TU Bergakademie Freiberg • Institut für angewandte Analysis

thesis for obtaining
the academic degree dipl. mat.

Quaternion Deep Learning

Johanna Luise Richter

July 18, 2022
Supervisors:
Prof. Dr. Swanhild Bernstein
Dr. Bettina Heise

Contents

thesis affidavit - Eidesstattliche Erklärung

1 Introduction

Neural networks have long been an established tool for many applications. From voice assistants [1], [2], prediction of purchasing behavior [3], [4], analysis of medical data [5], [6], [7], image processing [8], [9], and recognition [10], [11], in fact, everywhere where large amounts of data are generated, one encounters neural network approaches. The community worldwide is very active and new concepts and architectures for neural networks appear regularly. For a long time now, there has also been a great deal of economic interest in ever better networks for data evaluation and the automation of intelligent processes.

This thesis is concerned with image recognition and, to this end, explores a relatively new approach to constructing a network, which has potential applications beyond image processing. In conventional neural networks for image recognition, the three color layers are input separately and each is taught different parameters. This work will investigate an approach to input the color layers as coupled values, in the form of quaternions, and to learn common parameters.

After a detailed introduction of the mathematics of convolutional neural networks in general and quaternion networks in detail, the implementation in Python will be briefly discussed. Experiments are then conducted to better understand the learning process and its strengths and weaknesses. For this, the role of color will be considered in comparison to the role of structure in learning. Experiments on stability to noisy images will also be tested.
The aim is to provide a proof of concept that the learning functions in order to lay the foundation for further investigations of quaternion neural networks and potential application areas in which they could beat or enrich conventional networks.

2 Background

2.1 Neural Networks

2.1.1 Artificial Neural Networks

Artificial Neural Networks (ANN) provide a widespread method for information processing in Artificial Intelligence. The development of Artificial Neural Networks began around 1950 and continues to this day. Inspired by the biological model of a neuronal network, information is passed as input, signals are forwarded from neuron to neuron, and an output is obtained. Accordingly, the network structure is divided into input layer, hidden layer and output layer. Classically, they are represented in directed graphs from left to right, speaking of feedforward networks (2.1). A neuron is a node which receives, processes and forwards information. Superimposed nodes are grouped together as a layer. The edges between two layers correspond to an information transfer. Successive layers do not have to be completely connected.

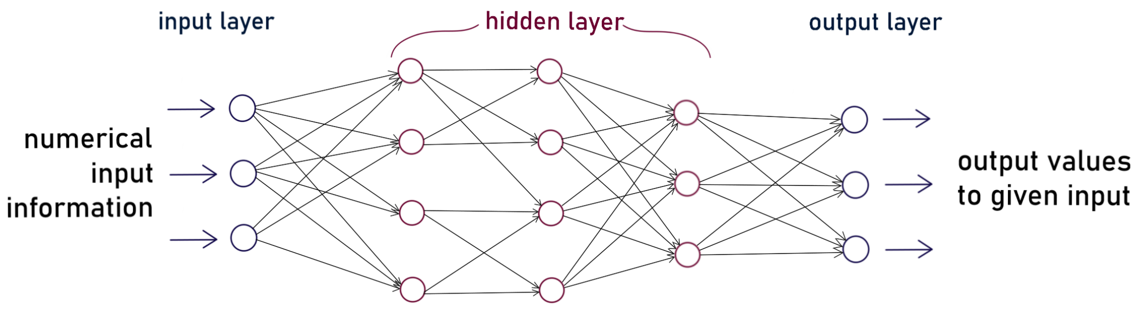


Figure 2.1: Basic structure of an artificial neural network

The number of layers, neurons per layer and edges between the neurons is called the architecture of the neural network. The edges, i.e. individual information transmissions, are assigned weights. These control the amount of influence of this information on the following neuron. Large weights suggest a large influence and small weights imply a small influence. Missing edges between two layers correspond to a non-influence of the information of a neuron on the following neuron and should be defined with a static weight of the value 0, so that the undefined value does not lead to errors in computation.
The incoming information into a neuron is the output values of the neurons of the previous layer each multiplied by the associated edge weights. These are added together in the neuron and processed as new information. This is accomplished by addition of a bias and subsequent processing by an activation function (2.2). The latter evaluates the information and assigns a new value to it. Usually, all values of the neurons of an ANN range between 0 and 1, and so does the image domain of the activation function. In general it should also be noted, that activation functions are monotonically increasing; small information causes a small output from the neuron, and large information causes a large output. Common activation functions are, for example

- hard limit: $\varphi^{\text{hlim}}(x) = \begin{cases} 1 & , \text{ for } x \geq 0 \\ 0 & , \text{ for } x < 0 \end{cases}$
- sigmoid function: $\varphi_a^{\text{sig}}(x) = \frac{1}{1+\exp(-ax)}$
- rectifier (ReLu): $\varphi^{\text{relu}}(x) = \max\{0, x\}$

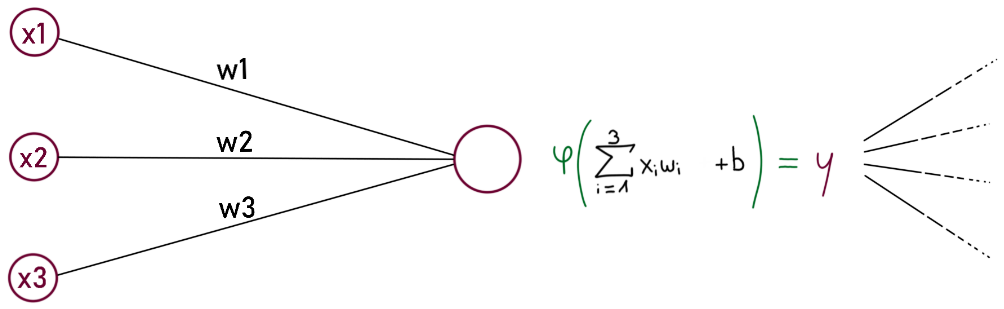


Figure 2.2: Basic processing of information at a neuron of an artificial neural network with the bias b and the activation function φ

The whole network can be written as a concatenation of sums, products and activation functions. For all input values, this structure results in certain output values between 0 and 1. These are often used to represent the probability that the feature assigned to a specific output node is characterized by the data (via softmax activation in the last layer). For example, given an input of various features of a text (certain words, sentence length, frequent structures) as representative numerical values, a suitable network could return a probability that the text is a non-fiction or literary text.
Prior to searching for a solution of how to learn or find a good network, it is necessary to determine, which facts can be approximated by which networks in the first place.

2.1.2 Universal Approximation Theorem

After simple models with some perceptrons (neurons with the jump function as their activation function, which only distinguishes an active or inactive state) already achieved good results in the approximation of certain facts, historically the question arose, which functions can be approximated by neural networks.
What mappings of an input vector (values in the input layer) to an output vector (values in the output layer) can be approximated by the concatenation of functions described by a neural network of a particular architecture?
The mathematical answer was given by Maxwell Stinchcombe, Halbert White and Kurt Hornik in 1989 in their paper *Multilayer Feedforward Networks are Universal Approximators* [12]. For continuous mappings from input to output, there exists a network which has a vanishingly small error to the mapping itself. Thus neural networks are universal approximators of such mappings. The proof is not limited to continuous activation functions. Therefore it is apparent that almost every problem can be represented by a network, as long as a sufficiently complex structure is chosen and a good learning algorithm is found.

The results were quickly expanded:
In 1991 Kurt Hornik showed that a restriction of the depth of the network (number of layers) at arbitrary width (number of neurons in a layer) still allows a universal approximation [13].
As recently as 2017, Zhou Lu et al. published evidence for the opposite model: bounded width and arbitrary depth of the network [14]. This gives justification and new momentum to the deep learning trend, which is already prospering at this point.

2.1.3 Learning through backpropagation

The universal approximation theorems allow to raise the question of finding an approximating network. First of all, the architecture of the network is important. This must be determined before the learning process. International competitions annually award prizes for the best architectures.
Given an architecture and a set of data (input and desired output), the goal is to choose the weights of the edges among the networks of an architecture in such a way that the existing data, but also new data likely to follow the same distribution, are approximated as well as possible.
The idea for this learning process is called backpropagation.
The basic concept is to compare for a given input the desired output value with the actual output of the network. Then one can calculate which change in the weights of the previous layer would lead to an improvement of the result. After that, one can again determine what change in the layer before might lead to an improvement in this previous layer. This stepwise correction from the output layer back through the network characterizes the name backpropagation.

If one obtains for an input $x_1, ..., x_n$ with the current configuration of the network the output $o_1, ..., o_m$ and has for comparison the correct output $y_1, ..., y_m$, the squared error is calculated as follows:

$$E = \frac{1}{2} \sum_{l=1}^{m} (o_l - y_l)^2 \tag{2.1}$$

The factor $\frac{1}{2}$ facilitates the derivation later on. The error E will now be partially derived according to all weights to determine their influence on the error and to correct them in the right direction. The partial derivative can be determined by the chain rule:

$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial\, \varphi\big(\sum_{i=1}^{k} x_i w_i\big)} \frac{\partial\, \varphi\big(\sum_{i=1}^{k} x_i w_i\big)}{\partial \sum_{i=1}^{k} x_i w_i} \frac{\partial \sum_{i=1}^{k} x_i w_i}{\partial w_i} \tag{2.2}$$

Weights of layers further ahead can now be determined step by step by partial derivation of the error of layers further to the right. With a selected learning rate η, which determines the amount of weight change in a learning step, the actual weight change can then be expressed like this:

$$\triangle w_i = -\eta \frac{\partial E}{\partial w_i} \tag{2.3}$$

The learning rate is crucial; a too high choice leads to surpassing the searched local minimum of the error. In this case, the network learns past a locally optimal parameter choice and the error may be larger than necessary or might even grow. Too small of a learning rate slows down the process unnecessarily because it takes many more iterations to move in a correct direction or you might even never surpass a local optimum that is not a very good parameter choice. There are different heuristics for static or variable learning rate choice.

2.2 Convolutional Neural Networks

Convolutional Neural Networks (CNN) can be translated as *folding neural network.* Here, some convolutional and pooling layers are added in front of the usual layers and thus neural networks are optimized for applications such as the processing of image or audio files. As a founder of the CNNs Yann LeCun is considered in the year 1997 [15]. In the following, we will mainly consider images as input information. Sound can be represented as a matrix similarly to a grayscale image by discretely ablating the strength of different frequencies at different points in time and does not require separate consideration here.

2.2.1 Convolutional Layer

In a CNN, the input layer is immediately followed by one or more convolutional layers. These can be understood as filtering the input for certain patterns. For example, in the figure below (4.8), the input is filtered for lines of darker or larger values running diagonally from top left to bottom right. The filter is shifted over the input (usually in matrix form) and scalar multiplied each time. This results in one value per comparison, which can indicate the correlation between the filter and the compared position in the input matrix. The value at position (i, j) of the convolution matrix B results from input matrix A and the $m \times n$-dimensional filter F as follows (first without shifting by padding, which is explained below):

$$B(i,j) = \sum_{k=1}^{m} \sum_{l=1}^{n} A(i+k-1, j+l-1) \cdot F(k,l) \tag{2.4}$$

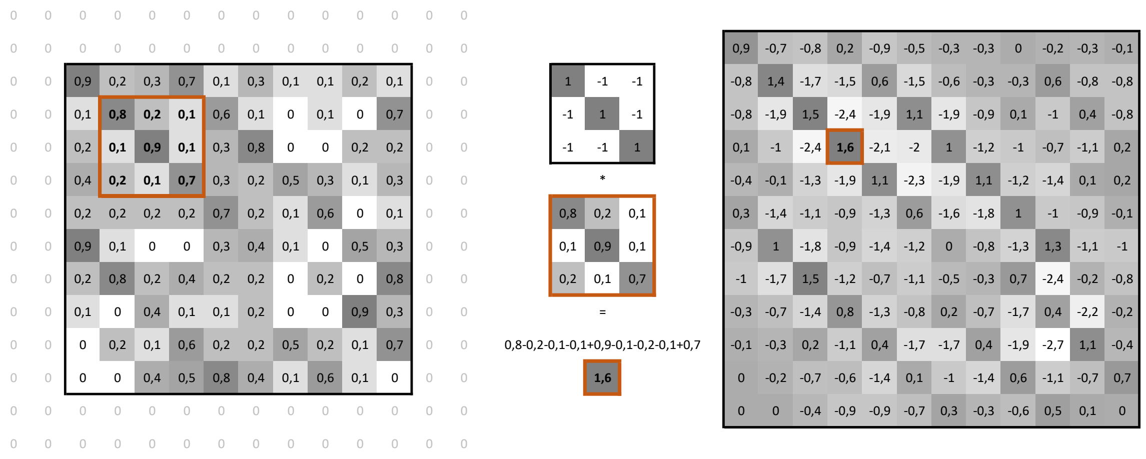


Figure 2.3: Example of convolution with MATLAB function *conv2* [16] and a filter for diagonal structures from top left to bottom right

```
Data: input image
      kernel matrix
Result: convolutional matrix
foreach image row in input image do
    foreach pixel in image row do
        set accumulator to zero;
        foreach kernel row in kernel matrix do
            foreach element in kernel row do
                if element position corresponding to pixel position then
                    multiply element value to pixel value;
                    add result to accumulator;
                end
            end
        end
        set output image pixel to accumulator;
    end
end
```

Algorithm 1: Pseudo code for one convolution
extracted from wikipedia.org/wiki/Kernel_(image_processing)

In order to consider the values at the edge equally and not to decrease the size of information, a padding is performed. In the example above, full zero padding was done; rows and columns of the value zero are added to the input matrix and the resulting larger matrix is sampled by the filter. But there are also other possibilities, like dimension-preserving padding. A distinction is made between valid, same and full padding; the differences will not be discussed in more detail here.
In a convolutional layer, many filters are applied. It is usually ensured that all of them output the same size, so that all outputs can be stored together in a tensor.

