QUATERNION TEMPORAL CONVOLUTIONAL NEURAL NETWORKS

Thesis

Submitted to

The School of Engineering of the

UNIVERSITY OF DAYTON

In Partial Fulfillment of the Requirements for

The Degree of

Master of Science in Computer Engineering

By

Cameron Ellsworth Long

Dayton, Ohio

August, 2019

QUATERNION TEMPORAL CONVOLUTIONAL NEURAL NETWORKS

Name: Long, Cameron Ellsworth

APPROVED BY:

Vijayan K. Asari, Ph.D.
Advisory Committee Chairman
Professor, Electrical and Computer Engineering

Theus H. Aspiras, Ph.D.
Committee Member
Advisory Committee, Electrical and Computer Engineering

Eric J. Balster, Ph.D.
Committee Member
Associate Professor and Chair,
Electrical and Computer Engineering

Robert J. Wilkens, Ph.D., P.E.
Associate Dean for Research and Innovation
Professor
School of Engineering

Eddy M. Rojas, Ph.D., M.A., P.E.
Dean, School of Engineering

ABSTRACT

QUATERNION TEMPORAL CONVOLUTIONAL NEURAL NETWORKS

Name: Long, Cameron Ellsworth
University of Dayton

Advisor: Dr. Vijayan K. Asari

Sequence Processing and Modeling are a domain of problems recently recieving significant attention for significant advancements in research and technology. While traditionally sequence processing using neural networks has been done using a recurrent neural network such as the long-short term memory cell. These recurrent networks have some fairly large drawbacks. issue in networks is increasingly large networks, which have been proven to learn features from useless noise in their input data. A network called the Temporal Convolutional Network seeks to fix the issues that the long-short term memory cell have. While other recent research has been put into quaternion neural networks, networks that dramatically reduce the number of parameters in a network while keeping the same performance.

This thesis combines both these recent advancements into a Quaternion Temporal Convolutional Network. The network performance is evaluated on a wide range of sequence processing and modeling tasks and compared to the base Temporal Convolutional Network.

Through testing and evaluation it is shown that although there is a reduction in the number of learned parameters in the Temporal Convolutional network by up to 4x, the network performance stays relatively close, and actually beats the base network on some tasks.

For my Friends and Family, who have provided me endless support

ACKNOWLEDGMENTS

I would like to thank Dr. Asari, my advisor, for providing support and guidance as I completed my Degree.

I would further like to thank Deep Lens for the gpu time on their system, without which this thesis would never be completed.

Finally, I would like to thank the KeyW corporation, who has provided the funding with which I am earning this degree with.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

INTRODUCTION

Processing of sequential data using neural networks is a field that has recently gain significant traction, providing large advancements in tasks such as sequence classification, text translation, and machine reading comprehension.

While traditional sequential neural network architecures involved the use of recurrent style networks such as the Long-Short Term Memory Cell (LSTM), which keep a "memory" of important information as the network processes data, These networks have significant trade-offs in their memory usage, lack of parallelization, and large number of parameters required to "learn" on a dataset.

This thesis aims to build off of two recent contributions to the deep learning field:

1. The development of the Temporal Convolutional Neural Network (TCN), a causal fully convolutional network that seeks to overcome the weaknesses of the LSTM with better parallelization, memory usage, and reduced parameter use.
2. The development of the Quaternion Convolutional Neural Network (QCNN), a modification on a traditional neural network that seeks to dramatically reduce the number of learned parameters for a neural network at minimal performance cost.

By combining these two contributions a Quaternion Temporal Convolutional Network (QTCN) is proposed, a network architecture that combines the best of both network architectures, a network architecture that is better parallelizable, with low memory usage, and an extremely low number of parameters, the contributions of this thesis can be summarized as follows:

1. A method for using weight normalization on quaternion neural networks is proposed
2. The Quaternion Temporal Convolutional Network (QTCN) is proposed
3. Several tasks from several domains are evaluated to show the performance of the network in different scenarios
4. It is shown that even though the number of parameters for the QTCN is significantly reduced over the base TCN, the performance remains relatively close on many tasks

The rest of the thesis is structured as follows: In Chapter II an overview of standard neural network architectures that led to the development of the Temporal Convolutional Neural Network. Chapter III discusses Quaternion algebra and neural networks. Chapter IV describes the architecture of the Quaternion Temporal Convolutional Network. Chapter V describes the testing methodology and results. Finally Chaper VI concludes and describes potential future work.

CHAPTER II

BACKGROUND

In this chapter, an introduction in given on neural network developments up to the temporal convolutional network.

2.1 The Perceptron

A Single Layer Perceptron is the simplest form of neural network, it produces a single output from multiple inputs by producing a summation over the inputs multiplied by internal weights that the network learns, usually followed by some sort of activation function [3]. The equation for this can be written out as

$$\hat{y} = \sum_{i=1}^{m} (w_i x_i) + b \tag{2.1}$$

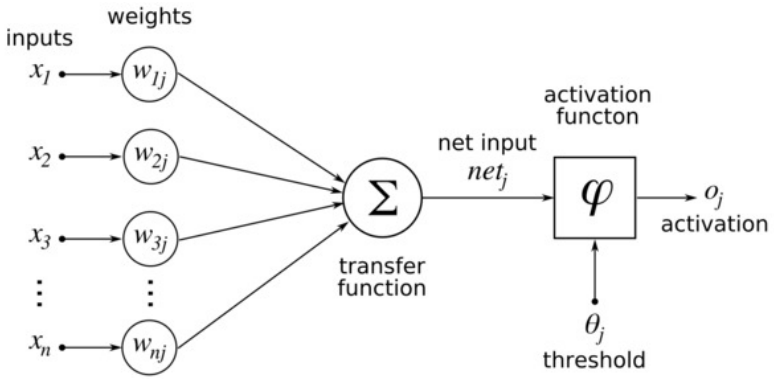


Figure 2.1: A Basic Perceptron

A single perceptron may be used to perform a binary classification problem, by producing a 0 or 1 for a result.

By using and 'stacking' many binary perceptrons a new type of network called a fully-connected network is formed, this network can be used to perform a multi-class classification problem by having a perceptron per class we which to classify, and then send the data through all the perceptrons, the perceptron with the highest value activation is the class you assign the input to.

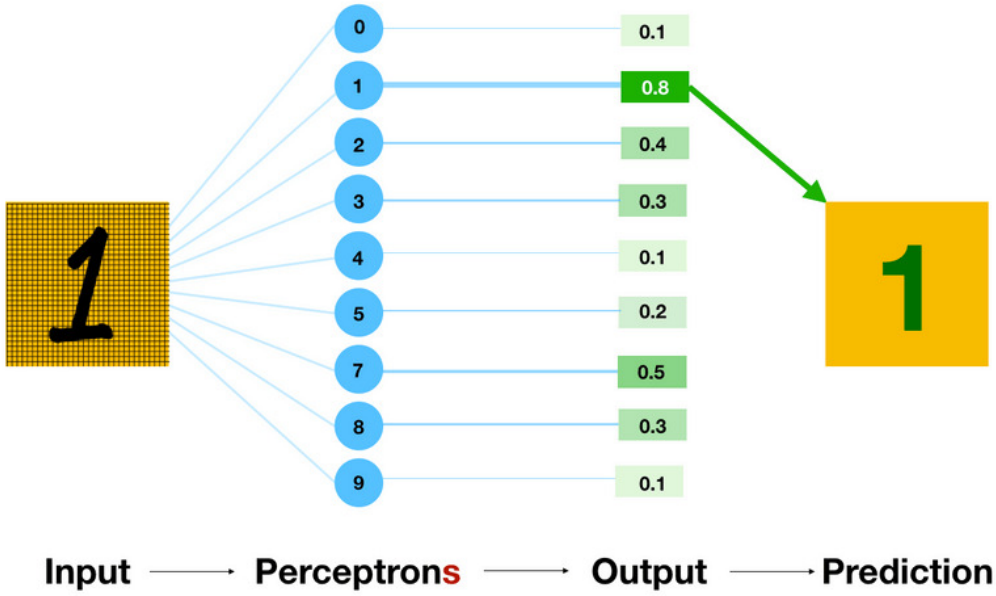


Figure 2.2: Example of Stacked Perceptrons

A common usage for a fully-connected network is using it at the end of other neural network operations to act as the final decision making layer in what class to select for a given input. In short, the network layers below the fully-connected layer act as a feature extractor to feed better data into the fully-connected network. When used this way the fully-connected network is referred to as a fully-connected layer, or dense layer.

A Perceptron may be turned into a Multi-layer Perceptron (MLP) by introducing layers of neurons called a hidden layer between the input and output layers, which allows for the network to learn "features" in the input data and typically increases performance to a certain point.

