USA, 1985.
7. Werbos, P. Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences. Ph.D. Thesis, Harvard University, Cambridge, MA, USA, 1974.
8. Karras, T.; Laine, S.; Aittala, M.; Hellsten, J.; Lehtinen, J.; Aila, T. Analyzing and Improving the Image Quality of StyleGAN. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, Seattle, WA, USA, 14–19 June 2020.
9. Deng, J.; Dong, W.; Socher, R.; Li, L.J.; Li, K.; Fei-Fei, L. ImageNet: A Large-Scale Hierarchical Image Database. In Proceedings of the 2009 IEEE Conference on Computer Vision and Pattern Recognition, Miami, FL, USA, 20–25 June 2009.
10. Oh, K.S.; Jung, K. GPU implementation of neural networks. *Pattern Recognit.* **2004**, *37*, 1311–1314. [\[CrossRef\]](#)
11. LeCun, Y.; Bottou, L.; Bengio, Y.; Haffner, P. Gradient-based learning applied to document recognition. *Proc. IEEE* **1998**, *86*, 2278–2324. [\[CrossRef\]](#)
12. Yin, Q.; Wang, J.; Luo, X.; Zhai, J.; Jha, S.K.; Shi, Y.Q. Quaternion convolutional neural network for color image classification and forensics. *IEEE Access* **2019**, *7*, 20293–20301. [\[CrossRef\]](#)
13. Matsui, N.; Isokawa, T.; Kusamichi, H.; Peper, F.; Nishimura, H. Quaternion neural network with geometrical operators. *J. Intell. Fuzzy Syst.* **2004**, *15*, 149–164.
14. Brown, J.W.; Churchill, R.V. *Complex Variables and Applications*; McGraw-Hill Higher Education: Boston, MA, USA, 2009.
15. Hamilton, W.R. LXXVIII. On quaternions; or on a new system of imaginaries in Algebra: To the editors of the Philosophical Magazine and Journal. *Lond. Edinb. Dublin Philos. Mag. J. Sci.* **1844**, *25*, 489–495. [\[CrossRef\]](#)
16. Kuipers, J.B. *Quaternions and Rotation Sequences: A Primer with Applications to Orbits, Aerospace, and Virtual Reality*; Princeton University Press: Princeton, NJ, USA, 1999.
17. Arena, P.; Fortuna, L.; Occhipinti, L.; Xibilia, M.G. Neural networks for quaternion-valued function approximation. In Proceedings of the IEEE International Symposium on Circuits and Systems-ISCAS'94, London, UK, 30 May–2 June 1994; Volume 6, pp. 307–310.
18. Ujang, B.C.; Jahanchahi, C.; Took, C.C.; Mandic, D. Quaternion valued neural networks and nonlinear adaptive filters. *IEEE Trans. Neural Netw.* **2010**, submission. [\[CrossRef\]](#) [\[PubMed\]](#)
19. Xu, D.; Zhang, L.; Zhang, H. Learning algorithms in quaternion neural networks using ghr calculus. *Neural Netw. World* **2017**, *27*, 271. [\[CrossRef\]](#)
20. Kim, T.; Adalı, T. Approximation by fully complex multilayer perceptrons. *Neural Comput.* **2003**, *15*, 1641–1666. [\[CrossRef\]](#) [\[PubMed\]](#)
21. Xu, D.; Jahanchahi, C.; Took, C.C.; Mandic, D.P. Enabling quaternion derivatives: The generalized HR calculus. *R. Soc. Open Sci.* **2015**, *2*, 150255. [\[CrossRef\]](#)
22. Arena, P.; Fortuna, L.; Re, R.; Xibilia, M.G. On the capability of neural networks with complex neurons in complex valued functions approximation. In Proceedings of the 1993 IEEE International Symposium on Circuits and Systems, Chicago, IL, USA, 3–6 May 1993; pp. 2168–2171.
23. Isokawa, T.; Kusakabe, T.; Matsui, N.; Peper, F. Quaternion neural network and its application. In Proceedings of the International Conference on Knowledge-Based and Intelligent Information and Engineering Systems, Oxford, UK, 3–5 September 2003; Springer: Berlin/Heidelberg, Germany, 2003; pp. 318–324.
24. Greenblatt, A.; Mosquera-Lopez, C.; Agaian, S. Quaternion neural networks applied to prostate cancer gleason grading. In Proceedings of the 2013 IEEE International Conference on Systems, Man, and Cybernetics, Manchester, UK, 13–16 October 2013; pp. 1144–1149.

25. Parcollet, T.; Morchid, M.; Bousquet, P.M.; Dufour, R.; Linarès, G.; De Mori, R. Quaternion neural networks for spoken language understanding. In Proceedings of the 2016 IEEE Spoken Language Technology Workshop (SLT), San Diego, CA, USA, 13–16 December 2016; pp. 362–368.
26. Buchholz, S.; Le Bihan, N. Optimal separation of polarized signals by quaternionic neural networks. In Proceedings of the 2006 14th European Signal Processing Conference, Florence, Italy, 4–8 September 2006; pp. 1–5.
