WaveTF: a fast 2D wavelet transform for machine learning in Keras[*]

Francesco Versaci

CRS4, Cagliari, Italy

March 15, 2021

Abstract

The wavelet transform is a powerful tool for performing multiscale analysis and it is a key subroutine in countless applications, from image processing to astronomy. Recently, it has extended its range of users to include the ever growing machine learning community. For a wavelet library to be efficiently adopted in this context, it needs to provide transformations which can be integrated seamlessly in already existing machine learning workflows and neural networks, being able to leverage the same libraries and run on the same hardware (e.g., CPU vs GPU) as the rest of the machine learning pipeline, without impacting training and evaluation performance. In this paper we present WaveTF, a wavelet library available as a Keras layer, which leverages TensorFlow to exploit GPU parallelism and can be used to enrich already existing machine learning workflows. To demonstrate its efficiency we compare its raw performance against other alternative libraries and finally measure the overhead it causes to the learning process when it is integrated in an already existing Convolutional Neural Network.

1 Introduction

The wavelet transform [18] is a powerful tool for multiscale analysis. It produces a mix of time/spatial and frequency data and has countless applications in many areas of science, including image compression, medical imaging, finance, geophysics, and astronomy [2]. Recently, the wavelet transform has also been applied to machine learning, for instance to extract the feature set to be used by a standard learning workflow [3, 16] and to enhance Convolutional Neural Networks (CNNs) [4, 12, 21, 15]. For many of these applications, and machine learning in particular, parallel execution on GPGPU accelerators is of critical importance to ensure the tractability of real-world problems. Therefore, a library that provides wavelet transform functionality for this context must efficiently integrate into existing computational pipelines, mitigating the loss of performance due to the cost of exchanging data between memories in different phases of the computation – e.g., if our pipeline runs on a GPU we would like to execute the

[*]The final authenticated version of this article is available online at `https://doi.org/10.1007/978-3-030-68763-2_46`

wavelet on the same device, without the need to repeatedly move data between the GPU and the main memory.

In this work we present WaveTF, a library providing a fast 1D and 2D wavelet implementation that provides scalable parallel execution on CPU and GPU devices. WaveTF enables full GPU execution of computational pipelines including wavelet transforms. The library is built on top of the popular TensorFlow framework and is exposed as a Keras layer, making it easy to integrate into existing Python workflows based on these widely adopted frameworks. Our evaluation shows that WaveTF improves upon the state of the art by providing faster routines and by adding only a negligible overhead to machine learning applications.

The rest of this manuscript is structured as follows. In Sec. 2 we provide a description of wavelet transforms, followed by a discussion of the related work in Sec. 3. Section 4 describes the implementation of the WaveTF library, while an evaluation of its performance is presented in Sec. 5. Finally, Sec. 6 points the reader to the software and Sec. 7 concludes the manuscript.

2 Background

2.1 Wavelet transform

Wavelet transforms are a family of invertible signal transformations that, given an input signal evolving in time, produce an output which mixes time and frequency information [8]. This paper will only focus on discrete transformations.

2.1.1 Haar transform

The simplest wavelet transform is the Haar transform, which, given in input a signal $x = (x_0, \dots, x_{n-1})$ (with n even) produces as output

$$H(x) := (l_0, \dots, l_{\frac{n}{2}-1}, h_0, \dots, h_{\frac{n}{2}-1}) = (l(x), h(x)) \quad ,$$

where

$$l_i := \frac{x_{2i} + x_{2i+1}}{\sqrt{2}} \quad , \qquad\qquad h_i := \frac{x_{2i} - x_{2i+1}}{\sqrt{2}} \quad , \tag{1}$$

with l_i and h_i containing low and high frequency components localized at times $2i$ and $2i + 1$ of the original signal. Note that when the input size is not even, the signal must be extended using some form of padding. The wavelet transform is often iterated on the low components to carry out a multiscale analysis:

$$H^d(x) := \left(H^{d-1}(l(x)), h(x)\right) \quad , \qquad\qquad \text{with } H^0(x) := x \quad . \tag{2}$$

2.1.2 Daubechies wavelet

The Haar transform can be extended so that l_i and h_i are linear functions of more than two terms, as done by the following Daubechies-N=2 (DB2) transform (see [7] for details):

$$\begin{aligned} l_i &= \lambda_0 x_{2i-1} + \lambda_1 x_{2i} + \lambda_2 x_{2i+1} + \lambda_3 x_{2i+2} \quad , \\ h_i &= \mu_0 x_{2i-1} + \mu_1 x_{2i} + \mu_2 x_{2i+1} + \mu_3 x_{2i+2} \quad , \end{aligned}$$

