University of Pisa

Artificial Intelligence and Data Engineering
Symbolic and Evolutionary Artificial Intelligence

Exploring TensorFlow's Frontier: Journeying Through Clifford Algebra's Realm

(2024_SEAI_project_C08)

Ricky Marinsalda 585094
Francesco Londretti 597086

June 19, 2024

Contents

Chapter 1

Clifford Algebra: a Theoretical Background

1.1 What is Clifford Algebra

Clifford Algebra is a type of algebra that extends the concept of real numbers, complex numbers, and quaternions to higher dimensions. It's named after the English mathematician William Kingdon Clifford. The fundamental idea of Clifford Algebra is to associate algebraic elements with geometric entities and operations.

In mathematical terms, a Clifford Algebra is generated by a vector space equipped with a quadratic form. This means that for a given vector space V over a field K, and a quadratic form $Q : V \to K$, the Clifford Algebra $Cl(V, Q)$ is the "freest" unital associative algebra generated by V subject to the condition:

$$v^2 = Q(v) \cdot 1 \tag{1.1}$$

Here, v is an element of V, v^2 is the inner(?) product of v with itself in the algebra, and 1 is the multiplicative identity of the algebra.

The algebraic structure of Clifford Algebras allows them to have applications in various fields, including geometry, theoretical physics, and digital image processing. They are particularly useful in the study of quadratic forms and orthogonal transformations.

Clifford Algebra can be seen as a generalization of several hypercomplex number systems and have deep connections with the theory of quadratic forms and orthogonal transformations. They are also used in proofs of the Atiyah-Singer index theorem, to provide double covers (spin groups) of the classical groups, and to generalize the Hilbert transform. [9]

1.1.1 Real numbers

In the context of Clifford algebra, particularly those defined over real and complex vector spaces with nondegenerate quadratic forms, the signature (p, q) plays a crucial role. The signature represents the characteristics of the quadratic form associated with the vector space on which the Clifford algebra is constructed.

The Clifford algebra $Cl_{p,q}(\mathbb{R})$ and $Cl_n(\mathbb{C})$ are structured around these quadratic forms and are isomorphic to either a full matrix ring A or a direct sum $A \oplus A$, where A consists of matrices with entries from $\mathbb{R}$, $\mathbb{C}$, or $\mathbb{H}$ (the real numbers, complex numbers, or quaternions, respectively) [9].

For a real vector space equipped with a nondegenerate quadratic form, this form can be expressed in the standard diagonal form:

$$Q(v) = v_1^2 + \cdots + v_p^2 - v_{p+1}^2 - \cdots - v_{p+q}^2,$$

where $n = p + q$ is the dimension of the vector space, and the integers p and q denote the number of positive and negative squares in the quadratic form, respectively. This is what is referred to as the signature (p, q) of the quadratic form.

In addition to p and q, we introduce the parameter r to account for the number of basis vectors that square to zero. These basis vectors are orthogonal to all other vectors, including themselves, and thus their square is zero. They are essential in defining degenerate quadratic forms and play a role in the structure of the corresponding Clifford algebra.

A standard basis for the vector space $R_{p,q,r}$ consists of $n = p + q + r$ mutually orthogonal vectors. Of these, p vectors square to $+1$, q vectors square to -1, and r vectors square to 0. The Clifford algebra $Cl_{p,q,r}(\mathbb{R})$ is then generated by these basis vectors, with p vectors squaring to $+1$, q vectors squaring to -1, and r vectors squaring to 0.

1.2 Multi Vectors

Multivectors are a fundamental concept in Clifford algebra, an extension of linear algebra that deals with vectors, planes, volumes, and other geometric entities in an n-dimensional space. To better understand multivectors, it is useful to start with vectors and build up from there.

Vectors: A vector is an element of a vector space. For example, in three-dimensional space $\mathbb{R}^3$, a vector can be represented as $\mathbf{v} = v_1\mathbf{e}_1 + v_2\mathbf{e}_2 + v_3\mathbf{e}_3$, where $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3$ are the basis vectors.

Bivectors: A bivector is an element generated by the wedge product of two vectors. For instance, if $\mathbf{u}$ and $\mathbf{v}$ are vectors, then $\mathbf{u} \wedge \mathbf{v}$ is a bivector. Bivectors represent oriented areas in a space. In $\mathbb{R}^3$, a bivector can be represented as a linear combination of wedge products of the basis vectors, e.g., $\mathbf{e}_1 \wedge \mathbf{e}_2, \mathbf{e}_2 \wedge \mathbf{e}_3, \mathbf{e}_3 \wedge \mathbf{e}_1$.

Trivectors: A trivector is generated by the wedge product of three vectors and represents an oriented volume in a space. In $\mathbb{R}^3$, a trivector can be written as a linear combination of triple wedge products of the basis vectors, e.g., $\mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_3$.

Multivectors A multivector is a linear combination of scalars, vectors, bivectors, trivectors, and so on. Generally, in an n-dimensional space, a multivector can be written as:

$$M = \alpha + \mathbf{v} + \mathbf{B} + \mathbf{T} + \dots$$

where:

- α is a scalar (a 0-vector),
- $\mathbf{v}$ is a vector (a 1-vector),
- $\mathbf{B}$ is a bivector (a 2-vector),
- $\mathbf{T}$ is a trivector (a 3-vector),
- and so on up to n-vectors.

In summary, a multivector is an element of Clifford algebra that can combine vectors, bivectors, trivectors, and other geometric objects into a unified structure, thus facilitating the description of complex geometries and geometric operations.

1.3 Pseudovectors and Pseudoscalars

In the previous section we have seen that one of the fundamental aspects is the use of multivectors, which are combinations of vectors of various dimensions. Pseudoscalars and pseudovectors are specific types of multivectors.

1.3.1 Pseudoscalars

The term *pseudoscalar* refers to an element of Clifford algebra that has the highest possible dimension in a given n-dimensional space. This means that a pseudoscalar in an n-dimensional space is the outer product of n linearly independent vectors. The main characteristic of a pseudoscalar is that it transforms like a scalar under rotations but may change sign under orientation-reversing transformations (reflections).

In the image 1.1, the pseudoscalar is represented as the element of dimension $\binom{n}{n} = 1$, located at the top vertex of the diamond-shaped structure. For example:

- In 1D, the pseudoscalar is simply e_1.
- In 2D, the pseudoscalar is e_1e_2, which can also be written as $-e_2e_1$ due to the anticommutative properties of the outer product.
- In 3D, the pseudoscalar is $e_1e_2e_3$.
- In 4D, the pseudoscalar is $e_1e_2e_3e_4$.

1.3.2 Pseudovectors

Pseudovectors, on the other hand, are elements of Clifford algebra of dimension $n - 1$. These are also called $(n - 1)$-vectors and are located just below the pseudoscalars in the diamond structure. Like pseudoscalars, pseudovectors transform in a particular way under orientation-reversing transformations. Instead of remaining invariant or changing sign like normal vectors, pseudovectors can transform in a way that combines both properties.

For example, in 3D, a pseudovector could be a combination such as e_1e_2, e_2e_3, or e_3e_1. These represent oriented planes and areas and are related to cross products and rotors in physics.

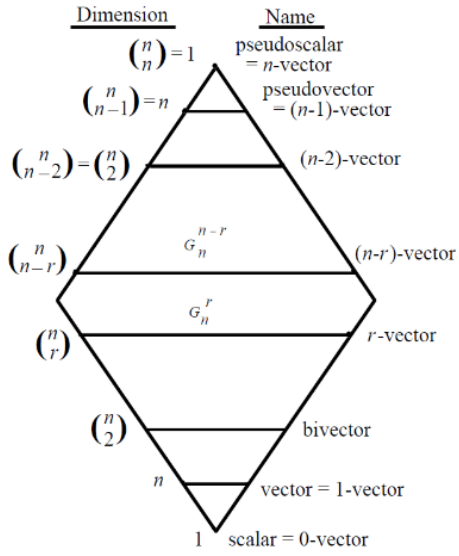


Figure 1.1: Diamond's structure of various types of multivectors in an n-dimensional Clifford algebra [3]

The image 1.1 from [3] shows the structure of various types of multivectors in an n-dimensional Clifford algebra, using binomial coefficients to determine the number of bases. This structure follows the rows of Tartaglia's triangle and shows the symmetry between the various grades of multivectors.

- **Scalars (0-vectors):** Elements of dimension 0.

- **Vectors (1-vectors):** Elements of dimension 1.
- **Bivectors (2-vectors):** Elements of dimension 2, and so on.
- r**-vectors:** Elements of dimension r.
- **Pseudovectors:** Elements of dimension $n-1$.
- **Pseudoscalars:** Elements of dimension n.

Blades Building blocks of multi vectors. Each blade has a grade as well as a dimension.

- Grade 0 is a scalar number (which has dimension 1) ,
- Grade 1 is a vector (whose dimension determines the number and dimension of the remaining grades) ,
- Grade 2 is a bivector (whose dimension depends on the vector) ,
- And so on for tri-vectors...

If we take, for example, $Cl_{2,0}$ we have that the multi vector $1+3*e_1+6*e_2+2*e_{12}$ is divided in:

- Grade 0, the scalar 1 (and has dimension 1)
- Grade 1, the vectors $3*e_1+6*e_2$ (and their dimension is 2)
- Grade 2, the bivector $2*e_{12}$ (and has dimension 1)

When blades of different grade are added the result cannot be simplified further and the multivector is just the sum of these with the '+' sign left in. This is the same principle as the notation for complex numbers or quaternions which might be denoted by say the number 2+i3 which cannot be further simplified [1].

Grades This representation helps us visualize how elements of different dimensions relate to each other in Clifford algebra and provides a better understanding of the properties of pseudoscalars and pseudovectors.

We have seen that in both 2D and 3D cases, the pseudoscalar squared is -1. However, this is not a general rule for all algebras with uniform signature. The sequence of signs for the pseudoscalar squared (I^2) depending on the dimension is as follows:

Dimension	1	2	3	4	5	6	7	8
I^2 Sign	+	-	-	+	+	-	-	+

In 2D, the pseudoscalar anticommutes with vectors, which is why it is not identified with the imaginary unit i. In 3D, the pseudoscalar not only commutes with vectors but with all elements. In short, the commutation property of the pseudoscalar depends on both the size of the space and the grade of the element, as shown in the following table:

Spaces / Multivectors	Even Grade	Odd Grade
Even Dimension	Commute	Anticommute
Odd Dimension	Commute	Commute

The name "pseudoscalar" derives from the fact that it behaves like a scalar with respect to rotations but undergoes parity transformation (inversion of all spatial axes). In 2D and 3D it is transformed into its negative.

In the context of $Cl_{3,0}$:

$$I^2 = -1$$

$$I^\dagger = -I$$

$$IM = MI \quad \text{for every multivector in } Cl_{3,0}$$

$$I^{-1} = \frac{I^\dagger}{|I^2|} = -I$$

where † denotes the reverse of x [1.4.1]. In general, in an n-dimensional space, if $I^{-1} = I^\dagger$, then

$$I^{-1} = (-1)^{\frac{n(n-1)}{2}} I$$

As we have seen, for 2D and 3D, $I^{-1} = -I$ and then the sequence of signs for higher dimensions alternates in pairs: $+, +, -, -, +, +, \ldots$.

1.3.3 Lipschitz groups, Pins and Spinors

Introduction to the Lipschitz Group

The Lipschitz group, denoted as Γ, is defined as the set of invertible elements x within the algebra that stabilize the vector space V under Clifford conjugation. This action is expressed mathematically as:

$$\alpha(x)vx^{-1} \in V \tag{1.2}$$

for all vectors v in V. The significance of this property lies in its ability to preserve the geometric relations encoded by the Clifford algebra, thereby enabling the representation of rotations and reflections in space.

The Pin Groups and Spinor Norm

Pin groups are subgroups of the Lipschitz group that are closely associated with the orthogonal groups of the vector space. They are defined as the elements of the Lipschitz group whose spinor norm is equal to 1.

The spinor norm is a function that maps elements of the Clifford algebra to the underlying field, typically the real numbers, and is given by the product of an element with its reverse. For an element x in the Pin group, the condition is:

$$N(x) = xx^\dagger = 1 \tag{1.3}$$

where $x^\dagger$ denotes the reverse of x (see section [1.4.1]).

The Pin group serves as a double cover of the orthogonal group, meaning each orthogonal transformation corresponds to two elements in the Pin group, reflecting the group's ability to represent both rotations and reflections.

Spinors and their role

Spinors are elements of a spinor space, which can be constructed as a minimal left ideal of a Clifford algebra. Spinors are characterized by their transformation properties under the action of the Spin group, which is a subgroup of the Pin group consisting of elements with a spinor norm of 1 and only composed of elements of even grade.

Let

$$Cl^{[0]}_{p,q} = \{x \in Cl_{p,q} | \alpha(x) = x\} \tag{1.4}$$

(These are precisely the elements of even degree in $Cl_{p,q}$). Then the spin group lies within $Cl^{[0]}_{p,q}$

Spinors are essential in representing quantum states, particularly the intrinsic angular momentum or 'spin' of elementary particles.

1.4 Important operations in Clifford Algebra

1.4.1 Automorphisms and Antiautomorphisms

In Clifford algebra, the concept of automorphism and antiautomorphism are pivotal in understanding the transformations of multivectors.

Grade Automorphism (Grade Involution)

The grade automorphism, also known as grade involution, is an operation that alters the sign of each element in a multivector according to its grade. The operation is defined by the following formula:

$$\alpha(x) = (-1)^i x$$

where i represents the grade of the element x. This operation is linear and involutory, meaning that applying it twice will return the original element.

Antiautomorphisms

Clifford algebra introduces two significant antiautomorphisms: reversion and conjugation.

Reversion: The reversion operation inverts the order of the basis elements in a multivector. Due to the anticommutative property of the Clifford product, this inversion changes the sign of an element based on its grade. If an element is a product of an odd number of vectors, its sign will be reversed. The reversion of a multivector A, denoted $A^\dagger$, is particularly important when dealing with the geometric product of vectors.

Conjugation: Conjugation is the last antiautomorphism and is essentially the combination of grade automorphism and reversion. It applies the grade involution followed by reversion, effectively changing the sign of elements based on their grade twice and then inverting the order of the basis elements. The conjugate of a multivector A, denoted $\bar{A}$, is given by:

$$\bar{A} = \alpha(A^\dagger)$$

Here's a table that summarizes the transformations of a multivector based on its grade:

Grade $k \mod 4$	0	1	2	3	General Rule
Grade Automorphism $\alpha(x)$	+	−	+	−	$(-1)^k$
Reversion $x^\dagger$	+	+	−	−	$(-1)^{k(k-1)/2}$
Conjugation $\bar{x}$	+	−	−	+	$(-1)^{k(k+1)/2}$

Table 1.1: Transformations of a multivector in Clifford algebra

Where:

- Grade Automorphism $\alpha(x)$ changes the sign of a multivector element based on its grade.
- Reversion $x^\dagger$ inverts the order of the basis elements and changes the sign based on the grade.
- Conjugation $\bar{x}$ combines grade automorphism and reversion, affecting the sign and order of basis elements.