2.2.2 Pooling

To reduce the size and remove noise or redundant information, pooling is performed after some or between some convolutional layers. Commonly used are *Max Pooling* and *Average Pooling*. In Max Pooling, submatrices of a certain size are pooled by sampling the maximum value, while in Average Pooling, the average value is pooled.

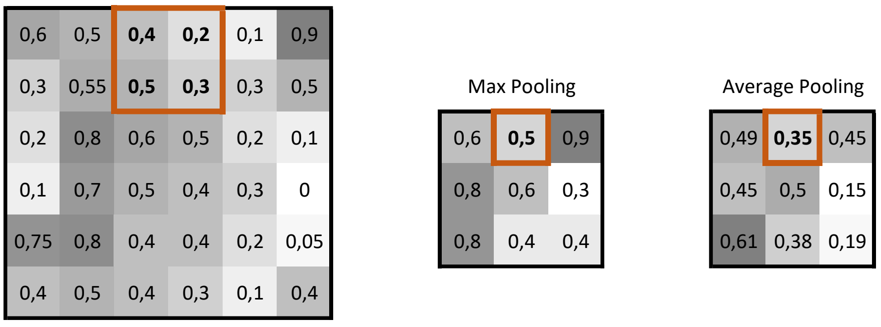


Figure 2.4: Input of a matrix (left) and comparison of 2x2 Max Pooling (middle) and 2x2 Average Pooling (right)

Different types of pooling have different properties in terms of removing noise, preventing overfitting, and size reduction. Max pooling has proven to be significantly more effective in practice and is primarily used.

2.2.3 Architecture

The selection of convolutional layers, pooling, flattening, fully connected layers including their arrangement and associated sizes is referred to as the architecture of the network. Some convolutional and pooling layers are usually followed by fully connected layers. These are basic artificial neural network layers as in 2.1, where all nodes of the previous layer have an edge to each node of the next layer. Before giving the information to the Fully Connected Layers, a flattening must be done, because the processing of the values in the tensor is complicated. For this purpose, all values are rearranged one after another in a vector or in small matrices.
CNNs are usually trained supervised, i.e. with data to which a correct output is available. Typically ReLu is used as an activation function.
A famous example for the introduction of CNNs is the handwriting recognition of digits (see 2.5) using the MNIST dataset [18]. Here all the components of the architecture discussed previously are found; two convolutional and pooling layers, flattening and fully connected layers. The output corresponds to a probability that a particular digit was recognized.
The same dataset is used in chapter 5.2 to gain insight into the structure learning of quaternion networks.

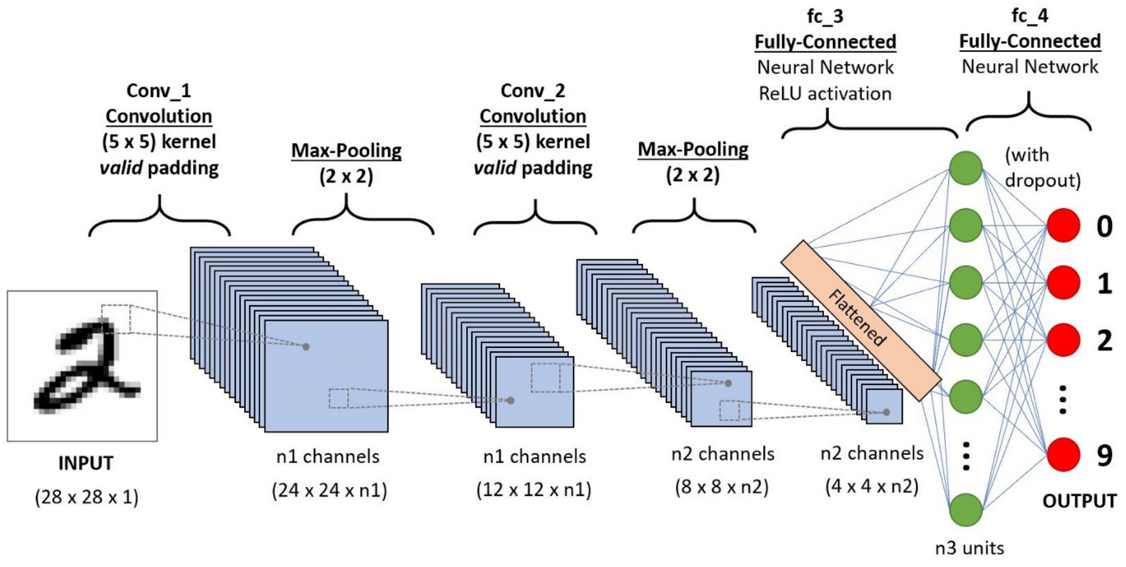


Figure 2.5: Example of Convolutional Neural Networks extracted from [19]
Handwriting recognition by an architecture with convolutional layers, max-pooling, flattening and fully connected layers

2.3 Batchlearning

If you calculate a learning step with only one image at a time, you run the risk of moving in a direction that generally does not reflect the context. Learning then takes a long time and only small steps can be taken at a time to create a meaningful learning process.
To avoid this, you learn in batches. Several images are fed forward through the network at the same time and the average of the gradients is implemented as weight change, making the learning process more robust. In general, large batches with relatively high learning rates are the fastest way to learn, but require correspondingly large data sets and a lot of memory. Also, there is a point where the batch size is so large that special properties of some classes are poorly learned because they get lost in the average. So an optimal batch size has to be found by trial and error depending on the dataset, architecture and learning rate.
It is usually a good choice to start with a batch size that is 3-10 times the number of classes to be learned, so that for each class you have a high probability of passing multiple data through the network with each batch. Also this choice is not too large, so that unique details of input data still lead to changes. Powers of two are recommended for memory management of the large tensors that arise in the intermediate steps of a forward pass.

2.4 Quaternions

In 1843 Sir William Rowan Hamilton was the first to propose a number range which extends the number range of the complex numbers. The set $\mathbb{H}$ of quaternion numbers allows an elegant description of rotations in three-dimensional euclidean space and new representations and calculation methods in many mathematical areas. Quaternions have the algebraic structure of a division ring, i.e. there exist zero and one, inversibility, and the rules of associativity and distributivity, but not commutativity. The set of quaternions can be described as:
$\mathbb{H} = \{q_0 + q_1 i + q_2 j + q_3 k, \quad q_0, q_1, q_2, q_3 \in \mathbb{R}\}$
with the real component q_0 and three imaginary components q_1, q_2, q_3 with the corresponding imaginary units i, j, k which are linearly independent of each other and obay the following conditions called the *Hamilton rules*:
$i^2 = j^2 = k^2 = ijk = -1$
The Hamilton rules lead by applying distributivity to multiplication rules:
$ij = k, jk = i, ki = j, ji = -k, kj = -i, ik = -j$

Operations for quaternions are defined in the following way:

For $\quad \hat{p} = p_0 + p_1 i + p_2 j + p_3 k \in \mathbb{H}, \quad \hat{q} = q_0 + q_1 i + q_2 j + q_3 k \in \mathbb{H}, \quad a \in \mathbb{R}$

addition	$\hat{p} + \hat{q} = (p_0 + q_0) + (p_1 + q_1)i + (p_2 + q_2)j + (p_3 + q_3)k$
scalar multiplication	$a\hat{p} = ap_0 + ap_1 i + ap_2 j + ap_3 k$
quaternion multiplication	$\hat{p}\hat{q} = (p_0q_0 - p_1q_1 - p_2q_2 - p_3q_3) + (p_0q_1 + p_1q_0 + p_2q_3 - p_3q_2)i$
	$+(p_0q_2 - p_1q_3 + p_2q_0 + p_3q_1)j + (p_0q_3 + p_1q_2 - p_2q_1 + p_3q_0)k$
conjugation	$\hat{p}^* = p_0 - p_1 i - p_2 j - p_3 k$

In the following we will represent pure quaternion numbers, i.e. quaternions without a real component p_0, in the notation $\hat{p} = p_1 i + p_2 j + p_3 k$ and their representation as a vector in the form $\mathbf{p} = (p_1, p_2, p_3)^T$.
The consideration of pure quaternions allows an interpretation in three-dimensional space. Rotating a vector $\mathbf{q} = (q_1, q_2, q_3)^T$ with an angle θ along a unit vector $\mathbf{w} = (w_1, w_2, w_3)^T$ is equivalent to the quaternion operation

$$\hat{p} = \hat{w}\hat{q}\hat{w}^* \qquad \text{with} \quad \hat{w} = \cos\frac{\theta}{2} + \sin\frac{\theta}{2}(w_1 i + w_2 j + w_3 k). \tag{2.5}$$

This property is fundamental to the idea of quaternion neural networks, as it can be used for rotations in color space.

3 Quaternion Neural Networks

The basic idea of quaternion networks is to no longer separate the color layers of an image, but to store the 3 colors of a pixel as imaginary parts in a quaternion and to pass them through the network in context. This is expected to result in image learning that corresponds more to the real-world recognition.
The idea came up after complex-valued neural networks achieved great results in the simultaneous processing of amplitude and frequency of signals [20], [21], [22]. Complex-valued networks are currently the subject of much research and have applications in various two-dimensional data [23], [24].

One interprets the input image as pure quaternions to create the quaternion valued input matrix $\hat{A} \in \mathbb{H}^{n \times n}$ with $\hat{A} = \mathbf{0} + \mathbf{R}i + \mathbf{G}j + \mathbf{B}k$, where $\mathbf{R}, \mathbf{G}, \mathbf{B}$ are the red, green and blue channels of the present image.

3.1 Quaternion Convolution

Convolving a $k \times k$ quaternion filter over a $n \times n$ quaternion input, basically works the same way as with real matrices. The filter is shifted with respect to the input, the values of the input matrix are transformed by the values in the overlaid filter and one value for the convolution matrix is calculated in each step (cf. figure 4.8 - real valued convolution).

Instead of multiplying a real pixel by a corresponding filter value, quaternion convolution rotates and scales a quaternion valued pixel within color space. One convolution with the quaternion filter shiftet to line l and column c of the input matrix is defined as:

$$\hat{f}_{lc} = \sum_{i=1}^{k} \sum_{j=1}^{k} \frac{1}{s_{ij}} \hat{w}_{ij} \hat{a}_{(l+i)(c+j)} \hat{w}_{ij}^{*} \tag{3.1}$$

where the rotation using quaternion multiplication is defined as in equation 2.5 with the corresponding rotation angle θ. The axis around which it is rotated is the gray axis in the color space, i.e. $\mu = \left(\frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3}\right)$.
Equation 3.1 is equivalent to the following representation using vector form and matrix multiplication:

$$\mathbf{f}_{lc} = \sum_{i=1}^{k} \sum_{j=1}^{k} s_{ij} \begin{pmatrix} f_1 & f_2 & f_3 \\ f_3 & f_1 & f_2 \\ f_2 & f_3 & f_1 \end{pmatrix} \mathbf{a}_{(l+i)(c+j)} \tag{3.2}$$

$$\text{with} \quad f_1 = \frac{1}{3} + \frac{2}{3}\cos\theta_{ij}, \quad f_2 = \frac{1}{3} - \frac{2}{3}\cos(\theta_{ij} - \frac{\pi}{3}), \quad f_3 = \frac{1}{3} - \frac{2}{3}\cos(\theta_{ij} + \frac{\pi}{3}) \tag{3.3}$$

Derivation of 3.2 from 3.1:

As described in 2.5, when choosing the scaling factor s_{ij} and selecting the grayscale axis in color space as rotation axis the element of the filter, $\hat{w}_{ij}$ has the form

$$\hat{w}_{ij} = s_{ij}\left(\cos\frac{\theta_{ij}}{2} + \sin\frac{\theta_{ij}}{2}\left(\frac{\sqrt{3}}{3}i + \frac{\sqrt{3}}{3}j + \frac{\sqrt{3}}{3}k\right)\right)$$

which will be written in the following way:

$$\hat{w}_{ij} = s_{ij}(y+xi+xj+xk) \text{ and } \hat{w}^*_{ij} = s_{ij}(y-xi-xj-xk) \text{ with } x := \frac{\sqrt{3}}{3}\sin\frac{\theta_{ij}}{2}, y := \cos\frac{\theta_{ij}}{2}$$

One particular pure quaternion input pixel can be represented in the form

$$\hat{a}_{(l+i)(c+j)} = 0 + ri + gj + bk$$

Using the rule for quaternion multiplication twice the operation can be extended:

$$\begin{aligned}
&\frac{1}{s_{ij}}\hat{w}_{ij}\hat{a}_{(l+i)(c+j)}\hat{w}^*_{ij} = s_{ij}(y + xi + xj + xk)(0 + ri + gj + bk)(y - xi - xj - xk)\\
&= s_{ij} \cdot \big((-rx - gx - bx) + (yr + bx - gx)i + (yg - bx + rx)j + (yb + gx - rx)k\big)(y - xi - xj - xk)\\
&= s_{ij} \cdot \big(0\\
&\qquad + (y^2r + 2ybx - 2ygx - rx^2 + 2gx^2 + 2bx^2)i\\
&\qquad + (y^2g - 2ybx + 2yrx + 2rx^2 - gx^2 + 2bx^2)j\\
&\qquad + (y^2b + 2ygx - 2yrx + 2rx^2 + 2gx^2 - bx^2)k\big)\\
&= s_{ij} \cdot \big(r(y^2 - x^2) + g(2x^2 - 2xy) + b(2x^2 + 2xy)\big)i\\
&\qquad + \big(r(2x^2 + 2xy) + g(y^2 - x^2) + b(2x^2 - 2xy)\big)j\\
&\qquad + \big(r(2x^2 - 2xy) + g(2x^2 + 2xy) + b(y^2 - x^2)\big)k\\
&= s_{ij}\begin{pmatrix} y^2 - x^2 & 2x^2 - 2xy & 2x^2 + 2xy\\ 2x^2 + 2xy & y^2 - x^2 & 2x^2 - 2xy\\ 2x^2 - 2xy & 2x^2 + 2xy & y^2 - x^2 \end{pmatrix}\begin{pmatrix} r\\ g\\ b\end{pmatrix} = s_{ij}\begin{pmatrix} f_1 & f_2 & f_3\\ f_3 & f_1 & f_2\\ f_2 & f_3 & f_1\end{pmatrix}\mathbf{a}_{(l+i)(c+j)}
\end{aligned}$$

because

$$y^2 - x^2 = \left(\cos\frac{\theta_{ij}}{2}\right)^2 - \left(\frac{\sqrt{3}}{3}\sin\frac{\theta_{ij}}{2}\right)^2 = \frac{1}{3} + \frac{2}{3}\cos\theta_{ij} = f_1$$

$$2x^2 - 2xy = 2\cdot\left(\frac{\sqrt{3}}{3}\sin\frac{\theta_{ij}}{2}\right)^2 - 2\cdot\left(\frac{\sqrt{3}}{3}\sin\frac{\theta_{ij}}{2}\right)\left(\cos\frac{\theta_{ij}}{2}\right) = \frac{1}{3} - \frac{2}{3}\cos(\theta_{ij} - \frac{\pi}{3}) = f_2$$

$$2x^2 + 2xy = 2\cdot\left(\frac{\sqrt{3}}{3}\sin\frac{\theta_{ij}}{2}\right)^2 + 2\cdot\left(\frac{\sqrt{3}}{3}\sin\frac{\theta_{ij}}{2}\right)\left(\cos\frac{\theta_{ij}}{2}\right) = \frac{1}{3} - \frac{2}{3}\cos(\theta_{ij} + \frac{\pi}{3}) = f_3$$

$\square$

Clearly the output of the quaternion convolution operation is again purely quaternion as there is no real component.

