1 Introduction

In this document we discuss the notions of Geometric Algebra (GA) (section Theory) and explain how GA is implemented in our Upstride engine (section Implementation).

2 Requirements

Upstride integrates seamlessly with TensorFlow 2.4. It is assumed that the user is experienced with development based on TensorFlow/Keras.

3 Theory

This document is not intended to provide an explanation on GA itself. It rather focuses on describing how the Upstride engine makes it frictionless to use GA in TensorFlow. It also provides the information needed to expand and explore beyond what is currently implemented in the engine in terms of GA sub-algebras. For details on GA, please refer to [3-5].

3.1 Definitions

Before we proceed let's look at some of the definitions and notations we will be using in this document.

- *Blade* - A blade is a generalization of the concept of scalars and vectors. Specifically, a k-blade is any object that can be expressed as the exterior product (or wedge $\wedge$ product) of k vectors, and is of grade k.
- *Multivector* - A multivector is a linear combination of k-blades.
- $\mathbb{G} \circ \mathbb{M}$ (Geometric Algebra over Matrices) - GA represented as real values in the matrix form. This is crucial as numerical computation frameworks like Tensorflow do not support GA yet.
- $\mathbb{M} \circ \mathbb{G}$ (Matrices over Geometric Algebra) - This representation has a difficult integration with frameworks like TensorFlow as we don't have a hardware-friendly data type that can natively represent multivectors.
- x - Inputs to the Neural Network layer
- y - Outputs to the Neural Network layer
- W - Weights to the Neural Network layer
- β_i - Represents the i-th blade
- $\mathbb{R}^3$ - a vector space of dimension 3 over the field $\mathbb{R}$ of real numbers.
- $\wedge$ - exterior product or wedge product

The GA representation is implemented in python as $\mathbb{G} \circ \mathbb{M}$. Code is written in TensorFlow 2.4 using Keras high-level API and supports Python 3.6 or higher.

3.2 Data representation

We stack the *blades* on the batch dimension of the tensor. So, for instance, if we are working in a GA with N blades, a image tensor will have the shape $(N \cdot BS, C, H, W)$, with:

- BS: the batch size
- C: number of channels
- H: height
- W: width

It's important to note:

- We distinguish between $(N \cdot BS)$ and $(BS \cdot N)$. The former means that we concatenate N tensors of size BS, whereas the latter means that we concatenate BS tensors of size N. Thus an image tensor will have its multivector components stacked along the first axis (and not interleaved).
- When performing the conversion between real and Upstride, the only change the user will notice is this increased batch size. Users need to update their model only if they use operations which don't treat independently different images in a batch (which is usually not the case).
- When performing the conversion between real and upstride, the only change the user will notice is this increased batch size.

- Although the above example follows `channels_first` data format convention, the Upstride engine supports `channels_last` as well.

3.3 Upstride data type

Upstride data types are GA-based types, in which GA sub-algebras like complex numbers or quaternions can also be expressed. It's defined by: - an *integer name* `uptype_id`; - an integer tuple triplet containing its geometrical definition `geometric_def`, i.e. the number of blades that square to 1, -1 and 0, in this order and excluding the first blade, which is the scalar; - a tuple of strings with the representation of each blade (e.g. `'12'` represents e_{12} and `''` represents the scalar)

The implementation and further details can be found at `uptypes_utilities.py` and extending to new higher dimensional geometric algebras only requires the instantiation of the desired algebra.

```
@dataclass
class UpstrideDatatype:
  uptype_id: int
  geometrical_def: tuple
  blade_indexes: tuple

UPTYPE0 = UpstrideDatatype(0, (0, 0, 0), ('',))                    # Real numbers
UPTYPE1 = UpstrideDatatype(1, (2, 0, 0), ('', '12'))               # Complex numbers
UPTYPE2 = UpstrideDatatype(2, (3, 0, 0), ('', '12', '23', '13'))  # Quaternions
# In particular for these Upstride data types:
# ''   is the real
# '12' has the same properties as the imaginary component i
# '23' has the same properties as the imaginary component j
# '13' has the same properties as the imaginary component k
```

3.4 Linear layers

This section describes how to implement any linear layer from any GA in TensorFlow, for instance:

- `Dense`
- `Conv2D`
- `DepthwiseConv2D`
- `Conv2DTranspose` (experimental)

The idea is to implement a very generic version of a linear layer, valid for any GA and any linear operation. Then all the specific implementations will benefit from generic implementation.

Note: `Conv2DTranspose` layer is considered experimental and we have not thoroughly validated it.