1.4.2 Inner Product ($\cdot$, | , inner_prod())

The inner product, also known as the dot product, is a fundamental operation in geometric algebra. It measures the projection of one vector onto another and is central to understanding the geometric relationships between vectors.

Given two vectors $\mathbf{a}$ and $\mathbf{b}$ in a geometric algebra Cl_n, the inner product $\mathbf{a} \cdot \mathbf{b}$ is defined as:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i$$

where a_i and b_i are the components of vectors $\mathbf{a}$ and $\mathbf{b}$, respectively.

Properties

- **Symmetry:** The inner product is symmetric, meaning $\mathbf{a} \cdot \mathbf{b} = \mathbf{b} \cdot \mathbf{a}$.
- **Linearity:** The inner product is linear in each argument, $(\lambda \mathbf{a}) \cdot \mathbf{b} = \lambda(\mathbf{a} \cdot \mathbf{b})$ and $\mathbf{a} \cdot (\mathbf{b} + \mathbf{c}) = \mathbf{a} \cdot \mathbf{b} + \mathbf{a} \cdot \mathbf{c}$.
- **Orthogonality:** Vectors $\mathbf{a}$ and $\mathbf{b}$ are orthogonal if and only if $\mathbf{a} \cdot \mathbf{b} = 0$.

Examples

Projection The inner product can be used to find the projection of one vector onto another. Given vectors $\mathbf{a}$ and $\mathbf{b}$, the projection of $\mathbf{a}$ onto $\mathbf{b}$ is given by:

$$\text{proj}_{\mathbf{b}}(\mathbf{a}) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{b}\|^2} \mathbf{b}$$

where $\|\mathbf{b}\|$ represents the magnitude of vector $\mathbf{b}$.

Angle Between Vectors The cosine of the angle θ between two vectors $\mathbf{a}$ and $\mathbf{b}$ can be expressed using the inner product:

$$\cos(\theta) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \|\mathbf{b}\|}$$

This relationship is fundamental in trigonometry and geometric calculations.

1.4.3 Outer Product ($\wedge, ext\_prod()$)

The outer product, also known as the wedge product, is a fundamental operation in geometric algebra. It represents the antisymmetric combination of vectors and is crucial for defining oriented subspaces and multivectors.

Given two vectors $\mathbf{a}$ and $\mathbf{b}$ in a geometric algebra Cl_n, the outer product $\mathbf{a} \wedge \mathbf{b}$ is defined as:

$$\mathbf{a} \wedge \mathbf{b} = \frac{1}{2}(\mathbf{a} * \mathbf{b} - \mathbf{b} * \mathbf{a})$$

where $\mathbf{ab}$ represents the geometric product of $\mathbf{a}$ and $\mathbf{b}$ [2].

Properties

- **Antisymmetry:** The outer product is antisymmetric, meaning $\mathbf{a} \wedge \mathbf{b} = -\mathbf{b} \wedge \mathbf{a}$.
- **Bilinearity:** The outer product is bilinear, $(\lambda \mathbf{a}) \wedge \mathbf{b} = \lambda(\mathbf{a} \wedge \mathbf{b})$ and $\mathbf{a} \wedge (\mathbf{b} + \mathbf{c}) = \mathbf{a} \wedge \mathbf{b} + \mathbf{a} \wedge \mathbf{c}$.
- **Idempotent:** The outer product of a vector with itself is zero, $\mathbf{a} \wedge \mathbf{a} = 0$.

Examples

Volume of Parallelepiped In 3D space, given three vectors $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$ forming the edges of a parallelepiped, the magnitude of their outer product $\mathbf{a} \wedge \mathbf{b} \wedge \mathbf{c}$ represents the volume of the parallelepiped.

Orientation of Planes The outer product of two vectors $\mathbf{a}$ and $\mathbf{b}$ can represent a bivector that indicates the orientation of a plane defined by $\mathbf{a}$ and $\mathbf{b}$.

1.4.4 Geometric Product (*, geom_prod())

The geometric product is the central operation in geometric algebra, combining both the inner (dot) product and the outer (wedge) product. It provides a unified framework for understanding various geometric transformations and relationships.

Given two vectors $\mathbf{a}$ and $\mathbf{b}$ in a geometric algebra Cl_n, the geometric product $\mathbf{ab}$ is defined as:

$$\mathbf{a} * \mathbf{b} = \mathbf{a} \cdot \mathbf{b} + \mathbf{a} \wedge \mathbf{b}$$

where $\mathbf{a} \cdot \mathbf{b}$ is the inner product and $\mathbf{a} \wedge \mathbf{b}$ is the outer product.

Properties

- **Associativity:** The geometric product is associative, meaning $(\mathbf{ab})\mathbf{c} = \mathbf{a}(\mathbf{bc})$.
- **Distributivity:** The geometric product distributes over addition, $\mathbf{a}(\mathbf{b} + \mathbf{c}) = \mathbf{ab} + \mathbf{ac}$.
- **Inverses:** Each non-zero vector $\mathbf{a}$ has an inverse $\mathbf{a}^{-1}$ such that $\mathbf{a}\mathbf{a}^{-1} = 1$.

Examples

Squares of Vectors For any vector $\mathbf{a}$:

$$\mathbf{a}^2 = \mathbf{a} \cdot \mathbf{a} + \mathbf{a} \wedge \mathbf{a} = \mathbf{a} \cdot \mathbf{a}$$

since the wedge product of a vector with itself is zero. Therefore, $\mathbf{a}^2$ equals the scalar $\mathbf{a} \cdot \mathbf{a}$, which is the squared magnitude of $\mathbf{a}$.

Rotation in the Plane Consider two orthonormal vectors $\mathbf{e}_1$ and $\mathbf{e}_2$ in 2D. Their geometric product is:

$$\mathbf{e}_1\mathbf{e}_2 = \mathbf{e}_1 \cdot \mathbf{e}_2 + \mathbf{e}_1 \wedge \mathbf{e}_2 = 0 + \mathbf{e}_1 \wedge \mathbf{e}_2$$

The term $\mathbf{e}_1 \wedge \mathbf{e}_2$ represents a bivector, which can be interpreted as the oriented area spanned by $\mathbf{e}_1$ and $\mathbf{e}_2$.

Reflecting a Vector Reflecting a vector $\mathbf{a}$ through another vector $\mathbf{b}$ is given by:

$$\mathbf{a}' = -\mathbf{bab}^{-1}$$

This operation is useful in understanding reflections and rotations in higher dimensions.

The geometric product's ability to combine inner and outer products makes it a powerful tool for various applications in physics, computer graphics, and engineering.

1.4.5 Regressive Product (∨,&, reg_prod())

The regressive product, also known as the meet, is a fundamental operation in geometric algebra, particularly in projective geometry. An example of this application is provided in 2.5. It is the dual of the outer (or wedge) product and is used to find the intersection of subspaces.

In a geometric algebra Cl_n, given two blades A and B, their regressive product $A \vee B$ is defined through the dual relationship:

$$A \vee B = (A^* \wedge B^*)^*$$

where A^* and B^* denote the duals of A and B, respectively, and $\wedge$ is the outer product.

Examples

Intersection of Lines in 3D Space Consider two lines in 3D space represented by bivectors L_1 and L_2. The intersection point P of these lines can be found using the regressive product:

$$P = L_1 \vee L_2$$

If the lines are not parallel, P will represent the point where the lines meet.

Intersection of a Plane and a Line in 3D Space Let Π be a plane represented by a trivector and L a line represented by a bivector. The regressive product gives the line-plane intersection point:

$$P = \Pi \vee L$$

Intersection of Planes in 3D Space For two planes Π_1 and Π_2 in 3D space, represented by trivectors, their intersection line L is:

$$L = \Pi_1 \vee \Pi_2$$

The regressive product is essential for geometric computations involving intersections, which are prevalent in computer graphics, robotics, and physics.

1.4.6 Inverse of a Multi Vector

Computing the inverse of a multivector in Clifford algebra can be approached in several ways, depending on the properties of the multivector and the dimensionality of the space. Here's a brief explanation:

1. **Reversion Method**: For a given multivector A, if it can be written as a geometric product of vectors (i.e., it is factorizable), then its inverse A^{-1} can often be found using its reverse $A^\dagger$. The inverse is given by:

   $$A^{-1} = \frac{A^\dagger}{AA^\dagger}$$

   This method works well for objects like rotors and spinors, which are factorizable.

2. **Matrix Representation**: Another approach is to represent the multivector as a matrix and then compute the inverse of that matrix. This method is more general and can be used even when the reversion method fails. It involves expressing the multivector as a $2^n \times 2^n$ matrix, where n is the dimension of the space, and solving the linear equation $Ax = \mathbf{1}$ to find x, which is the desired inverse A^{-1}.

3. **Using the Dual**: In some cases, especially when dealing with bivectors in three dimensions, one can use the dual of the multivector to compute the inverse. However, this method is not generalizable to higher dimensions or different signatures of the space.

It's important to note that not all multivectors have inverses. For example, multivectors that include null vectors (vectors v such that $vv = 0$) are not invertible. Additionally, the existence of an inverse can depend on the signature of the space, which is defined by the inner product of the basis vectors.

Thanks to the work of **Dmitry Shirokov**, it has been introduced a way to calculate the inverse of a Multi Vector in arbitrary dimensions. For further details we suggest to read the paper *On computing the determinant, other characteristic polynomial coefficients, and inverse in Clifford algebras of arbitrary dimension* [8].

Chapter 2

TensorFlow Geometric Algebra

2.1 Setup

2.1.1 Installation

To install `tfga` [4], you can use pip:

```
pip install tfga
```

Requirements:

- Python 3
- tensorflow 2
- numpy

2.1.2 Libraries

Additionally, it's recommended to utilize Google Colab for running the Python code within a Jupyter Notebook. This provides a more interactive development experience. The example code is available on GitHub[1].

```
%load_ext autoreload
%autoreload 2

import tensorflow as tf

# Make tensorflow not take over the entire GPU memory
for gpu in tf.config.experimental.list_physical_devices('GPU'):
    tf.config.experimental.set_memory_growth(gpu, True)

import numpy as np

!pip install tfga
from tfga import GeometricAlgebra
```

Listing 2.1: Libraries

2.2 Generic examples

2.2.1 Geometric Algebra's objects

The first thing to do in order to use Clifford's Algebra is, of course, creating the right vector subspace. With a custom vector space it is possible to create various elements, such as complex numbers and quaternions.

[1] `https://github.com/RobinKa/tfga`

To create a vector subspace, the command `GeometricAlgebra()` initializes a geometric algebra object using the `tfga` library:

```
sta = GeometricAlgebra([1, -1, -1, -1])
```

This line of code creates a geometric algebra with a specific signature. The list `[1, -1, -1, -1]` represents the spacetime signature, where the first element corresponds to time and the subsequent elements correspond to spatial dimensions. In this case, it's a 3D spacetime with one time dimension and three spatial dimensions, all with a negative sign.

This translates in a vector subspace composed of e_0, e_1, e_2, e_3 with the following properties:

- $\mathbf{e_0^2} = 1$
- $\mathbf{e_1^2} = -1$
- $\mathbf{e_2^2} = -1$
- $\mathbf{e_3^2} = -1$

2.2.2 Simple operations

```
sta.print(sta.geom_prod( sta.e0,sta.e1))
a = sta.geom_prod(sta.e0, sta.from_scalar(4.0))
b = sta.geom_prod(sta.from_scalar(9.0), sta.e1)
sta.print(a)
sta.print(b)
sta.print(a, b)
sta.print(
    sta.e0,
    sta.e1,
    sta.e("0", "1"),
    sta.e01,
    sta.e10
)
```

The result of computation:

```
MultiVector[1.00*e_01]
MultiVector[4.00*e_0]
MultiVector[9.00*e_1]
MultiVector[4.00*e_0] MultiVector[9.00*e_1]
MultiVector[1.00*e_0] MultiVector[1.00*e_1] MultiVector[1.00*e_0 + 1.00*e_1] MultiVector[1.00*e_01]
     MultiVector[-1.00*e_01]
```

Let's explain the simple operations used.

1. `sta.print()`: this is the correct way to print an element made from this library. If not used, the MultiVector elements are transformed in Tensors.
2. `sta.geom_prod()`: represents the **Geometric Product**(defined in 1.4.4) when operands are **Tensors**(when the operands are multivectors, the $\star$ operator **must** be used) . The geometric product combines both inner and outer products and is a fundamental operation in geometric algebra. As we can see, it is possible to compute this product between two MultiVectors (line 1), as well as a MultiVector and a scalar (line 2/3). Another possible way to write this product is through the $\star$ operator [2].

Now considering $a = 1.0 * e_0$ and $b = 1.0 * e_1$, let's illustrate two other important operations: the **inner product** and the **outer product**.

```
    sta.print(sta.inner_prod(a, b))
    sta.print(sta.ext_prod(a, b))
```

Output:

[2]It is important to note that `sta.geom_prod()` requires two tensors as input, whereas the $\star$ symbol requires two multivector instances. Not knowing this can lead to errors. A similar consideration applies to all other symbols such as the inner and outer product

```
    MultiVector[]
    MultiVector[36.00*e_01]
```

As we can see these operations implement the inner and the outer product for **Tensors**. Another possible way to write this product is through the | operator for the inner product and the $\wedge$ operator for the outer product when the operands are two **Multivectors**.

```
# Create geometric algebra tf.Tensor instances
a = ga.e123
b = ga.e1

# Wrap them as ‘MultiVector‘ instances
mv_a = ga(a)
mv_b = ga(b)

# Reversion ((~mv_a).tensor equivalent to ga.reversion(a))
print(~mv_a)

# Geometric / inner / outer product
print(mv_a * mv_b)
print(mv_a | mv_b)
print(mv_a ^ mv_b)
```

2.2.3 Automorphism and Antiautomorphisms

Here's the **tfga** implementation for the Automorphism and Antiautomorphisms, helpful to understand the theoretical details provided in section 1.4.1

```
    m = tf.ones(16)
    sta.print("m:", m)
    sta.print("alpha(m):", sta.grade_automorphism(m))
    sta.print("~m:", sta.reversion(m)) # this is the dagger operator
    sta.print("alpha(~m):", sta.conjugation(m))
```

Output:

```
    m:  MultiVector[1.00*1 + 1.00*e_0 + 1.00*e_1 + 1.00*e_2 + 1.00*e_3 + 1.00*e_01 + 1.00*e_02 +
     1.00*e_03 + 1.00*e_12 + 1.00*e_13 + 1.00*e_23 + 1.00*e_012 + 1.00*e_013 + 1.00*e_023 + 1.00*
     e_123 + 1.00*e_0123]

    alpha(m):  MultiVector[1.00*1 + -1.00*e_0 + -1.00*e_1 + -1.00*e_2 + -1.00*e_3 + 1.00*e_01 +
     1.00*e_02 + 1.00*e_03 + 1.00*e_12 + 1.00*e_13 + 1.00*e_23 + -1.00*e_012 + -1.00*e_013 + -1.00*
     e_023 + -1.00*e_123 + 1.00*e_0123]

    ~m:  MultiVector[1.00*1 + 1.00*e_0 + 1.00*e_1 + 1.00*e_2 + 1.00*e_3 + -1.00*e_01 + -1.00*e_02 +
     -1.00*e_03 + -1.00*e_12 + -1.00*e_13 + -1.00*e_23 + -1.00*e_012 + -1.00*e_013 + -1.00*e_023 +
     -1.00*e_123 + 1.00*e_0123]

    alpha(~m):  MultiVector[1.00*1 + -1.00*e_0 + -1.00*e_1 + -1.00*e_2 + -1.00*e_3 + -1.00*e_01 +
     -1.00*e_02 + -1.00*e_03 + -1.00*e_12 + -1.00*e_13 + -1.00*e_23 + 1.00*e_012 + 1.00*e_013 +
     1.00*e_023 + 1.00*e_123 + 1.00*e_0123]
```

As we see from the code example, all of these operations change the sign of each element differently.