This expression of the quaternion convolution as matrix multiplication greatly simplifies the implementation for learning an optimal quaternion network.
When comparing the parameter numbers of the real convolution with those of the quaternion convolution, it is noticable that twice as many parameters are needed for a quaternion filter of the same size. While in the real filter only $k \times k$ scalar values are needed, in the quaternion filter an angle $\theta \in [-\pi/2, \pi/2]$ and a scale $s \in \mathbb{R}$ are set for each entry. It should not be overlooked that with real convolution, in contrast to quaternion convolution, only the information of one color layer is processed. Keeping this in mind, one could argue that quaternion convolution is more parameter effective.
The joint processing of the 3 color channels is presumably more resistant to overfitting effects, since individual parameters are not taught for each channel, but a more realistic image perception takes place.
While real convolution detects edges, directional structures and coarse geometries, quaternion convolution can, for example, additionally detect adjacent different color patches.

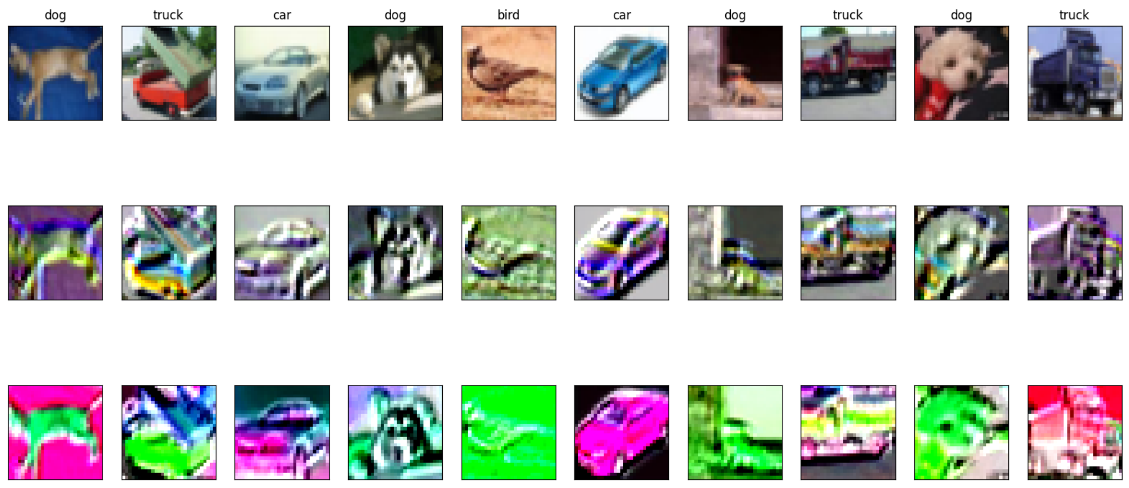


Figure 3.1: input CIFAR-10 images [25] and filtered images for 2 quaternion filters of size 4x4 with random unlearned angle θ and scale s

3.2 Pooling

To reduce the size, remove noise or redundant information and prevent overfitting effects, pooling layers are inserted between several convolutional layers and/or before the fully connected layers. This is done by sampling a certain value from submatrices of a certain size. Max-pooling or Average-Pooling as in the real valued case over the individual channels does not make sense here, since values of different layers are taken over and the color context is lost.
Instead, we maximize the magnitude of the pixels of a given submatrix. This preserves the color context, is very similar to max-pooling in the case of gray scales and provides good results. Even if filtered images and outputs of later layers are no longer real images, the idea of 'white' quaternions as the strongest firing of neurons and 'black' as a non-forwarding of information is helpful for visualizing the network.

So magnitude max-pooling of a submatrix of the size $k \times k$ can be calculated as follows:

$$f = \underset{\hat{a}^{1,1},\dots,\hat{a}^{k,k}}{\arg\max} \left((a_1^{i,j})^2 + (a_2^{i,j})^2 + (a_3^{i,j})^2 \right) \tag{3.4}$$

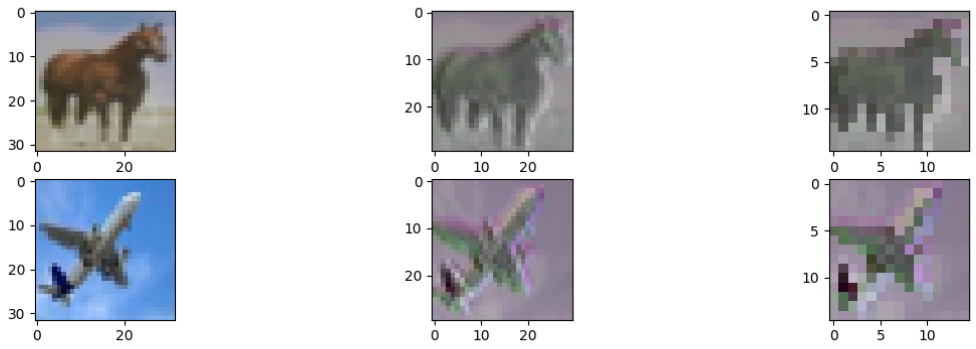


Figure 3.2: two CIFAR-10 images [25] (left) filtered with a 3x3 quaternion filter (middle) and then pooled using 2x2 quaternion max-pooling (right)

3.3 Quaternion Fully Connected Layers

In real networks, the operation on one neuron can be well described by a convolution through a filter as large as the input. Each input is multiplied by a weight and then all these product values are summed. We will not handle it differently on quaternion neurons and assign a rotation angle θ and a scaling s to each input.
The general output of a neuron with an input of k quaternion values $\hat{a}_0, \dots, \hat{a}_k$ can be calculated as follows:

$$\hat{f} = \sum_{i=1}^{k} \frac{1}{s_i} \hat{w}_i \hat{a}_i \hat{w}_i^* = \sum_{i=1}^{k} s_i \begin{pmatrix} f_1^i & f_2^i & f_3^i \\ f_3^i & f_1^i & f_2^i \\ f_2^i & f_3^i & f_1^i \end{pmatrix} \mathbf{a}_i \tag{3.5}$$

where f_1^i, f_2^i, f_3^i are defined as in 3.3 using the angle θ_i corresponding to the input value $\hat{a}_i$ We do not define a bias, because this function is taken over by the scaling s and using different summands for the different channels would detach the quaternions from their color context. If one compares the parameter numbers of real neurons with those of quaternion neurons, there roughly are twice as many parameters in the quaternion neurons. Real neurons have one weight for each input and a bias per neuron and thus $k + 1$ parameters. Quaternion neurons have an angle theta and a scale for each input and therefore $2k$ parameters. But, just as with quaternion convolution, one should not forget that each individual input consists of the three color channels and thus more information is processed simultaneously.

3.4 Activation Functions

For our architecture, ReLu was chosen as the activation function between fully connected layers. On the one hand, this is an obvious choice, since a large part of modern network architectures use ReLu, it performs well, and there is a lot of research data on it [26], [27], [28]. On the other hand, it is natural that one wants to avoid negative values for the three imaginary parts to ensure interpretation in our color space. The activation function is applied to the three imaginary parts individually.

3.5 Connection to Real Layers

To connect a quaternion layer to a real one there are different approaches. One could use the projection of the three color layers of given neurons on the gray axis as input information into a following real layer. Even though this would be a parameter efficient solution, information would be lost as three values would be reduced to one. Instead, now the three color layers are splitted as individual input values, as it is common in classical image processing networks right at the input layer. Since this is done only at the end of the network, there is still enough space to interpret the complete color information beforehand. Finally, the connection to real layers allows us to use the classic proven softmax function for class detection at the end of the network. Given a final layer with k neurons corresponding to the k classes to be classified, the softmax function returns a probability distribution for the given output. This is defined by:

$$\sigma : \mathbb{R}^k \to \left\{ z \in \mathbb{R}^k | z_i \geq 0, \sum_{i=1}^{k} z_i = 1 \right\}, \qquad \sigma(z)_j = \frac{e^{z_j}}{\sum_{i=1}^{k} e^{z_i}} \quad \text{for } j = 1, ..., k$$

Theoretically, one could also put a real gray value back into quaternion layers by simply assigning the same value to all imaginary components representing different colors. However, this and subsequent quaternion operations have little interpretation and will not be used here. The architecture used in this work only applies the transition from quaternion values to real ones.

3.6 Backpropagation

The learning process is supervised, i.e. we have the correct label for the training images. If one obtains for an input with the current configuration of the network the output $o_1, ..., o_m$ and has for comparison the correct output $y_1, ..., y_m$, the squared error is calculated as follows:

$$E = \frac{1}{2} \sum_{l=1}^{m} (o_l - y_l)^2$$

To perform updates that reduce this error derivatives of this error according to the angles θ and scales s of the network and according to posterior quaternion values $\mathbf{q}$ are needed:

$$\frac{\partial E}{\partial \theta}, \quad \frac{\partial E}{\partial s}, \quad \frac{\partial E}{\partial \mathbf{q}}$$

If one has the derivative of the error according to an intermediate quaternion value $\hat{p}$ with its vector representation $\mathbf{p}$, which is calculated from $\hat{p} = \frac{1}{s}\hat{w}\hat{q}\hat{w}^*$, then these derivatives result as follows:

$$\frac{\partial E}{\partial \theta} = \frac{\partial E}{\partial \mathbf{p}} \frac{\partial \mathbf{p}}{\partial \theta}, \quad \frac{\partial E}{\partial s} = \frac{\partial E}{\partial \mathbf{p}} \frac{\partial \mathbf{p}}{\partial s}, \quad \frac{\partial E}{\partial \mathbf{q}} = \frac{\partial E}{\partial \mathbf{p}} \frac{\partial \mathbf{p}}{\partial \mathbf{q}}$$

Using again the property

$$\hat{p} = \frac{1}{s}\hat{w}\hat{q}\hat{w}^* \;\hat{=}\; s \begin{pmatrix} f_1 & f_2 & f_3 \\ f_3 & f_1 & f_2 \\ f_2 & f_3 & f_1 \end{pmatrix} \mathbf{q}$$

the gradients can be calculated explicitly:

▶ $$\frac{\partial E}{\partial \theta} = \frac{\partial E}{\partial \mathbf{p}} \frac{\partial \mathbf{p}}{\partial \theta} = \frac{\partial E}{\partial \mathbf{p}}\; s \begin{pmatrix} f'_1 & f'_2 & f'_3 \\ f'_3 & f'_1 & f'_2 \\ f'_2 & f'_3 & f'_1 \end{pmatrix} \mathbf{q}$$

$$\text{as} \quad \frac{\partial \mathbf{p}}{\partial \theta} = \begin{pmatrix} \frac{\partial p_1}{\partial \theta} \\ \frac{\partial p_2}{\partial \theta} \\ \frac{\partial p_3}{\partial \theta} \end{pmatrix} = \begin{pmatrix} \frac{\partial(sf_1q_1 + sf_2q_2 + sf_3q_3)}{\partial \theta} \\ \frac{\partial(sf_3q_1 + sf_1q_2 + sf_2q_3)}{\partial \theta} \\ \frac{\partial(sf_2q_1 + sf_3q_2 + sf_1q_3)}{\partial \theta} \end{pmatrix}$$

$$= \begin{pmatrix} s(f'_1q_1 + f'_2q_2 + f'_3q_3) \\ s(f'_3q_1 + f'_1q_2 + f'_2q_3) \\ s(f'_2q_1 + f'_3q_2 + f'_1q_3) \end{pmatrix} = s \begin{pmatrix} f'_1 & f'_2 & f'_3 \\ f'_3 & f'_1 & f'_2 \\ f'_2 & f'_3 & f'_1 \end{pmatrix} \mathbf{q}$$

$$\text{with} \quad f'_1 = \frac{\partial\left(\frac{1}{3} + \frac{2}{3}\cos\theta\right)}{\partial \theta} = -\frac{2}{3}\sin\theta$$

$$f'_2 = \frac{\partial\left(\frac{1}{3} - \frac{2}{3}\cos(\theta - \frac{\pi}{3})\right)}{\partial \theta} = -\frac{2}{3}\cos\left(\theta + \frac{\pi}{6}\right)$$

$$f'_3 = \frac{\partial\left(\frac{1}{3} - \frac{2}{3}\cos(\theta + \frac{\pi}{3})\right)}{\partial \theta} = \frac{2}{3}\cos\left(\frac{\pi}{6} - \theta\right)$$