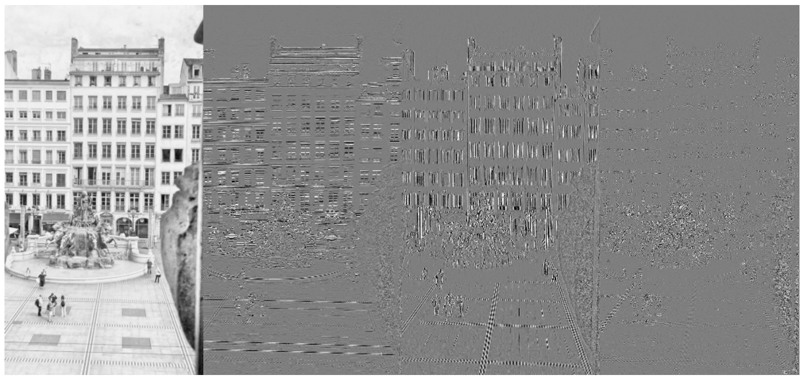

Figure 1: The four components (LL, LH, HL and HH) of a two-dimensional Daubechies-N=2 wavelet transform. LH, HL and HH have been contrasted to emphasize their structure.

where

$$\begin{array}{llll} \lambda_0 = \frac{1+\sqrt{3}}{2\sqrt{2}} & \lambda_1 = \frac{3+\sqrt{3}}{2\sqrt{2}} & \lambda_2 = \frac{3-\sqrt{3}}{2\sqrt{2}} & \lambda_3 = \frac{1-\sqrt{3}}{2\sqrt{2}} \\ \mu_0 = \lambda_3 & \mu_1 = -\lambda_2 & \mu_2 = \lambda_1 & \mu_3 = -\lambda_0 \quad , \end{array} \tag{3}$$

and vectors $\vec{\lambda} := (\lambda_0, \lambda_1, \lambda_2, \lambda_3)$ and $\vec{\mu} := (\mu_0, \mu_1, \mu_2, \mu_3)$ being orthonormal.

When working with larger kernels (4×2 in this case, where Haar was 2×2) the border of the signal must always be extended with padding to be able to invert the transformation.

2.1.3 Multidimensional transform

The wavelet transform is extended to multidimensional signals by executing it orderly in all the dimensions. For instance, in the two-dimensional case the input is a matrix and the output is obtained by first transforming the rows and then the columns; it is thus formed by 4 matrices (conventionally called LL, LH, HL, and HH), containing the low and high components for the horizontal and vertical directions (an example can be seen in Fig. 1). As with the 1D case, the multidimensional transformations can also be iterated for perform a multilevel analysis (see Fig. 2). When the input is a multichannel image (e.g., RGB or HSV), transformations are performed independently for each channel.

WaveTF supports batched, multichannel inputs, i.e., for the two-dimensional case it accepts a tensor of shape [batch_size, dim_x, dim_y, channels], and returns a tensor of shape [batch_size, new_x, new_y, 4×channels].

2.2 TensorFlow and Keras

TensorFlow [1] is a powerful framework for efficiently manipulating multidimensional arrays (i.e., tensors) in parallel, and it provides APIs for Python, C++, Java and JavaScript. It has been developed as a fast and scalable framework for machine learning, and for this purpose it is complemented by the higher level Keras library [6]. However, TensorFlow offers many powerful algebraic routines which can be used independently of the application and its Python API can be seen as a parallel, GPU-enabled version of NumPy [23], with which it shares many similarities in the syntax and names of its methods. Note that TensorFlow supports a wide variety of computing hardware: it can run on multiple CPUs,

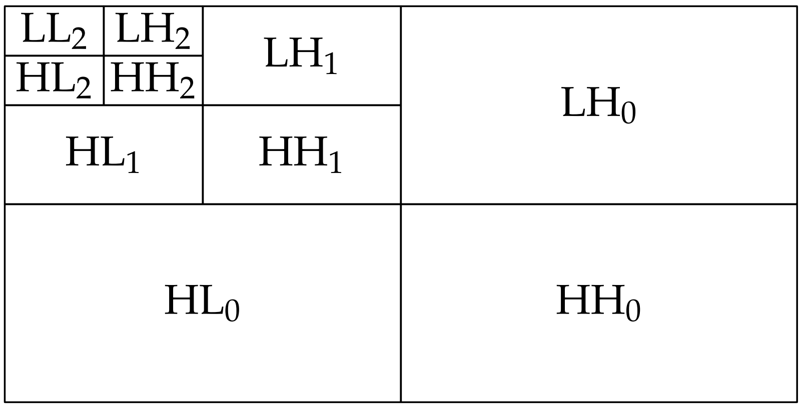

Figure 2: Recursive structure of a 2D multilevel wavelet transform: each new level is obtained by transforming the LL component of the previous level (which can be preserved or discarded, as in this case).

GPUs and also on specialized ASICs known as TPUs [13], which are now available for end-users as part of the Google Cloud infrastructure.

We have chosen to implement WaveTF leveraging TensorFlow's rich API and scalability, so that it can easily exploit available parallelism, be easily and efficiently integrated with other programs that use TensorFlow and Keras and provide its functions to the growing machine learning community.