2.2.4 Inverse

To compute the inverse in Clifford's Algebra is all but an easy task, as we've seen in 1.4.6. Fortunately for us, the **tfga** library provides two easy operations to calculate it.

```
    sta.print("c:", c)
    sta.print("c^-1:", sta.simple_inverse(c)) # Faster, only works if c ~c is a scalar
    sta.print("c^-1 shirokov:", sta.inverse(c)) # Always works if an inverse exists
```

Output:

```
    c: MultiVector[36.00*e_01]
    c^-1: MultiVector[0.03*e_01]
    c^-1 shirokov: MultiVector[0.03*e_01]
```

As we can see from the code, the simple inverse works only if $c\tilde{c}$ is a scalar. This is because it uses the following simplified formula:

$$c^{-1} = \frac{\tilde{c}}{c\tilde{c}}$$

If $c\tilde{c}$ is not a scalar, it's mandatory to use the **Shirokov's formula** [8], which is slower but works for every multivector.

2.2.5 Rotation

It is also possible to implement the rotation using the geometric product as follows.

```
1  complex_ga = GeometricAlgebra([1, 1])
2  x = complex_ga.from_scalar(5.0)
3  imag = complex_ga.e01
4  r = complex_ga.approx_exp(complex_ga.geom_prod(complex_ga.from_scalar(np.deg2rad(45).astype(np.
   float32)), imag))
5  complex_ga.print("x:", x)
6  complex_ga.print("e0:", complex_ga.e0)
7  complex_ga.print("e1:", complex_ga.e1)
8  complex_ga.print("i = e01:", imag)
9  complex_ga.print("i^2:", complex_ga.geom_prod(imag, imag))
10 complex_ga.print("r = e^(45° * e12):", r)
11 complex_ga.print("x * r (x rotated 45°):", complex_ga.geom_prod(x, r))
12 complex_ga.print("x * ~r (x rotated -45°):", complex_ga.geom_prod(x, complex_ga.reversion(r)))
```

Output:

```
1  x: MultiVector[5.00*1]
2  e0: MultiVector[1.00*e_0]
3  e1: MultiVector[1.00*e_1]
4  i = e01: MultiVector[1.00*e_01]
5  i^2: MultiVector[-1.00*1]
6  r = e^(45° * e12): MultiVector[0.71*1 + 0.71*e_01]
7  x * r (x rotated 45°): MultiVector[3.54*1 + 3.54*e_01]
8  x * ~r (x rotated -45°): MultiVector[3.54*1 + -3.54*e_01]
```

As we can see in line 4, the rotation of a number is just a matter of doing a geometric product between the number and the radiants.

Note that the operation i^2 translates to $e_{01} * e_{01}$. This is equal to -1 because:

$$e_{01} * e_{01} = e_0e_1e_0e_1 = -e_0e_0e_1e_1 = -1$$

2.2.6 Logarithm and Exponential functions

The **tfga** library also provides approximations of both logarithmic and exponential functions.

```
1  y = complex_ga.from_scalar(0.8)
2  complex_ga.print(y)
3  complex_ga.print(complex_ga.approx_log(y), "expected", np.log(0.8))
4  complex_ga.print(complex_ga.approx_exp(complex_ga.approx_log(y)), "expected", 0.8)
5  complex_ga.print(complex_ga.approx_log(complex_ga.approx_exp(y)), "expected", 0.8) # doesn't
   work because approx_log only works for |x -1| < 1
```

Output:

```
1  MultiVector[0.80*1]
2  MultiVector[-0.22*1] expected -0.2231435513142097
3  MultiVector[0.80*1] expected 0.8
4  MultiVector[-283.90*1] expected 0.8
```

As we can see in the code, the function *approx_ log* approximates the logarithm up to the second digit. Furthermore, the two functions are the opposite of each other. Finally, it's important to remember that, while using *approx_ log*, the input should follow the following rule: $|x - 1| < 1$.

2.3 Use case: Quaternions

2.3.1 Quaternions and Vectors in Geometric Algebra

In this example, we demonstrate how to create a quaternion and compare it to a standard vector using the geometric algebra library. The code illustrates this process.

```

# Create an algebra with 3 basis vectors given their metric.
# Contains geometric algebra operations.
ga = GeometricAlgebra(metric=[1, 1, 1])
```

In this line, you are creating an instance of a geometric algebra with a metric $[1, 1, 1]$, which defines the properties of the three basis vectors. This metric specifies that the basis vectors are orthogonal to each other and have unit norms.

```
# Create geometric algebra tf.Tensor for vector blades (ie. e_0 + e_1 + e_2).
# Represented as tf.Tensor with shape [8] (one value for each blade of the algebra).
# tf.Tensor: [0, 1, 1, 1, 0, 0, 0, 0]
ordinary_vector = ga.from_tensor_with_kind(tf.ones(3), kind="vector")
```

Here, you create a standard vector in the geometric algebra using `tf.ones(3)`, which generates a tensor of dimension 3 with all elements equal to 1. This vector is represented as a linear combination of the three bases (e_0, e_1, e_2) and converted into a geometric vector with `kind="vector"`. The result is a tensor with 8 components, corresponding to the values for each blade of the algebra.

```
# 5 + 5 e_01 + 5 e_02 + 5 e_12
quaternion = ga.from_tensor_with_kind(tf.fill(dims=4, value=5), kind="even")
```

In this line, you create a quaternion using `tf.fill(dims=4, value=5)`, which generates a tensor of dimension 4 with all elements equal to 5. With `kind="even"`, you specify that you want to create a quaternion, which in this context is represented by the scalar and bivector components (e_{01}, e_{02}, e_{12}). Starting from the defined metrics, you have 8 elements available because the algebra was created with the metrics `ga = GeometricAlgebra(metric=[1, 1, 1])`.

By specifying `kind="even"`, you are effectively selecting only the even combinations of the bases from the 8 available to form the quaternion. This produces a quaternion in the form $5 + 5e_{01} + 5e_{02} + 5e_{12}$.

2.3.2 Theoretical Background

The Clifford algebra $\mathrm{Cl}_{3,0,0}(\mathbb{R})$ includes elements that can represent vectors, bivectors, and trivectors in three-dimensional space.

The general element of the Clifford algebra $\mathrm{Cl}_{3,0,0}(\mathbb{R})$ is given by:

$$A = a_0 + a_1e_1 + a_2e_2 + a_3e_3 + a_4e_2e_3 + a_5e_1e_3 + a_6e_1e_2 + a_7e_1e_2e_3$$

The linear combination of the even degree elements of $\mathrm{Cl}_{3,0,0}(\mathbb{R})$ defines the even subalgebra $\mathrm{Cl}^{[0]}_{3,0,0}(\mathbb{R})$ with the general element:

$$q = q_0 + q_1e_2e_3 + q_2e_1e_3 + q_3e_1e_2$$

The basis elements can be identified with the quaternion basis elements i, j, k as:

$$i = e_2e_3, \quad j = e_1e_3, \quad k = e_1e_2$$

This shows that the even subalgebra $\mathrm{Cl}^{[0]}_{3,0,0}(\mathbb{R})$ is Hamilton's real quaternion algebra.

Quaternions are a generalization of complex numbers introduced by William Rowan Hamilton in 1843. A quaternion q is usually written in the form:

$$q = a + bi + cj + dk$$

where a, b, c, d are real numbers, and i, j, k are imaginary units satisfying the following relationships:

$$i^2 = j^2 = k^2 = ijk = -1$$

$$ij = k,\ ji = -k$$

$$jk = i,\ kj = -i$$

$$ki = j,\ ik = -j$$

Quaternions extend complex numbers and are useful in various applications, such as computer graphics, robotics, and physics, especially for representing rotations in three-dimensional space as we will see in some next examples.

In our Clifford Algebra implementation the quaternion properties are satisfied, because:

$$i^2 = (e_2e_3)^2 = e_2e_3e_2e_3 = -e_2e_2e_3e_3 = -1$$

this works also for j and k.

Furthermore

$$ij = e_2e_3e_1e_3 = -e_2e_3e_3e_1 = -e_2e_1 = e_1e_2 = k$$

Finally,

$$ijk = e_2e_3e_1e_3e_1e_2 = -e_2e_1e_3e_3e_1e_2 = -e_2e_1e_1e_2 = -e_2e_2 = -1$$

2.3.3 Differences between a Vector and a Quaternion

- **Vector**: A standard vector in three-dimensional space can be represented as a linear combination of its components along each axis (e.g., $\mathbf{v} = x\mathbf{e}_1 + y\mathbf{e}_2 + z\mathbf{e}_3$).
- **Quaternion**: A quaternion includes a scalar component and three imaginary components. It represents not only a direction in space but also a rotation around that axis.

The example presented above in 2.3.1 clearly illustrates the difference between a standard vector and a quaternion in the context of geometric algebra. A standard vector is a simple directional entity, while a quaternion represents a more complex structure that can be used to model rotations and other transformations in three-dimensional space.

2.4 Neural Network Application: Estimating the Area of a Triangle

The following code imports necessary libraries, configures TensorFlow to manage GPU memory efficiently, installs and imports the specialized package for geometric algebra operations in TensorFlow:

```
import matplotlib.pyplot as plt
import tensorflow as tf

# Make tensorflow not take over the entire GPU memory
for gpu in tf.config.experimental.list_physical_devices('GPU'):
    tf.config.experimental.set_memory_growth(gpu, True)

!pip install tfga
from tfga import GeometricAlgebra
from tfga.layers import GeometricProductDense, TensorToGeometric, GeometricToTensor
```

Now we are going to do several tasks including generating random triangles, calculating their areas using Heron's formula, visualizing them, and then comparing the performance of neural network models with and without Clifford algebra for area calculation.

2.4.1 Generating random triangles and calculating their areas

The function `make_batch` generates random triangles and calculates their areas using Heron's formula:

```
def make_batch(batch_size):
    triangle_points = tf.random.uniform([batch_size, 3, 2], minval=-1, maxval=1)
    x, y = triangle_points[..., 0], triangle_points[..., 1]
    ax, ay, bx, by, cx, cy = x[..., 0], y[..., 0], x[..., 1], y[..., 1], x[..., 2], y[..., 2]
    triangle_areas = 0.5 * tf.abs(ax * (by - cy) + bx * (cy - ay) + cx * (ay - by))
    return triangle_points, triangle_areas
```

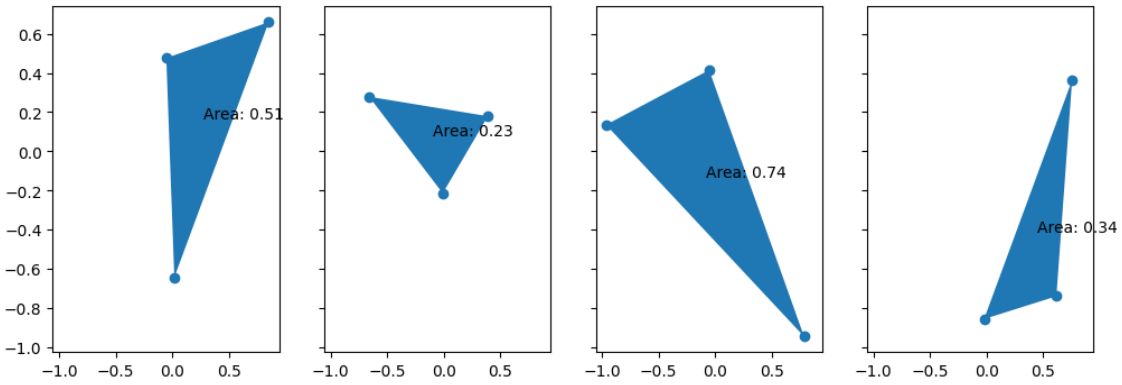


Figure 2.1: Triangles randomly created

- `triangle_points` generates random coordinates for the vertices of the triangles.
- The coordinates are then separated into individual points (ax, ay), (bx, by), and (cx, cy).
- `triangle_areas` calculates the area of each triangle using Heron's formula.

2.4.2 Sample Data Generation and Visualization

The following code generates and visualizes sample triangles along with their areas:

```
num_samples = 4
sample_points, sample_areas = make_batch(num_samples)

fig, axes = plt.subplots(1, num_samples, figsize=(12, 4), sharex=True, sharey=True)
for i, ax in enumerate(axes):
    points = sample_points[i]
    area = sample_areas[i]
    center = tf.reduce_mean(points, axis=0)
    ax.scatter(points[..., 0], points[..., 1])
    ax.add_patch(plt.Polygon(points))
    ax.annotate("Area: %.2f" % area, center)
fig.show()
```

- `num_samples` specifies the number of triangles to generate.
- `sample_points` and `sample_areas` store the vertices and areas of the triangles.
- The code plots each triangle and annotates it with its calculated area.

2.4.3 Neural Network Model with Clifford Algebra

Using the Clifford Algebra library `tfga`, we define a neural network model to predict the areas of triangles:

```
ga = GeometricAlgebra([1, 1])
s_indices = [0]
v_indices = [1, 2]
ga.print(ga.num_blades)
mv_indices = tf.range(ga.num_blades)

model = tf.keras.Sequential([
    TensorToGeometric(ga, blade_indices=v_indices),
    GeometricProductDense(
        ga, units=64, activation="relu",
        blade_indices_kernel=mv_indices,
        blade_indices_bias=mv_indices,
    ),
    GeometricProductDense(
        ga, units=64, activation="relu",
        blade_indices_kernel=mv_indices,
        blade_indices_bias=mv_indices,
    ),
    GeometricProductDense(
        ga, units=1,
        blade_indices_kernel=mv_indices,
        blade_indices_bias=s_indices,
    ),
    GeometricToTensor(ga, blade_indices=s_indices)
])
```

- `GeometricAlgebra` initializes the Clifford algebra structure.
- `TensorToGeometric` converts tensor inputs to geometric objects.
- `GeometricProductDense` layers perform dense layer operations using geometric products.
- `GeometricToTensor` converts the geometric objects back to tensors.

The Clifford Algebra is advantageous in this context because it allows for more natural and efficient handling of geometric transformations and relationships, potentially leading to more accurate and interpretable models.