Note: `SeparableConv2D` is an exception. It is computed by going through 2 linear functions, but moving these 2 linear functions to hypercomplex is not the same as moving the combination of the function to hypercomplex. Currently not supported by the Upstride engine for any GA.

Let's go over an example on how generic linear layer works. In the following two sections, we describe two specific GAs, that is complex numbers and quaternions.

3.4.1 Complex numbers

Let's define:

$x = x_R + ix_I$, the complex input of a linear layer

$y = y_R + iy_I$, the complex output of the same layer

$W = W_R + iW_I$, the kernel of the layer

Computing a linear layer means to compute the product: $y = xW$

So to compute y, we first need to compute all the cross-product between the components of x and the components of W. This can be done in a single call to the TensorFlow API.

Indeed, as we saw in the data representation section, for TensorFlow x is a single tensor which is the concatenation of x_R and x_I on axis 0 (the BS axis). Now we need to concatenate the component of the kernel along the output channel axis.

So, for instance a linear layer will have:

- x_R and x_I are tensors of shape (BS, C)
- x is a tensor of shape $(2 \cdot BS, C)$
- W_R and W_I are tensors of shape (C, C')
- W is a tensor of shape $(C, C' \cdot 2)$

Note: Beware that we distinguish between $(2 \cdot C)$ and $(C \cdot 2)$, as explained in Data representation.

The output of the linear product $y = xW$ will be a tensor of shape $(2 \cdot BS, C' \cdot 2)$ equal to:

$$\begin{bmatrix} x_R W_R, x_R W_I \\ x_I W_R, x_I W_I \end{bmatrix}$$

We need to split the matrix and aggregate the component results:

$y_R = x_R W_R - x_I W_I$

$y_I = x_R W_I + x_I W_R$

3.4.2 Quaternions

Let's look at example of computing the linear product for quaternions.

Given two quaternions $u = u_1 + u_2 i + u_3 j + u_4 k$ and $v = v_1 + v_2 i + v_3 j + v_4 k$, the naive way to compute the product c is :

$$\begin{aligned} c_1 = &\quad u_1 v_1 - u_2 v_2 - u_3 v_3 - u_4 v_4 \\ c_2 = &\quad u_1 v_2 + u_2 v_1 + u_3 v_4 - u_4 v_3 \\ c_3 = &\quad u_1 v_3 + u_3 v_1 + u_4 v_2 - u_2 v_4 \\ c_4 = &\quad u_1 v_4 + u_4 v_1 + u_2 v_3 - u_3 v_2 \end{aligned}$$

This computation includes 16 multiplications and 12 additions. Due to the isomorphism between $\mathbb{M} \circ \mathbb{G}$ and $\mathbb{G} \circ \mathbb{M}$, this corresponds to 16 calls to the TensorFlow linear layer of choice.

3.4.3 General case

Let's work with a generic geometrical algebra defined by a set of blades $\beta_i, i \in [0, n]$.

A blade is a generalization of the concept of scalars and vectors to include multivectors. Specifically, a k-blade is any object that can be expressed as the exterior product (or wedge $\wedge$ product) of k vectors, and is of grade k. For example, the complex number $x = x_R + i x_I$ can be expressed in terms of blades as $x = x_R \beta_0 + x_I \beta_1$.

In general, we can then write our 3 quantities x, y and W as:

- $x = \sum_i x_i \beta_i$
- $y = \sum_i y_i \beta_i$
- $W = \sum_i W_i \beta_i$

Then,

$y = \sum_i \sum_j x_i W_j \beta_i \beta_j$

the product $\beta_i \beta_j$ is defined by the structure of the GA and can be expressed as: $\beta_i \beta_j = s \beta_k, (s, k) \in \{-1, 0, 1\} \times [0, n]$

Note that the set of blades $\beta_i, i \in [0, n]$ does not correspond necessarily to an orthonormal set . For example, given the orthonormal set $\{e_1, e_2, e_3\}$ in the space $\mathbb{R}^3$, a valid basis for a GA on the same space is $\{1, e_1, e_2, e_3, e_1 \wedge e_2, e_2 \wedge e_3, e_3 \wedge e_1, e_1 \wedge e_2 \wedge e_3\}$. We can express this basis in terms of blades as $\beta_i, i \in [0, 7]$ where is β_0 a 0-blade, $\beta_1, \beta_2, \beta_3$ are 1-blades, $\beta_4, \beta_5, \beta_6$ are 2-blades and β_7 is a 3-blade.