3 Related work

In this section we briefly describe three alternative wavelet libraries available for Python and published as open source software: PyWavelets, pypwt and TF-Wavelets. In Sec. 5.1 we will compare their raw performance to our library.

3.1 PyWavelets

PyWavelets [14] is probably the most widely used Python library for wavelet transforms. Its core routines are written in C and made available to Python through Cython. It supports 1D and 2D transformations and provides over 100 built-in wavelet kernels and 9 signal extension modes. Unlike WaveTF, it is a sequential library and runs exclusively on CPUs.

3.2 pypwt

pypwt [20] is a Python wrapper of PDWT, which in turn is a C++ wavelet transform library, written using the parallel CUDA platform and running on NVIDIA GPUs. It implements 1D and 2D transforms, (though it does not support batched 2D transforms) supports 72 wavelet kernels and adopts periodic padding for signal extension.

3.3 TF-Wavelets

TF-Wavelets [10, 17] is a Python wavelet implementation which, like WaveTF, leverages the TensorFlow framework. It features two wavelet kernels (Haar and DB2) and implements periodic padding for signal extension. It is the library more conceptually similar to ours, allowing, for instance, both input and output to reside in GPU memory, and it is thus the best match for a raw performance comparison against WaveTF. However, it lacks support of batched, multichannel, 2D transforms, which are typically required for machine learning applications in Keras. As a consequence, it does not provide a network layer for that framework.

4 Implementation

WaveTF is written in Python using the TensorFlow API. It exposes its functions via a Keras layer which can either be called directly or can be plugged easily into already existing neural networks. The library currently implements the Haar (Eq. (1)) and DB2 (Eq. (3)) wavelet kernels – which are the two most commonly used ones. To handle border effects, anti-symmetric-reflect padding (known as *asym* in MATLAB) has been implemented, which extends the signal by preserving its first-order finite difference at the border. WaveTF supports both 32- and 64-bit floats transparently at runtime.

4.1 Direct transform

In order to efficiently implement the wavelet transform in TensorFlow we first reshape it as a matrix operation. Let us consider, as an example, the 1D DB2 transform with input size n, where n is a multiple of 4. The original formulation of the transform presented in Sec. 2.1.2 can be rewritten as a matrix multiplication in the following form:

$$\begin{pmatrix} l_0 & h_0 \\ l_1 & h_1 \\ l_2 & h_2 \\ l_3 & h_3 \\ \vdots & \vdots \\ l_{\frac{n}{2}-1} & h_{\frac{n}{2}-1} \end{pmatrix} = \begin{pmatrix} 2x_0 - x_1 & x_0 & x_1 & x_2 \\ x_1 & x_2 & x_3 & x_4 \\ x_3 & x_4 & x_5 & x_6 \\ x_5 & x_6 & x_7 & x_8 \\ \vdots & \vdots & \vdots & \vdots \\ x_{n-3} & x_{n-2} & x_{n-1} & 2x_{n-1} - x_{n-2} \end{pmatrix} \begin{pmatrix} \lambda_0 & \mu_0 \\ \lambda_1 & \mu_1 \\ \lambda_2 & \mu_2 \\ \lambda_3 & \mu_3 \end{pmatrix} .$$

In order to generate the data matrix above we need to group the data vector by 4 and interleave it with a copy of itself, shifted left by two (plus some constant operations for the padding at the border). This operation can be implemented with the `reshape`, `concat` and `stack` methods provided by TensorFlow. Alternatively, the specialized `conv1d` method can be employed instead of the standard matrix multiplication, somewhat simplifying the data rearrangement. We have implemented both the variants and we have seen that the convolution one is faster in all considered cases, except for the 1D-Haar transform (for which we have thus adopted the matrix multiplication algorithm).

Note that when n is not a multiple of 4, the border values are arranged slightly differently, but the procedural steps remain the same.

4.2 Inverse transform

In this section we show how to properly invert the DB2 wavelet transform, taking into account the border effects while keeping the padding as small as possible. This is done both to justify the exact algorithmic steps we adopted and to offer a future reference for alternative implementations by other authors. To the best of our knowledge the following derivation, at this level of detail, is original, though it is likely that it might be already present, at least implicitly, in the vast literature on Wavelet transform.