2.4.4 Training the Model

We generate training and testing data, compile and train the model:

```
train_points, train_areas = make_batch(1024)
test_points, test_areas = make_batch(128)

model.compile(optimizer="Adam", loss="mse", metrics=["mae"])
model.fit(x=train_points, y=train_areas, validation_data=(test_points, test_areas), epochs=100)
```

- `train_points` and `train_areas` generate training data.
- `test_points` and `test_areas` generate testing data.
- The model is compiled with the Adam optimizer, mean squared error loss, and mean absolute error metric, and then trained for 100 epochs.

2.4.5 Model Summary

The following code prints the model summary:

```
print(model.summary())
```

The model summary shows the architecture and the number of parameters:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 tensor_to_geometric (Tenso  (None, 3, 4)              0
 rToGeometric)

 geometric_product_dense (G  (None, 64, 4)             1024
 eometricProductDense)

 geometric_product_dense_1   (None, 64, 4)             16640
 (GeometricProductDense)

 geometric_product_dense_2   (None, 1, 4)              257
 (GeometricProductDense)

 geometric_to_tensor (Geome  (None, 1, 1)              0
 tricToTensor)

=================================================================
Total params: 17921 (70.00 KB)
Trainable params: 17921 (70.00 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
None
```

2.4.6 Comparing Predicted and Actual Areas

After training, we compare the predicted and actual areas of the sample triangles:

```python
predicted_sample_areas = model(sample_points)

fig, axes = plt.subplots(1, num_samples, figsize=(20, 5), sharex=True, sharey=True)
for i, ax in enumerate(axes):
    points = sample_points[i]
    area = sample_areas[i]
    predicted_area = predicted_sample_areas[i]
    center = tf.reduce_mean(points, axis=0)
    ax.scatter(points[..., 0], points[..., 1])
    ax.add_patch(plt.Polygon(points))
    ax.annotate("Area: %.2f" % area, center)
    ax.annotate("Predicted area: %.2f" % predicted_area, center + tf.constant([0, -0.1]))
fig.show()
```

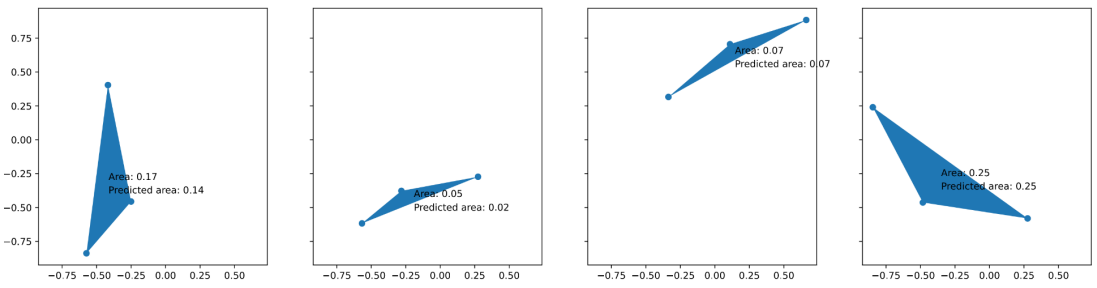


Figure 2.2: predicted and actual areas

- `predicted_sample_areas` calculates the predicted areas using the trained model.

- The code plots each triangle, annotating it with both the actual and predicted areas for comparison.

2.4.7 Neural Network Model Without Clifford Algebra

Next, we define a standard neural network model without using Clifford algebra to predict the areas of triangles:

```
model_normal = tf.keras.Sequential([
    tf.keras.layers.Dense(64*2, activation="relu"),
    tf.keras.layers.Dense(64*2, activation="relu"),
    tf.keras.layers.Dense(1)
])

model_normal.compile(optimizer="Adam", loss="mse", metrics=["mae"])
model_normal.fit(x=tf.reshape(train_points, [-1, 6]), y=train_areas, validation_data=(tf.reshape(
    test_points, [-1, 6]), test_areas), epochs=100)
```

- The model consists of three dense layers, each with ReLU activation except for the final layer.
- The input points are reshaped from $[3, 2]$ to $[6]$ to match the dense layer input requirements.
- The model is compiled and trained in a similar manner to the Clifford algebra model.

2.4.8 Model Summary for Standard Neural Network

The following code prints the model summary:

```
print(model_normal.summary())
```

The model summary shows the architecture and the number of parameters:

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (32, 128)                 896

 dense_1 (Dense)             (32, 128)                 16512

 dense_2 (Dense)             (32, 1)                   129

=================================================================
Total params: 17537 (68.50 KB)
Trainable params: 17537 (68.50 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
None
```

2.4.9 Comparing Predicted and Actual Areas for Standard Neural Network

After training, we compare the predicted and actual areas of the sample triangles:

```
predicted_sample_areas = model_normal(tf.reshape(sample_points, [-1, 6]))

fig, axes = plt.subplots(1, num_samples, figsize=(20, 5), sharex=True, sharey=True)
for i, ax in enumerate(axes):
    points = sample_points[i]
    area = sample_areas[i]
    predicted_area = predicted_sample_areas[i]
    center = tf.reduce_mean(points, axis=0)
    ax.scatter(points[..., 0], points[..., 1])
    ax.add_patch(plt.Polygon(points))
    ax.annotate("Area: %.2f" % area, center)
```

```
    ax.annotate("Predicted area: %.2f" % predicted_area, center + tf.constant([0, -0.1]))
fig.show()
```

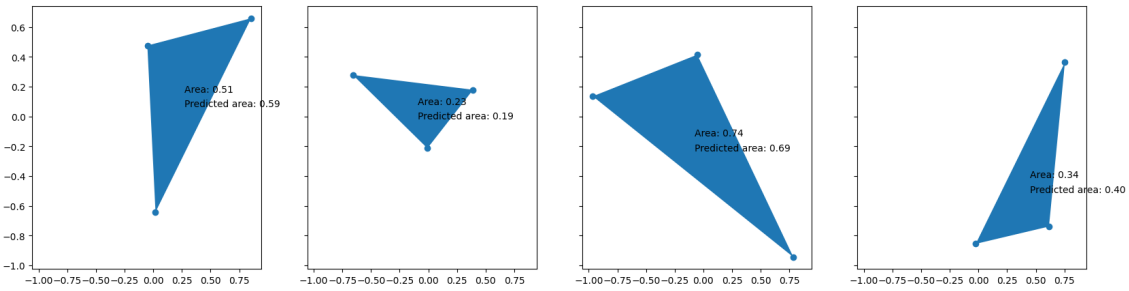


Figure 2.3: Comparing predicted and actual areas

2.4.10 Comparison and Observations

From the experiments conducted, it can be observed that the neural network model utilizing Clifford algebra is significantly slower compared to the standard neural network model. This performance difference is primarily due to the lack of hardware acceleration for geometric algebra computations, which makes the operations more computationally intensive.

In the standard neural network model, the calculation of triangle areas can be directly learned from the input points using simpler matrix multiplications and activation functions. On the other hand, the Clifford algebra model involves geometric products between multivectors (bivectors), which are more complex and not as well-optimized on conventional hardware.

Despite the performance drawbacks, Clifford algebra provides a more geometrically intuitive framework for representing and manipulating shapes, potentially leading to more accurate and interpretable models in certain applications.

However, the predictions of the Clifford algebra model are less accurate compared to the standard model. Several factors contribute to this discrepancy:

- **Complexity of Geometric Operations:** Clifford algebra introduces complex geometric operations that can be harder to optimize during training compared to simpler matrix operations used in standard models.
- **Representation and Transformations:** The abstract representations in Clifford algebra might make it more challenging for the model to directly map input points to triangle areas.
- **Computational Infrastructure:** Current deep learning libraries and hardware are primarily optimized for matrix and tensor operations, not for the specific needs of geometric algebra, leading to less efficient computations.
- **Generalization Capacity:** While Clifford algebra can capture more complex geometric relationships, it may require more data to reach optimal performance.

In conclusion, while Clifford algebra offers a powerful framework for geometric computations, its application in neural networks faces challenges in optimization, efficiency, and accuracy on conventional hardware. With further research and optimization, it could become more competitive and beneficial in specific applications, as we will see with further examples later in this discussion.

2.4.11 Challenges

While the use of Clifford Algebra layers, such as the **GeometricProductDense**, has shown promise in estimating the area of a triangle, several challenges remain. It would be worthwhile to explore in greater detail the underlying reasons for employing Clifford Algebra layers in this context. Specifically, investigating why the **GeometricProductDense** layer is effective in this estimation task could provide deeper insights. Furthermore, understanding the specific advantages of this approach, compared to more traditional neural network layers, could reveal significant benefits or limitations.

2.5 Geometric Algebra Operations for Points and Lines

First, the necessary libraries are imported and the GPU memory settings for TensorFlow are configured:

```
%load_ext autoreload
%autoreload 2

import tensorflow as tf

# Make tensorflow not take over the entire GPU memory
for gpu in tf.config.experimental.list_physical_devices('GPU'):
    tf.config.experimental.set_memory_growth(gpu, True)

from matplotlib import pyplot as plt
from tfga import GeometricAlgebra
```

An instance of geometric algebra is defined with a specific metric, and the basis of multivector elements is printed:

```
ga = GeometricAlgebra([0, 1, 1])
print(ga.basis_mvs)
```

The line

```
ga = GeometricAlgebra([0, 1, 1])
```

defines the metric for the geometric algebra. The array `[0, 1, 1]` specifies the signature of the metric, which determines the behavior of the geometric product of vectors in this algebra.

Explanation of the Metric

- **Signature:** The array `[0, 1, 1]` represents a metric with:
  - One null vector dimension (squares to zero).
  - Two Euclidean dimensions (vectors square to positive values).
- **Interpretation:** This defines a geometric algebra with a 2D Euclidean space and an additional null dimension. The null vector component facilitates the representation of projective geometry elements, such as points and lines in a plane.

2.5.1 Applications

This metric allows for various geometric computations:

- **Points Definition:** Points are defined using combinations of basis vectors. For example:

```
p_1 = ga(ga.e12 - ga.e01)

```

- **Line Definition:** Lines are defined by a regression product (`&`) between two points. For example:

```
l_14 = p_1 & p_4

```

- **Geometric Operations:** Operations such as distance calculations, projections, and intersections use the metric for accurate geometric relationships. For example:

```
def dist_point_line(point, line):
    point_normalized = point.tensor / mv_length(point)
    line_normalized = line.tensor / mv_length(line)
    return ga(point_normalized) & ga(line_normalized)

```

In summary, the metric `[0, 1, 1]` is chosen to facilitate computations in a 2D Euclidean space with an additional null dimension, useful for geometric operations involving points and lines in projective geometry.

Next, several utility functions are defined:

- `mv_length`: Computes the length of a multivector.

```
def mv_length(mv):
    return tf.sqrt((mv * ~mv).tensor)[..., 0]
```

- `dist_point_line`: Computes the signed distance between a point and a line.

```
def dist_point_line(point, line):
    point_normalized = point.tensor / mv_length(point)
    line_normalized = line.tensor / mv_length(line)
    return ga(point_normalized) | ga(line_normalized)
```

- `dist_points`: Computes the signed distance between two points.

```
def dist_points(point_a, point_b):
    point_a_normalized = point_a.tensor / mv_length(point_a)
    point_b_normalized = point_b.tensor / mv_length(point_b)
    return ga(point_a_normalized) | ga(point_b_normalized)
```

- `proj_point_line`: Projects a point onto a line.

```
def proj_point_line(point, line):
    return (point | line) * line
```

- `intersect_lines`: Computes the intersection point of two lines.

```
def intersect_lines(line_a, line_b):
    return line_a ^ line_b
```

- `point_coordinates`: Extracts the homogeneous coordinates of a point.

```
def point_coordinates(point):
    z = point("12")
    x = point("20") / z
    y = point("01") / z
    return x, y
```

Next, five points and two lines are defined using linear combinations of the algebra's bases. The code also computes and prints the signed distance between points and lines and the projection of a point onto a line:

```
# Shift up vertically
shift_23 = 0.5 * ga.e01

p_1 = ga(ga.e12 - ga.e01)
p_2 = ga(ga.e12 - ga.e20 + shift_23)
p_3 = ga(ga.e12 + ga.e20 + shift_23)
p_4 = ga(ga.e12 + ga.e01)
p_5 = ga(ga.e12)

l_14 = p_1 & p_4
l_23 = p_2 & p_3

p2_on_l14 = proj_point_line(p_2, l_14)

print("P1:", p_1)
print("P2:", p_2)
```

```
print("P3:", p_3)
print("P4:", p_4)
print("P5:", p_5)
print("L14:", l_14)
print("Signed distance between P2 and L14:", dist_point_line(p_2, l_14))
print("Signed distance between P3 and L14:", dist_point_line(p_3, l_14))
print("P2 on L14:", p2_on_l14)
```

Giving the following output:

```
P1: MultiVector[-1.00e_01 + 1.00e_12]
P2: MultiVector[0.50e_01 + 1.00e_02 + 1.00e_12]
P3: MultiVector[0.50e_01 + -1.00e_02 + 1.00e_12]
P4: MultiVector[1.00e_01 + 1.00e_12]
P5: MultiVector[1.00e_12]
L14: MultiVector[-2.00e_1]
Signed distance between P2 and L14: MultiVector[1.001]
Signed distance between P3 and L14: MultiVector[-1.001]
P2 on L14: MultiVector[2.00e_01 + 4.00e_12]
```

If we wanted to visualize the points produced in the example, we can do so with the following code:

```
# Plot the results

def plot_point(point, name):
    xy = point_coordinates(point)
    plt.scatter(*xy, marker="x")
    plt.annotate(name, xy)

plt.figure(figsize=(8, 8))
plot_point(p_1, "P1")
plot_point(p_2, "P2")
plot_point(p_3, "P3")
plot_point(p_4, "P4")
plot_point(p_5, "P5")
plot_point(p2_on_l14, "P2 on L14")
plt.xlabel("X")
plt.ylabel("Y")
plt.title("Points")
plt.show()
```

This code snippet utilizes a plotting function to visualize the points generated in the example. Each point is represented by a marker on a 2D plane, with annotations denoting their names. Finally, the plot is displayed with labeled axes and a title.