Computing the intermediate $x_i W_j$ results is done in the same way as for complex numbers, that is by running a single linear operation with input tensor obtained as concatenated components x_i, and filter obtained as concatenated weights W_j. The matrix is then split and the intermediate results are aggregated according

to the GA formulation, based on an internal function `unit_multiplier`, which computes the product of two unit blades.

```
def unit_multiplier(uptype, i: int, j: int) -> Tuple[int, int]:
```

This function takes as input the Upstride data type `uptype`, the indexes `i` and `j` of the two blades to be multiplied and returns the index `k` and the sign `s` of such product.

Let's dive into the following example: `unit_multiplier(UPTYPE2, 2, 1) == (3, -1)`.

Recall that `UPTYPE2` represents quaternions and that we have `UPTYPE2 = UpstrideDatatype(2, (3, 0, 0), ('', '12', '23', '13'))`. This means that the blade indexes for `UPTYPE2` is a tuple `('', '12', '23', '13')`, elements of which can be respectively interpreted as the quaternions 1, i, j, k. We thus have that `UPTYPE2.blade_indexes[2] == '23'` (interpreted as j) and that `UPTYPE2.blade_indexes[1] == '12'` (interpreted as i).

`unit_multiplier(UPTYPE2, 2, 1)` is interpreted as the multiplication $j \cdot i$ of the quaternions j and i. This is equal to $-k$, and indeed this function call returns `(3, -1)`, which corresponds to: - the quaternion k, as `3` is the first returned value and `UPTYPE2.blade_indexes[3] == '13'` is interpreted as k - a negative sign, as `-1` is the second returned value.

Now, we have everything to code the GenericLinear operation. Note that we do not need to know which linear TensorFlow operation will be used. We can pass this information as an argument.

3.4.4 The Bias handling

In deep learning, we often add a bias term after a linear operation. In Keras, this bias is handled by the linear layer itself, which is a problem here.

Let's take the example of complex number. If the bias was in the TensorFlow layer the computation would be:

$y_R = x_R W_R - x_I W_I + b_R - b_I$

$y_I = x_I W_R + x_R W_I + b_R + b_I$

where b_R, b_I are the bias terms.

This formulation has two issues:

1. we perform two more operations (the bias terms) than needed.
2. If we worked with a single blade of the multivector (e.g: y_R as the final output of the softmax layer) then we would have two variables (b_R and b_I) and only one constraint. This kind of situation can hurt the performance.

One solution to prevent this is to detect when the user applies the bias term and handle it by following the steps:

1. The user asks for the creation of a linear layer with bias
2. We intercept it, save the bias-related parameters and do not forward them to Keras
3. Keras creates the linear layer without the bias
4. We perform the GA multiplication
5. We add the bias at the end with the user's parameters

3.5 Non-linear layers

Examples of non-linear layers are:

- `MaxPooling2D`
- `GlobalAveragePooling2D`
- `Reshape`
- `BatchNormalization`
- `Activation`
- `Flatten`
- `ZeroPadding2D`
- `Add`
- `Concatenate`

For most of the non-linear layers, the equation is simply:

$y = \sum_i f_i(x_i)\beta_i$

where,

y is the output

x_i is the input

β_i is the blade

f_i is the non-linear function

With the way we encode hypercomplex tensors, the non-linear hypercomplex layer is the same as the real layer. The sum in the previous equation is handled naturally as all blades are stacked along the batch axis. However, there are some exceptions, such as BatchNormalization and Dropout. We describe those in the following two sections.

3.5.1 BatchNormalization

For batch normalization to work, the blades should not be stacked along the first axis so that the normalization is not computed as if the blades belonged to the batch axis. We decided to stack the blades on the channel axis.

Also, note that the current (BatchNormalization) implementation works on the several components of the multivector in a correlated way. In case you would like to compute it differently (e.g. for ensuring equal variance in all the blades),

you could opt for BatchNormalizationC (Complex) or BatchNormalizationH (Quaternion) where the real and imaginary parts are independent.

BatchNormalizationC: Trabelsi et al [1]

- Ensures the equal variance for both real and imaginary parts, unlike applying real valued BatchNormalization.

It's recommended to read Section 3.5 in the Deep Complex Networks paper for further details.

BatchNormalizationH: Gaudet et al [2]

- Similar to Complex BatchNormalization. The Quaternion BatchNormalization uses the same idea to ensure all 4 components to have equal variance.

It's recommended to read section 3.4 in the Deep Quaternion Networks paper for further details.