To better understand how to properly handle the border effect when computing the inverse, let us reshape the transformation above in a slightly different way: i.e., as $\vec{w} = W\vec{x} = KP\vec{x}$, with K being the $n \times (n+2)$ kernel matrix and P the $(n+2) \times n$ (anti-symmetric-reflect) padding matrix:

$$\underbrace{\begin{pmatrix} l_0 \\ h_0 \\ l_1 \\ h_1 \\ \vdots \\ l_{\frac{n}{2}-1} \\ h_{\frac{n}{2}-1} \end{pmatrix}}_{\vec{w}} = \underbrace{\begin{pmatrix} \lambda_0 & \lambda_1 & \lambda_2 & \lambda_3 & & & & \\ \mu_0 & \mu_1 & \mu_2 & \mu_3 & & & & \\ & & \lambda_0 & \lambda_1 & \lambda_2 & \lambda_3 & & \\ & & \mu_0 & \mu_1 & \mu_2 & \mu_3 & & \\ & & & & \ddots & \ddots & \ddots & \ddots \end{pmatrix}}_{K} \underbrace{\begin{pmatrix} 2 & -1 & & & & \\ 1 & & & & & \\ & 1 & & & & \\ & & \ddots & & & \\ & & & & 1 & \\ & & & & & 1 \\ & & & & -1 & 2 \end{pmatrix}}_{P} \underbrace{\begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix}}_{\vec{x}} .$$

We can then decompose K, P and W in (non-square) blocks (with each block shape shown between parentheses):

$$K = \left(\begin{array}{c|c|c} \underset{(4\times 3)}{K_{00}} & \underset{(4\times n-4)}{K_{01}} & \underset{(4\times 3)}{0} \\ \hline \underset{(n-8\times 3)}{0} & \underset{(n-8\times n-4)}{K_{11}} & \underset{(n-8\times 3)}{0} \\ \hline \underset{(4\times 3)}{0} & \underset{(4\times n-4)}{K_{21}} & \underset{(4\times 3)}{K_{22}} \end{array}\right) , \quad P = \left(\begin{array}{c|c|c} \underset{(3\times 2)}{P_{00}} & \underset{(3\times n-4)}{0} & \underset{(3\times 2)}{0} \\ \hline \underset{(n-4\times 2)}{0} & \underset{(n-4\times n-4)}{\mathbb{I}_{n-4}} & \underset{(n-4\times 2)}{0} \\ \hline \underset{(3\times 2)}{0} & \underset{(3\times n-4)}{0} & \underset{(3\times 2)}{P_{22}} \end{array}\right) ,$$

$$W = KP = \left(\begin{array}{c|c|c} \underset{(4\times 2)}{K_{00}P_{00}} & \underset{(4\times n-4)}{K_{01}} & \underset{(4\times 2)}{0} \\ \hline \underset{(n-8\times 2)}{0} & \underset{(n-8\times n-4)}{K_{11}} & \underset{(n-8\times 2)}{0} \\ \hline \underset{(4\times 2)}{0} & \underset{(4\times n-4)}{K_{21}} & \underset{(4\times 2)}{K_{22}P_{22}} \end{array}\right) .$$

To invert W we first note that K_{11} has orthonormal rows and thus admits its transpose as a right inverse: $K_{11}K_{11}^t = \mathbb{I}_{n-4}$. Furthermore, $W_{00} := K_{00}P_{00}$ and $W_{22} := K_{22}P_{22}$ have linearly independent columns and thus admit a (Moore–Penrose) left inverse: $W_{00}^+ W_{00} = W_{22}^+ W_{22} = \mathbb{I}_2$. Finally, because of the choice of coefficients in Eq. (3), we have

$$W_{00}^+ K_{01} = K_{01}^t W_{00} = W_{22}^+ K_{21} = K_{21}^t W_{22} = 0 \quad ,$$
$$K_{01}^t K_{01} + K_{11}^t K_{11} + K_{21}^t K_{21} = \mathbb{I}_{n-4} \quad .$$

Table 1: Hardware configuration of the test machine.

CPU	Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz (24 SMT cores)
RAM	250 GiB
GPU	GeForce RTX 2080 Ti (11 GB GDDR6)

We can now verify that W is inverted by

$$W^{-1} = \left(\begin{array}{c|c|c} \underset{(2\times4)}{W^{+}_{00}} & \underset{(2\times n-8)}{0} & \underset{(2\times4)}{0} \\ \hline \underset{(n-4\times4)}{K^{t}_{01}} & \underset{(n-4\times n-8)}{K^{t}_{11}} & \underset{(n-4\times4)}{K^{t}_{21}} \\ \hline \underset{(2\times4)}{0} & \underset{(2\times n-8)}{0} & \underset{(2\times4)}{W^{+}_{22}} \end{array}\right) ,$$

and that we can compute its non-border elements similarly to the direct transform case:

$$\begin{pmatrix} x_1 & x_2 \\ x_3 & x_4 \\ \vdots & \vdots \\ x_{n-3} & x_{n-2} \end{pmatrix} = \begin{pmatrix} l_0 & h_0 & l_1 & h_1 \\ l_1 & h_1 & l_2 & h_2 \\ \vdots & \vdots & \vdots & \vdots \\ l_{\frac{n}{2}-3} & h_{\frac{n}{2}-3} & l_{\frac{n}{2}-2} & h_{\frac{n}{2}-2} \\ l_{\frac{n}{2}-2} & h_{\frac{n}{2}-2} & l_{\frac{n}{2}-1} & h_{\frac{n}{2}-1} \end{pmatrix} \begin{pmatrix} \lambda_2 & \lambda_3 \\ \mu_2 & \mu_3 \\ \lambda_0 & \lambda_1 \\ \mu_0 & \mu_1 \end{pmatrix}$$