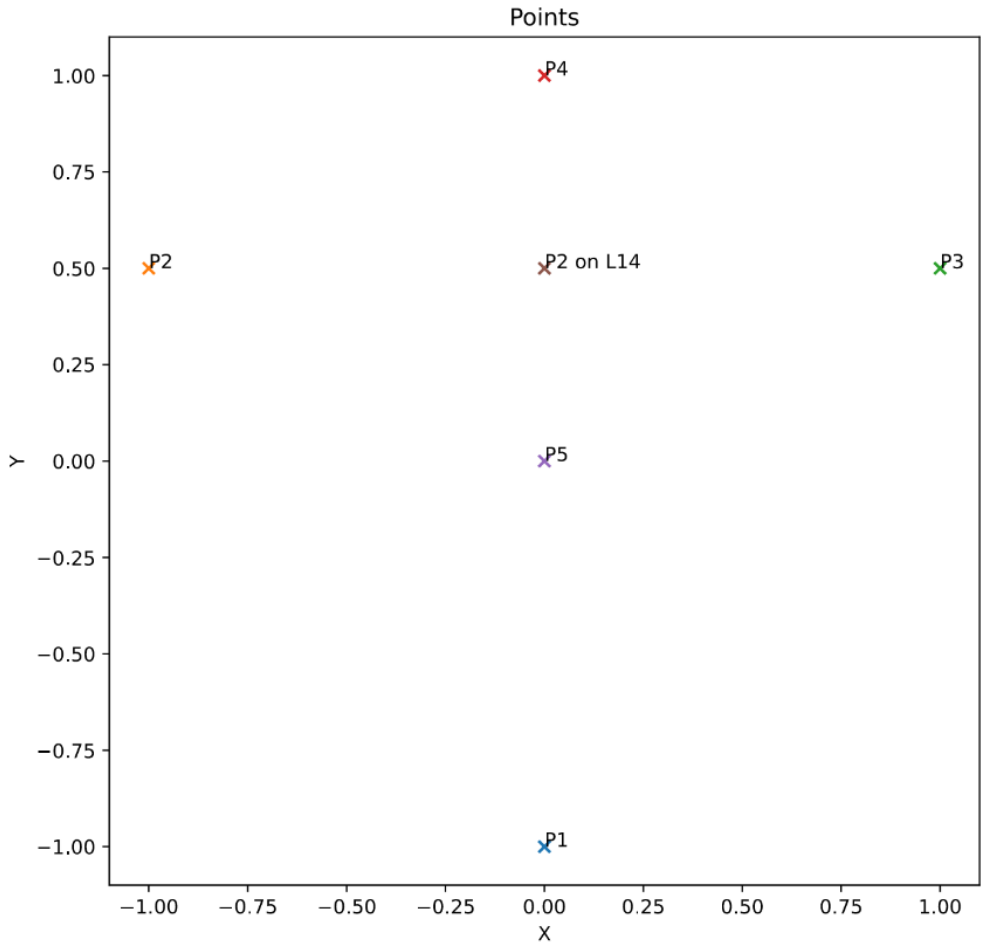


Figure 2.4: Points of the example

If you want to see the created lines drawn on the graph, just add the following Python code:

```
def plot_line(point1, point2, name, color='r'):
    x1, y1 = point_coordinates(point1)
    x2, y2 = point_coordinates(point2)
    plt.plot([x1, x2], [y1, y2], color=color, label=name)

# Find two points on each line
p1_on_l14 = proj_point_line(p_1, l_14)
p4_on_l14 = proj_point_line(p_4, l_14)
p2_on_l23 = proj_point_line(p_2, l_23)
p3_on_l23 = proj_point_line(p_3, l_23)

plt.figure(figsize=(8, 8))
plot_point(p_1, "P1")
plot_point(p_2, "P2")
plot_point(p_3, "P3")
plot_point(p_4, "P4")
plot_point(p_5, "P5")
plot_point(p2_on_l14, "P2 on L14")
plot_line(p1_on_l14, p4_on_l14, "L14", color='b')  # Blue line
plot_line(p2_on_l23, p3_on_l23, "L23", color='g')  # Green line
plt.xlabel("X")
plt.ylabel("Y")
plt.title("Points and Lines")
plt.legend()
plt.show()
```

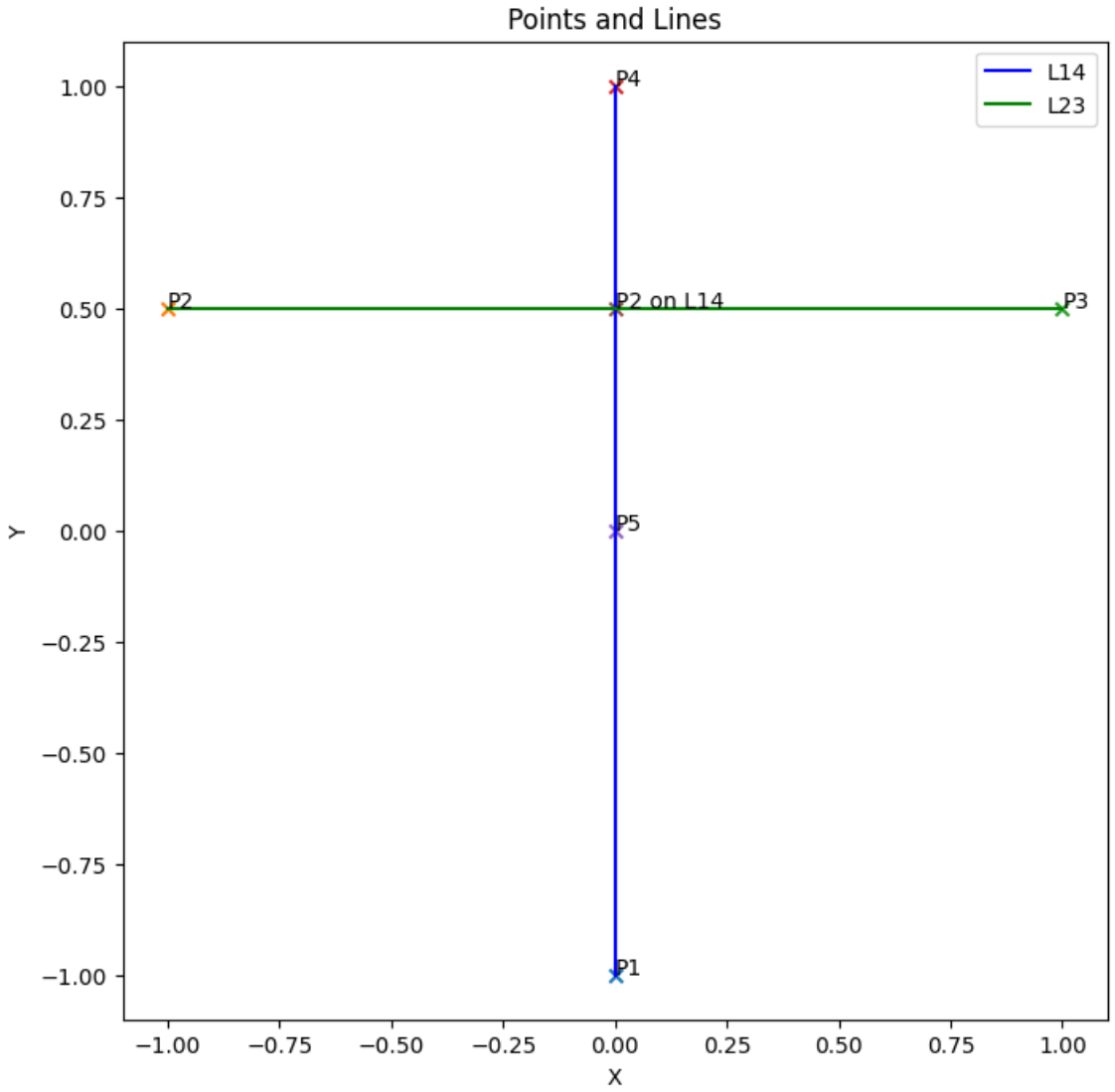


Figure 2.5: Points and lines of the example

2.6 Utilizing Clifford Algebra in TensorFlow with 1D Convolutions

This example demonstrates the integration of Clifford algebra into a TensorFlow-based machine learning pipeline using the `tfga` library. Clifford algebra extends traditional linear algebra by providing a richer mathematical framework that can model complex geometric transformations and interactions more effectively. This can lead to more efficient computations and potentially more powerful representations in certain machine learning tasks.

First, we initialize the Clifford algebra with a signature $[0, 1, 1, 1]$, which corresponds to a 3-dimensional vector space with one scalar dimension (representing time or a scalar field) and three spatial dimensions.

```
ga = GeometricAlgebra([0, 1, 1, 1])
```

We define parameters for a 1D geometric convolution: a batch size of 2, sequence length of 8, 3 input channels, 4 output channels, and a kernel size of 3.

```
batch_size = 2
sequence_length = 8
c_in = 3
c_out = 4
```

```
kernel_size = 3
```

Next, we create input and kernel tensors filled with ones, each having an associated multivector kind from the Clifford algebra.

```
a = ga.from_tensor_with_kind(tf.ones([batch_size, sequence_length, c_in, ga.num_blades]), BladeKind.
    MV)
k = ga.from_tensor_with_kind(tf.ones([kernel_size, c_in, c_out, ga.num_blades]), BladeKind.MV)
```

We then perform a geometric convolution operation on these tensors. The `geom_conv1d` function performs the convolution with a stride of 2 and "SAME" padding.

```
y = ga.geom_conv1d(a, k, 2, "SAME")
print(y.shape)
print(y)
```

To further illustrate, we create a `GeometricProductConv1D` layer, which encapsulates this convolution operation. We specify the same parameters and provide the necessary blade indices for the kernel and bias.

```
conv_layer = GeometricProductConv1D(
    ga, filters=c_out, kernel_size=kernel_size, stride=2, padding="SAME",
    blade_indices_kernel=tf.range(ga.num_blades, dtype=tf.int64),
    blade_indices_bias=tf.range(ga.num_blades, dtype=tf.int64)
)
y2 = conv_layer(a)
print(y2.shape)
ga.print(y2)
ga.print(y2[0, 0, 0])
```

2.6.1 Advantages of Using Clifford's Geometric Algebra in 1D Convolutions

- **Geometric Representation**: Clifford algebra provides a unified framework for complex geometric transformations, beneficial in 1D convolutions.
- **Computational Efficiency**: Clifford algebra offers:
  - **Algorithmic Efficiency**: More compact representations of geometric operations for faster calculations.
  - **Code Simplicity**: Reduced code complexity for representing and manipulating geometric objects.
  - **Numerical Stability**: Enhanced numerical stability of operations, reducing rounding errors.
  - **Parallelism**: Natural parallelism of operations, beneficial for parallel architectures like GPUs.
- **Mathematical Structure**: Clifford algebra extends linear algebra with elements like multivectors and rotors, enhancing data pattern capture.
- **1D Convolutions**: Clifford algebra improves efficiency and accuracy of 1D convolutions, reducing the need for complex transformations.

Application of 1D Convolution to Audio Signals

A concrete example of using 1D convolution is in audio signal processing, such as speech recognition or command classification. It is possible to build a 1D convolutional neural network to classify audio commands, leveraging Clifford algebra for improved representations.

This technique offers the potential for application in various temporal sequence domains, such as biological signal processing and financial data analysis. By leveraging Clifford algebra, we can achieve more advanced geometric representations that may improve model performance on complex tasks.

2.6.2 Challenges

While the use of Clifford Algebra layers, such as the GeometricProductConv1D, has shown promise in various applications, challenges remain. It would be worthwhile to explore in greater detail the underlying reasons for employing Clifford Algebra layers in the context of 1D convolutions. Specifically, investigating why the GeometricProductConv1D layer is effective in these applications could provide deeper insights. Furthermore, understanding the specific advantages of this approach, compared to more traditional convolutional neural network layers, could reveal significant benefits or limitations.

Chapter 3

Modeling Dynamical Systems with Clifford Algebra Neural Networks: The Case of Tetris

In this chapter, we explore the use of Clifford Algebra Neural Networks (GCANs) in modeling dynamical systems, with a particular focus on their application in predicting the trajectories of objects in a modified version of the Tetris game. We discuss the advantages of GCANs over traditional neural network models and present the experimental results from the paper *Geometric Clifford Algebra Networks* that demonstrate their superior performance in complex trajectory prediction tasks.

3.1 Clifford Algebra Neural Networks

GCANs utilize the mathematical framework of Clifford algebras to handle higher-dimensional data and transformations more effectively. This approach is particularly beneficial in tasks involving rotations and translations in three-dimensional space. By employing the $Cl_{3,0,1}$ algebra, GCANs can represent isometries in $\mathbb{R}^3$, thus providing a robust method for capturing the dynamics of objects subjected to complex motions.

3.2 The Tetris Experiment

To illustrate the capabilities of GCANs, they consider a scenario inspired by the classic game of Tetris. In this experiment, Tetris objects, initially positioned at the origin, undergo random translations and rotations around their centers of mass. These transformations are sampled conditionally, creating correlations between the objects. Additionally, conditional Gaussian noise is applied to the individual parts of each object, simulating a more realistic and challenging environment.

The objects move outward from the origin in an exploding fashion, continuously rotating around their centers of mass. Given four input time steps, the model's objective is to predict the subsequent four time steps accurately. This requires the model to infer the positions, velocities, rotation axes, and angular velocities, and apply these inferred parameters to future time steps.

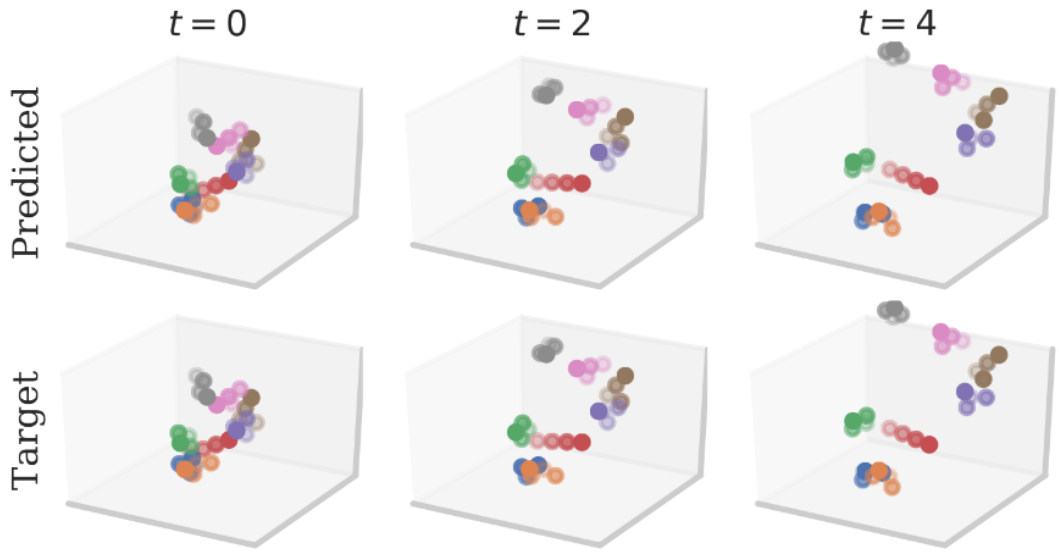


Figure 3.1: Tetris trajectories. Exemplary predicted (top) and ground- truth (bottom) states. Predictions are obtained by the GCA-GNN model when using 16384 training trajectories.

3.3 Methodology

In this study, the authors used the $Cl_{3,0,1}$ algebra to represent the necessary isometries in $\mathbb{R}^3$. They compared the performance of Multi-Layer Perceptrons (MLPs) and Graph Neural Networks (GNNs) against their GCAN-enhanced counterparts. While GCAN-MLPs only showed marginal improvements over the baseline models, GCAN-GNNs significantly outperformed all traditional approaches.