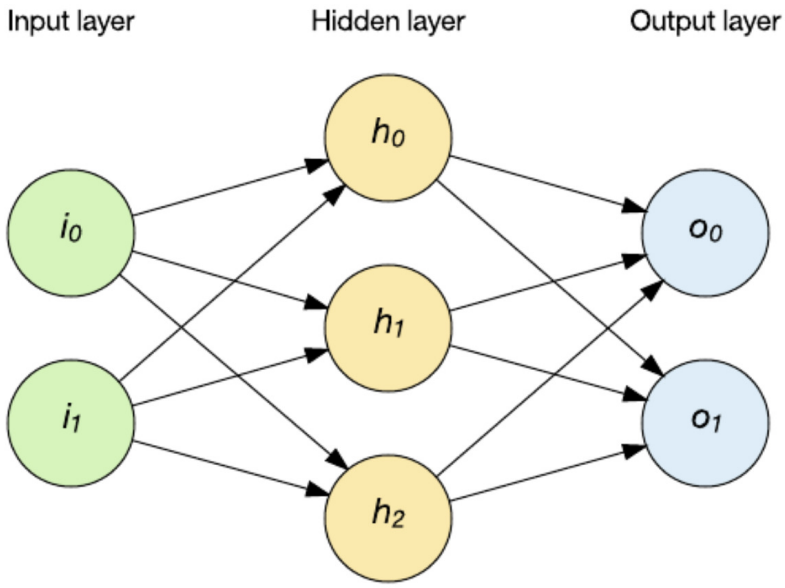


Figure 2.3: Simple MLP Network

2.2 Back Propagation and Learning

Weights for a perceptron can be hand selected and calculated for very small toy problems, however to use perceptrons and neural networks for practical usage we need to take advantage of a technique called back propagation. First popularized in [4], back propagation is a technique that allows networks to select weights themselves through a process known as "training".

If we have some way to evaluate how well a network is doing on a given task, we can update the weights of a network by calculating the gradients for any weight in a network by calculating the partial derivative of our cost function C with respect to any weight w in our network

$$\frac{\partial C}{\partial w} \tag{2.2}$$

By using the chain rule, weights across any arbitrarily sized network can be updated as long as every layer of a network is differentiable.

2.3 Activation Layers

Activation layers are typically placed in neural networks to stabalize and speed up the learning of neural networks during the training phase. Activation layers operate on an element by element basis on a layer and constrain the layer to a certain range of outputs. Relavent to this thesis are the following for activation layers

2.3.1 Sigmoid

The sigmoid function used to be one of the most common activations used in perceptron style networks and can be described with the following equation

$$f(z) = \frac{1}{1 + e^{-z}} \tag{2.3}$$

This layer limits outputs to [0, 1], as the input is more negative, the result approaches 0, and as the input becomes more positive, the result approaches 1.

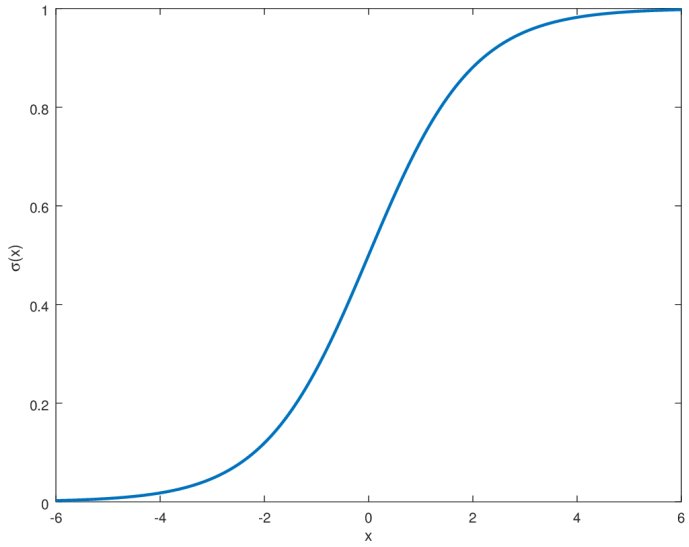


Figure 2.4: The Sigmoid Function

2.3.2 Hyperbolic Tangent

The hyperbolic tangent (tanh) activation function is another function that is more practical than sigmoid, as it outputs values that are easier to work with when settings up a network.

The layer can be defined with the following equation

$$f(z) = tanh(z) = \frac{2}{1 + e^{-2z}} - 1 \tag{2.4}$$

This activation layer changes the outputs to [-1, 1], as the input is more negative, the result approaches -1, and as the input becomes more positive, the result approaches 1.

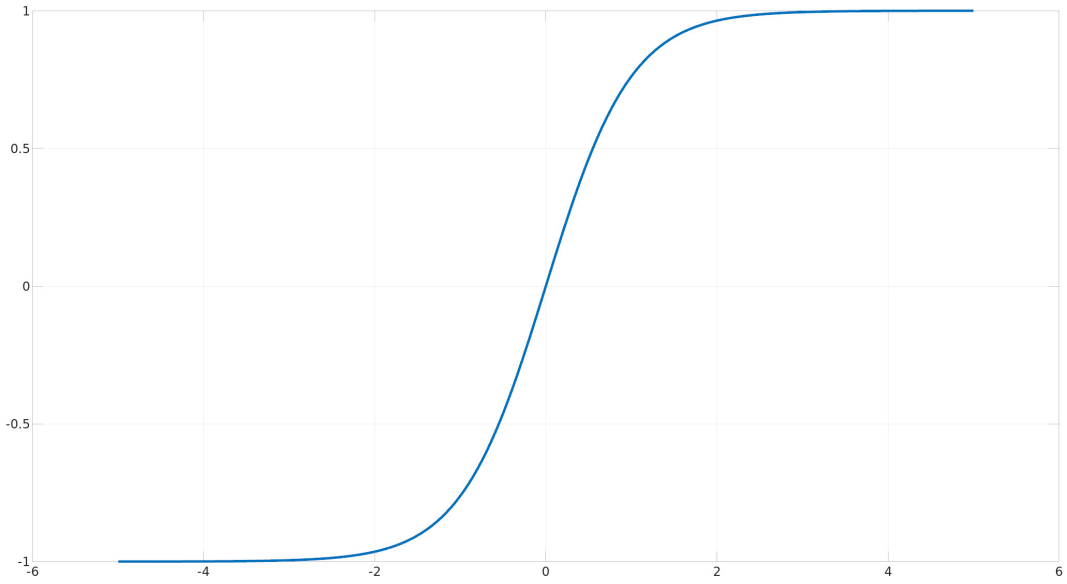


Figure 2.5: The Tanh Function

The Tanh activation function is regarded to be much better than the sigmoid function, as having a zero centered layer outputs will make learning much faster and more stable [5].

2.3.3 Rectified Linear Unit

The Rectified Linear unit (ReLU) is an activation function that was first shown to increase performance of neural network training in [6]. The algorithm is written out as

$$f(x) = \begin{cases} x & \text{if } x > 0, \\ 0 & \text{otherwise} \end{cases} \tag{2.5}$$

ReLU brings advantages during training because it tends to avoid the vanishing gradient problem, where as networks get deeper and larger, gradients for learning tend to shrink, and can prevent a network from converging as the important first layers of a network are unable to update their weights.

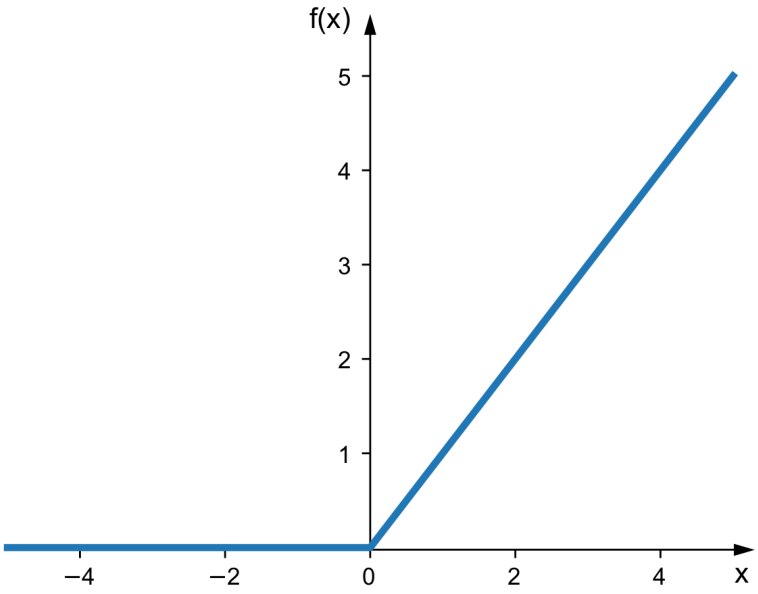


Figure 2.6: The ReLU Function

2.3.4 Parametric Rectified Linear Unit

The Parametric Rectified Linear Unit (PReLU) is a modification of the ReLU layer that adds a learned parameter to the layer instead of zeroing out inputs less than 0, the network now learns the proper value to promote training stability and performance [7]

$$f(x) = \begin{cases} x & \text{if } x > 0, \\ ax & \text{otherwise} \end{cases} \tag{2.6}$$

Where x is the input value and a is a learned parameter.

This layer seeks to overcome issues with ReLU by allowing a gradient to flow through the layer even if the layer is not activated. This even further reduces the issue of vanishing gradients, while also attempts to solve the dying ReLU problem, where ReLU may cover up important gradients even if they are important.