and its border values as:

$$\begin{pmatrix} x_0 \\ x_1 \end{pmatrix} = W^{+}_{00} \begin{pmatrix} l_0 \\ h_0 \\ l_1 \\ h_1 \end{pmatrix} , \qquad \begin{pmatrix} x_{n-2} \\ x_{n-1} \end{pmatrix} = W^{+}_{22} \begin{pmatrix} l_{\frac{n}{2}-2} \\ h_{\frac{n}{2}-2} \\ l_{\frac{n}{2}-1} \\ h_{\frac{n}{2}-1} \end{pmatrix} .$$

4.3 Correctness

In addition to the formal derivation given above, we have tested our implementation for consistency against PyWavelets, and we have composed direct and inverse transforms to verify that they result in an identity map (up to numerical precision errors). The randomized test code is included with the source code and is runnable with the *pytest* framework [19].

Note that, contrary to PyWavelets, WaveTF always uses a minimal padding when transforming: e.g., WaveTF's output for an input vector of size 10 is a 2×5 matrix, whereas PyWavelets produces a 2×6 matrix when using the DB2 kernel and a 2×5 one when using the Haar kernel.

5 Performance results

The performance of WaveTF has been tested in two ways:

- by executing raw signal transforms, leaving the output data available for the user either in RAM or in the GPU memory;

Table 2: Versions of the software used in this work.

Package	Version	Source
WaveTF	0.1	`https://github.com/crs4/WaveTF`
PyWavelets	1.1.1	`https://github.com/PyWavelets/pywt`
pypwt	d225e09	`https://github.com/pierrepaleo/pypwt`
TF-Wavelets	ac4f357	`https://github.com/UiO-CS/tf-wavelets`
TensorFlow	2.1.0	`https://www.tensorflow.org/install`
CUDA	V10.1.243	`https://developer.nvidia.com/cuda-downloads`
NVIDIA driver	435.21	`https://www.nvidia.com/Download/Find.aspx`

- as a Keras layer, integrated in a simple neural network for a training task.

In the first test, we also computed the same transformations with the PyWavelets, pypwt and TF-Wavelets libraries to compare their performance to WaveTF's.

In order to better exploit the computation power provided by the GPU [5], the tests have been run with single-precision floating-point types: `np.float32` for PyWavelets, `tf.float32` for WaveTF and TF-Wavelets, and pypwt compiled to use 32-bit floats.

The hardware and software used in the tests are detailed in Tables 1 and 2.

5.1 Raw transformation

PyWavelets operates in RAM and pypwt uses RAM for input and output but runs its computation in the GPU. On the other hand, WaveTF and TF-Wavelets operate on TensorFlow tensors which, when GPUs are available and used, reside in the GPU memory. We expect to see this difference reflect on the runtimes, because of the overhead of moving data between GPU and RAM.

We have recorded the wall clock time of one- and two-dimensional Haar and DB2 wavelet transforms using WaveTF, PyWavelets and pypwt and TF-Wavelets. For WaveTF, we have measured both the time required when leaving the data in the GPU memory and when input and output are required to be in main memory. For TF-Wavelets we have instead focused on the fastest case of working only on GPU memory, to offer a fair comparison for WaveTF. The test procedure for the one-dimensional case is as follows:

- A random array of n elements is created, with n ranging from $5 \cdot 10^6$ to 10^8,
- For the non-batched case the array is used as is (i.e., shape = $[n]$), for the batched case it is reshaped to $[b, n/b]$, with $b = 100$,
- The transform, on the same input array, is executed from a minimum of 500 up to a maximum of 10000 times for smaller data size; the total time is measured and the time per iteration is recorded.

For the two-dimensional case, the input matrix is chosen to be as square as possible given the target total size of n elements, i.e., shape = $\left[\lfloor\sqrt{n}\rfloor, \lceil\sqrt{n}\rceil\right]$.

Note that we have not measured the time to execute a single transformation, but instead the time to execute many of them grouped together (up to 10000), because the single execution time when working in GPU memory would have been completely overshadowed by the setup time required for the library calls.

Table 3: Runtimes, for the largest tested size, i.e., 10^8 elements, normalized against WaveTF.

Operation	**WaveTF**	**TF-Wavelets**	**PyWavelets**	**pypwt**
1D Haar	**1**	2.98	74.81	73.55
1D DB2	**1**	1.58	42.91	36.04
1D Haar, batched	**1**	3.21	73.69	72.37
1D DB2, batched	**1**	1.62	39.85	33.63
2D Haar	**1**	2.58	45.59	14.30
2D DB2	**1**	2.30	44.61	12.27
2D Haar, batched	**1**	*n.a.*	42.55	*n.a.*
2D DB2, batched	**1**	*n.a.*	41.08	*n.a.*

The standard deviations for these grouped measures are all well below 1%, so they are not shown in the plots.