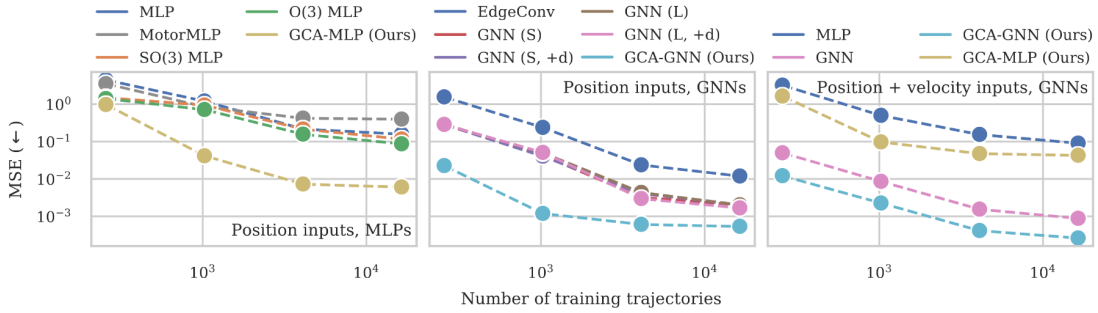


Figure 3.2: Test MSE results of the Tetris experiment as a function of the number of training trajectories. Left: comparison of different MLP models, center: comparison of different GNN models, right: comparison of the best MLP and GNN models when velocities are included.

3.4 Results and Discussion

The experimental results indicate that GCANs, particularly when integrated with GNNs, excel at modeling the complex trajectories of Tetris objects. The ability of GCAN-GNNs to outperform classical methods underscores the potential of Clifford algebras in enhancing the modeling capabilities of neural networks. This improvement is attributed to the more effective handling of rotational and

translational transformations in three-dimensional space, which are common in many real-world dynamical systems.

In conclusion the application of GCANs in modeling dynamical systems, as demonstrated through the Tetris experiment, highlights their potential in advancing the state of the art. By leveraging the properties of Clifford algebras, GCANs provide a powerful tool for accurately predicting complex object trajectories. Future research could explore further refinements and applications of this approach across various domains.

Chapter 4

CGAPoseNet+GCAN

In this chapter, we explore the innovative approach proposed in the paper titled *CGAPoseNet+GCAN: A Geometric Clifford Algebra Network for Geometry-aware Camera Pose Regression* [6] by the authors Pepe et al.

4.1 Paper Overview

The paper introduces CGAPoseNet+GCAN as an extension of CGAPoseNet, a framework designed for camera pose regression. CGAPoseNet is founded on Clifford Geometric Algebra, which unifies quaternions and translation vectors into a single mathematical entity called a motor, facilitating the unique description of camera poses.

4.1.1 Motor

A motor, as mentioned in the paper *CGAposeNet+GCAN: A geometric clifford algebra network for geometry-aware camera pose regression* [6], fuses both rotation and translation. Its construction is defined in the following way.

A point $x \in Cl_{3,0}$, is mapped to $X \in Cl_{4,0}$ through the function $f : x \to X$

$$X = f(x) = \left(\frac{2\lambda}{\lambda^2 + x^2}\right) x + \left(\frac{\lambda^2 - x^2}{\lambda^2 + x^2}\right) e_4. \tag{4.1}$$

It can be shown that translating and rotating in $Cl_{4,0}$ can both be done through rotors. Given a translation vector $t \in Cl_{3,0}$, its corresponding rotor in 4D spherical geometry is given by:

$$T = g(t) = \frac{\lambda + te_4}{\sqrt{\lambda^2 + t^2}} \tag{4.2}$$

A rotor R in 3D Euclidean geometry is still R in 4D spherical geometry. The rigid body motion, i.e. translation and rotation, of an object X into X' in the 1D-Up CGA can hence be expressed as the combination of two sandwich products:

$$X' = TRX\tilde{R}\tilde{T} = MX\tilde{M} \tag{4.3}$$

The geometric product $M = TR$ yields a *motor*, which represents a rotation and a translation. Note how rotations and translations are now expressed in the same units. Motors are objects (multivectors) in $Cl_{4,0}$ with only even blades, presenting 1 scalar, 6 bivector and 1 quadrivector components:

$$M = \underbrace{x_0 1}_{\text{scalar}} + \underbrace{x_{12}e_{12} + x_{13}e_{13} + x_{14}e_{14} + x_{23}e_{23} + x_{24}e_{24} + x_{34}e_{34}}_{\text{bivector}} + \underbrace{x_{1234}e_{1234}}_{\text{quadrivector}} \tag{4.4}$$

Since motors combine translations and rotations, they can be employed as a pose representation with 8 parameters (i.e. the 8 coefficients) [6].

Code implementation

```
# Converting the rotation matrix M into a rotor R

M = np.array(1)
B = [
    ga(ga.from_scalar(float(M[0, 0]))) * ga(ga.e0) + ga(ga.from_scalar(float(M[1, 0]))) * ga(ga.e1)
     + ga(ga.from_scalar(float(M[2, 0]))) * ga(ga.e2),
    ga(ga.from_scalar(float(M[0, 1]))) * ga(ga.e0) + ga(ga.from_scalar(float(M[1, 1]))) * ga(ga.e1)
     + ga(ga.from_scalar(float(M[2, 1]))) * ga(ga.e2),
    ga(ga.from_scalar(float(M[0, 2]))) * ga(ga.e0) + ga(ga.from_scalar(float(M[1, 2]))) * ga(ga.e1)
     + ga(ga.from_scalar(float(M[2, 2]))) * ga(ga.e2)
]

A = [ga(ga.e0), ga(ga.e1), ga(ga.e2)]
R = ga(ga.from_scalar(1.0)) + A[0] * B[0] + A[1] * B[1] + A[2] * B[2]

# Normalization
R = R.tensor / mv_length(R)

# Use the tf.math.is_nan() function to create a mask of the same shape as 'R',
# where each element is True if that element is NaN and False otherwise.
mask = tf.math.is_nan(R)

# Use this mask to replace the NaN values in 'R' with 0
R = tf.where(mask, 0.0, R)
R = ga(R)

# Mapping the position from Euclidean into spherical space
Ta = translation_rotor(
    ga(ga.from_scalar(float(M[0, 3]))) * ga(ga.e0) +
    ga(ga.from_scalar(float(M[1, 3]))) * ga(ga.e1) +
    ga(ga.from_scalar(float(M[2, 3]))) * ga(ga.e2)
)

N = Ta * R
N = N.tensor
y = np.append(y, [N[0], N[5], N[6], N[7], N[8], N[9], N[10], N[15]])
```

Translation_rotor function

```
# Create an algebra with 4 basis vectors all having positive squares (Euclidean signature).
ga = GeometricAlgebra(metric=[1, 1, 1, 1])

lambda_coeff = 10.0

# From Euclidean to 1D Up CGA.
def translation_rotor(a, L = lambda_coeff):
    e4 = ga(ga.e3)  # e4 is the fourth basis vector, but indexed as 3 (0-based index)
    L_mv = ga(ga.from_scalar(L))
    L_mv_sq = ga(ga.from_scalar(L**2))
    a_sq = a|a
    norm_factor = tf.sqrt(L_mv_sq.tensor + a_sq.tensor )
    Ta = (L_mv + a*e4) / ga(norm_factor)
    return Ta
```

4.1.2 Benefits

Unlike traditional methods, CGAPoseNet doesn't require extensive fine-tuning of loss functions or additional scene information like 3D point clouds. However, it solely predicts motor coefficients without a deep understanding of their geometric significance.

In response, CGAPoseNet+GCAN integrates Geometric Clifford Algebra Network (GCAN) to imbue the pipeline with geometric awareness. Leveraging advancements in Geometric Deep Learning, CGAPoseNet+GCAN refines predictions through a series of GCAN layers, operating within the mathematical space of $Cl_{4,0}$. By doing so, it preserves the geometric properties of

inputs, weights, and biases, thereby enhancing performance and reducing the number of trainable parameters.

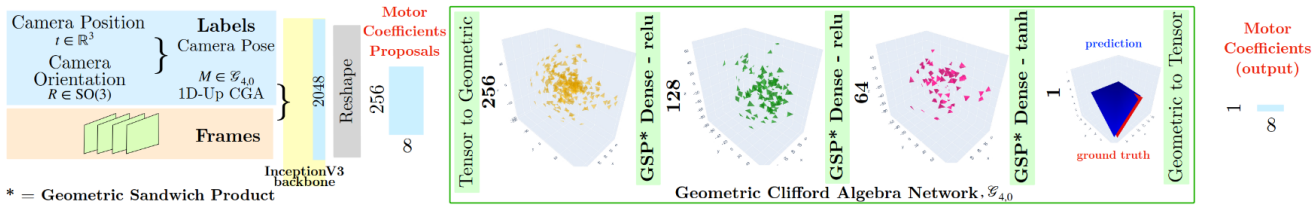


Figure 1. The CGAPoseNet+GCAN architecture. The output of the InceptionV3 network is reshaped to obtain a set of motor coefficients proposals. Motors are objects in the 1D-Up Conformal Geometric Algebra (CGA) $\mathcal{G}_{4,0}$ with scalar, bivector and quadrivector parts, giving a total of 8 real coefficients. These coefficients are used to build motors $\in \mathcal{G}_{4,0}$ in input to the Geometric Clifford Algebra Network (GCAN). A motor represents a rotation and a translation, and it is hence a suitable representation for camera poses. The GCAN works in $\mathcal{G}_{4,0}$ space and has weights, biases and outputs that are also motors, and hence interpretable as poses. The GCAN narrows down the proposals to a single motor through a geometric understanding of the scene.

4.2 Contributions and Results

The authors of the paper discovered that CGAPoseNet+GCAN surpasses CGAPoseNet's performance by significant margins. It reduces the average rotation error by 41% and the average translation error by 8.8%, while also outperforming the best PoseNet strategy by 32.6% in rotation error and 19.9% in translation error. Furthermore, it achieves state-of-the-art results across 13 commonly used datasets for camera pose regression. Our results differ from those reported in the paper, as we conducted a proof of concept on only one dataset, whereas the authors evaluated their model across 13 commonly used datasets for camera pose regression. Additionally, due to resource constraints such as limited computational resources (RAM and GPU) and the use of Colab with the free version, we downsized the dataset. Despite these limitations, our approach still demonstrates promising results.

4.3 Detailed Analysis

In this section, we will explain how CGAPoseNet+GCAN was created, focusing primarily on the implementation of GCAN, which is the core component of the paper and our proposed implementation using the **tfga** package. The tfga package provides us with the necessary tools to utilize Clifford algebra and implements the layers required to replicate the methodology proposed in the paper.

Among the layers we utilized from tfga are:

- **TensorToGeometric**: Converts from a `tf.Tensor` to the geometric algebra `tf.Tensor` (MultiVector) with as many blades on the last axis as basis blades in the algebra, where blade indices determine which basis blades the input's values belong to.
- **GeometricToTensor**: Converts from a geometric algebra `tf.Tensor` (MultiVector) with as many blades on the last axis as basis blades in the algebra to a `tf.Tensor`, where blade indices determine which basis blades we extract for the output.
- **GeometricSandwichProductDense**: Analogous to Keras' `Dense` with multivector-valued weights and biases. Each term in the matrix multiplication performs the geometric product $w \cdot x \cdot \sim w$.

With these elements provided by tfga, we were able to implement the final layer of the neural network and train it using the reduced dataset.

Our CGAPoseNet+GCAN model implementation

```
import tensorflow as tf
!pip install tfga
from tfga import GeometricAlgebra
from tfga.layers import TensorToGeometric, GeometricToTensor, GeometricSandwichProductDense
from tfga.blades import BladeKind
import keras
from keras.applications.inception_v3 import InceptionV3
from keras.models import Model
from keras.layers import Input, Conv2D, MaxPooling2D, AveragePooling2D, Flatten,
    GlobalAveragePooling2D, Dense, Dropout, concatenate


# Define the geometric algebra
sta = GeometricAlgebra([1, 1, 1, 1])

# Load the InceptionV3 model with ImageNet weights
base_model = InceptionV3(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

# Add custom layers
x = base_model.output
x = tf.keras.layers.GlobalAveragePooling2D()(x)


# Reshape from 2048 to 256x8 to create the motor proposals
x = tf.keras.layers.Dense(2048)(x)
x = tf.reshape(x, (-1, 256, 8))

# Define the model up to the intermediate output
intermediate_model = tf.keras.Model(inputs=base_model.input, outputs=x)

# Convert to geometric algebra representation
vector_blade_indices = sta.get_kind_blade_indices(BladeKind.EVEN)

x = TensorToGeometric(sta, blade_indices=vector_blade_indices)(x)


# Apply geometric layers
x = GeometricSandwichProductDense(sta, units=128, blade_indices_kernel=vector_blade_indices,
    activation = "relu", blade_indices_bias=vector_blade_indices)(x)

x = GeometricSandwichProductDense(sta, units=64, blade_indices_kernel=vector_blade_indices,
    activation = "relu", blade_indices_bias=vector_blade_indices)(x)

x = GeometricSandwichProductDense(sta, units=1, blade_indices_kernel=vector_blade_indices,
    activation = "tanh", blade_indices_bias=vector_blade_indices)(x)


# Check that the indices are correct and concatenate the indices
result_indices = sta.get_kind_blade_indices(BladeKind.EVEN)


# Convert back to standard tensor
x = GeometricToTensor(sta, blade_indices=result_indices)(x)


# Final layer to produce the 8 motor coefficients
output = tf.keras.layers.Dense(8)(x)

# Define the complete model
CGAPoseNet_GCAN = tf.keras.Model(inputs=base_model.input, outputs=output)
CGAPoseNet_GCAN.summary()
```

4.4 Considerations about Resource Limitations

Due to the limited resources available to us compared to those of the authors of the paper, we decided to rewrite the code related to data reading from the dataset. The original code proposed

by the authors required excessive RAM as it loaded the entire dataset into memory. We replaced it with a `PoseDataset` class that manages the dataset in a modular way and, more importantly, processes data in batches as needed, thereby avoiding the need to keep all the data in memory. For example, the position and y data are kept in RAM, while the more substantial x data are handled in batches when necessary.

Additionally, the class includes methods to handle the split between training and test datasets. The complete code can be found in Appendix A.

4.5 Library Replacement Justification

We decided to replace the library used by the authors of the paper (*Pepe et al.*) with the `tfga` library for two main purposes: to add the final layers of the geometric sandwich product and to transpose the output values (y) to make them compatible with Clifford Algebra. This decision is also supported by benchmark tests conducted by RobinKa (author of the `tfga` library), who compared the `tfga` library with the `clifford` library used by the authors of the paper. The results of these benchmarks are shown in the graphs (image 4.1) below.