▶ $$\frac{\partial E}{\partial s} = \frac{\partial E}{\partial \mathbf{p}} \frac{\partial \mathbf{p}}{\partial s} = \frac{\partial E}{\partial \mathbf{p}} \begin{pmatrix} f_1 & f_2 & f_3 \\ f_3 & f_1 & f_2 \\ f_2 & f_3 & f_1 \end{pmatrix} \mathbf{q}$$

$$\text{as} \quad \frac{\partial \mathbf{p}}{\partial s} = \begin{pmatrix} \frac{\partial p_1}{\partial s} \\ \frac{\partial p_2}{\partial s} \\ \frac{\partial p_3}{\partial s} \end{pmatrix} = \begin{pmatrix} \frac{\partial(sf_1q_1 + sf_2q_2 + sf_3q_3)}{\partial s} \\ \frac{\partial(sf_3q_1 + sf_1q_2 + sf_2q_3)}{\partial s} \\ \frac{\partial(sf_2q_1 + sf_3q_2 + sf_1q_3)}{\partial s} \end{pmatrix}$$

$$= \begin{pmatrix} f'_1q_1 + f'_2q_2 + f'_3q_3 \\ f'_3q_1 + f'_1q_2 + f'_2q_3 \\ f'_2q_1 + f'_3q_2 + f'_1q_3 \end{pmatrix} = \begin{pmatrix} f_1 & f_2 & f_3 \\ f_3 & f_1 & f_2 \\ f_2 & f_3 & f_1 \end{pmatrix} \mathbf{q}$$

▶ $$\frac{\partial E}{\partial \mathbf{q}} = \frac{\partial E}{\partial \mathbf{p}} \frac{\partial \mathbf{p}}{\partial \mathbf{q}} = \frac{\partial E}{\partial \mathbf{p}} \; s \begin{pmatrix} f_1 & f_2 & f_3 \\ f_3 & f_1 & f_2 \\ f_2 & f_3 & f_1 \end{pmatrix}$$

$$\text{as} \quad \frac{\partial \mathbf{p}}{\partial \mathbf{q}} = \begin{pmatrix} \frac{\partial \mathbf{p}}{\partial q_1} & \frac{\partial \mathbf{p}}{\partial q_2} & \frac{\partial \mathbf{p}}{\partial q_3} \end{pmatrix} = \begin{pmatrix} \frac{\partial p_1}{\partial q_1} & \frac{\partial p_1}{\partial q_2} & \frac{\partial p_1}{\partial q_3} \\ \frac{\partial p_2}{\partial q_1} & \frac{\partial p_2}{\partial q_2} & \frac{\partial p_2}{\partial q_3} \\ \frac{\partial p_3}{\partial q_1} & \frac{\partial p_3}{\partial q_2} & \frac{\partial p_3}{\partial q_3} \end{pmatrix}$$

$$= \begin{pmatrix} \frac{\partial(sf_1q_1 + sf_2q_2 + sf_3q_3)}{\partial q_1} & \frac{\partial(sf_1q_1 + sf_2q_2 + sf_3q_3)}{\partial q_2} & \frac{\partial(sf_1q_1 + sf_2q_2 + sf_3q_3)}{\partial q_3} \\ \frac{\partial(sf_3q_1 + sf_1q_2 + sf_2q_3)}{\partial q_1} & \frac{\partial(sf_3q_1 + sf_1q_2 + sf_2q_3)}{\partial q_2} & \frac{\partial(sf_3q_1 + sf_1q_2 + sf_2q_3)}{\partial q_3} \\ \frac{\partial(sf_2q_1 + sf_3q_2 + sf_1q_3)}{\partial q_1} & \frac{\partial(sf_2q_1 + sf_3q_2 + sf_1q_3)}{\partial q_2} & \frac{\partial(sf_2q_1 + sf_3q_2 + sf_1q_3)}{\partial q_3} \end{pmatrix}$$

$$= s \begin{pmatrix} f_1 & f_2 & f_3 \\ f_3 & f_1 & f_2 \\ f_2 & f_3 & f_1 \end{pmatrix}$$

While the derivation of quaternion functions can be generally very complicated due to the lack of commutativity, the rotation of pure quaternions by this explicit form is very convenient in this respect.

Now that all the components are defined, one can assemble a complete architecture of a quatenion convolutional neural network. Input images are interpreted as pure quaternions with the RGB-channels on each of the imaginary axes and given into a layer of multiple convolutions. The output of the convolutional layer can either be given into further convolutions or reduced with magnitude max-pooling. After these convolutional and pooling layers, a flattening is performed, which arranges the individual resulting pixels made of pure quaternion values one below the other as a vector. This provides input to the following fully connected quaternion layers, each of whose neurons are endowed with a ReLu activation function. Finally, moving to real layers by flattening the quaternion values each to three real values. The last layer of the network consists of k neurons corresponding to the k classes to be assigned classification. The final softmax function assigns probabilities to the various predictions. This setup relates heavily on the work of Xuanyu Zhu and Yi Xu [29].
A scheme for this basic structure can be found in the following figure 3.3.

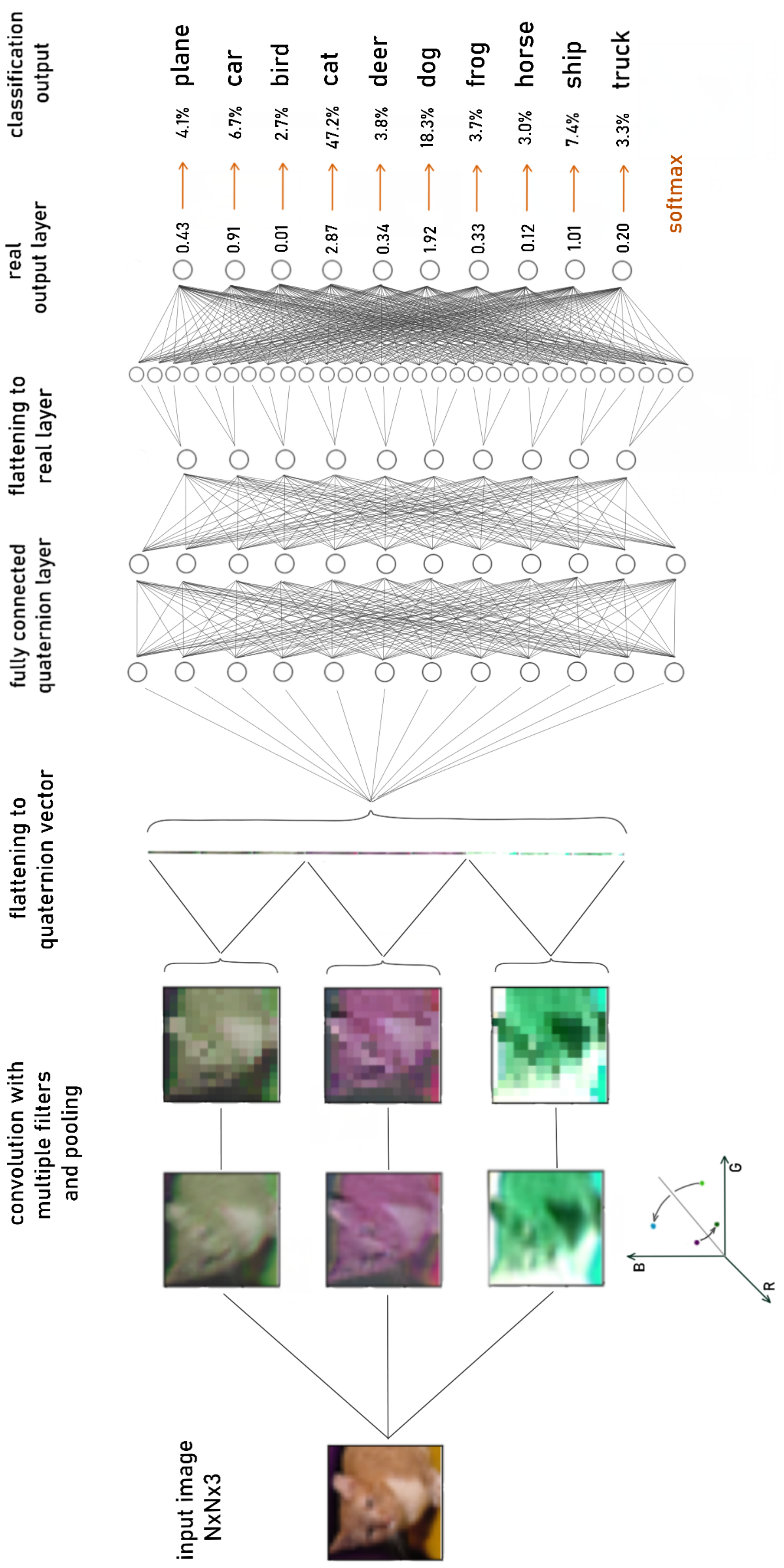


Figure 3.3:
Scheme for the Architecture of the Quaternion Neural Network

4 Implementation in Python

Python was chosen for the implementation of a quaternion network because it is popular for artificial intelligence and image processing applications, it is multiparadigmatic and effective. It also has many libraries that facilitate the handling of image datasets, large tensors and neural networks. Despite using their functions occasionally, the network itself will be implemented from scratch, starting with simple quaternion neurons, moving to layers, and finally to convolution. In doing so, we have access to all data, including intermediate steps. Among other things, this enables a detailed visualization of the learning process and all steps in the network, as already shown in some figures of this work. Care was taken to program as modularly as possible in order to guarantee testability, clarity and rapid adaptation.

For some operations, such as data handeling, backpropagation, parameter optimization steps, and working with large tensors, the Python package PyTorch was used [30]. The essential structure, consisting of the most important functions for setting up the network and one basic example, will be shown in this work by means of code snippets. For the complete code including executable scripts there is a freely accessible Git repository corresponding to this work. There you can also find which libraries and versions were used in detail.

First, simple quaternion dummy neurons are to be implemented and then merged into layers:

```python
# prepare the entity quaternion neuron
class QuatNeuron:
    pass

# create a whole layer of quaternion neurons passing the desired width
def createquatlayer(numberneurons):
    layer = []
    for i in range(numberneurons):
        this = QuatNeuron()
        layer.append(this)
    return layer
```

code sample 4.1: Preparation of a quaternion layer

When connecting two such layers, parameters must be set that regulate the weighting of input information from the first layer into the respective neurons of the second layer. Between two quaternion layers, these parameters are an angle theta and a scale s respectively for each connection of the two layers. Using the Xavier initialization method for weights initialization, a normalized initialization considering the size of input and output for the scale [31], and a uniform distribution for theta, the first setup of the network is ready.

```python
# connecting two adjacent layers and initializing all necessary parameters
# this version of the function is not final
def connectlayers(layer1, layer2):
     params = torch.zeros(len(layer1), len(layer2), 2)
     for i in range(len(layer1)):
          for j in range(len(layer2)):
              # initializing the angle theta
              params[i, j, 0] = torch.tensor(np.random.uniform(
                -np.pi/2, np.pi/2))
              # initializing the scale s
              params[i, j, 1] = torch.tensor(np.random.uniform(
                -np.sqrt(6) / np.sqrt(len(layer1) + len(layer2)),
                +np.sqrt(6) / np.sqrt(len(layer1) + len(layer2))))
    params.requires_grad = True
    return params
```

code sample 4.2: Connecting two quaternion layers and initializing necessary parameters

These functions are not programmed to be super effective, as they are only executed once at the beginning of the program to prepare the network and take less than a second. The functions can be called as follows to create a simple architecture with 3 layers of relatively small width and initialize associated parameters:

```python
# call 'createquatlayer' to create a quaternion layer of certain width
layer1 = createquatlayer(64)
layer2 = createquatlayer(32)
layer3 = createquatlayer(16)
# connect two layers by calling 'connectlayers' and passing the layers
# this returns the related parameters for the connection
params12 = connectlayers(layer1, layer2)
params23 = connectlayers(layer2, layer3)
```

code sample 4.3: Creating a simple 64-32-16 quaternion layer architecture

Since the input to the network is considered as an input to a new quaternion layer, the first layer must be defined and initialized with parameters depending on the size of the input, just like between two quaternion layers:

```python
# first layer must be declared seperately to prepare the first parameters
def declarefirstlayer(inputsize, layer):
    params = torch.zeros(inputsize, len(layer), 2)
    for i in range(inputsize):
        for j in range(len(layer)):
            # initializing the angle theta
            params[i, j, 0] = torch.tensor(np.random.uniform(
              -np.pi/2, np.pi/2))
            # initializing the scale s
            params[i, j, 1] = torch.tensor(np.random.uniform(
              -np.sqrt(6) / np.sqrt(inputsize + len(layer)),
              +np.sqrt(6) / np.sqrt(inputsize + len(layer))))
    params.requires_grad = True
    return params
```