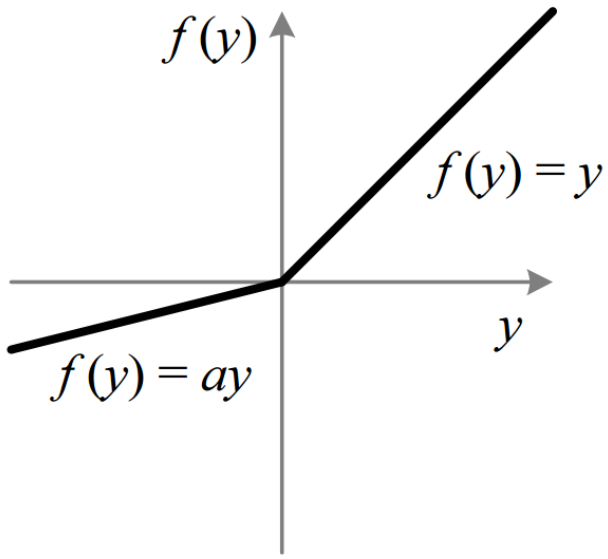


Figure 2.7: The PReLU Function

2.4 Convolutional Neural Networks

A Convolutional Neural Network (CNN) expands on the idea of the fully-connected layer by learning a collection of filters that will be used to convolve the input and identify key features in the data to feed into the dense layer.

CNNs provide some key advantages over large MLP in that they dramatically reduce the number of parameters required to learn, and also take into account local features of an input better. This allows CNNs to learn local features at lower layers in a network, and much higher end features as the layers of convolution increase.

Another important layer to a CNN is the concept of a pooling layer. These layers act to subsample the outputs of the convolutions and add provide additional resiliency to the neural network.

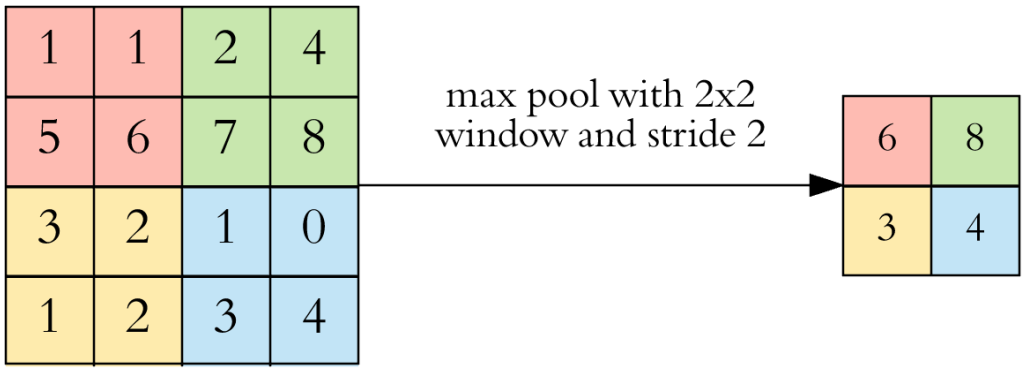


Figure 2.8: Example of a Maximum Pooling Operation

The above figure is an example of a "max pooling" layer, where the maximum value at each sliding window is taken to represent the area. This provides translation invariance as the pooling will nullify the effects of small shifts in the inputs.

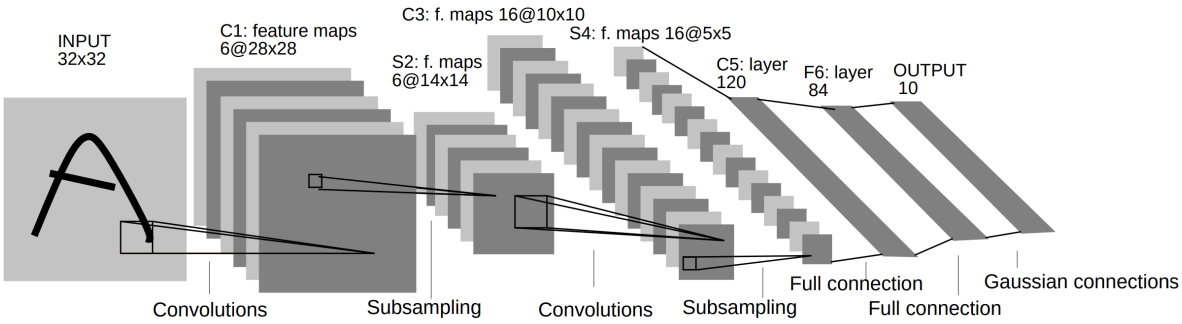


Figure 2.9: The LeNet-5 CNN Architecture for Character Recognition

When a full CNN is trained on a given dataset, the resulting network will be much more shift invariant than a comparatively sized MLP [8].

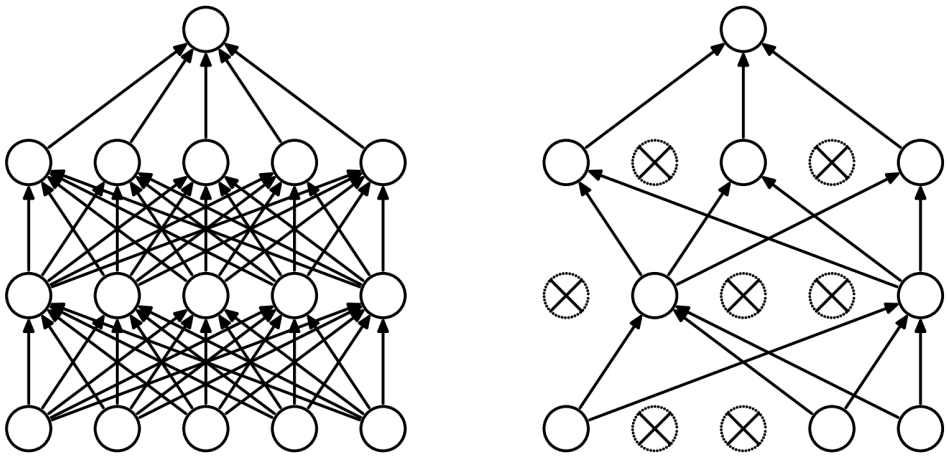
2.5 Normalization Functions

This section will discuss normalization and regularization techiniques that will be relavent to this thesis.

2.5.1 Dropout

As Neural networks increase in size and number of learned parameters, they tend to begin learning on noise in the data as opposed to actually important features, which leads to a state called overfitting there the network works extremely well on the training dataset, but then does not generalize out to data is was not trained on.

The most common way to attempt to avoid this problem is called dropout, where at each layer of a network, during network training only, layer outputs are set to zero at a predefined probability.



(a) Standard Neural Net (b) After applying dropout.

Figure 2.10: The Effects of Dropout During Training Time

This forces the network to not learn on noise in the input data, and learn actual important features. This leads to major performance improvements to network performance once networks are done training [9].

2.5.2 Batch Normalization

Batch normalization is a normalization technique that operates on the current batch of data being put through a network is 0-centered and 1-variance scaled. It is calculated through the following set of equations

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_m \tag{2.7}$$

To calculate the mean of a batch of data

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \tag{2.8}$$

To calculate the variance of the data

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \tag{2.9}$$

Where ϵ is a very small number to prevent division by 0. This equation normalizes the data to the 0-centered and 1-variance scaled, which is used in the following final equation

$$y_i = \gamma\hat{x}_i + \beta \tag{2.10}$$

Where γ and β are both learned parameters for each dimension of the data.

Batch normalization works to stabilize training and brings improvements in performance. The original paper argues that these performance enhancements are due to reducing the internal covariance shift of the data, or smoothing out the differences between inputs at each layer of a neural network[10]. This is disputed by others however, who argue that instead it smooths out the objective function and makes the gradient traversal much faster and stable[11].

2.5.3 Weight Normalization

Weight normalization is another technique for normalization, however instead of operating on the input data, weight normalization works on the weights of the layers. Weight normalization works by reparameterizing the weights on a given layer into a learnable scalar g and a learnable k-dimensional vector v, then during each forward pass of the network, the true w matrix is recalculated using the equation

$$\boldsymbol{w} = \frac{g}{||\boldsymbol{v}||}\boldsymbol{v} \tag{2.11}$$

This has the effect of fixing the Euclidean norm of w, independent of the parameters in v. This also has the effect of disconnecting the norm of the weight vector g, from the direction $(v/||\boldsymbol{v}||)$. These two changes result in improved convergence, while also providing faster learning speeds.

Weight normalization is significantly faster to calculate than batch normalization while providing very similar performance improvements [12].

Of note is that [12] also proposes a version of batch normalization refered to as mean-only batch normalization, which simply subtracts the mean of batch to 0-center the data. They

showed an increased performance by combining weight normalization and this mean-only batch normalization.

$$y = x - \mu_B \tag{2.12}$$

and argued that using both the mean-only batch normalization and weight normalization would provide the greatest training performance. Adding this mean-only batch normalization has the effect of centering the gradients used for backpropogation, which reduces noise in the gradients and leads to improved performance at test time [12].

2.6 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are networks that operate on sequences of data, they are usually causal, meaning that they can only see what they have processed up to their current location in the sequence, they react to incoming data based on what they have previously seen.

2.6.1 Elman Network

The Elman network is the most basic of RNN, which adds a very slight modification to the equations for a standard MLP to the following

$$\begin{aligned} h_t &= \sigma_h(W_h i_t + U_h h_{t-1} + b_h) \\ y_t &= \sigma_y(W_y h_t + b_y) \end{aligned} \tag{2.13}$$

Where i_t is the input data at timestep t, W_h and U_h are learned parameter matricies, h_{t-1} is the value of the hidden state at the last time step, and b are the bias values.

