




# PyDTNN: A user-friendly and extensible framework for distributed deep learning

Sergio Barrachina<sup>1</sup> · Adrián Castelló<sup>1</sup> · Mar Catalán<sup>1</sup> · Manuel F. Dolz<sup>1</sup>  · Jose I. Mestre<sup>1</sup>

Accepted: 4 February 2021 / Published online: 22 February 2021

© The Author(s), under exclusive licence to Springer Science+Business Media, LLC part of Springer Nature 2021

## Abstract

We introduce a framework for training deep neural networks on clusters of computers with the following appealing properties: (1) It is developed in Python, exposing an amiable interface that provides an accessible entry point for the newcomer; (2) it is extensible, offering a customizable tool for the more advanced user in deep learning; (3) it covers the main functionality appearing in convolutional neural networks; and (4) it delivers reasonable inter-node parallel performance exploiting data parallelism by leveraging MPI via `MPI4PY` for communication and `NumPY` for the efficient execution of (multithreaded) numerical kernels.

**Keywords** Deep neural networks · Distributed parallel training · Python

## 1 Introduction

The recent outburst in machine learning via deep neural networks (DNNs) is largely due to the combined effect of new algorithmic techniques, vast amounts of computational capacity in current hardware, and the explosion in the amount of training data [1, 2]. The myriad of applications of deep learning (DL) and the computational complexity of the training process have pushed the industry to design customized

---

✉ Manuel F. Dolz  
dolzm@uji.es

Sergio Barrachina  
barrachi@uji.es

Adrián Castelló  
adcastel@uji.es

Mar Catalán  
catalama@uji.es

Jose I. Mestre  
jmiravet@uji.es

<sup>1</sup> Universitat Jaume I, Castellón de la Plana, Spain

architectures and hardware components as well as very sophisticated frameworks for DL. In the latter category, we can identify Google's TensorFlow, Facebook's PyTorch and Caffe2, Microsoft's CNTK, Theano, and Keras, among others.

While these frameworks have doubtlessly contributed to the adoption of DL, we also find that the level of internal intricacy of these packages turns their customization into a fairly difficult task. A particular problem that we are concerned with is the realization of distributed training for DNNs, which entails an extra degree of complexity to DL frameworks. To tackle this, we present a lightweight framework for distributed DL training and inference, named PyDTNN (*Python Distributed Training of Neural Networks*),<sup>1</sup> with the following features:

1. Amiable user interface: PyDTNN is developed in a high-level language, such as Python, offering an interface that is similar to that exposed by popular packages, such as Keras, to provide a flat accessing curve for the novice.
2. Extensible: PyDTNN prioritizes simplicity while facilitating user customization of the framework.
3. Functional: PyDTNN covers fully connected, convolutional and pooling layers, dropout, batch normalization, a variety of popular nonlinear functions, etc.
4. Moderately efficient: PyDTNN exploits data parallelism, using MPI for message-passing and multithreaded kernels for the major linear algebra operations.

Note that we do not claim about PyDTNN offering an alternative for distributed training that is competitive, for example, with TensorFlow enhanced with Horovod from the point of view of parallel performance. Instead, we claim that PyDTNN offers an accessible solution for basic training of simple DNN models on clusters that can be more easily customized to prototype and experiment with new ideas.

The rest of the paper is organized as follows. In Sect. 2, we provide a brief overview of distributed training for DNNs and the exploitation of data parallelism. In Sect. 3, we discuss the internal organization and functionality of PyDTNN, and in Sect. 4, we describe its user interface. Next, in Sect. 5, we illustrate the flexibility of PyDTNN as a tool to prototype ideas, and in Sect. 6, we elaborate on its efficiency. Finally, in Sect. 7, we summarize the main properties of PyDTNN as part of our concluding remarks.

## 2 Distributed training of DNNs

In this section, we provide a short review of DNNs and distributed training.

**Overview of DNNs.** Consider a collection of input vectors (or samples) given by  $x_1, x_2, \dots, x_s \in \mathbb{R}^n$ , respectively, classified using labels  $y_1, y_2, \dots, y_s \in \mathbb{R}^m$  (also known as target outputs or ground truth). A neural network comprises a number of interconnected neurons, organized into multiple layers, which define a nonlinear

<sup>1</sup> The source code for PyDTNN is available at <https://github.com/hpca-uji/PyDTNN>.

function  $\mathcal{F} : \mathbb{R}^n \rightarrow \mathbb{R}^m$  performing the mapping  $\mathcal{F}(x_r) = \tilde{y}_r$ , where we expect that  $\tilde{y}_r \approx y_r$ ,  $r = 1, 2, \dots, s$ . For performance reasons, the input–output mapping realized by a DNN (also known as forward pass, or FP) is performed in batches of  $b$  samples at a time [3].

The goal of the training process is to minimize the difference between the output(s) computed by the NN and the ground truth, given by  $\sum_{r=1}^s \frac{1}{s} \|y_r - \tilde{y}_r\|$ . This optimization problem is usually solved via the *stochastic gradient descent* (SGD) method, which implements an iterative “back-propagation” (BP) procedure that realizes the gradient computation (GC) which minimizes the difference and performs the corresponding weight updates (WU).