5.1.1 Discussion

As can be seen from the data in Fig. 3 and Table 3, there is a huge gap in performance between PyWavelets and pypwt and the TensorFlow programs. The performance of PyWavelets is explained by the fact that it is a serial program and that it does not exploit the parallelism available in the GPU. pypwt, on the other hand, does use the GPU but incurs a big overhead caused by the data movement between GPU and main memory – as demonstrated by the similar performance achieved by WaveTF when it is forced to have both input and output in RAM.

When working directly in GPU memory WaveTF and TF-Wavelets have a big performance advantage over the other evaluated libraries, with WaveTF being about 70x faster than PyWavelets and pypwt on 1D Haar and 30-40x on 1D DB2. For the 2D cases WaveTF has a speedup greater than 40x over PyWavelets and a 12-14x one over pypwt. This test scenario mirrors the common situation in TensorFlow-based machine learning workflows using wavelet transforms.

The speedup of WaveTF against TF-Wavelets is still quite impressive, considered that both libraries adopt the same general strategy, and it ranges from 1.6x up to 3.2x. This improvement is mainly due to a careful algorithmic implementation as to avoid redundant computations.

5.2 Machine learning

In this section we quantify the overhead of integrating WaveTF in machine learning workflows. For this purpose we consider a classification problem on a standard image dataset solved by a simple CNN. In our experiment we measure the training and evaluation times before and after enriching the CNN with wavelet layers.

For this test we have adopted the Imagenette2-320 dataset [11] – a subset of 10 classes from ImageNet [22] – consisting of 9469 training and 3925 validation RGB images. For the classification task we used a basic CNN network featuring 5 levels of convolution, followed by downscaling which halves the spatial feature dimensions at each level (i.e., 320x320 $\rightarrow$ 160x160 $\rightarrow$ 80x80 $\rightarrow$ 40x40 $\rightarrow$ 20x20). To enrich this network with the wavelet transform, each newly downscaled layer