By saving the hidden layer of a MLP at each time step of a sequence and then including it as inputs from the input layer, the network keeps a context of what it has processed and can make more informed decisions at each time step [13].

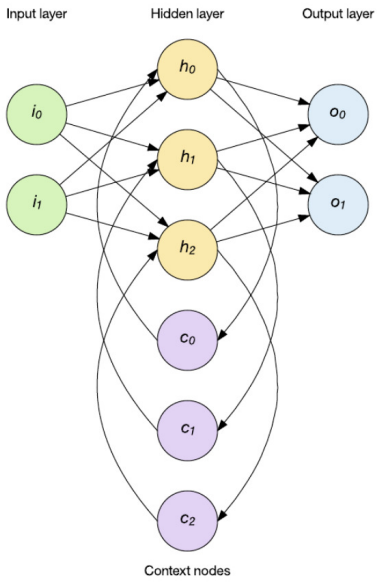


Figure 2.11: The Basic Elman Network

2.6.2 Long-Short Term Memory Cell Networks

While basic RNN's such as the Elman network are able to function relatively well at remembering data that is significant over a short number of time steps, once the distance between increases to a higher number of time steps, they are quickly unable to retain needed information that they need.

The introduction of the Long-Short Term Memory (LSTM) cell proved a significant advancement over simple RNN designs such as the Elman network. Seeking to resolve the issue of long term data dependency loss from Elman networks, The LSTM cell changes the

flow of data by using four network layers and introducing another vector that is saved at each time step, the cell state. The equations for the LSTM are as follows

$$
\begin{aligned}
f_t &= \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \\
i_t &= \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \\
o_t &= \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \\
c_t &= f_t \circ c_{t-1} + i_t \circ \sigma_c(W_d x_t + U_d h_{t-1} + b_c) \\
h_t &= o_t \circ \sigma_h(c_t)
\end{aligned}
\tag{2.14}
$$

Where $\circ$ represents element-wise multiplication, f_t represents the forget gate, i_t represents the input gate, o_t represents the output gate, c_t represents the cell state, and finally h_t represents the hidden state.

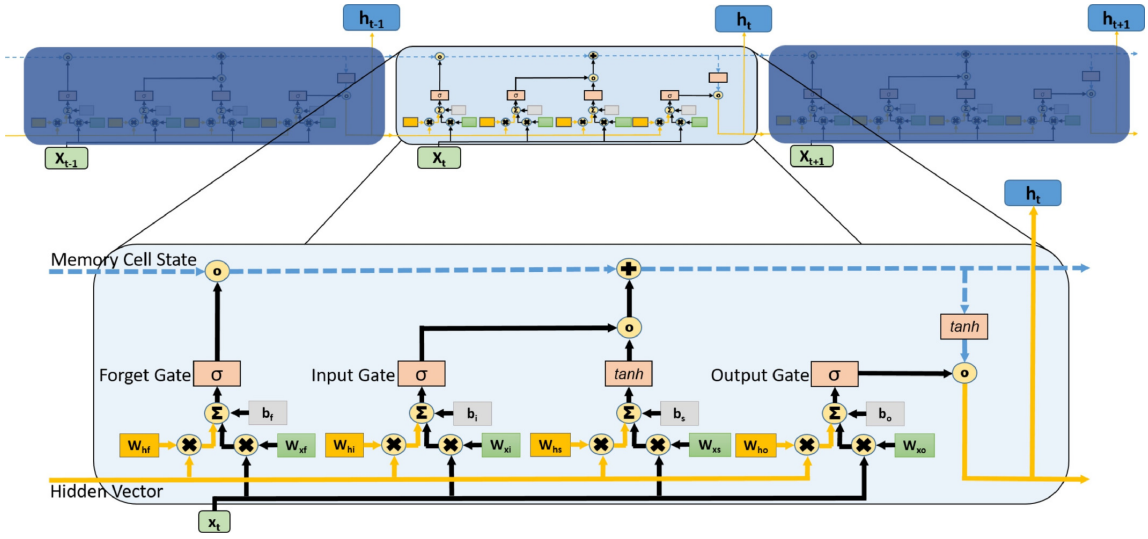


Figure 2.12: The LSTM Cell [1]

As the memory cell state is barely touched as data flows through the network, it is able to carry the long term data relationships that basic RNNs struggle to connect [14].

2.6.3 Temporal Convolutional Networks

While LSTM's provided advancement in long term data dependency and are the current standard of sequence processing, they have some significant trade offs. They are very computationally intensive, with a large amount of parameters to learn, require a large number of operations per time step, and finally the fact they are not easily parallelizable, as you must wait for the previous time step to process before moving onto the next one.

First introduced as ByteNet[15] and later refined under [16], The Temporal Convolutional Network (TCN) sought to fix the issues of the LSTM through a radical shift in idea. Instead of using multiple matrix multiplications to process time steps, a TCN uses layers of causal dilated convolutions to process data.

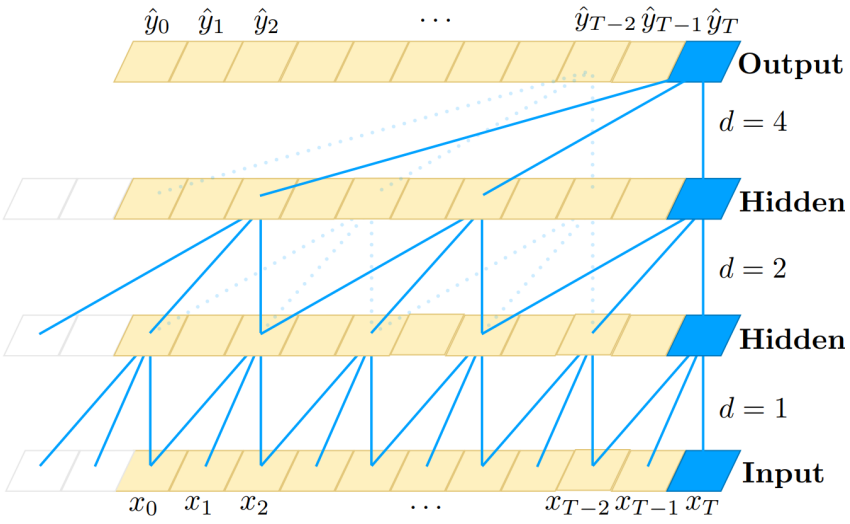


Figure 2.13: The TCN Layer Structure

Each layer consists of a residual convolution block, which preform several operations on data flowing into them. As the number of layers increases so does the dialation factor

for the TCN. This allows the receptive field of higher layers to be much higher, and given enough layers, the receptive field will cover the entire input. The output of each layer is an output the same length as the input signal.

Given a number of layers and a kernel size, if the dilation at each layer is 2^{layer_number} , then the last residual block will have a receptive field of

$$1 + 2 * (kernel\_size - 1) * (2^{n-1}) \tag{2.15}$$

Each residual block contains a dilated causal convolution layer using weight normalization, followed by ReLU and then dropout. These 4 operations are repeated twice for each block. If the number of channels needs to change then an optional 1x1 Convolution will be used to change the number of channels.

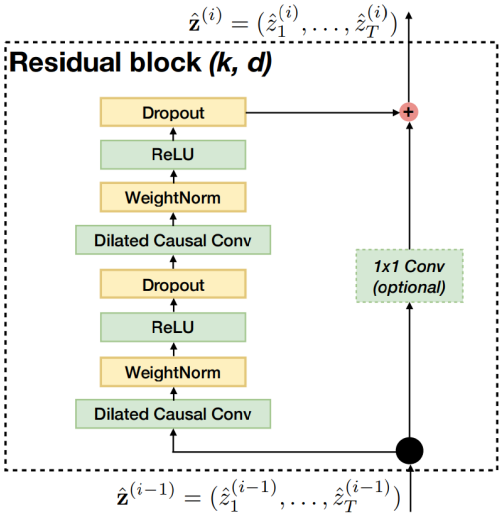


Figure 2.14: The TCN Residual Block Structure

It is argued that this network architecure fixes many of the issues that LSTM has, as this network is a fully convolutional network, it is fully parallelizable, has perfect memory over the input sequence assuming a proper size, and has many times less operations per layer.

TCNs were shown to outperform LSTMs in a variety of tasks in [16], who argue that the TCN could become the new standard for recurrent networks, however it does not seem to be a perfect drop-in, as LSTMs outperform TCN networks in some tasks such as language modeling.

CHAPTER III

QUATERNION BACKGROUND

While traditionally neural networks have always operated in the real valued space. Recent work has found that by representing neural networks in a quaternion space, there is a drastic reduction in the number of learned parameters required to gain the same level of performance in a real valued network.

This chapter provides an overview on quaternions and quaternion convolutional neural networks.