code sample 4.4: Preparation of the first layer of the network

Passing the output information of an earlier layer through the next one is done as described in chapter 3.3 - Quaternion Fully Connected Layers. Each connection between the layers is weighted by rotation and scaling. By working with large tensors, loops across the neurons of both layers can be avoided, which would have meant a large increase in runtime. The only loop that cannot be avoided is the one over the elements of the batch. The following code belongs to the function that takes the output of a previous layer and the parameters of the connection between the two layers as input values and returns the output of the neurons of the later layer.

```python
# one forward step between two quaternion layers
def forwardlayerquat(input, params):
    # input has the form [k,3,i] with k - element of the batch,
    # i - output of neuron i from the earlier layer,
    # and respectively the three color values

    # params has the form [i,j,2] and is the parameters for connecting
    # the earlier layer with i neurons to the later layer with j neurons,
    # assigning theta and scale respectively

    # preparing to have an output of size [k,3,j] with
    # k - element of the batch, j - output of neuron j of the later layer
    # (before activation) and respectively the three color values
    output = torch.zeros(len(input[:, 0, 0]), 3, len(params[0, :, 0]))

    # preparation of the rotation for every neuron i from the earlier layer
    # and output neuron j from the later one
    # using the angle theta from params
    f1 = 1/3 + 2 / 3 * torch.cos(params[:, :, 0])
    f2 = 1/3 - 2 / 3 * torch.cos(params[:, :, 0] - torch.tensor(np.pi) / 3)
    f3 = 1/3 - 2 / 3 * torch.cos(params[:, :, 0] + torch.tensor(np.pi) / 3)
    ff = torch.stack(
        [f1, f2, f3, f3, f1, f2, f2, f3, f1],
        dim=2).unflatten(dim=2, sizes=(3, 3))
    # ff has the size [i,j,3,3] assigning a 3x3 rotation matrix to every
    # network connection (i,j)

    # stacking the scale from params for each connection (i,j) three times
    # to facilitate multiplication with the tensor later on
    scal = torch.stack([params[:, :, 1], params[:, :, 1], params[:, :, 1]])
    scal = scal.permute(1, 0, 2)
    # scal has the size [i,j,3]

    # looping over the elements k of the batch
    for k in range(len(input[:, 0, 0])):
        # multiplication of the rotation matrix with the input vector
        # processing all the connections between the layers simultaneously
        rot = torch.matmul(ff[:, :, :, :], input[k, :, :])
        rot = torch.diagonal(rot, offset=0, dim1=0, dim2=3)
        rot = rot.permute(2, 1, 0)

        # scaling the rotated values, summing up the values of each
        # output neuron j and adding them to the output
        output[k, :, :] = torch.add(output[k, :, :],
                              torch.sum((scal[:,:,:] * rot[:,:,:]), dim=0))

    # manually deleting big tensors that are no longer neededeed
    del ff, scal, rot
    gc.collect()

    return output
```

code sample 4.5: Passing the output of an earlier quaternion layer through the next one

At the end the goal is to perform a flattening along the 3 color components and connect them to a real layer to finally calculate the different class predictions with softmax. For this we need to define real neurons and layers and associated parameter initialization. The function *connectlayers* will be adapted to distinguish the different types of layers that are to be connected.

```
# prepare the entity real neuron
class RealNeuron:
    pass

# create a whole layer of real neurons passing the desired width
def createreallayer(numberneurons):
    layer = []
    for i in range(numberneurons):
        this = RealNeuron()
        layer.append(this)
    return layer

# connecting two adjacent layers and initializing all necessary parameters
# this is done with differentiation of the different types of layers
def connectlayers(layer1, layer2):

    if type(layer1[0]) == QuatNeuron and type(layer2[0]) == RealNeuron:
        # parameters between a quaternion and a real layer
        # manually flattening the quaternion output before calling
        # this function is required
        # proceeding as if each quaternion neuron was 3 real neurons
        params = torch.zeros(3 * len(layer1), len(layer2), 2)
        for i in range(3 * len(layer1)):
            for j in range(len(layer2)):
                # initializing the weight w
                params[i, j, 0] = torch.tensor(np.random.uniform(
                  -np.sqrt(6) / np.sqrt(3*len(layer1) + len(layer2)),
                  +np.sqrt(6) / np.sqrt(3*len(layer1) + len(layer2))))
                # initializing the bias b
                params[i, j, 1] = torch.tensor(0.1)

    elif type(layer1[0]) == RealNeuron and type(layer2[0]) == RealNeuron:
        # parameters between two real layers
        params = torch.zeros(len(layer1), len(layer2), 2)
        for i in range(len(layer1)):
            for j in range(len(layer2)):
                # initializing the weight w
                params[i, j, 0] = torch.tensor(np.random.uniform(
                  -np.sqrt(6) / np.sqrt(len(layer1) + len(layer2)),
                  +np.sqrt(6) / np.sqrt(len(layer1) + len(layer2))))
                # initializing the bias b
                params[i, j, 1] = torch.tensor(0.1)

    else:
        # parameters between two quaternion layers
        params = torch.zeros(len(layer1), len(layer2), 2)
        for i in range(len(layer1)):
            for j in range(len(layer2)):
                # initializing the angle theta
                params[i, j, 0] = torch.tensor(np.random.uniform(
                  -np.pi/2, np.pi/2))
                # initializing the scale s
                params[i, j, 1] = torch.tensor(np.random.uniform(
                  -np.sqrt(6) / np.sqrt(len(layer1) + len(layer2)),
                  +np.sqrt(6) / np.sqrt(len(layer1) + len(layer2))))
    params.requires_grad = True
    return params


# one forward step through a real layer
# not very effective because of nested loops, but in the runtime
# irrelevant compared to other parts of the network
def forwardlayerreal(input, params):
    output = torch.zeros(len(input[:, 0]), len(params[0, :, 0]))
    # looping over the elements k of the batch
    for k in range(len(input[:, 0])):
        # looping over the neurons of the output layer
        for j in range(len(params[0, :, 0])):
            # looping over the input
            for i in range(len(params[:, 0, 0])):
                # weighing the input with the corresponding weight
                output[k, j] = output[k, j] + (params[i, j, 0]*input[k, i])
            # adding the bias once per neuron
            output[k, j] = output[k, j] + params[0, j, 1]
    return output
```

code sample 4.6: Preparing real neurons and layers

Having all these functions to construction and forward pass, one can already build a simple quaternion neural network:

```
# creating the layers of the networklayers:
layer1 = createquatlayer(128) # quaternion layer (128 neurons)
layer2 = createquatlayer(64)  # quaternion layer (64 neurons)
```

```
layer3 = createreallayer(10)  # real layer (10 neurons) output layer

# input size of the first layer must match the size of the input images
params1 = declarefirstlayer(1024, layer1)

# initializing all parameters between the layers
params12 = connectlayers(layer1, layer2)
params23 = connectlayers(layer2, layer3)

# passing the parameters to be optimized by backpropagation
# to the stochastic gradient descent optimizer from torch
# also passing the learning rate and the momentum for learning rate change
optimizer = optim.SGD([params1, params12, params23], lr=0.01, momentum=0.9)

# defining functions used in the network
# activation function ReLu
activation = torch.nn.ReLU()
# softmax for the last layer to get a probability distribution
m = nn.Softmax(dim=1)
# calculating the loss betwenn output und labels with cross entropy loss
loss = nn.CrossEntropyLoss()

# how many batches we want to learn, counting the batch starting with i=0
batchnumber = 500
i=0

# the actual learning loop
while i < batchnumber:
  # empty the optimizer saving all the gradients
  optimizer.zero_grad()

  # load a new batch of images
  # for this a corresponding dataloader must be defined
  images, y = dataiter.next()
  # flattening the images to act as quaternion vectors
  # as input format for the first layer
  flattened = torch.flatten(images, start_dim=2)

  # passing the input through the first layer and applying activation
  out1 = activation(forwardquatlayer(flattened, params1))
  # pass trough the next layer
  out2 = activation(forwardquatlayer(out1, params12))

  # flattening the 3 values to act as 3 real inputs for the real layer
  out2flat = torch.transpose(out2, 2, 1).flatten(start_dim=1)

  # passing the flattened values through the real layer
  out = activation(forwardreallayer(out2flat, params23))

  # calculating the prediction as it is the maximum of the softmax
  prediction = torch.argmax(m(out), 1)
  likelihood = torch.max(m(out), 1)

  # calculating the loss to the correct labels
  l = loss(out.float(), y.long())

  # calculating the learning steps via backpropagation
  l.backward()
  # implementing the learning step on the parameters of the network
  optimizer.step()

  # collecting the variables no longer needed to clear memory
  del out1, out2, out, flattened, out2flat, prediction, likelihood
  gc.collect()

  # counting up one batch
  i += 1
```

code sample 4.7: Skript for a working feedforward quaternion network

This flow is just the pure learning on several batches of training images to briefly illustrate the principle way of workflow. Details on including packages, loading the data, visual evaluation and working with a validation dataset can be found in the associated repository.

The following functions are used to prepare the quaternion convolution. They allow initializing a whole layer of several convolutions of the same size and the corresponding parameters:

```
# creation of a single convolutional matrix
def createquatconv(size):
    convparams = torch.zeros(size, size, 2)
    for i in range(size):
        for j in range(size):
            # initializing the angle theta
            convparams[i, j, 0]=torch.tensor(np.random.uniform(
                  -np.pi/2, np.pi/2))
            # initializing the scale s
            convparams[i, j, 1]=torch.tensor(np.random.uniform(0.0,2.0))
    return convparams


# creation of an entire layer of similar size convolutions
def createquatconvlayer(number, size):
    convparams = torch.zeros(number, size, size, 2)
    # calling 'createquatconv' for every convolution matrix of the layer
    for i in range(number):
        convparams[i, :, :, :] = createquatconv(size)
    convparams.requires_grad = True
    return convparams
```

code sample 4.8: Preparation of quaternion convolutional layers

```
def forwardconv(input, convparams):
    # checking whether it is the first convolutional layer, in this case
    # the input has no dimension stacking the different
    # convolutional outputs of the previous layer
    if len(input.size()) == 4:
        # getting the number of convolutions
        number = len(convparams[:, 0, 0, 0])
        # getting the size of the convolutions
        size = len(convparams[0, :, 0, 0])
        # preparation of the output having dimensions for the batchsize,
        # the number of convolutions
        # and the precalculated size of convolution output matrix
        output = torch.zeros(len(input[:, 0, 0, 0]), number, 3,
          len(input[0, 0, :, 0]) - size + 1,
          len(input[0, 0, 0, :]) - size + 1)
        # preparing the rotation for every filter matrix and
        # all corresponding pixels, using the angle theta from convparams
        f1 = 1/3+2/3*torch.cos(convparams[:,:,:,0])
        f2 = 1/3-2/3*torch.cos(convparams[:,:,:,0]-torch.tensor(np.pi)/3)
        f3 = 1/3-2/3*torch.cos(convparams[:,:,:,0]+torch.tensor(np.pi)/3)
        ff = torch.stack([f1,f2,f3,f3,f1,f2,f2,f3,f1], dim=3)
          .unflatten(dim=3, sizes=(3, 3))
        # stacking the scale from convparams for each rotation three times
        # to facilitate multiplication with the tensor later on
        scal = torch.stack([convparams[:, :, :, 1], convparams[:, :, :, 1],
          convparams[:, :, :, 1]])
        scal = scal.permute(1,0,2,3)
        # preparing the input to fit the tensor multiplication
        input = input.permute(1, 0, 2, 3)

        # looping over the dimensions of the filter
        for p in range(size):
            for l in range(size):
                # looping over the output of the convolution
                for i in range(len(output[0, 0, 0, :, 0])):
                    for j in range(len(output[0, 0, 0, 0, :])):
                        # calculating the rotated values
                        rot = torch.matmul(ff[:, p, l, :, :],
                          input[:, :, i + p, j + l])
                        rot = rot.permute(2, 0, 1)
                        # scaling and adding rotated values to the output
                        output[:,:,:,i,j] = torch.add(output[:,:,:,i,j],
                          scal[:, :, p, l]/(size*size) * rot[:, :, :])

    # if the input is 5-dimensional we already had different convolutions
    # and need to loop over the according outputs as well
    else:
        # getting the number of different inputs from the last convolution
        inputnum = len(convparams[:, 0, 0, 0])
        # getting the number of convolutions to do in this layer
        filtnum = len(convparams[:, 0, 0, 0])
        # calculating the resulting number of different convolution outputs
        number = inputnum * filtnum
        # getting the size of the convolutions
        size = len(convparams[0, :, 0, 0])
        # preparation of the output having dimensions for the batchsize,
        # the number of convolutions
        # and the precalculated size of convolution output matrix
        output = torch.zeros(len(input[:, 0, 0, 0, 0]), number, 3,
          len(input[0, 0, 0, :, 0]) - size + 1,
          len(input[0, 0, 0, :, 0]) - size + 1)
        # preparing the rotation for every filter matrix and
        # all corresponding pixels, using the angle theta from convparams
        f1 = 1/3+2/3*torch.cos(convparams[:,:,:,0])
        f2 = 1/3-2/3*torch.cos(convparams[:,:,:,0]-torch.tensor(np.pi)/3)
        f3 = 1/3-2/3*torch.cos(convparams[:,:,:,0]+torch.tensor(np.pi)/3)
        ff = torch.stack([f1,f2,f3,f3,f1,f2,f2,f3,f1], dim=3)
          .unflatten(dim=3, sizes=(3, 3))
        # stacking the scale from convparams for each rotation three times
        # to facilitate multiplication with the tensor later on
        scal = torch.stack([convparams[:, :, :, 1], convparams[:, :, :, 1],
          convparams[:, :, :, 1]])
        scal = scal.permute(1,0,2,3)
        # preparing the input to fit the tensor multiplication
        input = input.permute(2, 0, 1, 3, 4)

        # looping over the dimensions of the filter
        for p in range(size):
            for l in range(size):
                # looping over the output of the convolution
                for i in range(len(output[0, 0, 0, :, 0])):
                    for j in range(len(output[0, 0, 0, 0, :])):
                        # looping over the different convolution
                        # outputs of the last layer
                        for t in range(inputnum):
                            # calculating the rotated values
                            rot = torch.matmul(ff[:, p, l, :, :],
                            input[:, :, t, i + p, j + l])
                            rot = rot.permute(2,0,1)
                            # scaling the values and
                            # adding to the output
                            output[:,t*filtnum:(t+1)*filtnum,:,i,j] =
                              torch.add(
                              output[:,t*filtnum:(t+1)*filtnum,:,i,j],
                              scal[:, :, p, l]/(size*size) * rot[:,:, :])
    # manually deleting big tensors that are no longer needed
    del input, number, size, scal, ff, rot, f1, f2, f3
    gc.collect()
    return output
```

code sample 4.9: Passing an input through one convolutional layer

This provides all the tools to assemble a network and run the forward pass. For the back-propagation *torch.autograd* was used, a PyTorch tool which automatically calculates the gradient and updates the parameters [30]. This is possible here because all operations were expressed as tensor operations.
To prove that the model is able to learn and approximate a relationship by backpropa-