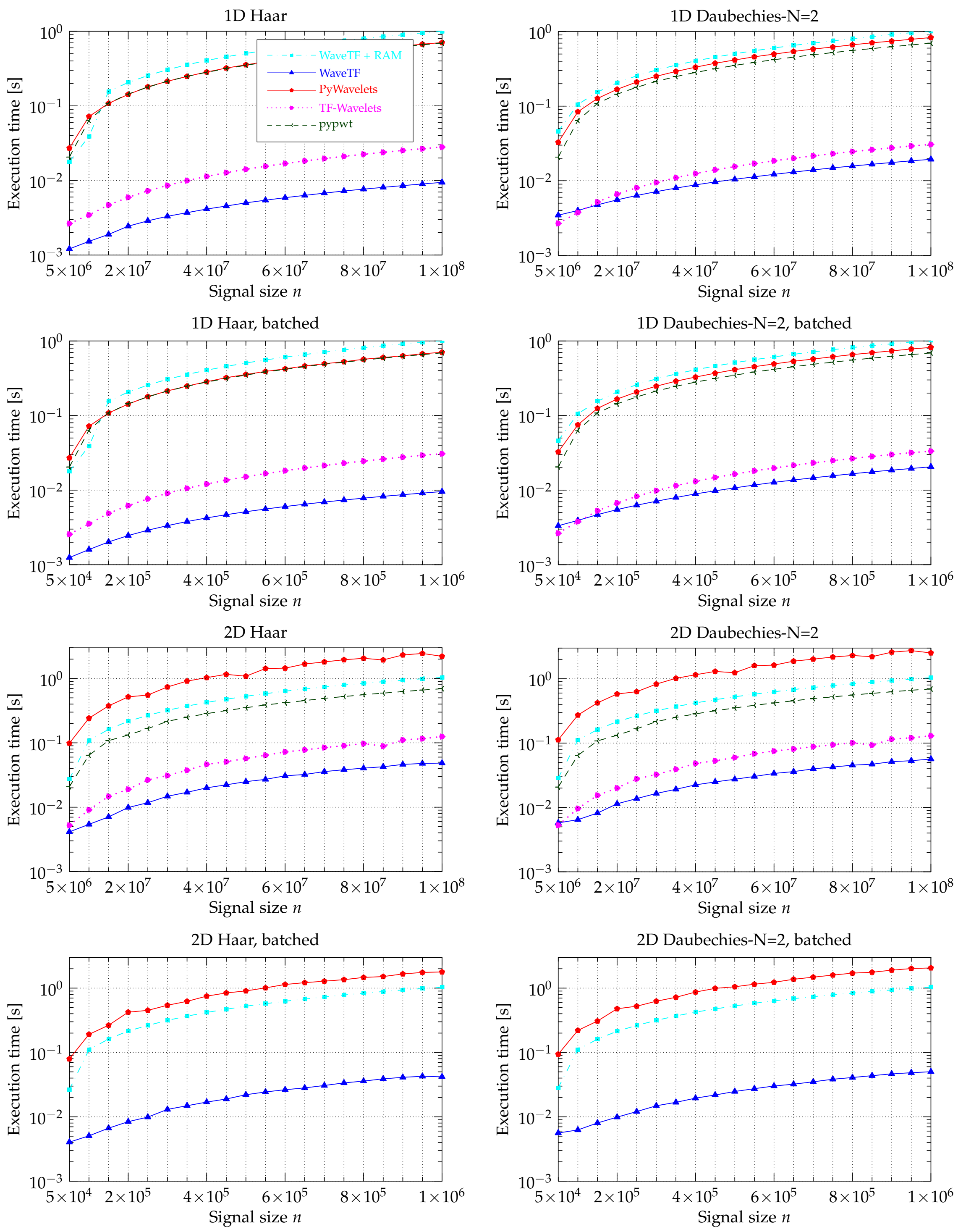

Figure 3: Runtime of wavelet transforms: WaveTF vs. PyWavelets vs. pypwt vs. TF-Wavelets – Wall time of execution, for Haar and Daubechies-N=2 kernels, one- and two-dimensional, batched and non-batched. For WaveTF we show two runtimes: i) when working directly in GPU memory, ii) when input and output are required to be in RAM. Standard deviation is below 1% in all cases.

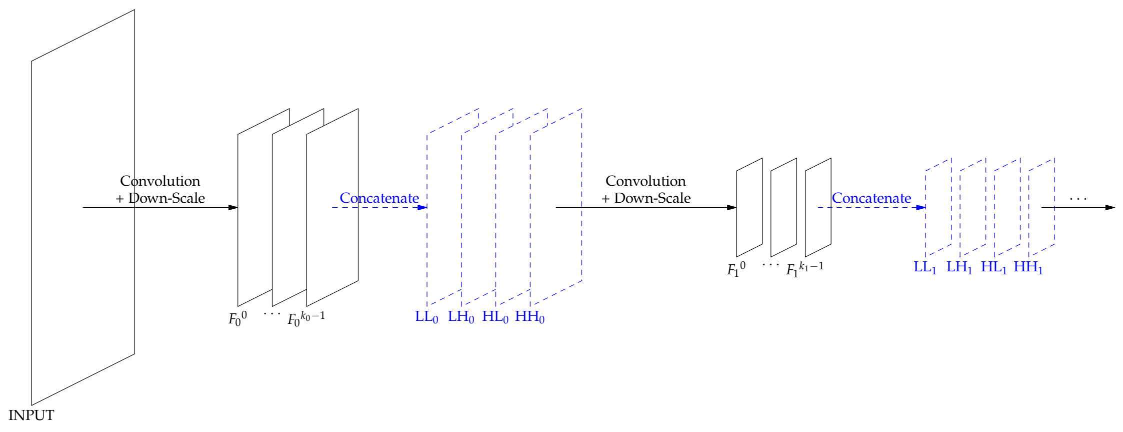
Convolution
+ Down-Scale
Concatenate
Convolution
+ Down-Scale
Concatenate
$F_0{}^0$ ··· $F_0{}^{k_0-1}$
LL$_0$ LH$_0$ HL$_0$ HH$_0$
$F_1{}^0$ ··· $F_1{}^{k_1-1}$
LL$_1$ LH$_1$ HL$_1$ HH$_1$
···
INPUT

Figure 4: The first steps of a wavelet-enriched CNN: after down-scaling at level l, the k_l output features $(F_l^0, \ldots, F_l^{k_l-1})$ are concatenated with the wavelet components (LL$_l$, LH$_l$, HL$_l$, HH$_l$,) at the corresponding level of scale, before the following convolution is performed.

Table 4: Running times (with standard deviation) of a 5-level CNN on the Imagenette dataset, with and without enriching the network with wavelet features computed by the WaveTF Keras layer.

Operation	**Baseline**	**With wavelet**	**Overhead**
Training time [s]	1581 ± 18	1593 ± 14	<1%
Evaluation time [s]	78.5 ± 0.5	78.7 ± 0.8	<1%

is concatenated with the corresponding level from the output of WaveTF (see Fig. 4), launched iteratively as shown in Eq. (2). This approach has been used, e.g., for improving texture classification [9].

Since the objective of our experiment is only to quantify the computational overhead of adding wavelet features via WaveTF to the network, we disabled all forms of data augmentation for the training – these procedures would add their own considerable overhead which would confound our results. To compute the training overhead, we measured the wall clock time required to train the model for 20 epochs, with and without enriching the network with the wavelet features. We repeated this training process 20 times (after a first, unmeasured run, used to set the memory buffering to a stationary state). On the other hand, to measure the overhead incurred in evaluation we used the trained network to evaluate all the images in the dataset and repeated the process 20 times.

5.2.1 Discussion

As can be seen from the results shown in Table 4, the overhead of adding wavelet features to the existing 5-level CNN is below 1%, both in training and evaluation, thus allowing its use at an almost negligible cost.