3.1 Quaternion Algebra

Quaternions were first described in [17] and are a hyper-complex system of numbers that are interpreted as points in four-dimensional space. A quaternion can be represented as the following:

$$\hat{q} = r + x\boldsymbol{i} + y\boldsymbol{j} + z\boldsymbol{k} \tag{3.1}$$

Where each q value is a real value r, and imaginary i,j, and k values that obey the property:

$$i^2 = j^2 = k^2 = ijk = -1 \tag{3.2}$$

From this definition we can also represent a quaternion as the following matrix:

$$Q_{mat} = \begin{bmatrix} r & -x & -y & -z \\ x & r & -z & y \\ y & z & r & -x \\ z & -y & x & r \end{bmatrix} \tag{3.3}$$

This is described as an injective homomorphism from quaternion space to a 4x4 real matrix.

Quaternions also have operations similar to real numbers:

Addition:

$$\hat{p} + \hat{q} = (r_1 + r_2) + (x_1 + x_2)i + (y_1 + y_2)j + (z_1 + z_2)k \tag{3.4}$$

Scalar Multiplication:

$$\lambda\hat{q} = \lambda r + \lambda xi + \lambda yj + \lambda zk \tag{3.5}$$

Conjugation:

$$\hat{q}^* = r - xi - yj - zk \tag{3.6}$$

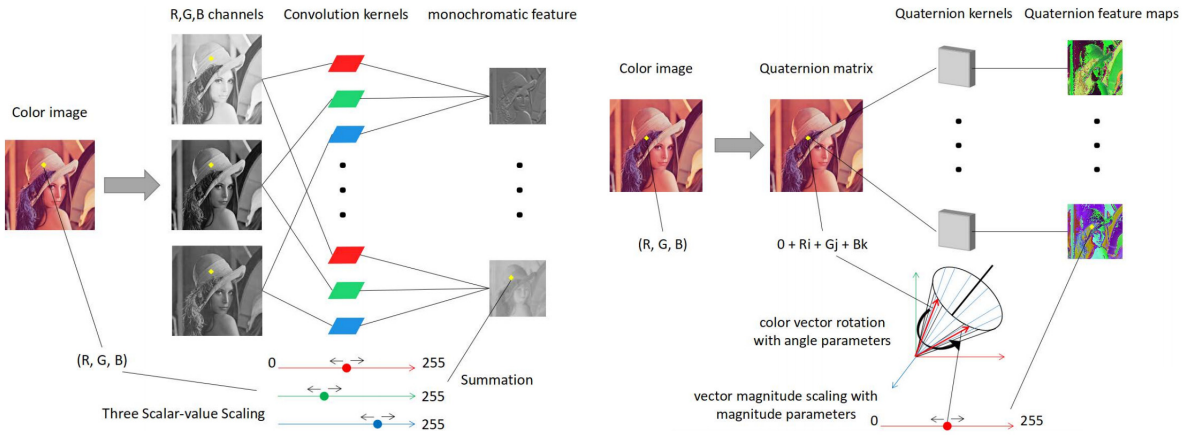
Normalization:

$$\hat{q}^{\triangleleft} = \frac{\hat{q}}{\sqrt{r^2 + x^2 + y^2 + z^2}} \tag{3.7}$$

3.2 Quaternion Neural Networks

Introduced by [18], Quaternion Neural Networks have some components that are very similar to traditional neural networks, and other parts that require reformulations to account for the change in representation.

While traditional convolutions operate on channels of an input independently, Quaternion networks are able to incorporate information from multiple channels at once.



(a) Real-valued CNN (b) Quaternion CNN

Figure 3.1: A Visual Description on the Differences Between Real and Quaternion Valued Networks [2]

3.3 Quaternion Data Representation

Quaternion networks put a constraint on the number of input and output channels, as they must be divisible by 4 in order to represent the input in four channels for r, i, j, and k. For any input layer or intermediate layer, the first N/4 feature maps are the r channels, the next N/4 are the i channel, the next N/4 are the j channel, and finally the last N/4 are the k channel.

3.4 Quaternion Convolution Layers

Quaternion Convolution is done by convolving a quaternion filter matrix $W = A + Bi + Cj + Dk$ by a quaternion vector $h = r + xi + yj + zk$, where A, B, C, and D are real-valued matrices, while r, x, y, z are all real-valued vectors.

Quaternion filter weight sizes are formed by selecting a weight size for a non quaternion network, once the input and output channel size for a convolution layer, each of them is divided by 4 and that becomes one of the found quaternion filter matrix. This means that each filter matrix is a 16^{th} the number of parameters of a standard neural network, and the total number of parameters at a given Quaternion layer is a four fold reduction over standard convolution layers.

By arranging the quaternion matrices into the structure from equation 3.3, We can take the four filter matrices and concatenate and stack them to be back to the size of the standard network filter matrix. Which is then used as the filter matrix for performing convolution.

$$\begin{bmatrix} (\boldsymbol{W} * \boldsymbol{h})R \\ (\boldsymbol{W} * \boldsymbol{h})I \\ (\boldsymbol{W} * \boldsymbol{h})J \\ (\boldsymbol{W} * \boldsymbol{h})K \end{bmatrix} = \begin{bmatrix} \boldsymbol{A} & -\boldsymbol{B} & -\boldsymbol{C} & -\boldsymbol{D} \\ \boldsymbol{B} & \boldsymbol{A} & -\boldsymbol{D} & \boldsymbol{C} \\ \boldsymbol{C} & \boldsymbol{D} & \boldsymbol{A} & -\boldsymbol{B} \\ \boldsymbol{D} & -\boldsymbol{C} & \boldsymbol{B} & \boldsymbol{A} \end{bmatrix} * \begin{bmatrix} \boldsymbol{w} \\ \boldsymbol{x} \\ \boldsymbol{y} \\ \boldsymbol{z} \end{bmatrix} \tag{3.8}$$

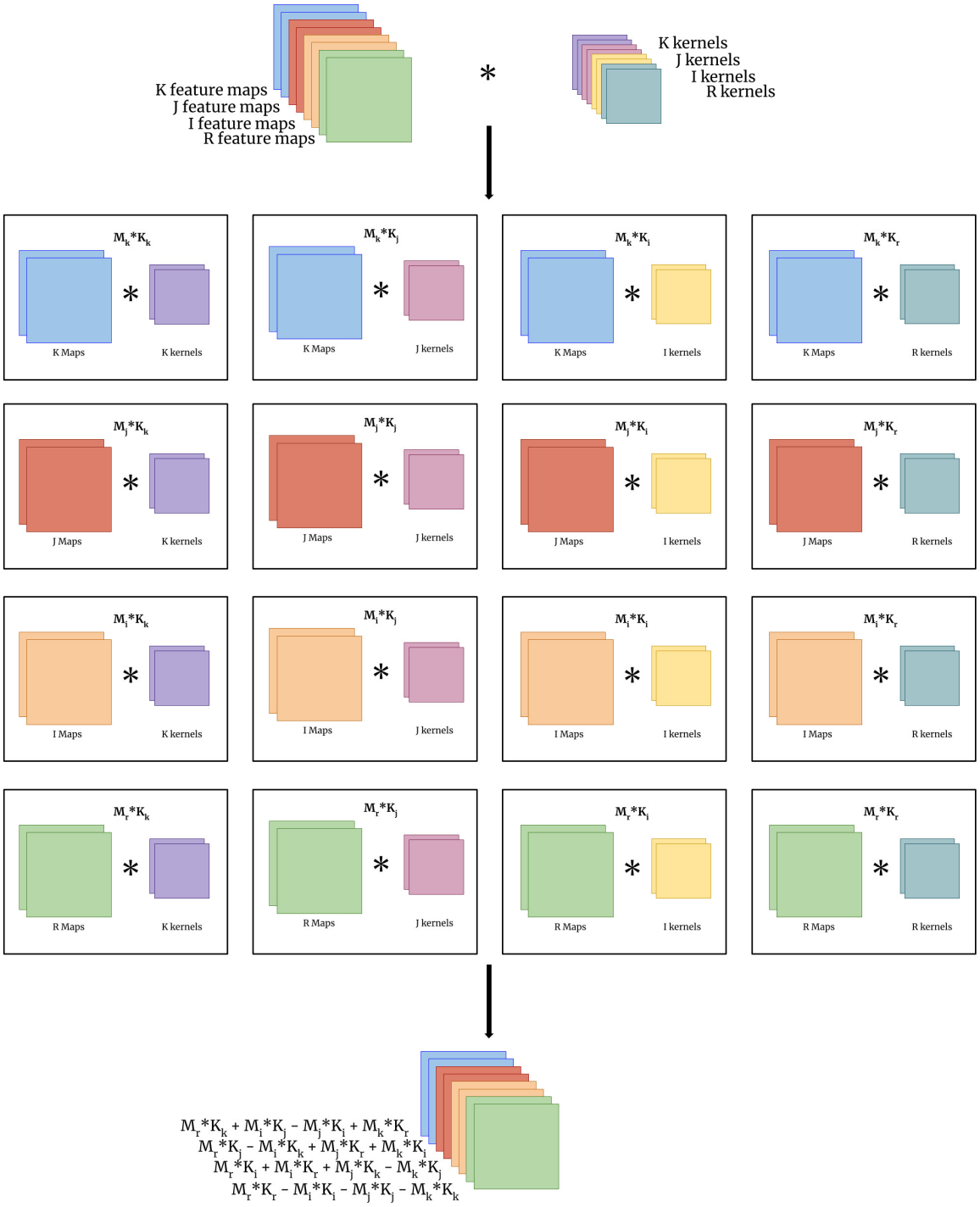


Figure 3.2: A Visual Description of Quaternion Convolution

3.5 Quaternion Batch Normalization

While batch normalization for real values is clearly defined, the algorithm does not cleanly translate over to quaternion numbers. Because of the special relationships between the different weights of the network, simply translating and scaling the quaternions will not work.