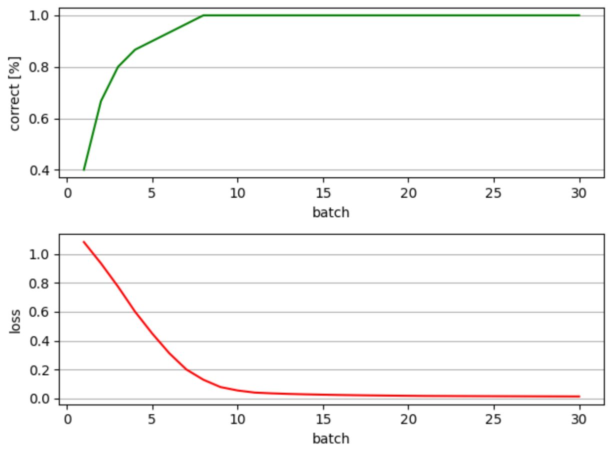
gation, it was first trained on a small subset, which was also used for testing. This does not make sense if a real relation is to be learned, since thus only for exactly the training data an overfitted relation is learned, but proves the concept of the approximation by the network. The used data set Cifar-10 is explained in more detail in chapter 5.1 [32].

```python
from fullyconnectedlayer import createreallayer, connectlayers,
    createquatlayer, declarefirstlayer, forwardlayerreal, forwardlayerquat
from convolutionallayer import createquatconvlayer, forwardconv,
    quatmaxpool
import torch
import numpy as np
import matplotlib.pyplot as plt
from tools import topncorrect, convert_to_imshow_format
import torchvision
import torchvision.transforms as transforms
from torch import optim
import torch.nn as nn
from sklearn import metrics
import seaborn as sns
import gc

# proof of concept on static dataset with now train-test-split

# preparation of Cifar10 dataset
transform = transforms.Compose(
    [transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
batchsize = 30

trainset = torchvision.datasets.CIFAR10(root='/home/CIFAR-10PyTorch/data/',
                                        train=True,
                                        download=True,
                                        transform=transform)
trainloader = torch.utils.data.DataLoader(trainset,
                                          batch_size=batchsize,
                                          shuffle=True)
classes = ['plane', 'car', 'bird', 'cat', 'deer',
           'dog', 'frog', 'horse', 'ship', 'truck']

# select subset of classes to train on for now
classes = ['cat', 'deer', 'ship']
chosenclasses = [3, 4, 8]
mapping = {3: 0, 4: 1, 8: 2}
trainset.data = [trainset.data[ind] for ind, x in
    enumerate(trainset.targets) if (x == 3) | (x == 4) | (x == 8)]
trainset.targets = [x for ind, x in
    enumerate(trainset.targets) if (x == 3) | (x == 4) | (x == 8)]
trainset.targets = [mapping.get(number, number) for number in trainset.
    targets]
dataiter = iter(trainloader)

# creating the architecture of the easy network consisting of 1 convolution
```

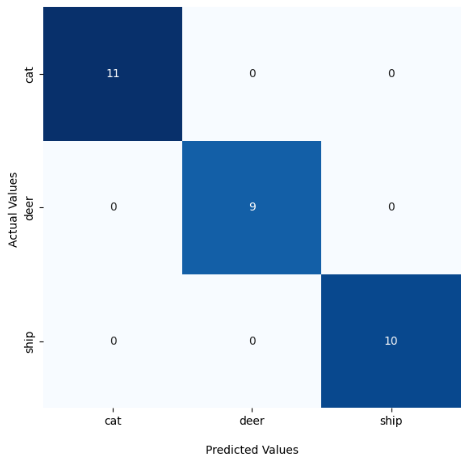
```
    with 2 filters, one quaternion layer and one real layer
conv1 = createquatconvlayer(2, 4)
layer1 = createquatlayer(32)
layer2 = createreallayer(3)
params1 = declarefirstlayer(162, layer1)
params12 = connectlayers(layer1, layer2)

# defining optimizer and loss using torch, passing in learnable parameters
optimizer = optim.SGD([params1, params12, conv1], lr=0.5, momentum=0.5)

# training and testing on the same data every step to prove, that learning
    an approximation is possible
images, y = dataiter.next()

# collecting training, loss scores for plot
training = []
lossplot = []

# preparations for learning loop
loss = nn.CrossEntropyLoss()
l = 1000
m = nn.Softmax(dim=1)
i = 0
activation = torch.nn.ReLU()
batchnumber = 30
currentrate = optimizer.param_groups[0]['lr']
print(f'Learning {batchnumber} batches of size {batchsize} starting with a
    learning rate of {currentrate}.')

# learning loop
while i < batchnumber:
    i += 1
    optimizer.zero_grad()
    # forward pass
    convout = forwardconv(images, conv1)
    convout = quatmaxpool(convout, 3, 3)
    convout = convout.permute(0, 2, 1, 3, 4)
    flattened = torch.flatten(convout, start_dim=2)

    try:
        out = activation(forwardlayerquat(flattened, params1))
    except:
        print("adjust size of input into first neuronal layer to fit size
    of flattened in 'declarefirstlayer'")
        print(flattened.size())
        break

    outflat = torch.transpose(out, 2, 1).flatten(start_dim=1)
    out = activation(forwardlayerreal(outflat, params12))
    output = out.float()

    prediction = torch.argmax(m(output), 1)
    likelihood = torch.max(m(out), 1)
    l = loss(output, y.long())
    difference = prediction-y
    correct = torch.numel(difference[difference==0])
    training.append(correct / len(y))

    # backpropagation and update
    l.backward()
    optimizer.step()
    ll = l.detach()
    lossplot.append(ll)
    currentrate = optimizer.param_groups[0]['lr']
    if (i) % 8 == 0:
        optimizer.param_groups[0]['lr'] =
    currentrate * optimizer.param_groups[0]['momentum']
    print(f'\n\nbatch {i}:    correct: {correct}/{len(y)},
    learning rate: {currentrate:.5f}')

    del convout, flattened, outflat
    gc.collect()

# plot training curve and confusion matrix
# full code in git
```

code sample 4.10: Learning process on a static data set with out train-test-split for proof of concept

The results show a clear ability to approximate the present context.



(a) Learning curve for a static batch



(b) Confusion matrix for a static batch

Figure 4.1: Testing the ability to approximate a context by using a static batch with no train-test-split

Since a typical learning curve is obtained and it has been proven that the code is capable of learning, some properties of the learning process can now be investigated.

5 Experiments and Results

5.1 The influence of noise on the learning process

To test the resistance of the learning process to noise and gain a better understanding of the learning process, training was performed on the Cifar-10 dataset [32]. This dataset is widely used for training new machine learning methods. It consists of 60,000 color images of size 32x32 pixel representing 10 different classes with respectively 6,000 images. The 10 different classes are airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. The training is challenging because the images are vague and not standardized in terms of light, angle, distance and color. Training on such a challenging dataset is expected to produce a robust model with a deeper understanding of the issue, rather than a very specific one that gives poor results even with small deviations. Comparison of different models on this data set is standard. Prediction accuracies range from 50% for simpler models to 99.4% through networks with up to more than 200 million parameters [33][34]. We experiment here with a simpler model with about a million parameters. We also limit ourselves to a subset of 4 classes: Cat, Dog, Horse, Deer. These are not easily distinguishable classes, especially between Horse and Deer and between Cat and Dog errors are to be expected.

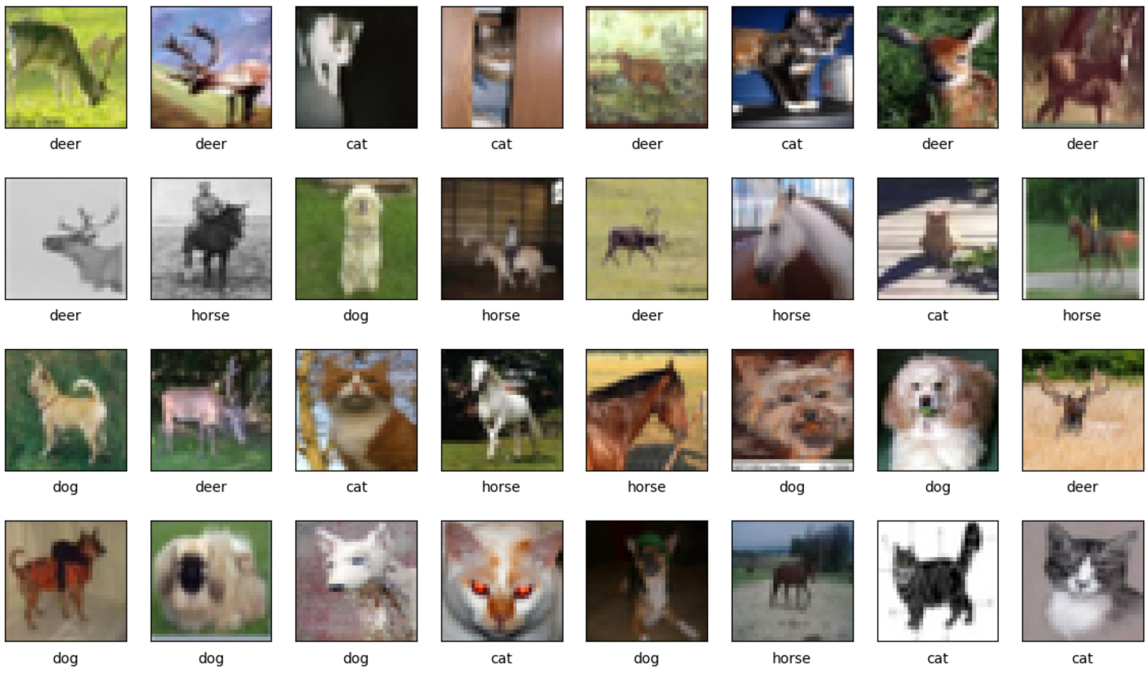


Figure 5.1:
Random sample of the used subset of Cifar-10

To study the influence of noise a common gaussian noise was used.

The density function of this is defined by:

$$p(z) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(z-\mu)^2}{2\sigma^2}}$$

, where μ is the mean gray value and σ is the standard deviation and thus a measure of the strength of the noise. For a color image, the 3 color channels are individually processed with the same standard deviation. The resultant noisy images look like this:

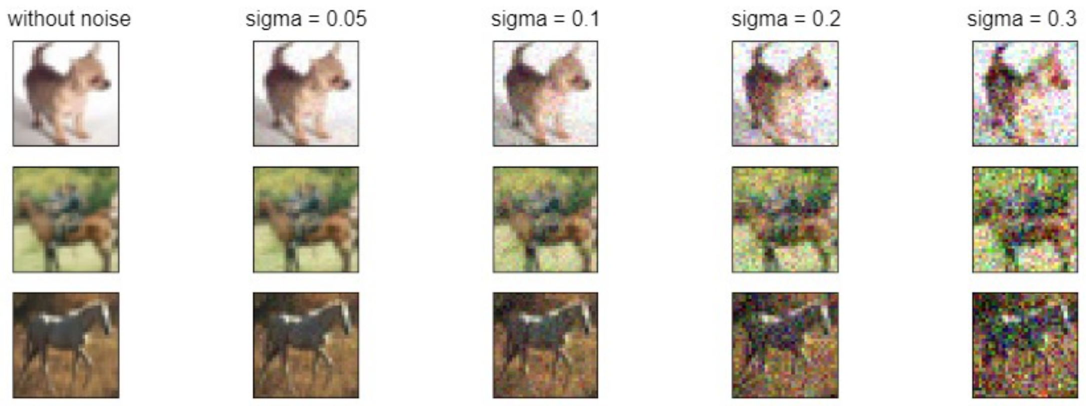


Figure 5.2:
Examples for the different levels of gaussian noise

The same architecture was trained on the Cifar-10 subset without noise and with noise levels of strengths sigma = 0.05, 0.1, 0.2 and 0.3. The ability to deal with a slight noise without a significant loss of contextual understanding is given:

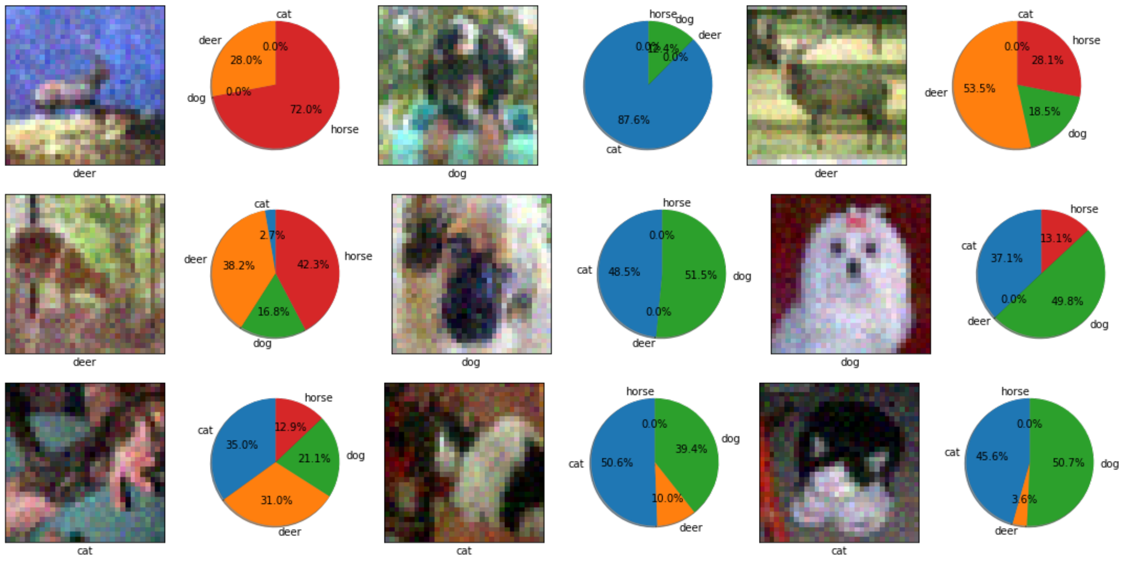


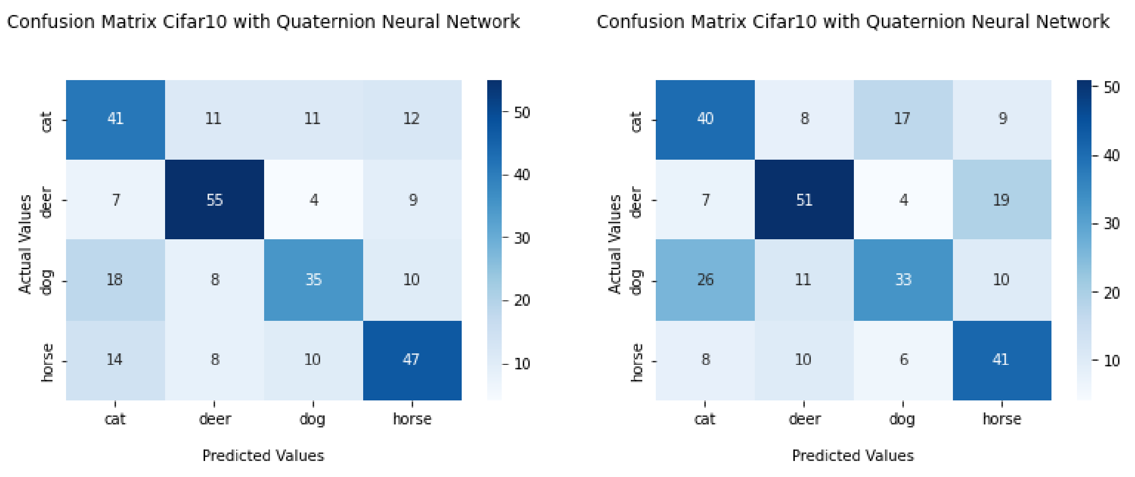
Figure 5.3: Examples of correct and incorrect classification of images with sigma = 0.1 gaussian noise after training

The results in final accuracy and loss dependent on the noise level are shown in the following table:

	$\sigma = 0$	$\sigma = 0.05$	$\sigma = 0.1$	$\sigma = 0.2$	$\sigma = 0.3$
final accuracy	59.3%		55.0%		
final loss	1.12		1.18		

Table 5.1: Results in training a network with different levels of noise

die Tabelle wird selbstverständlich noch gefüllt, das entsprechende Programm läuft ;) One can clearly observe a decrease in accuracy and an increase in loss with higher noise levels. However, an interesting development can also be observed in the confusion matrices. As expected, the confusion between similar classes (cat-dog, horse-deer) is increased. This amplification seems to increase even more with higher noise. This effect could come from the fact that with stronger noise, the value of color learning shifts in comparison to that of structure learning. This hypothesis should be tested as a basis for further experiments. hier kommen am Ende confusion matrices aller noise levels hin, um zu zeigen, dass mit zunehmendem Rauschen die Verwechslung zwischen den ähnlichen Klassen Cat-Dog und Horse-Deer zunimmt.



(a) without noise (b) sigma = 0.1

Figure 5.4: The confusion between similar classes increasewith higher noise

wenn alle confusion matrices da sind werden sie noch zugeschnitten und die Kästchen Cat-Dog, Horse-Deer Verwechslung werden grün umrandet, um zu markieren worauf ich hinaus will

5.2 The role of color in learning quaternion networks

5.2.1 Universality for the use of black and white images

One question that naturally arises is whether such a network can nevertheless learn contexts that are not colored. The interpretation of the rotation makes little sense for images that are on the gray scale. However, the weighting by scales does, this is comparable to real networks. When entering a grayscale image for the rgb channel equally, the context should be trainable with the quaternion network, even if no color information is provided. To verify this experimentally, the MNIST dataset, consisting of black and white digits, was trained in such a quaternion convolutional neural network [18]. The architecture for all subsequent experiments consists of a convolutional layer with 8 filters of size 3x3, a maxpooling of size 3x3, two quaternion layers with 256 and 128 neurons, and the real 10-class output layer. This architecture is small, straightforward and follows simple basic structures. It is well suited for observing effects of different data sets. The following figures (5.5) display the training process in consideration of the current error and the current accuracy. The latter is distinguished in the accuracy of the training batch and the independent validation data set, which was never used for training. Furthermore, in addition to the proportion of correct predictions, the proportion of correct tips among the three most likely tips per image is given, which is also a measure of a well-learned classification.

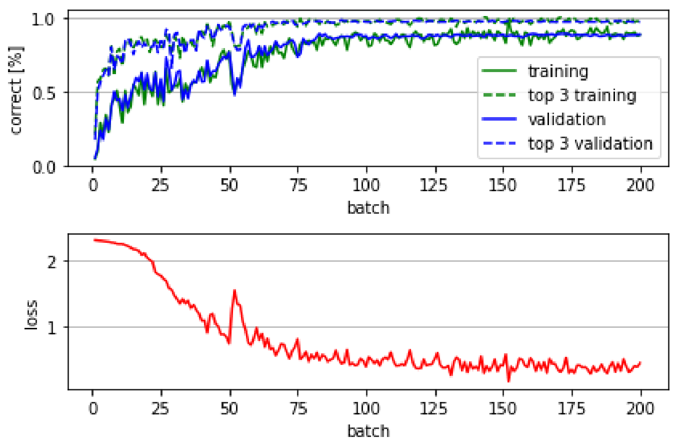


Figure 5.5: Evolution of loss and accuracy in the course of the learning MNIST data

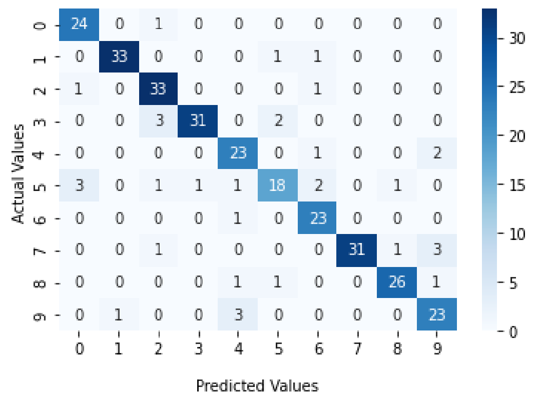


Figure 5.6: Final confusion matrix for 300 validation images of MNIST data

The following figure (5.7) shows in detail some predictions of randomly selected images. The resulting predictions show a good understanding of the digits. For example, for the digit 7, a 2 is often predicted as the second tip, since this is the closest to the shape. The situation is similar with 9 and 4. This explains why it makes sense to also look at the accuracy among the top three predictions.

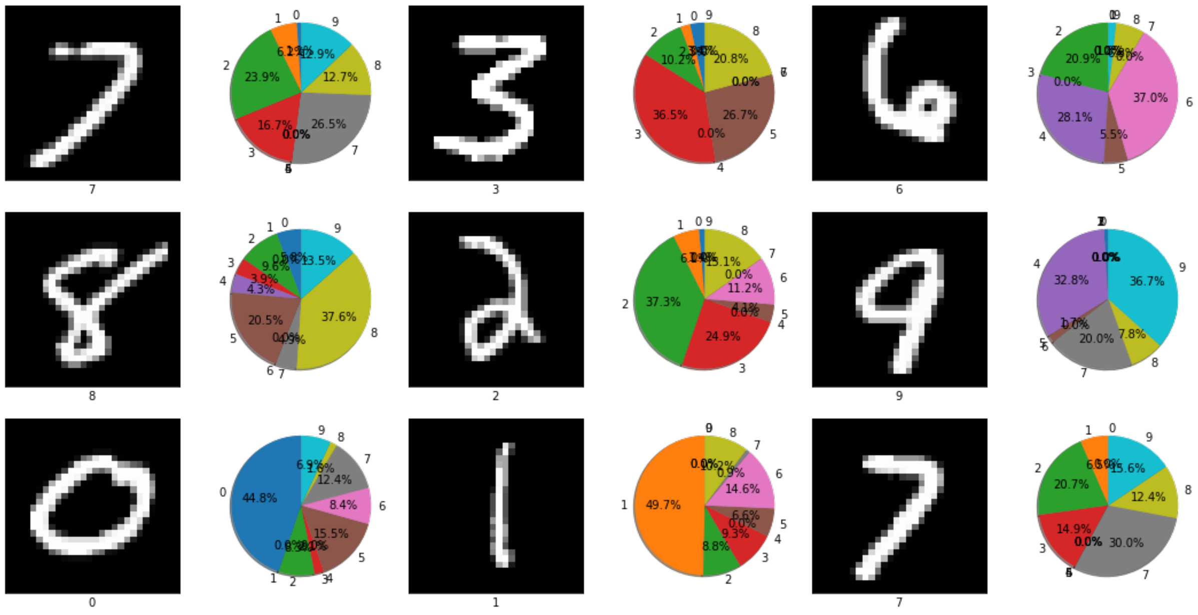


Figure 5.7:
Random sample of predictions of validation data on the trained network on MNISTdata

5.2.2 Proof of color learning

The fact that color information is nevertheless learned decisively can be shown with a simple experiment. For this purpose, the different digits instead of black and white are colored in an unique color combination. This should lead to 100% recognition in only a few batches. The corresponding results are displayed in figure (5.9).

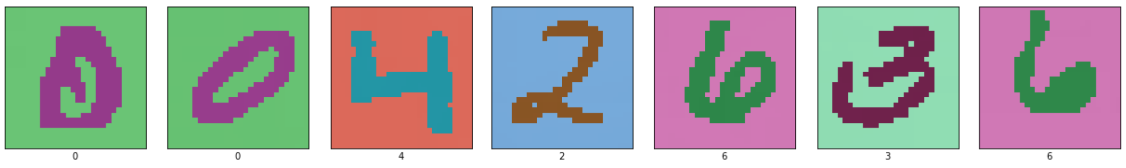
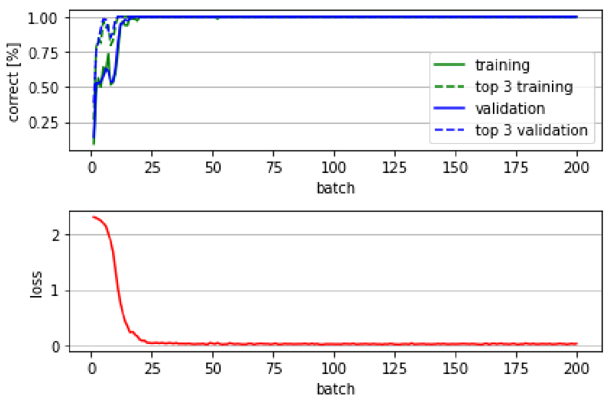
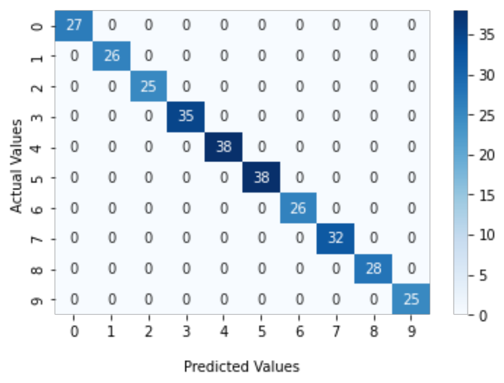


Figure 5.8: Examples of the colored digits

The network is able to indentify the colors and learn the context fast with almost no fluctuation in the loss.



(a) Learning process of colored digits

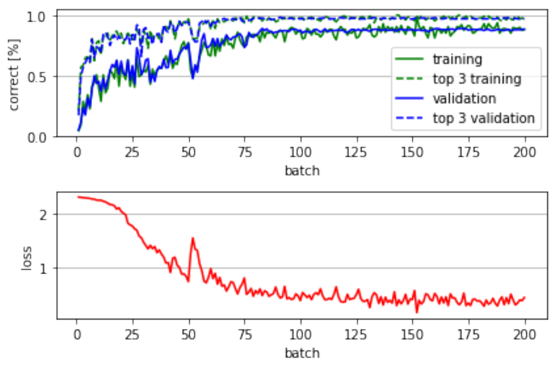


(b) Confusion matrix for colored digits, no mispredictions

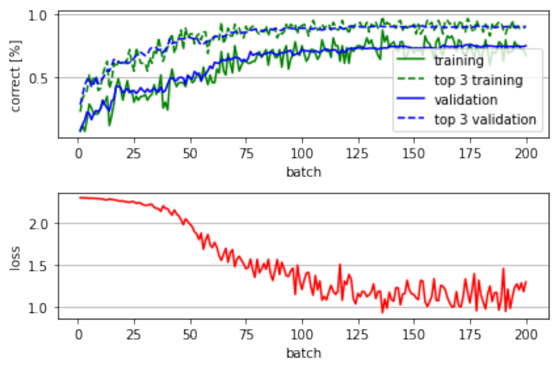
Figure 5.9: Assigning colors to the different MNIST-digits makes the learning very fast and clear

5.2.3 Resistance against false color information

To complete the series of these experiments, a similar model was trained on the MNIST data with random colors that have no relation to the digits. Unlike the black and white images, color information is present, but it has no correlation with the image classes. In figure (5.10), the learning process on these data is compared to that on the black and white MNIST images.