6 Software availability

WaveTF is released under the open source Apache License Version 2.0. Its source code is available for download from the GitHub platform, together with accompanying documentation and some usage examples, which also include the CNN used in this paper. The link to the GitHub repository is shown in Table 2.

7 Conclusion and future work

In this work we have presented an efficient wavelet library which leverages TensorFlow and Keras to exploit GPU parallelism and allows for easy integration in already existing machine learning workflows. Since the wavelet transform is characterized by high parallelism and low computational complexity (time complexity being $O(n)$ for an input of size n), minimizing communication is pivotal to achieve good performance, and in this work we have shown how to do it by limiting the transfer between GPU and memory whenever is possible.

In future we plan to extend the library to include other popular wavelet kernels and padding extensions, as well as extending it to 3D signals.

Acknowledgments

I'd like to thank G. Busonera and L. Pireddu for reviewing the draft and S. Leo for his suggestions on structuring the Python code. This work has been funded by the European Commission under the H2020 program grant DeepHealth (n. 825111).

References

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al.: Tensorflow: A system for large-scale machine learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). pp. 265–283 (2016)

[2] Addison, P.S.: The illustrated wavelet transform handbook: introductory theory and applications in science, engineering, medicine and finance. CRC press (2017)

[3] Amin, H.U., Malik, A.S., Ahmad, R.F., Badruddin, N., Kamel, N., Hussain, M., Chooi, W.T.: Feature extraction and classification for eeg signals using wavelet transform and machine learning techniques. Australasian physical & engineering sciences in medicine **38**(1), 139–149 (2015)

[4] Bruna, J., Mallat, S.: Invariant scattering convolution networks. IEEE transactions on pattern analysis and machine intelligence **35**(8), 1872–1886 (2013)

[5] Burgess, J.: Rtx on—the nvidia turing gpu. IEEE Micro **40**(2), 36–44 (2020)

[6] Chollet, F., et al.: Keras: The python deep learning library. Astrophysics Source Code Library (2018)

[7] Daubechies, I.: Orthonormal bases of compactly supported wavelets. Communications on pure and applied mathematics **41**(7), 909–996 (1988)

[8] Daubechies, I.: Ten lectures on wavelets, vol. 61. Siam (1992)

[9] Fujieda, S., Takayama, K., Hachisuka, T.: Wavelet convolutional neural networks for texture classification (2017)

[10] Haug, K.M.: Stability of Adaptive Neural Networks for Image Reconstruction. Master's thesis (2019)

[11] Howard, J.: Fastai's imagenette and imagewoof datasets (2020), `https://github.com/fastai/imagenette`

[12] Huang, H., He, R., Sun, Z., Tan, T.: Wavelet-srnet: A wavelet-based cnn for multi-scale face super resolution. In: The IEEE International Conference on Computer Vision (ICCV) (10 2017)

[13] Jouppi, N., Young, C., Patil, N., Patterson, D.: Motivation for and evaluation of the first tensor processing unit. IEEE Micro **38**(3), 10–19 (2018)

[14] Lee, G., Gommers, R., Waselewski, F., Wohlfahrt, K., O'Leary, A.: Pywavelets: A python package for wavelet analysis. Journal of Open Source Software **4**(36), 1237 (2019)

[15] Liu, P., Zhang, H., Lian, W., Zuo, W.: Multi-level wavelet convolutional neural networks. IEEE Access **7**, 74973–74985 (2019)

[16] Livani, H., Evrenosoglu, C.Y.: A machine learning and wavelet-based fault location method for hybrid transmission lines. IEEE Transactions on Smart Grid **5**(1), 51–59 (2013)

[17] Lohne, M.: Parseval Reconstruction Networks. Master's thesis (2019)

[18] Mallat, S.G.: A theory for multiresolution signal decomposition: the wavelet representation. IEEE transactions on pattern analysis and machine intelligence **11**(7), 674–693 (1989)

[19] Oliveira, B.: pytest Quick Start Guide: Write better Python code with simple and maintainable tests. Packt Publishing Ltd (2018)

[20] Paleo, P.: pypwt, parallel discrete wavelet transform (2020), `https://github.com/pierrepaleo/pypwt`

[21] Rodriguez, M.X.B., Gruson, A., Polania, L., Fujieda, S., Prieto, F., Takayama, K., Hachisuka, T.: Deep adaptive wavelet network. In: The IEEE Winter Conference on Applications of Computer Vision. pp. 3111–3119 (2020)

[22] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A.C., Fei-Fei, L.: ImageNet Large Scale Visual Recognition Challenge. International Journal of Computer Vision (IJCV) **115**(3), 211–252 (2015)

[23] Walt, S.v.d., Colbert, S.C., Varoquaux, G.: The NumPy array: a structure for efficient numerical computation. Computing in Science & Engineering **13**(2), 22–30 (2011)