In [18] it is proposed to use a matrix whitening approach:

First a covariance matrix must be calculated that holds all of the covariance values for the four quaternion parts

$$V = \begin{bmatrix} V_{rr} & V_{ri} & V_{rj} & V_{rk} \\ V_{ri} & V_{ii} & V_{ij} & V_{ik} \\ V_{rj} & V_{ij} & V_{jj} & V_{jk} \\ V_{rk} & V_{ik} & V_{jk} & V_{kk} \end{bmatrix} \tag{3.9}$$

Where V is the covariance between two parts of the quaternion components.

A matrix W is calculated by taking the Cholesky decomposition of V^{-1}. Then this matrix is used to whiten the input data

$$\tilde{x} = \boldsymbol{W}(x - \mu_x) \tag{3.10}$$

Then the shift operation will be applied for the final result

$$\boldsymbol{BN}(\tilde{\boldsymbol{x}}) = \gamma\tilde{\boldsymbol{x}} + \beta \tag{3.11}$$

Where β and γ are learned parameters. The β parameter is a quaternion value itself, while the γ parameter is a matrix described as

$$\gamma = \begin{bmatrix} \gamma_{rr} & \gamma_{ri} & \gamma_{rj} & \gamma_{rk} \\ \gamma_{ri} & \gamma_{ii} & \gamma_{ij} & \gamma_{ik} \\ \gamma_{rj} & \gamma_{ij} & \gamma_{jj} & \gamma_{jk} \\ \gamma_{rk} & \gamma_{ik} & \gamma_{jk} & \gamma_{kk} \end{bmatrix} \tag{3.12}$$

3.6 Quaternion Weight Initialization

A very important factor of quaternion networks is that they require special weight initialization for the network to learn properly. While real value neural networks have established initialization techniques through work such as [7] and [19], these initialization techniques can also provide a basic for Quaternion Initialization.

The Algorithm for quaternion initialization is as such:

Input: W: weights, n_{in}: number of features in, n_{out}: number of features out
$\sigma \leftarrow \frac{1}{\sqrt{2(n_{in}+n_{out})}}$
for w ***in*** W **do**
  $\theta \leftarrow rand(-\pi, \pi)$
  $\phi \leftarrow rand(-\sigma, \sigma)$
  $x, y, z \leftarrow rand(0, 1)$
  $q_{imag} \leftarrow Quaternion(0, x, y, z)$
  $q^{\triangleleft}_{imag} \leftarrow \frac{q_{imag}}{\sqrt{x^2+y^2+z^2}}$
  $w_r \leftarrow \phi \times \cos\theta$
  $w_i \leftarrow \phi \times q^{\triangleleft}_{imag_i} \times \sin\theta$
  $w_j \leftarrow \phi \times q^{\triangleleft}_{imag_j} \times \sin\theta$
  $w_k \leftarrow \phi \times q^{\triangleleft}_{imag_k} \times \sin\theta$
  $w \leftarrow Quaternion(w_r, w_i, w_j, w_k)$
end
return W

Algorithm 1: QINIT initializes Quaternion Values

This provides a distribution of weights that match the properties of a Quaternion [18].

CHAPTER IV

QUATERNION NETWORK ARCHITECTURE AND METHODOLOGY

This chapter will introduce the Quaternion Temporal Convolutional Network architecture, and the methodology used to evaluate its performance.

4.1 Quaternion Weight Normalization

While previous work with quaternion neural networks has shown that quaternion batch normalization improves the performance and training of networks. It comes with tradeoffs for usage in the network:

Significant Network Slowdown The calculation of the multitude of covariances at each step, the Cholesky decomposition, and other operations of quaternion batch normalization significantly slow down the network by over 50% during training Quaternion Network Architecture and Methodology[18]. While this is just an annoyance, a network that takes significantly longer to train for the same performance doesn't seem practical.

It Conflicts with Dropout Batch normalization by nature of relying on the mean and variance of the input data is at odds with any operation that modifies those statistics. Dropout modifies the statistics of a batch in a way that cases network instability and lack of generalizing performance when employing both techniques [20].

While Batch Normalization can be used in a centering only mode to alleviate much of the slowdown, not being able to use the most common form of regularization on a network is concerning, as there is a large risk of overfitting as training goes on.

To overcome these two problems Weight Normalization can be used as opposed to Batch normalization. This paper defines Quaternion Weight Normalization as the procedure of applying weight normalization to each of the four weight matrices independently before they are used.

This will allow for the usage of dropout while also avoiding the massive slowdown. While [12] discussed combining mean-only batch normalization and weight normalization to increase performance, this would prevent using dropout, which through experimentation was shown to be more valuable than combining the two normalization techniques.

4.2 Quaternion Temporal Convolutional Network

The Quaternion Temporal Convolutional Network will take advantage of the previous work of the Temporal Convolutional Network and the Quaternion Convolutional Neural Network as the basis for it's architecture.

The basic changeset of the network will be to replace all standard convolutions with Quaternion 1D convolutions, all Activation layers will be replaced by PReLU, and weight normalization in the base network will be replaced with a "Quaternion weight normalization" where we simply apply weight normalization to each of the four weight matrices for a given layer.

PReLU is chosen as the residual block activation function because it will prevent the network from ignoring negative activations due to the differences of the Quaternion valued networks.

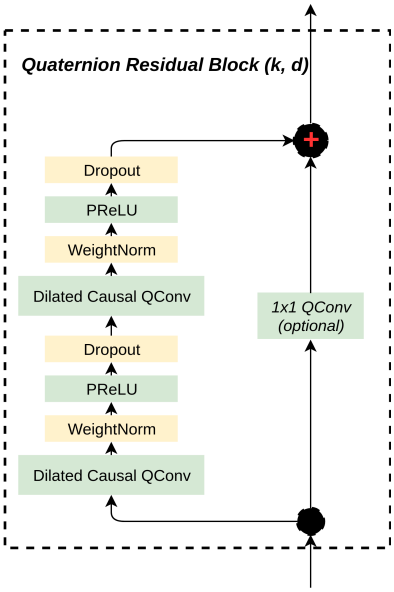


Figure 4.1: The Quaternion Residual Block

4.3 Testing Methodology

To test network performance we follow the methodology used in [16], and attempt to match the results for datasets where the number of channels allows us to use the QTCN, or easily manipulate the data in a way that allows the network to read it.

The goal of this thesis is to demonstrate performance of the network as a drop-in replacement to the base TCN, using the hyperparameters from [16]. Unfortunatly as Quaternion networks require an input with a channel dimension divisible by 4, there will be some minimal changes the input data to get the correct input size.

It should be noted that there may be other portions to networks for particular problems, such as fully connected layers, these network components will be left as real valued to allow the performance of the two TCNs to be directly compared.

4.3.1 Sequential MNIST

The Modified Natitional Institute of Standards and Technology (MNIST) database is a collection of 70,000 handwritten digits that are typically used for image classification. The images are 28x28 greyscale pixels and are one of the most common datasets to test a neural network on [8].

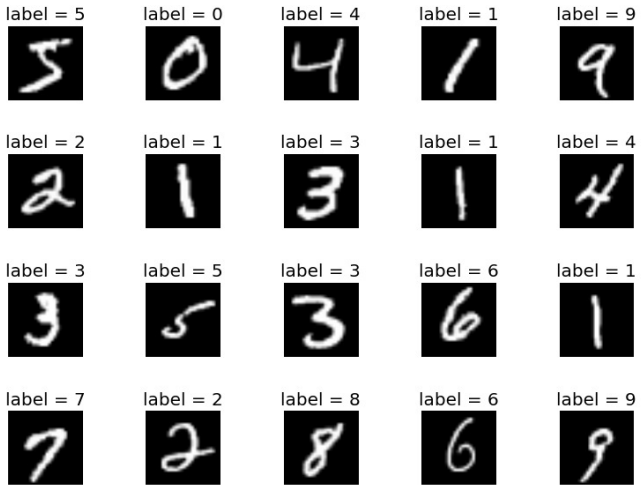


Figure 4.2: Examples of MNIST Imagery and Their Respective Classifications

Because the QTCN is designed for sequence processing, we use a modification known to the dataset known as Sequential MNIST where the images are flattened down to a 784 long vector, and then classification is then done with that vector in a causal manner [21]. This is a fairly difficult problem as a network must keep the image context for the entire length of the sequence.

As each image in the Sequential MNIST dataset will be a 784 length vector, we duplicate the data to 4 channels to allow the QTCN to be able to process the dataset. We use an 8

layer network with a kernel size of 7, providing a receptive field of $1 + 2 * 6 * 2^7 = 1537$ time steps, more than covering the input size. At the end of the network a fully connected layer to make the final classification of the images into one of 10 classes.

Each convolution block is allowed 25 filters for the Base TCN, The QTCN is given 24 to keep things as close as possible. No dropout is used for this test, and the optimizer selected was the Adam optimizer [22] with a learning rate of 2e-3.