(a) Learning on black and white images



(b) Learning on images with random color

Figure 5.10: Comparing the learning on images with no color information to learning on images with random color

The main differences between the learning processes are, that learning on random colored images has a slower start and ends in a slightly less prediction accuracy. Also, the loss varies more with random colors because more changes need to be implemented for each batch. Nevertheless, the digit context is learned well even with randomized colors. The network can represent the insignificance of the color to some extent.

In order to experimentally exploit this resistance to false information, training has finally been carried out with significant amounts of disruptive information. For this the idea from [35] is used, taking a random crop of a different image, coloring it and using it as background for the MNIST digits.

This results in images that have disruptive information in color and texture.

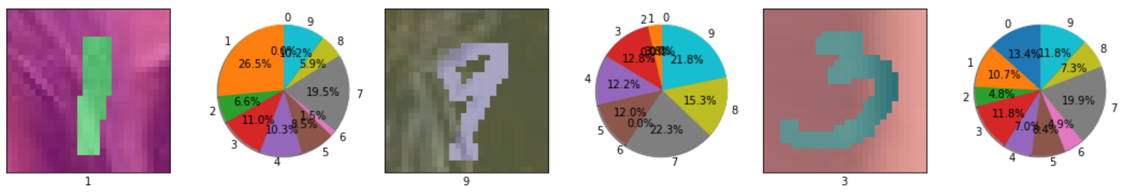
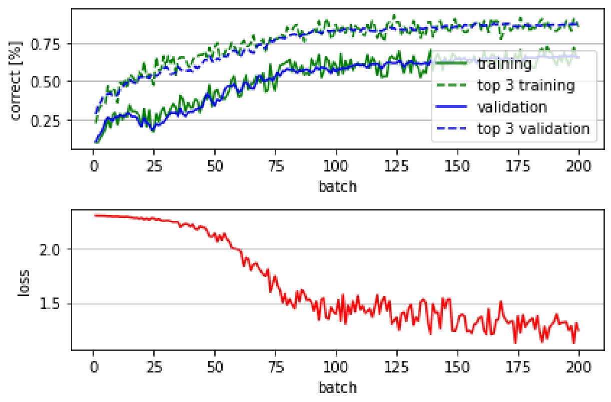
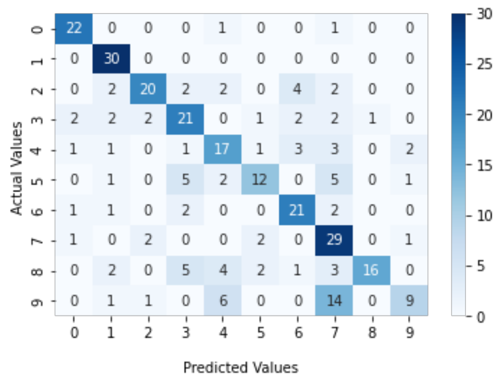


Figure 5.11: Examples of the predictions on disrupted digits



(a) Learning process of disturbed digits



(b) Confusion matrix for the disturbed images

Figure 5.12: Learning process of the MNIST digits with disruptive information in color and texture

The effects that were already observed with only interfering color information continue to a greater extent. The learning process has an even slower start and ends in a lower prediction accuracy. The loss varies even more with this disruptive information because more changes are to be implemented for each batch.

To conclude this section, it can be stated that both structure information and color information are learned by the built quaternion network.
A disturbance in color or structural information does not lead to unlearnability; to a certain extent, the irrelevance of such information is learned. The more the image information is disturbed, the slower the learning process runs and ends in lower accuracy.
The observed effects are consistent with expectations for the quaternion network and justify using quaternion neural networks on more complicated data sets and more with complicated network architecture.

6 Summary and Conclusion

The quaternion network with the structural elements quaternion convolution quaternion pooling and quaternion fully connected layers as proposed is able to approximate a context by backpropagation. It was also established that both structural and color information are learned and included in the classification of color images by the network.
In addition the learning of the quaternion network was still accomplished when false textural or color information was applied to the data set. The learning process becomes slower with increasing false information and ends in lower accuracy and the learning process fluctuates more. Nevertheless, even with significant interference, a rough context is still learned.
The stability against gaussian noise was also tested. The result is similar to that for color and structure noise. A context is still approximated with a slightly inferior learning process and result. Furthermore, it was found that with stronger noise the confusion between similar classes increases.

The experiments have clear limitations in that they were only applied to a few data sets. The implementation of this work serves only as a proof of concept and is not optimized for performance. For more complicated arrangements, a large speed up must be applied.

All these results provide a basis for experiments with quaternion networks of different architectures as found in real valued or complex valued experimental setups. Possible combinations of real and quaternionic networks, deeper architectures or even advanced approaches like ResNet should be tested [36]. Other image datasets and potentially even other datasets that can be represented three-dimensionally as quaternions are to be probed.

Bibliography

[1] Shrutika Singh, Harshita Arya, and P. Arun Kumar. Voice assistant for ubuntu implementation using deep neural network. In Hari Vasudevan, Antonis Michalas, Narendra Shekokar, and Meera Narvekar, editors, *Advanced Computing Technologies and Applications*, Algorithms for Intelligent Systems, pages 11–20. Springer Singapore, Singapore, 2020.

[2] Ali Bou Nassif, Ismail Shahin, Imtinan Attili, Mohammad Azzeh, and Khaled Shaalan. Speech recognition using deep neural networks: A systematic review. *IEEE Access*, 7:19143–19165, 2019.

[3] Laura Maria Badea. Predicting consumer behavior with artificial neural networks. *Procedia Economics and Finance*, 15:238–246, 2014.

[4] Max N. Greene, Peter H. Morgan, and Gordon R. Foxall. Neural networks and consumer behavior: Neural models, logistic regression, and the behavioral perspective model. *The Behavior analyst*, 40(2):393–418, 2017.

[5] Shivappriya S N, Navaneethakrishnan R, Raj K. Sangeetha, Abirami M., and Chidhambaram S. Neural network based heart disease prediction. In *2021 International Conference on Advancements in Electrical, Electronics, Communication, Computing and Automation (ICAECA)*, pages 1–6, 2021.

[6] Hongquan Peng, Haibin Zhu, Chi Wa Ao Ieong, Tao Tao, Tsung Yang Tsai, and Zhi Liu. A two-stage neural network prediction of chronic kidney disease. *IET systems biology*, 15(5):163–171, 2021.

[7] Yu-Dong Zhang, Chichun Pan, Junding Sun, and Chaosheng Tang. Multiple sclerosis identification by convolutional neural network with dropout and parametric relu. *Journal of Computational Science*, 28:1–10, 2018.

[8] Jaakko Lehtinen, Jacob Munkberg, Jon Hasselgren, Samuli Laine, Tero Karras, Miika Aittala, and Timo Aila. Noise2noise: Learning image restoration without clean data. 2018.

[9] Peiran Ren, Yue Dong, Stephen Lin, Xin Tong, and Baining Guo. Image based relighting using neural networks. *ACM Transactions on Graphics*, 34:111:1–111:12, 2015.

[10] Shaoqiang Wang, Gewen He, Shudong Wang, Song Zhang, and Fangfang Fan. Research on recognition of medical image detection based on neural network. *IEEE Access*, 8:94947–94955, 2020.

[11] *2017 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE, 19.10.2017 - 21.10.2017.

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

[13] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991.

[14] Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. The expressive power of neural networks: A view from the width. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc, 2017.

[15] Y. Bengio and Yann Lecun. Convolutional networks for images, speech, and time-series. 1997.

[16] Matlab version 9.10.0.1613233 (r2021a), 2021.

[17] Florentin Bieder, Robin Sandkühler, and Philippe C. Cattin. Comparison of methods generalizing max- and average-pooling, 2021.

[18] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[19] Sumit Saha. A comprehensive guide to convolutional neural networks: the eli5 way. *towards data science*, 2018.

[20] Akira Hirose and Shotaro Yoshida. Generalization characteristics of complex-valued feedforward neural networks in relation to signal coherence. *IEEE Transactions on Neural Networks and Learning Systems*, 23:541–551, 2012.

[21] Joshua Bassey, Lijun Qian, and Xianfang Li. A survey of complex-valued neural networks, 2021.

[22] Hyeong-Seok Choi, Jang-Hyun Kim, Jaesung Huh, Adrian Kim, Jung-Woo Ha, and Kyogu Lee. Phase-aware speech enhancement with deep complex u-net, 2019.

[23] Jesper Sören Dramsch, Mikael Lüthje, and Anders Nymark Christensen. Complex-valued neural networks for machine learning on non-stationary physical data. *Computers & Geosciences*, 146:104643, 2021.

[24] Mario Lopez-Pacheco and Wen Yu. Complex valued deep neural networks for nonlinear system modeling. *Neural processing letters*, pages 1–22, 2021.

[25] Alex Krizhevsky. Learning multiple layers of features from tiny images. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 20.06.2009 - 25.06.2009.

[26] Kunihiko Fukushima. Visual feature extraction by a multilayered network of analog threshold elements. *IEEE Transactions on Systems Science and Cybernetics*, 5(4):322–333, 1969.

[27] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for activation functions, 2017.

[28] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification, 2015.

[29] Xuanyu Zhu, Yi Xu, Hongteng Xu, and Changjian Chen. Quaternion convolutional neural networks, 2019.

[30] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc, 2019.

[31] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 2010. PMLR.

[32] Alex Krizhevsky. Cifar-10: Learning multiple layers of features from tiny images, 2010.

[33] paperswithcode from MetaAI. Image classification on cifar-10 leaderboard.

[34] Hugo Touvron, Matthieu Cord, Alexandre Sablayrolles, Gabriel Synnaeve, and Hervé Jégou. Going deeper with image transformers, 2021.

[35] Wouter Bulten. Colorful mnist: Getting started with gans part 2: Colorful mnist, 1998.

[36] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. 2015.

[37] *2009 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 20.06.2009 - 25.06.2009.

[38] *12th 5USENIX6 Symposium on Operating Systems Design and Implementation (5OSDI6 16)*, 2016.

[39] *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc, 2019.

[40] *2021 International Conference on Advancements in Electrical, Electronics, Communication, Computing and Automation (ICAECA)*, 2021.

[41] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow:

A system for large-scale machine learning. In *12th 5USENIX6 Symposium on Operating Systems Design and Implementation (5OSDI6 16)*, pages 265–283, 2016.

[42] Cen Chen, Kenli Li, Mingxing Duan, Keqin L, editor. *Big Data Analytics for Sensor-Network Collected Intelligence: Chapter 6 Extreme Learning Machine and its applications in big data processing.* Elsevier, 2017.

[43] Cen Chen, Kenli Li, Mingxing Duan, and Keqin Li. Extreme learning machine and its applications in big data processing. In Cen Chen, Kenli Li, Mingxing Duan, Keqin L, editor, *Big Data Analytics for Sensor-Network Collected Intelligence: Chapter 6 Extreme Learning Machine and its applications in big data processing*, pages 117–150. Elsevier, 2017.

[44] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255. IEEE, 20.06.2009 - 25.06.2009.

[45] Chase Gaudet and Anthony Maida. Deep quaternion networks, 2017.

[46] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with numpy. *Nature*, 585:357–362, 2020.

[47] Sepp Hochreiter. Untersuchungen zu dynamischen neuronalen netzen. 1991.

[48] John D. Hunter. Matplotlib: A 2d graphics environment. *Computing in science & engineering*, 9(3):90–95, 2007.

[49] I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors. *Advances in Neural Information Processing Systems.* Curran Associates, Inc, 2017.

[50] Jesús Utrera. Deep learning using python + keras (chapter 3): Resnet, 2018.

[51] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

[52] Michael Waskom, Olga Botvinnik, Drew O'Kane, Paul Hobson, Saulius Lukauskas, David C Gemperline, Tom Augspurger, Yaroslav Halchenko, John B. Cole, Jordi Warmenhoven, Julian de Ruiter, Cameron Pye, Stephan Hoyer, Jake Vanderplas, Santi Villalba, Gero Kunter, Eric Quintero, Pete Bachant, Marcel Martin, Kyle Meyer, Alistair Miles, Yoav Ram, Tal Yarkoni, Mike Lee Williams, Constantine Evans, Clark Fitzgerald, Brian, Chris Fonnesbeck, Antony Lee, and Adel Qalieh. mwaskom/seaborn: v0.8.1 (september 2017), 2017.

[53] Toshifumi Minemoto, Teijiro Isokawa, Haruhiko Nishimura, and Nobuyuki Matsui. Feed forward neural network with random quaternionic neurons. *Signal Processing*, 136:59–68, 2017.

[54] Hung Nguyen, Sarah J. Maclagan, Tu Dinh Nguyen, Thin Nguyen, Paul Flemons, Kylie Andrews, Euan G. Ritchie, and Dinh Phung. Animal recognition and identification with deep convolutional neural networks for automated wildlife monitoring. In *2017 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pages 40–49. IEEE, 19.10.2017 - 21.10.2017.

[55] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[56] Siddharth Das. Cnn architectures: Lenet, alexnet, vgg, googlenet, resnet and morełdots. *Analytics Vidhya*, 2017.

[57] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions, 2014.

[58] Yee Whye Teh and Mike Titterington, editors. *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, Proceedings of Machine Learning Research, Chia Laguna Resort, Sardinia, Italy, 2010. PMLR.

[59] P. Umesh. Image processing in python. *CSI Communications*, 23, 2012.

[60] Guido van Rossum and Fred L. Drake. *Python 3 Reference Manual.* CreateSpace, Scotts Valley, CA, 2009.

[61] Hari Vasudevan, Antonis Michalas, Narendra Shekokar, and Meera Narvekar, editors. *Advanced Computing Technologies and Applications.* Algorithms for Intelligent Systems. Springer Singapore, Singapore, 2020.