27. Fortuna, L.; Muscato, G.; Xibilia, M.G. A comparison between HMLP and HRBF for attitude control. *IEEE Trans. Neural Netw.* **2001**, *12*, 318–328. [[CrossRef](#)] [[PubMed](#)]
28. Parcollet, T.; Morchid, M.; Linares, G. A survey of quaternion neural networks. *Artif. Intell. Rev.* **2020**, *53*, 2957–2982. [[CrossRef](#)]
29. Gaudet, C.J.; Maida, A.S. Deep quaternion networks. In Proceedings of the 2018 International Joint Conference on Neural Networks (IJCNN), Rio de Janeiro, Brazil, 8–13 July 2018; pp. 1–8.
30. Parcollet, T.; Ravanelli, M.; Morchid, M.; Linarès, G.; Trabelsi, C.; De Mori, R.; Bengio, Y. Quaternion recurrent neural networks. *arXiv* **2018**, arXiv:1806.04418.
31. Stanley, K.O.; Miikkulainen, R. Evolving Neural Networks through Augmenting Topologies. *Evol. Comput.* **2002**, *10*, 99–127. [[CrossRef](#)]
32. Stanley, K.O.; D'Ambrosio, D.B.; Gauci, J. A hypercube-based encoding for evolving large-scale neural networks. *Artif. Life* **2009**, *15*, 185–212. [[CrossRef](#)]
33. 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* **2017**, arXiv:1712.06567.
34. Ojha, V.K.; Abraham, A.; Snášel, V. Metaheuristic design of feedforward neural networks: A review of two decades of research. *Eng. Appl. Artif. Intell.* **2017**, *60*, 97–116. [[CrossRef](#)]
35. Fister, I.; Yang, X.S.; Brest, J.; Fister, I., Jr. Modified firefly algorithm using quaternion representation. *Expert Syst. Appl.* **2013**, *40*, 7220–7230. [[CrossRef](#)]
36. Papa, J.; Pereira, D.; Baldassin, A.; Yang, X.S. On the harmony search using quaternions. In Proceedings of the IAPR Workshop on Artificial Neural Networks in Pattern Recognition, Ulm, Germany, 31 August–2 September 2006; Springer: Berlin/Heidelberg, Germany, 2016; pp. 126–137.
37. Papa, J.P.; de Rosa, G.H.; Yang, X.S. On the Hypercomplex-Based Search Spaces for Optimization Purposes. In *Nature-Inspired Algorithms and Applied Optimization*; Springer: Berlin/Heidelberg, Germany, 2018; pp. 119–147.
38. Khuat, T.T.; Le, M.H. A genetic algorithm with multi-parent crossover using quaternion representation for numerical function optimization. *Appl. Intell.* **2017**, *46*, 810–826. [[CrossRef](#)]
39. Papa, J.P.; Rosa, G.H.; Pereira, D.R.; Yang, X.S. Quaternion-based deep belief networks fine-tuning. *Appl. Soft Comput.* **2017**, *60*, 328–335. [[CrossRef](#)]
40. Ackley, D. *A Connectionist Machine for Genetic Hillclimbing*; Springer Science & Business Media: Berlin/Heidelberg, Germany, 2012; Volume 28.
41. Lorenz, E.N. Deterministic nonperiodic flow. *J. Atmos. Sci.* **1963**, *20*, 130–141. [[CrossRef](#)]
42. Arena, P.; Fortuna, L.; Muscato, G.; Xibilia, M.G. *Neural Networks in Multidimensional Domains: Fundamentals and New Trends in Modelling and Control*; Springer: Berlin/Heidelberg, Germany, 1998; Volume 234.
43. Arena, P.; Caponetto, R.; Fortuna, L.; Muscato, G.; Xibilia, M.G. Quaternionic multilayer perceptrons for chaotic time series prediction. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.* **1996**, *79*, 1682–1688.
44. Arena, P.; Baglio, S.; Fortuna, L.; Xibilia, M. Chaotic time series prediction via quaternionic multilayer perceptrons. In Proceedings of the 1995 IEEE International Conference on Systems, Man and Cybernetics. Intelligent Systems for the 21st Century, Vancouver, BC, Canada, 22–25 October 1995; Volume 2, pp. 1790–1794.
45. Ujang, B.C.; Took, C.C.; Mandic, D.P. Split quaternion nonlinear adaptive filtering. *Neural Netw.* **2010**, *23*, 426–434. [[CrossRef](#)]
46. Isokawa, T.; Nishimura, H.; Matsui, N. Quaternionic Multilayer Perceptron with Local Analyticity. *Information* **2012**, *3*, 756–770. [[CrossRef](#)]
47. Morais, J.P.; Georgiev, S.; Sprößig, W. *Real Quaternionic Calculus Handbook*; Springer: Berlin/Heidelberg, Germany, 2014.
48. Innes, M. Flux: Elegant Machine Learning with Julia. *J. Open Source Softw.* **2018**. [[CrossRef](#)]