A modified version of Sequential MNIST called permuted MNIST or P-MNIST also exists where the input data for an image is permuted randomly [21], results for both versions of the problem are shown.

The reporting metric for these two problems is accuracy, or how many samples were correctly classified versus total number of samples in the testing set, a higher value is better.

4.3.2 Sequential Cifar10/Cifar100

Cifar10 and Cifar100 datasets are very similar to the MNIST dataset in that they are small images (32x32 pixels) and seperated into classes (Cifar10 has 10 classes of objects, while Cifar100 has 100) [23].

A large difference here is that these images are RBG as opposed to MNIST's grayscale imagery, This fits very well with quaternion networks. To generate the sequential Cifar10 and 100 dataset the base imagery should be flattened the down to a 1024 vector with R, G, and B channels. To match the 4 channel design of quaternions a grayscale vector is formed from the 3 channels and added to the top of each image matrix to act as the 'Real' channel. This provides input is 4x1024 per image. To allow the network to process the data correctly

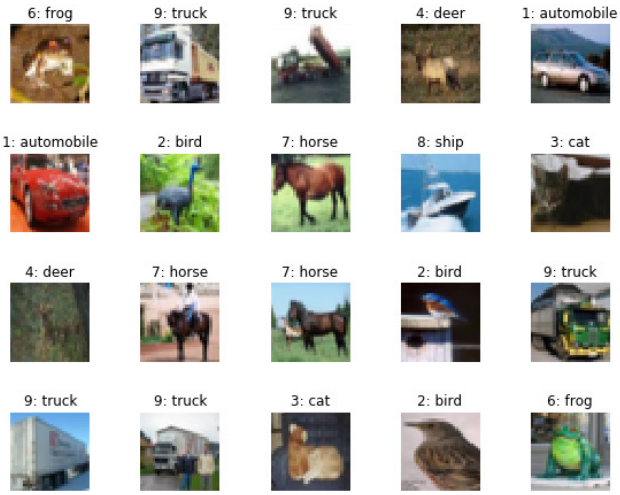


Figure 4.3: Examples of Cifar10 Imagery and Their Respective Classifications

an 8 layer network with a kernel size of 10 is used, giving the network a receptive field of $1 + 2 * 9 * 2^7 = 2305$ time steps. At the end of the network a fully connected layer is added to make the final classification of the images into one of 10 or 100 classes.

Each convolution block is allowed 64 filters, given a dropout rate of .2, trained for 200 epochs and learning is performed using the Stochastic Gradient Descent [24] algorithm, with a learning rate of .01, which is multiplied by .1 at epochs 75, and 150.

The reporting metric for these two problems is accuracy, or how many samples were correctly classified versus total number of samples in the testing set, a higher value is better.

4.3.3 Adding Problem

First introduced in [14] the adding problem is a test of a networks ability to retain and process information in sequences of data.

In the adding problem a 2 channel matrix is provided as an input, the first channel is randomly filled with values in [0,1], while the second channel is all zeros except for two elements which are one. The task for this problem is to have the network produce the sum of the elements where the two elements are one.

Due to the 2 channel nature of the input this task does not directly translate to the QTCN without some modification. The translation approach involved duplicating the mask channel twice, so that the real, i, and j components are the mask, and the k channel is the randomly initialized values. The actual task remains the same. A fully connected layer is added at the end to make the final prediction.

The specific variation of the adding problem used here is with 600 elements in the vector. An 8 layer network with a kernel size of 8 is used, giving a receptive field of $1+2*(7)*(2^7) = 1793$. Each convolution block uses 24 filters, with no dropout. The network is trained for 10 epochs with a learning rate of 4e-3, while using the Adam optimizer.

The reporting metric for these two problems is loss, or how close on average the network's result to the actual answer is. A lower value is better.

4.3.4 Copy Problem

The Copy Memory Problem is another test that tests a networks ability to remember information. In this task the input is a vector that is of length 20+T where T is some arbitrary number. The first 10 values of the vector are randomly filled with numbers ranging [0,8], the last 11 are filled with the digit 9, and then the middle is filled with 0. The task is to fill in the final 10 values with the randomly selected digits from the beginning of the vector [25].

The input for this problem is only 1 channel, the channel must be duplicated into 4 channels for the network to read it. A T of 1000 is selected for direct comparison to [16] and the results of the base TCN. The networks were given 8 layers and a kernel size of 8, giving a receptive field of $1 + 2 * 7 * 2^7 = 1793$. A fully connected layer is added at the end to make the final prediction.

Each convolution block was allowed 10 filters, a dropout of 0.05 was used. The networks were trained for 50 epochs with a learning rate of 5e-4 using the RMSProp algorithm [26].

The reporting metric for these two problems is loss, or how close on average was the network's result to the actual answer. A lower value is better.

4.3.5 Language Modeling

Introduced in [27], Language modeling is the task of assigning a probability to sentences in a language, more importantly a language model can be used to predict the next token in a sequence based on the previous tokens in a sequence [28].

This task is incredibly useful as the language models produced are often used for many NLP tasks, such as translation [29], image captioning [30], and reading comprehension [31] tasks.

There are two main 'modes' of language modeling, a word based token system and a character based token system. In a word based token system the task is to predict the next word based on the previous sequence, while a character based system simply predicts the next letter or digit that comes next based on the previous context. Both versions have tradeoffs, a word based language model does not easily handle words it has never been

trained on, while a character based system is significantly more computationally expensive and is much more demanding on the networks memory of sequences.

The method to train a language model takes a large corpus of data and tokenizes sentences to a numerical form representing either words or characters, a learning look-up table is used to turn these indexes into vectors that networks can process. The network then reads over the sequence of vectors to produce features for a fully connected layer at the end to make the prediction for the next word. The weights for the fully connected layer and embedding table are typically "tied" together and use the same weights.

A common dataset for language modeling is the PennTreebank (PTB) dataset [32], a relatively small set with a vocabulary size of 10K words.

Results for PTB word and character based language modeling are shown.

Settings for each mode is shown below, based on [16]:

Table 4.1: Language Modeling Network Parameters

Mode	Layers	Kernel Size	Embedding Size	Receptive Field
Word-Based	4	3	600	33
Character-Based	3	3	100	17

The receptive field corresponds to how long of a sentence or word the network can read, and embedding size is the size of the vector produced for a given word or character out of the look-up table.

Hyperparameters for the two tests are also given below

The reporting metric for a word based language model is perplexity, which is defined as

Table 4.2: Language Modeling Hyperparameters

Mode	Number of Filters	Dropout	Epochs	Learning Rate	Optimizer
Word-Based	600	.5	100	4	SGD
Character-Based	450	.1	100	4	SGD

$$PP(W) = P(w_1 w_2 ... w_N)^{\frac{-1}{N}} \tag{4.1}$$

where N is the inverse probability of the test set, normalized by the number of words. A lower score is representative of a language model being able to predict the next word in a sentence better.

The reporting metric for a character based language model is bits per character or BPC. Which is calculated as

$$bpc(string) = -\frac{1}{T}\sum_{t=1}^{T} \log_2 \hat{P}_t(x_t) \tag{4.2}$$

Where $\hat{P}_t$ is the probability distribution of all character outputs at a given character. A lower score is better, as it signifies the network is more confident in predicting the next token.

4.3.6 Polyphonic Music Modeling

The J. S. Bach (JSB) Chorales dataset [33] and the much larger Nottingham dataset [1] are both used as polyphonic music sequencing tasks, very similar to language modeling,

[1] https://ifdo.ca/ seymour/nottingham/nottingham.html

the task of music modeling is to predict the next note to be played based on the previous sequence of music.

Much like language modeling the input to the network is a sequence of 88-length vectors, where each index in a vector corresponds to a key on a piano, this task is slightly different than language modeling, as multiple tokens must be produced at each timestep.

We follow [16] and use the following parameters for the QTCN

Table 4.3: Music Modeling Network Parameters

Dataset	Layers	Kernel Size	Receptive Field
JSB Chorales	3	2	9
Nottingham	4	6	81

Where the receptive field is the number of notes in a sequence.

Hyperparameters for the two tests are also given below

Table 4.4: Music Modeling Hyperparameters

Dataset	Number of Filters	Dropout	Epochs	Learning Rate	Optimizer
JSB Chorales	150	.5	100	1e-3	Adam
Nottingham	150	.2	100	1e-3	Adam

The reporting metric for music modeling is loss. Where we determine how well the network was at predicting that a note was supposed to be played at a given time step.

CHAPTER V

RESULTS AND DISCUSSION

This section contains results of the network evaluation tasks to directly compare the performance and parameter count of the TCN and proposed QTCN. All results for the base TCN were taken from [16] except for those of Cifar10/100, which are new results generated for this thesis.

Note that a ^{h} in the Metric column signifies that higher values are better, and a ^{l} signifies the opposite.

Parameter counts cover every learned parameter in a network, smaller reductions in learned parameters occur if the TCN consisted of a smaller part parameter wise of the entire network for a given task.