The benchmarks were conducted with the following specifications:

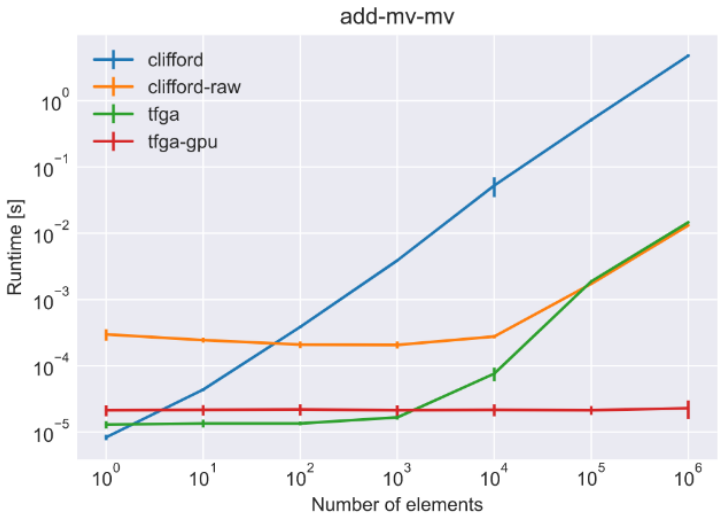
- CPU: AMD Ryzen 7 2700X
- GPU: Nvidia GTX 1070
- RAM: 2x16GB (2800MHz)
- OS: Windows 10 Pro 1903

And relevant libraries:

- `tfga`: 0.1.10
- `tf-gpu-nightly`: 2.3.0-dev20200515
- `clifford`: 1.3.0
- `numpy (mkl)`: 1.18.1
- `numba`: 0.49.1

Additionally, the environment variable `MKL_DEBUG_CPU_TYPE` was set to 5 to disable the crippling of AMD CPUs by MKL.

Addition A + B, Algebra=STA, A=Full Multivector, B=Full Multivector



Geometric Product A * B, Algebra=STA, A=Full Multivector, B=Full Multivector

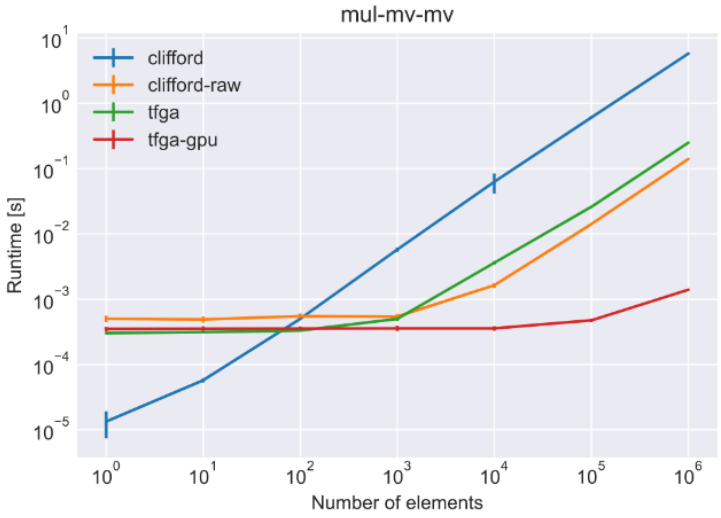


Figure 4.1: Benchmark results comparing `tfga` and `clifford` libraries for addition and geometric product operations.

4.6 Results

The results obtained in our work were more than acceptable. Albeit the tests were done using a fraction of the original dataset (2/12 directories) the resulting loss was comparable regarding the network from the original paper.

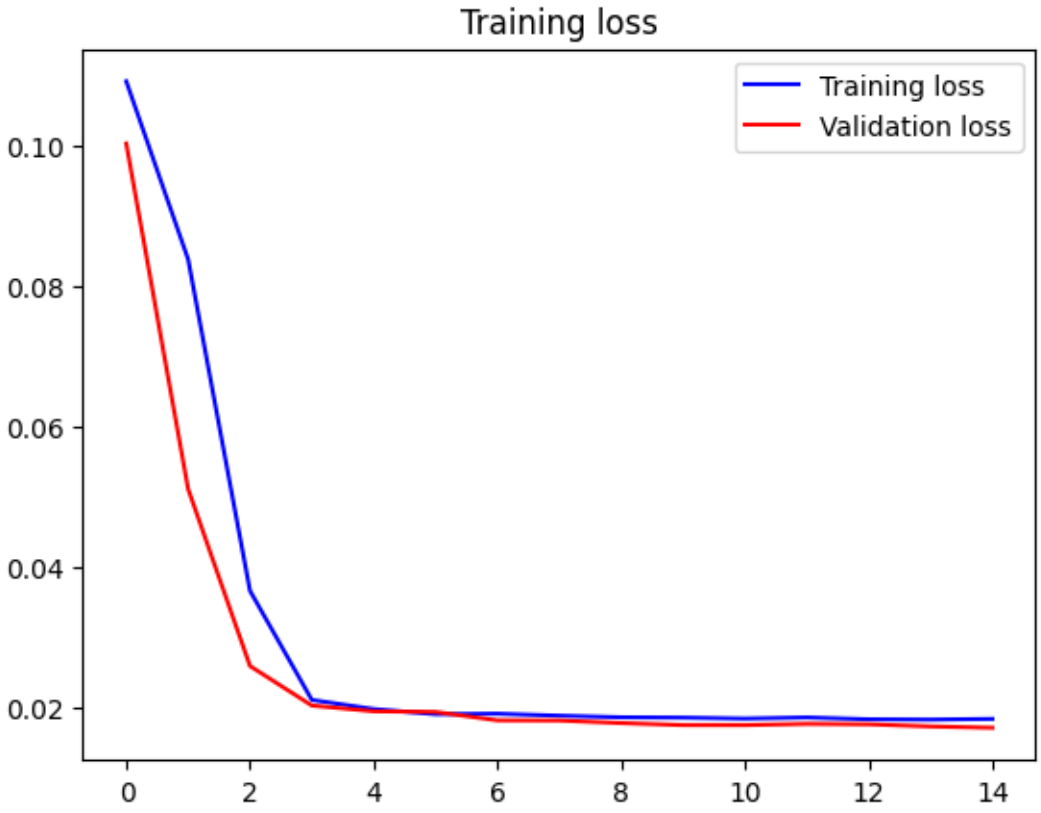


Figure 4.2: Training and Validation Loss

```
37         tot += mse
38
39 print(tot)
40 np.save("/content/drive/MyDrive/MSE_2pkg_29_05_24.npy", MSE)
41
```

```
2.0438181838253513
```

Figure 4.3: Mean Squared Error of our model

Finally, as we said before, the speed of the network is significantly faster than the original model thanks to the **tfga** library.

Due to the lack of resources it wasn't possible to have a full confrontation with the original paper, but since we managed to obtain similar results with a small dataset it's entirely possible to obtain excellent results with more computational power.

Chapter 5

Conclusion

In this report, we have explored various aspects of Clifford Algebra and its practical applications in computational and engineering contexts. Our work can be summarized as follows:

1. **Quick review of Clifford Algebra theory**:
   - We started with a theoretical background on Clifford Algebra, providing an overview of its fundamental concepts such as real numbers, multivectors, pseudovectors, and pseudoscalars.
   - Essential operations within Clifford Algebra, including inner, outer, geometric, and regressive products, were discussed, along with the notions of inverse of a multivector and the various automorphisms and antiautomorphisms that exist within this algebraic framework.
2. **Illustration of TensorFlow Geometric Algebra (TFGA)**:
   - TFGA is a library designed to handle Clifford Algebra computations within the TensorFlow environment developed by Robin Kahlow.
   - Various examples were provided to demonstrate the practical use of TFGA, including basic operations, inverse calculations, and more complex functions like rotations, logarithms, and exponentials in Clifford Algebra.
   - A specific use case involving quaternions was analyzed to highlight the differences between vectors and quaternions in geometric algebra.
3. **Increasingly Complex Applications**:
   - We examined the application of Clifford Algebra in modeling dynamical systems, using the game Tetris as a case study.
   - A detailed methodology was provided for implementing Clifford Algebra in neural networks, focusing on how this algebra can be used to model and analyze the behavior of such systems effectively.
   - Results from the Tetris experiment demonstrated the potential of using Clifford Algebra in real-world computational problems.
4. **Implementation of a Neural Network Model**:
   - We implemented a neural network model incorporating layers that perform Clifford Algebra operations, replicating the methodology described in the paper by *Pepe et al.*
   - This model demonstrated the advantages of integrating Clifford Algebra into neural network architectures, providing insights into its benefits for complex problem-solving and pattern recognition tasks.
   - Comparative analysis was conducted to evaluate the performance of the Clifford Algebra-based neural network against a standard neural network model, highlighting the improvements and efficiencies gained.

In conclusion, for anyone interested in delving deeper into Clifford Algebra and its practical applications, we recommend the book *Numerical Calculations in Clifford Algebra: A Practical Guide for Engineers and Scientists* [7]. This text offers a unique combination of theory and practice, enriched with numerous worked examples and computational calculations, making it an indispensable reference for engineers and scientists.

Appendix A

Appendix

This is the complete code for the implementation of the model proposed in the paper *CGAPoseNet+GCAN: A Geometric Clifford Algebra Network for Geometry-aware Camera Pose Regression* [6] realized by us: we started from the code available at the following link from the paper's author, which provides the foundation of the neural network, CGAPoseNet [5]. As mentioned, we converted the code to the `tfga` package and rewrote some parts of the code to optimize data extraction from the dataset. Additionally, we created the final layer using the layers with Clifford algebra knowledge from the `tfga` package, implementing the missing layer in the provided code that implements GCAN as illustrated in the paper.