3.5.2 Dropout

We enable two approaches to dropout in the Upstride engine: it can be applied either independently on different blades, or in a "synchronized" fashion, dropping the same input units across blades. By default, `Dropout` employs the independent variant; it can be changed by setting the `synchronized` argument while constructing the layer. The "synchronized" variant requires some tensors reshaping in order to work correctly.

Note: `Dropout` layer is considered experimental and we have not thoroughly tested it.

3.6 Initialization:

Upstride engine supports all the standard weight initialization techniques that are supported by TensorFlow/Keras API.

The weight initialization techniques for Complex (Trabelsi el al [1]) and Quaternion (Gaudet el al [2]) are supported. We have used them in the past, but they are not extensively validated and are considered experimental.

3.7 Data conversion

Two operations are defined for converting data between TensorFlow and Upstride: `TF2Upstride` and `Upstride2TF`. The output of `TF2Upstride` is N times bigger than the input tensor along the batch dimension, where N is the number of blades of the multivector. Conversely, the output of `Upstride2TF` is N times smaller along the batch dimension. The other dimensions depend on the conversion strategy provided to the layer and are not enforced to be the same between the input and the output for all the strategies.

Note: - The blades are stacked (and NOT interleaved) with regards to batch axis. Therefore, for an 2D image the underlying Upstride data representation is of shape `(n_blades * batch_size, channels, height, width)` and not `(batch_size * n_blades, channels, height, width)`. Consequently, to get the first full feature map you need to type `my_tensor[::batch_size]`, whereas to get all the real values you need to type `my_tensor[:batch_size]`.

These 2 operations support several strategies depending on the Upstride type we're using.

For **TF2Upstride**:

- `default` or `basic`: The TensorFlow Tensor is placed in the real component and all the other blades are initialized with zeros.
- `learned`: A resnet block (BN -> ReLU -> CONV -> BN -> ReLU -> CONV) is used to learn the imaginary parts. (Trabelsi et al [1])

For **Upstride2TF**:

- `basic`: outputs a tensor that keeps only the real component of the tensor.
- `concat`: generates a vector by concatenating the imaginary components on the channel dimension.
- `max_pool`: outputs a tensor that takes the maximum values across the real and imaginary parts.
- `avg_pool`: outputs a tensor that takes the average across the real and imaginary parts.

4 Using Upstride Engine

This is a simple neural network that uses Upstride layers:

```
import tensorflow as tf
from upstride.type2.tf.keras import layers

inputs = tf.keras.layers.Input(shape=(224, 224, 3))
x = layers.TF2Upstride()(inputs)
x = layers.Conv2D(32, (3, 3))(x)
x = layers.Activation('relu')(x)
x = layers.Conv2D(64, (3, 3))(x)
x = layers.Activation('relu')(x)
x = layers.Flatten()(x)
x = layers.Dense(100)(x)
x = layers.Upstride2TF()(x)

model = tf.keras.Model(inputs=[inputs], outputs=[x])
```

Upstride engine provides layers that transforms real valued tensor into the following representations.

- `type1` - Complex
- `type2` - Quaternion

Note: we also have a `type0` which is the same as using TensorFlow tensors with real values.

Each module encapsulates an API similar to Keras.

To use it, - import the layers package from the upstride type you want to use, e.g.`import upstride.type2.tf.keras import layers`. (The environment variable `PYTHONPATH` might need to be set as the path to the directory with the engine.) - define an Input to the network. - convert TensorFlow tensor to Upstride type by calling `layers.TF2Upstride` - build the neural network the same way you do with Keras. - at the end of the neural network call `layers.Upstride2TF` to convert from upstride to TensorFlow tensor.

For training and inference, all TensorFlow tools can be used (distribute strategies, mixed-precision training…)

Upstride's engine is divided in three main modules `type0`, `type1`, `type2`. All the modules have the same type of layers and functions. They can be imported in the following way:

```
# type 0
from upstride.type0.tf.keras import layers
# type 1
from upstride.type1.tf.keras import layers
# type 2
from upstride.type2.tf.keras import layers
```

When a network is built with `type0`, it is equivalent to the real valued network or just using TensorFlow layers without Upstride engine.

In the following three sections we describe some specific features implemented in our framework, such as the `factor` parameter, and conversion strategies: `Upstride2TF` and `TF2Upstride`.