5.1 Sequence Classification

Table 5.1: Sequence Classification Results

Sequence Task	Metric	TCN param #	TCN Result	QTCN Param #	QTCN Result
Seq. MNIST	Accuracy^{h}	21K	99.0	16K	98.91
Permuted MNIST	Accuracy	42K	97.2	16K	97.12
Seq. Cifar10	Accuracy	500K	77.33	160K	77.48
Seq. Cifar100	Accuracy	500K	48.50	160K	47.46

The QTCN was able to significantly reduce the number of parameters for all classification tasks while suffering very little performance impact, in some cases even performing slightly better. Of note would be the interesting datapoint that the QTCN edges out the TCN on Cifar10, however slightly loses on Cifar100, while keeping the parameter numbers exactly

the same, it could be possible that the QTCN either requires more data per class than the base TCN, or could be that the parameters count of the QTCN is at capacity for the 100 class problem, Causing the performance to fall off slightly. collecting results at higher parameter counts for both networks could cause the QTCN to edge out in performance again.

5.2 Adding and Copy Problem

Table 5.2: Adding and Copy Results

Sequence Task	Metric	TCN param #	TCN Result	QTCN Param #	QTCN Result
Adding Problem	Loss[1]	70K	5.8e-5	17K	1.8e-4
Copy Problem	Loss	13K	3.5e-5	2K	5.6e-5

The performance of the QTCN on these task show that the network keeps the same traits and characteristics of the base TCN, as the network was able to succeed in both tasks and match very close to the performance of the base network.

Interestingly the performance on the adding problem did not seem to match the TCN as well as expected. An experiment was done to double the training time to see if the network was not done learning yet

Table 5.3: Adding Problem Epoch Modification Results

Sequence Task	Epochs	QTCN Test Loss
Adding Problem	10	1.8e-4
Adding Problem	20	9.5e-5

By modifying the number of epochs the performance was improved, under some circumstances it appears the network is not a complete drop in replacement for the TCN, and minor hyperparameter tuning may be required.

5.3 Music Modeling

Table 5.4: Music Modeling Results

Sequence Task	Metric	TCN param #	TCN Result	QTCN Param #	QTCN Result
Music JSB Chorales	Loss	300K	8.10	84K	8.27
Music Nottingham	Loss	1M	3.07	282K	2.65

While the QTCN did not quite match the performance of the TCN on the JSB task, it performed significantly better than the base TCN on the Nottingham task. The JSB dataset has very few training samples, so it is possible there was simply not enough information to learn, another potential explanation is that there were too few iterations for the network to learn to the data, as we saw in the adding problem.

5.4 Language Modeling

Table 5.5: Language Modeling Results

Sequence Task	Metric	TCN param #	TCN Result	QTCN Param #	QTCN Result
Word-level PTB	Perplexity[l]	13M	88.68	8.2M	96.81
Char-level PTB	bpc[l]	3M	1.31	560K	1.41

Once again the QTCN fails to match the performance of the base network on Modeling tasks, performing slightly worse on the Word-modeling task, while performing significantly

worse on the Character level Task. A more thorough grid search on hyper parameters may be needed to find the correct amount of dropout to get the network perform better on sequence modeling tasks. Another potential cause of a problem on this task is the embedding table that the word embeddings are first input through before the TCN. Because neither the embeddings before the QTCN or the fully connected layer after the QTCN store data in a quaternion representation, there may be information loss. If a word embedding table is created, it would be very interesting to see the results of a fully quaternion language model, and see if it is able to reach the base TCN model with the same parameters, to understand if there is information loss in the conversion of representation.

CHAPTER VI

CONCLUSION AND FUTURE WORK

In this thesis, it was seen that quaternion 1D convolutions can be integrated into the Temporal Convolutional Network design, that when combined with Quaternion Weight normalization, create a new Quaternion Temporal Convolutional Network architecture. Using this new network architecture for sequence processing tasks was shown to significantly reduce the number of learned parameters for a network, while remaining fairly similar in accuracy on a wide range of tasks, allowing it to act as a minimal modification replacement to the base network.

For future work, research into higher order hyper-complex representations such as the octonion representation[34] could be integrated to verify that the patterns of the QTCN follow into other representations. Another area of interest would be looking into mixed representation networks, Figure 3.1 shows the great difference between feature maps in different network representations, stacking quaternion and regular convolutions may provide some benefit on top of the reduced number of parameters. Finally, a Quaternion RNN and LSTM cell were demonstrated in [35] for the task of Automatic Speech Recognition, A comparison between the QTCN and QRNN cells would be of interest.

BIBLIOGRAPHY

[1] R. G. Hefron, B. J. Borghetti, J. C. Christensen, and C. M. S. Kabban, "Deep long short-term memory structures model temporal dependencies improving cognitive workload estimation," *Pattern Recognition Letters*, vol. 94, pp. 96–104, 2017.

[2] X. Zhu, Y. Xu, H. Xu, and C. Chen, "Quaternion convolutional neural networks," *CoRR*, vol. abs/1903.00658, 2019. [Online]. Available: http://arxiv.org/abs/1903.00658

[3] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain." *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[4] D. E. Rumelhart, G. E. Hinton, R. J. Williams *et al.*, "Learning representations by back-propagating errors," *Cognitive modeling*, vol. 5, no. 3, p. 1.

[5] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient backprop," in *Neural networks: Tricks of the trade*. Springer, 2012, pp. 9–48.

[6] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, 2011, pp. 315–323.

[7] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," *CoRR*, vol. abs/1502.01852, 2015. [Online]. Available: http://arxiv.org/abs/1502.01852

[8] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner *et al.*, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[9] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[10] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015. [Online]. Available: http://arxiv.org/abs/1502.03167

[11] S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry, "How does batch normalization help optimization?" in *Advances in Neural Information Processing Systems*, 2018, pp. 2483–2493.

[12] T. Salimans and D. P. Kingma, "Weight normalization: A simple reparameterization to accelerate training of deep neural networks," *CoRR*, vol. abs/1602.07868, 2016. [Online]. Available: http://arxiv.org/abs/1602.07868

[13] J. L. Elman, "Finding structure in time," *Cognitive science*, vol. 14, no. 2, pp. 179–211, 1990.

[14] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[15] N. Kalchbrenner, L. Espeholt, K. Simonyan, A. van den Oord, A. Graves, and K. Kavukcuoglu, "Neural machine translation in linear time," *CoRR*, vol. abs/1610.10099, 2016. [Online]. Available: http://arxiv.org/abs/1610.10099

[16] S. Bai, J. Z. Kolter, and V. Koltun, "An empirical evaluation of generic convolutional and recurrent networks for sequence modeling," *CoRR*, vol. abs/1803.01271, 2018. [Online]. Available: http://arxiv.org/abs/1803.01271

[17] W. R. Hamilton, "Ii. on quaternions; or on a new system of imaginaries in algebra," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 25, no. 163, pp. 10–13, 1844.

[18] C. J. Gaudet and A. S. Maida, "Deep quaternion networks," *CoRR*, vol. abs/1712.04604, 2017. [Online]. Available: http://arxiv.org/abs/1712.04604

[19] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 2010, pp. 249–256.

[20] X. Li, S. Chen, X. Hu, and J. Yang, "Understanding the disharmony between dropout and batch normalization by variance shift," *CoRR*, vol. abs/1801.05134, 2018. [Online]. Available: http://arxiv.org/abs/1801.05134

[21] Q. V. Le, N. Jaitly, and G. E. Hinton, "A simple way to initialize recurrent networks of rectified linear units," *arXiv preprint arXiv:1504.00941*, 2015.

[22] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[23] A. Krizhevsky *et al.*, "Learning multiple layers of features from tiny images," Citeseer, Tech. Rep., 2009.

[24] H. Robbins and S. Monro, "A stochastic approximation method," *The annals of mathematical statistics*, pp. 400–407, 1951.

[25] S. Zhang, Y. Wu, T. Che, Z. Lin, R. Memisevic, R. R. Salakhutdinov, and Y. Bengio, "Architectural complexity measures of recurrent neural networks," in *Advances in neural information processing systems*, 2016, pp. 1822–1830.

[26] G. Hinton, N. Srivastava, and K. Swersky, "Neural networks for machine learning lecture 6a overview of mini-batch gradient descent."

[27] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, "A neural probabilistic language model," *Journal of machine learning research*, vol. 3, no. Feb, pp. 1137–1155, 2003.

[28] Y. Goldberg, "Neural network methods for natural language processing," *Synthesis Lectures on Human Language Technologies*, vol. 10, no. 1, pp. 1–309, 2017.

[29] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, 2014, pp. 3104–3112.

[30] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan, "Show and tell: A neural image caption generator," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 3156–3164.

[31] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "Squad: 100,000+ questions for machine comprehension of text," *arXiv preprint arXiv:1606.05250*, 2016.

[32] M. Marcus, B. Santorini, and M. A. Marcinkiewicz, "Building a large annotated corpus of english: The penn treebank," 1993.

[33] M. Allan and C. Williams, "Harmonising chorales by probabilistic inference," in *Advances in neural information processing systems*, 2005, pp. 25–32.

[34] J. Wu, L. Xu, Y. Kong, L. Senhadji, and H. Shu, "Deep octonion networks," *CoRR*, vol. abs/1903.08478, 2019. [Online]. Available: http://arxiv.org/abs/1903.08478

[35] T. Parcollet, M. Ravanelli, M. Morchid, G. Linarès, C. Trabelsi, R. De Mori, and Y. Bengio, "Quaternion recurrent neural networks," *arXiv preprint arXiv:1806.04418*, 2018.