Link to the dataset: rgb-d-dataset-7-scenes

```
import tensorflow as tf
!pip install tfga
from tfga import GeometricAlgebra
from tfga.layers import TensorToGeometric, GeometricToTensor, GeometricSandwichProductDense
from tfga.blades import BladeKind
import keras
from keras.applications.inception_v3 import InceptionV3
from keras.models import Model
from keras.layers import Input, Conv2D, MaxPooling2D, AveragePooling2D, Flatten,
    GlobalAveragePooling2D, Dense, Dropout, concatenate


# Define the geometric algebra
sta = GeometricAlgebra([1, 1, 1, 1])

# Load the InceptionV3 model with ImageNet weights
base_model = InceptionV3(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

# Add custom layers
x = base_model.output
x = tf.keras.layers.GlobalAveragePooling2D()(x)


# Reshape from 2048 to 256x8 to create the motor proposals
x = tf.keras.layers.Dense(2048)(x)
x = tf.reshape(x, (-1, 256, 8))

# Define the model up to the intermediate output
intermediate_model = tf.keras.Model(inputs=base_model.input, outputs=x)

# Convert to geometric algebra representation
vector_blade_indices = sta.get_kind_blade_indices(BladeKind.EVEN)

x = TensorToGeometric(sta, blade_indices=vector_blade_indices)(x)


# Apply geometric layers
x = GeometricSandwichProductDense(sta, units=128, blade_indices_kernel=vector_blade_indices,
    activation = "relu", blade_indices_bias=vector_blade_indices)(x)
```

```

x = GeometricSandwichProductDense(sta, units=64, blade_indices_kernel=vector_blade_indices,
    activation = "relu", blade_indices_bias=vector_blade_indices)(x)

x = GeometricSandwichProductDense(sta, units=1, blade_indices_kernel=vector_blade_indices,
    activation = "tanh", blade_indices_bias=vector_blade_indices)(x)


# Check that the indices are correct and concatenate the indices
result_indices = sta.get_kind_blade_indices(BladeKind.EVEN)


# Convert back to standard tensor
x = GeometricToTensor(sta, blade_indices=result_indices)(x)


# Final layer to produce the 8 motor coefficients
output = tf.keras.layers.Dense(8)(x)

# Define the complete model
CGAPoseNet_GCAN = tf.keras.Model(inputs=base_model.input, outputs=output)
CGAPoseNet_GCAN.summary()
```

```
FOLDER = "redkitchen" #Change the name to change the dataset

from google.colab import drive
drive.mount('/content/drive')
```

```
import tensorflow as tf
from tfga import GeometricAlgebra
import numpy as np
import os
import cv2

# Create an algebra with 4 basis vectors all having positive squares (Euclidean signature).
ga = GeometricAlgebra(metric=[1, 1, 1, 1])

lambda_coeff = 10.0

# From Euclidean to 1D Up CGA.
def translation_rotor(a, L = lambda_coeff):
    # a_vec = ga.from_tensor_with_kind(a , "vector")
    e4 = ga(ga.e3)  # e4 is the fourth basis vector, but indexed as 3 (0-based index)
    L_mv = ga(ga.from_scalar(L))
    L_mv_sq = ga(ga.from_scalar(L**2))
    a_sq = a|a
    norm_factor = tf.sqrt(L_mv_sq.tensor + a_sq.tensor )
    Ta = (L_mv + a*e4) / ga(norm_factor)
    return Ta

# From Euclidean to 1D Up CGA. function implementing the Eq. 10 (X = f(x))
def up1D(x, L = lambda_coeff):
    L_tensor = tf.convert_to_tensor(L, dtype=tf.float32)
    x_tensor = tf.convert_to_tensor(x, dtype=tf.float32)
    e4 = ga.e3
    Y = (2 * L_tensor / (L_tensor**2 + x_tensor**2)) * x_tensor + ((L_tensor**2 - x_tensor**2) / (
    L_tensor**2 + x_tensor**2)) * e4
    return Y

# From 1D Up CGA to Euclidean. function implementing the inverse of Eq. 10 (x = f^{-1}(X))
def down1D(Y, x, L = lambda_coeff):
    L_tensor = tf.convert_to_tensor(L, dtype=tf.float32)
    x_tensor = tf.convert_to_tensor(x, dtype=tf.float32)
    alpha = (2 * L_tensor / (L_tensor**2 + x_tensor**2))
    beta  =  (L_tensor**2 - x_tensor**2) / (L_tensor**2 + x_tensor**2)
    e4 = ga.e3
    x = (Y - beta * e4) / alpha
    return x

def mv_length(mv):
    return tf.sqrt((mv * ~mv).tensor)[..., 0]
```

```
import glob
import os
from typing import List
import tensorflow as tf
import math
import numpy as np

import cv2
import  sklearn.utils as skutlis

path = "/content/drive/MyDrive/"+ FOLDER

class PoseDataset(tf.keras.utils.Sequence):

    def __init__(self, root = "", numFolders = 1, seed=34, batch_size=32, img_size=(224, 224)):
        if root == "":
            return

        self.seed : int = seed
        self.batch_size = batch_size
        self.img_size = img_size

        self.x : list[str] = []
        self.y : list[float] = []
        self.position : list[float] = []

        for i, folder_name in enumerate(glob.glob(os.path.join(root, 'seq-*'))):

            if i >= numFolders:
              self.x = self.x[:len(self.y)]
              return

            print(f"Processing {folder_name}")

            # Define the pattern for the files to be processed
            pose_files = glob.glob(os.path.join(folder_name, 'frame-*.pose.txt'))

            self.x += [x.replace(".pose.txt", ".color.png") for x in pose_files]

            for j, pose_file in enumerate(pose_files):
                if j % 100 == 0:
                    print("  ", j, "/", str(len(pose_files)))

                y,position = PoseDataset._read_pose_file(pose_file)
                self.y.append(y)
                self.position.append(position)



    def __len__(self):
        return math.floor(len(self.x) / self.batch_size)

    def _read_pose_file(path : str):

        y = []
        position = []

        with open(path, 'r') as f:
            l = [[float(num) for num in line.split('\t ')] for line in f]

        #converting the rotation matrix M into a rotor R

        M = np.array(l)

        B =[ga(ga.from_scalar(float(M[0,0]))) * ga(ga.e0) + ga(ga.from_scalar(float(M[1,0])))* ga(ga
    .e1) + ga(ga.from_scalar(float(M[2,0])))* ga(ga.e2),
            ga(ga.from_scalar(float(M[0,1]))) * ga(ga.e0) + ga(ga.from_scalar(float(M[1,1])))* ga(ga
    .e1) + ga(ga.from_scalar(float(M[2,1])))* ga(ga.e2),
            ga(ga.from_scalar(float(M[0,2]))) * ga(ga.e0) + ga(ga.from_scalar(float(M[1,2])))* ga(ga
    .e1) + ga(ga.from_scalar(float(M[2,2])))* ga(ga.e2)]
```

```
        A = [ga(ga.e0),ga(ga.e1), ga(ga.e2)]
        R = ga(ga.from_scalar(1.0)) + A[0]*B[0] + A[1]*B[1] + A[2]*B[2]

        R = R.tensor / mv_length(R)


        # Use the tf.math.is_nan() function to create a mask of the same shape as 'R',
        # where each element is True if that element is NaN and False otherwise.
        mask = tf.math.is_nan(R)

        # Use this mask to replace the NaN values in 'R' with 0
        R = tf.where(mask, 0.0, R)
        R = ga(R)
        # Mapping the position from Euclidean into spherical space
        Ta = translation_rotor(ga(ga.from_scalar(float(M[0,3]))) * ga(ga.e0) + ga(ga.from_scalar(float(M[1,3]))) * ga(ga.e1) + ga(ga.from_scalar(float(M[2,3]))) * ga(ga.e2))

        N = Ta*R
        N = N.tensor

        y = np.append(y, [N[0], N[5], N[6], N[7], N[8], N[9], N[10], N[15]])

        position = np.append(position, [M[0,3], M[1,3], M[2,3]])

        return y, position


    def __getitem__(self, idx):
        start = idx * self.batch_size
        end = (idx + 1) * self.batch_size

        batch_x_paths = self.x[start:end]
        batch_y = self.y[start:end]

        # Calculate the number of samples in the batch
        batch_size = len(batch_x_paths)

        # Initialize the arrays for data_x and data_y with zeros
        data_x = np.zeros((self.batch_size, *self.img_size, 3), dtype=np.float32)
        data_y = np.zeros((self.batch_size, 8), dtype=np.float32)  # Adjust the shape (8) based on your target shape

        # Load and preprocess the images and labels
        for i, img_path in enumerate(batch_x_paths):
            img = cv2.imread(img_path)
            if(img is not None):
              img = cv2.resize(img, self.img_size)
              img = cv2.normalize(img, None, alpha=0, beta=1, norm_type=cv2.NORM_MINMAX, dtype=cv2.CV_32F)
              data_x[i] = img

            # Assuming batch_y is a list of numpy arrays
            data_y[i] = batch_y[i]

        # If the batch is smaller than the batch_size, the remaining part is already zero-padded
        return data_x, data_y


    def on_epoch_end(self):
        self.x = skutlis.shuffle(self.x, random_state=self.seed)
        self.y = skutlis.shuffle(self.y, random_state=self.seed)
        self.position = skutlis.shuffle(self.position, random_state=self.seed)

    def split(self, split = 0.8):
        DATASET_SIZE = len(self.x)

        # Create permutation
        permutation = np.random.RandomState(seed=self.seed).permutation(DATASET_SIZE)
```

```
        # Apply the SAME permutation to both images and masks
        self.x = [self.x[i] for i in permutation]
        self.y = [self.y[i] for i in permutation]
        self.position = [self.position[i] for i in permutation]

        training = PoseDataset()
        validation = PoseDataset()

        training.x = self.x[:int(DATASET_SIZE*split)]
        training.y = self.y[:int(DATASET_SIZE*split)]
        training.position = self.position[:int(DATASET_SIZE*split)]
        training.batch_size = self.batch_size
        training.seed = self.seed
        training.img_size = self.img_size

        validation.x = self.x[int(DATASET_SIZE*split):]
        validation.y = self.y[int(DATASET_SIZE*split):]
        validation.position = self.position[int(DATASET_SIZE*split):]
        validation.batch_size = self.batch_size
        validation.seed = self.seed
        validation.img_size = self.img_size


        return training, validation
```

```
# In this object declaration we specify:
# - 1st attr -> path of the dataset
# - 2nd attr -> # of directories to read
# - 3rd attr -> batch size of the dataset
dataset = PoseDataset(path, 7, batch_size = 64)

# TRAINING
dataset, test_dataset = dataset.split()

train_dataset, val_dataset = dataset.split()

#defining hyperparameters

nb_epoch = 20

initial_learning_rate = 1e-4
lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(
    initial_learning_rate,
    decay_steps=10000,
    decay_rate=0.90,
    staircase=True)

#compiling the model
CGAPoseNet_GCAN.compile(optimizer = tf.keras.optimizers.Adam(learning_rate=lr_schedule),
                loss="mse", run_eagerly=True)

es_callback = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=10)

#training
model_train = CGAPoseNet_GCAN.fit(train_dataset,
                            validation_data = val_dataset,
                           epochs=nb_epoch,
                           verbose=1,
                           shuffle=True,
                           callbacks = es_callback,
                            batch_size = 32,
                            reduce_retracing = True)

#plotting losses
loss = model_train.history['loss']
val_loss = model_train.history['val_loss']
epochs = range(0,np.size(loss))


import matplotlib.pyplot as plt
```

```

plt.figure()
plt.plot(epochs, loss, 'b-', label='Training loss')
plt.plot(epochs, val_loss, 'r-', label='Validation loss')
plt.title('Training loss')
plt.legend()
plt.show()



#storing losses
np.save("train_loss.npy", loss)
np.save("val_loss.npy", val_loss)

#saving weights
CGAPoseNet.save('weights.h5')
from tensorflow.python.client import device_lib
device_lib.list_local_devices()
```

```
#Storing the MSE between \hat{M}, M over the test set

MSE = []

tot = 0
cnt = 0
for i, y_batch in enumerate(test_dataset):
    for j, y_real in enumerate(y_batch[1]):
        mse = (np.square(y_real - y_pred[i * test_dataset.batch_size + j])).mean()

        MSE = np.append(MSE, mse)

        #printing the first 20 motors M, \hat{M} if the MSE between them is close
        if cnt < 20 and mse < 0.0008:
            print("original:" , test_dataset)

            X = test_dataset[i][1][j]
            Y = y_pred[i * test_dataset.batch_size + j]

            M_real = X[0] + ga.geom_prod(X[1], ga.e12) + ga.geom_prod(X[2], ga.e13) + ga.geom_prod(X
    [3], ga.e14) + ga.geom_prod(X[4], ga.e23) + ga.geom_prod(X[5], ga.e24) +ga.geom_prod(X[6], ga.
    e34) + ga.geom_prod(X[7], ga.e1234)
            M_pred = Y[0] + ga.geom_prod(Y[1], ga.e12) + ga.geom_prod(Y[2], ga.e13) + ga.geom_prod(Y
    [3], ga.e14) + ga.geom_prod(Y[4], ga.e23) + ga.geom_prod(Y[5], ga.e24) +ga.geom_prod(Y[6], ga.
    e34) + ga.geom_prod(Y[7], ga.e1234)

            print("prediction:", y_pred[i * test_dataset.batch_size + j])
            print("****")
            cnt += 1

        tot += mse

print(tot)
np.save("MSE.npy", MSE)
```

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf

# Assuming ga and other necessary modules are imported and initialized

origin = ga.e3
fig = plt.figure(figsize=(12, 12))
ax = fig.add_subplot(projection='3d')

positional_error = []
rotational_error = []

translation = []
translation_pred = []

```

```
rotation = []
rotation_pred = []

# Assuming y_pred is a flattened list of predictions corresponding to the test dataset
y_pred_index = 0

for batch_index in range(len(test_dataset)):
    # Get a batch of test data
    _, y_test_batch = test_dataset[batch_index]
    position_batch = test_dataset.position[batch_index * test_dataset.batch_size:(batch_index + 1) *
      test_dataset.batch_size]

    for i in range(len(y_test_batch)):
        x = ga.geom_prod(ga.from_scalar(float(position_batch[i][0])), ga.e0) + ga.geom_prod(ga.
    from_scalar(float(position_batch[i][1])), ga.e1) + ga.geom_prod(ga.from_scalar(float(
    position_batch[i][2])), ga.e2)

        X = y_test_batch[i]
        Y = y_pred[y_pred_index][0]

        # Construct M and \hat{M}
        M_real = X[0] + ga.geom_prod(ga.from_scalar(float(X[1])), ga.e01) + ga.geom_prod(ga.
    from_scalar(float(X[2])), ga.e02) + ga.geom_prod(ga.from_scalar(float(X[3])), ga.e03) + ga.
    geom_prod(ga.from_scalar(float(X[4])), ga.e12) + ga.geom_prod(ga.from_scalar(float(X[5])), ga.
    e13) + ga.geom_prod(ga.from_scalar(float(X[6])), ga.e23) + ga.geom_prod(ga.from_scalar(float(X
    [7])), ga.e0123)
        M_pred = Y[0] + ga.geom_prod(ga.from_scalar(Y[1]), ga.e01) + ga.geom_prod(ga.from_scalar(Y
    [2]), ga.e02) + ga.geom_prod(ga.from_scalar(Y[3]), ga.e03) + ga.geom_prod(ga.from_scalar(Y[4]),
     ga.e12) + ga.geom_prod(ga.from_scalar(Y[5]), ga.e13) + ga.geom_prod(ga.from_scalar(Y[6]), ga.
    e23) + ga.geom_prod(ga.from_scalar(Y[7]), ga.e0123)

        # Normalizing
        M_pred = M_pred / mv_length(ga(M_pred))

        # Predicted and real displacement vector \hat{D}, D in spherical space
        S = ga.geom_prod(ga.geom_prod(M_pred, origin), ga.reversion(M_pred))
        T = ga.geom_prod(ga.geom_prod(M_real, origin), ga.reversion(M_real))

        # Predicted and real displacement vector \hat{d}, d in Euclidean space
        s = down1D(S, x)
        t = down1D(T, x)

        # POSITIONAL ERROR
        mae = np.mean(np.abs(np.array([t[1], t[2], t[3]]) - np.array([s[1], s[2], s[3]])))
        positional_error.append(mae)

        translation.extend([t[1], t[2], t[3]])
        translation_pred.extend([s[1], s[2], s[3]])

        # Plotting the camera trace
        ax.scatter(t[1], t[2], t[3], s=20, c="r")
        ax.scatter(s[1], s[2], s[3], s=20, c="b", alpha=0.5)

        Tup = translation_rotor(ga(ga.geom_prod(ga.from_scalar(t[1]), ga.e0)) + ga(ga.geom_prod(ga.
    from_scalar(t[2]), ga.e1)) + ga(ga.geom_prod(ga.from_scalar(t[3]), ga.e2)))
        Sup = translation_rotor(ga(ga.geom_prod(ga.from_scalar(s[1]), ga.e0) + ga.geom_prod(ga.
    from_scalar(s[2]), ga.e1) + ga.geom_prod(ga.from_scalar(s[3]), ga.e2)))

        # Predicted and real rotors \hat{R}, R
        R_pred = ga.geom_prod(ga.reversion(Sup.tensor), M_pred)
        R_real = ga.geom_prod(ga.reversion(Tup.tensor), M_real)

        # ROTATIONAL ERROR
        error = np.degrees(np.arccos(tf.clip_by_value((ga.geom_prod(R_real, ga.reversion(R_pred)))
    [0], -1.0, 1.0)))
        rotational_error.append(error)

        rotation.extend([R_real[0], R_real[6], R_real[7], R_real[10]])
        rotation_pred.extend([R_pred[0], R_pred[6], R_pred[7], R_pred[10]])

```

```
        y_pred_index += 1

plt.show()

# Storing rotational and translational errors
np.save("/content/drive/MyDrive/translation_error_7pkg_30_05_24.npy", positional_error)
np.save("/content/drive/MyDrive/rotational_error_7pkg_30_05_24.npy", rotational_error)

# Storing original and predicted translations
np.save("/content/drive/MyDrive/T_7pkg_30_05_24.npy", translation)
np.save("/content/drive/MyDrive/S_7pkg_30_05_24.npy", translation_pred)

# Storing original and predicted rotations
np.save("/content/drive/MyDrive/R_7pkg_30_05_24.npy", rotation)
np.save("/content/drive/MyDrive/Q_7pkg_30_05_24.npy", rotation_pred)
```

```
#plotting the camera orientation (coefficients e_{01}, e_{02}, e_{12} of rotor R)
fig = plt.figure(figsize=(12, 12))
ax = fig.add_subplot(projection='3d')

N=200
stride=1

u = np.linspace(0, 2 * np.pi, N)
v = np.linspace(0, np.pi, N)
x = np.outer(np.cos(u), np.sin(v))
y = np.outer(np.sin(u), np.sin(v))
z = np.outer(np.ones(np.size(u)), np.cos(v))
ax.plot_surface(x, y, z, linewidth=0.0, alpha = 0.1, cstride=stride, rstride=stride)

ax.scatter(0, 0, 0, c = "k", marker = "s", label = "O")

rotation = np.reshape(rotation, (-1, 4))
rotation_pred = np.reshape(rotation_pred, (-1, 4))
ax.scatter(rotation[:,1], rotation[:, 2], rotation[:,3], s = 15, c = "r")
ax.scatter(rotation_pred[:,1], rotation_pred[:, 2], rotation_pred[:,3], s = 15, c = "b")
plt.show()
```

```
print(np.median(positional_error))
print(np.mean(positional_error))

print(np.median(rotational_error))
print(np.mean(rotational_error))
```

Bibliography

[1] Maths - clifford / geometric algebra. https://euclideanspace.com/maths/algebra/clifford/index.htm.

[2] Geometric algebra, 2024. https://en.wikipedia.org/w/index.php?title=Geometric_algebra&oldid=1225453837.

[3] Geometrica. The pseudoscalar, 2023. Accessed: 2024. https://geometrica.vialattea.net/en/the-pseudoscalar/.

[4] Robin Kahlow. Tensorflow geometric algebra.

[5] Alberto Pepe. CGA PoseNet (7 Scenes), 2024. [Online; accessed 2024] https://github.com/albertomariapepe/CGA-PoseNet/blob/main/CGA_PoseNet_(7_Scenes).ipynb.

[6] Alberto Pepe, Joan Lasenby, and Sven Buchholz. CGAPoseNet+GCAN: A geometric clifford algebra network for geometry-aware camera pose regression. *2024 IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, Jan 2024.

[7] Andrew Seagar. *Numerical Calculations in Clifford Algebra: A Practical Guide for Engineers and Scientists*. Wiley, 2023.

[8] Dmitry Shirokov. On computing the determinant, other characteristic polynomial coefficients, and inverse in clifford algebras of arbitrary dimension. May 2020.

[9] Wikipedia contributors. Clifford algebra — Wikipedia, the free encyclopedia, 2024. https://en.wikipedia.org/w/index.php?title=Clifford_algebra&oldid=1225054795.