4.1 Factor parameter

This is a simple neural network that uses Upstride layers (type 2) with `factor` parameter:

```
import tensorflow as tf
from upstride.type2.tf.keras import layers

factor = 4

inputs = tf.keras.layers.Input(shape=(224, 224, 3))
x = layers.TF2Upstride()(inputs)
x = layers.Conv2D(32 // factor, (3, 3))(x)
```

```
x = layers.Activation('relu')(x)
x = layers.Conv2D(64 // factor, (3, 3))(x)
x = layers.Activation('relu')(x)
x = layers.Flatten()(x)
x = layers.Dense(100 // factor)(x)
x = layers.Upstride2TF()(x)

model = tf.keras.Model(inputs=[inputs], outputs=[x])
```

The `factor` parameter is used to linearly scale the number of feature maps in the linear layers. Higher factor value results in less number of feature maps and vice versa. The `factor` parameter allows to easily tradeoff between the overall accuracy performance and the total number of free parameters of a model. The factor can be applied to all the linear layers except the final logits layer.

For example,

```
x = layers.Conv2D(32 // factor, (3, 3))(x)
```

Here, `factor = 2` reduces the number of channels to 16, `factor = 4` reduces the number of channels to 8.

Due to the way the UpStride engine is implemented, the vanilla approach (without using the `factor` i.e. when `factor == 1`) results in a model that contains more free parameters than its pure TensorFlow counterpart. Usually, to roughly match the number of parameters of a real network with a network based on an algebra with k blades, factor $\sqrt{k}$ should be used (though some layers do not comply with that, for example for a network with only DepthwiseConv2D layers, factor k should be used for that aim).

Our classification-api repository contains a parameter `factor` to automatically scale the models we use. Ensure the value is not so large that it hinders the learning capability of the network. In the example above, the output channels for the first `Conv2D` is `32 // factor`. If `factor = 32` is used then resulting output will be `1`. The network will struggle to extract features with just 1 output channel.

4.2 Initialization

Weight initialization is done similarly to TensorFlow/Keras. We utilize the `kernel_initializer` parameter in the linear layers.

For example,

```
from upstride.type2.keras import layers
# ...
# ...
x = layers.Conv2D(32 // factor, (3, 3), kernel_initializer='glorot')(x)
```

Note: There are specific initializers for each `type0`, `type1` and `type2` Upstride types.

Refer to `InitializersFactory` in the folder `upstride/initializers.py` for further information.

4.3 TF2Upstride

As we have seen from data conversion section there are 2 strategies available to convert TensorFlow tensors to Upstride tensors.

Each strategy can be passed as parameters when invoking `TF2Upstride`

```
x = layers.TF2Upstride(strategy="default")(inputs)
```

```
x = layers.TF2Upstride(strategy="learned")(inputs)
```

4.4 Upstride2TF

As we have seen from data conversion section there are 4 strategies available to convert Upstride tensors to TensorFlow tensors.

Each strategy can be passed as parameters when invoking `Upstride2TF`

e.g:

```
x = layers.Upstride2TF(strategy="default")(x)
```

```
x = layers.Upstride2TF(strategy="concat")(x)
```

```
x = layers.Upstride2TF(strategy="max_pool")(x)
```

```
x = layers.Upstride2TF(strategy="avg_pool")(x)
```

4.5 Tests

To run tests, install pip package `pytest` and run the following in the main directory:

```
python test.py
```

Pytest offers the following useful command line arguments:

- `-v` increases verbosity, e.g. prints test names
- `-s` prints standard output
- `-k Simple` runs only tests with "Simple" in their name
- `-k "Conv2D and (not basic or not grouped)"` runs only tests which have "Conv2D" in their name and have neither "basic" or "grouped" in their name
- `-m exhaustive` runs only tests marked as "exhaustive" in the code
- `... filename` runs only tests from a given filename

5 References

1. Chiheb Trabelsi, Olexa Bilaniuk, Ying Zhang, Dmitriy Serdyuk, Sandeep Subramanian, João Felipe Santos, Soroush Mehri, Negar Rostamzadeh, Yoshua Bengio, Christopher J Pal. "Deep Complex Networks". In Internation Conference on Learning Representations (ICLR), 2018

2. Gaudet, Chase J., and Anthony S. Maida. "Deep quaternion networks." 2018 International Joint Conference on Neural Networks (IJCNN). IEEE, 2018.

3. Hitzer, Eckhard. "Introduction to Clifford's geometric algebra." Journal of the Society of Instrument and Control Engineers 51, no. 4 (2012): 338-350.

4. Hildenbrand, Dietmar. "Foundations of geometric algebra computing." In AIP Conference Proceedings, vol. 1479, no. 1, pp. 27-30. American Institute of Physics, 2012.

5. Dorst, Leo, Chris Doran, and Joan Lasenby, eds. Applications of geometric algebra in computer science and engineering. Springer Science & Business Media, 2012.