In practice, current DNNs often combine convolutional layers (Conv) in the initial stages followed by fully connected layers (FC) layers in the last ones. A Conv layer consists of multiple filters that operate on a (sub)tensor of the inputs, of the same dimension as the filters, to produce a single scalar value. The filters are repeatedly applied in a sliding window manner to the whole input, in order to produce all the output values [1]. An efficient realization of Conv can be obtained by means of a re-organization of the proper input operand via an *im2col* transform [3–5]. The result of the convolution operation can then be achieved using a general matrix–matrix multiplication (Gemm).

**Distributed training.** There exist strict data dependencies between the outputs of one layer and the inputs to the next layer, both in the FP and BP stages of DNN training. Thus, the only parallelization option is to exploit the intra-layer concurrency, which corresponds to parallelizing the individual Gemm inside each layer.

In the data-parallel (DP) scheme [3], concurrency is extracted across the batch dimension. This benefits from the fact that provided some algorithmic issues related to the training convergence are conveniently tackled, the batch dimension ( $b$ ) can be linearly increased with the number of processes, up to values of  $b$  in the range 32k–64k [6, 7].

In short detail, the DP scheme replicates the weight matrices that define the NN model in all processes while the remaining matrix operands (input/output activations to each layer) are distributed in the batch dimension by blocks of columns. Therefore, in the FP and GC stages, there is no need for any inter-process communication. In contrast, the WU stage requires an Allreduce [8] exchange to aggregate the local updates, across all processes into the model (weights) before the computation with the next batch.

The current version of PyDTNN comprises a distributed DP realization of the training that relies on the MPI4Py Python package for the inter-node communication layer. The development of an alternative model-parallel scheme is part of ongoing work.

### 3 A glimpse of PyDTNN

In this section, we provide an overview of PyDTNN and describe how this framework exploits data parallelism.

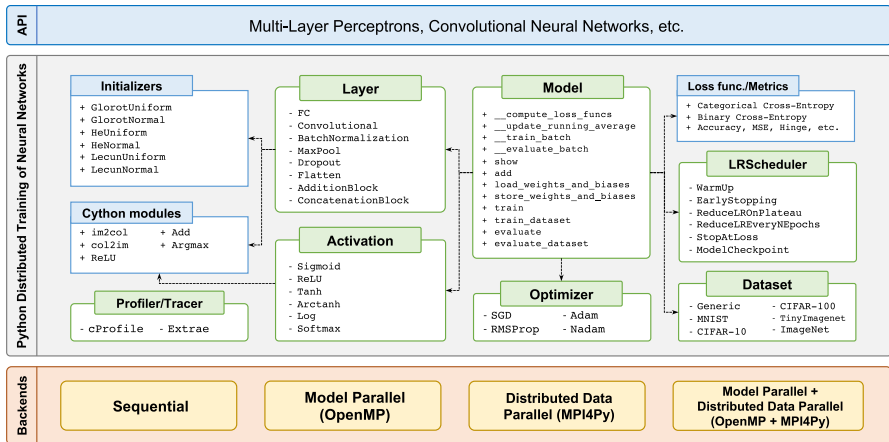


Fig. 1 PyDTNN architecture

### 3.1 Overview

**Functionality** PyDTNN supports basic DL modules to create, train, and perform inference with MLPs and CNNs such as, for example, the VGG models, and the residual neural networks (ResNet), among other types of convolutional DNNs. We have oriented our design to obtain a customizable environment. Some plans toward extending the current functionality, for example, in order to cover more involved models, include developing the classes of modules that appear in recurrent DNNs.

Figure 1 offers an overview of the PyDTNN architecture. The top box illustrates the application programming interface (API) exposed to the user, that he/she can then leverage to create, train, and evaluate DNNs. The middle (gray) box comprises the distinct PyDTNN modules, such as layers, activations, and models, among others, that realize the training and inference processes. In addition to Python, Cython is used to exploit intra-node parallelism via OpenMP. As shown in the bottom boxes, these DNNs can be trained while exploiting: (1) model parallelism at intra-node level; (2) DP at inter-/intra-node levels; or a combination of both (1) and (2), for example, for clusters of nodes equipped with multicore processors.

**Basic classes and methods** The PyDTNN framework defines two main classes: Model and Layer. The former class contains the model features and defines the most relevant methods including, among others, `train_dataset()` for performing the training. This method receives several input parameters—such as the dataset and optimizer objects, the number of epochs, the batch size  $b$  (per process), and a list of loss metrics and learning rate schedulers, which set the training configuration. The fragments of code in Listings 1 and 2 illustrate the main aspects of this method. The first listing shows the implementation of the training cycle over the epochs and training batches, returned by the corresponding dataset generator. The second one corresponds to the training of a single batch.

The Layer class contains a generic definition of the three main methods: `forward()`, `backward()`, and `update_weights()` (for FP, GC, and WU,

respectively). Each type of layer specializes these functions. For example, only those layers that operate with weights (Conv and FC) will redefine the `update_weights()` method. The main methods of the FC layer, derived from the `Layer` class, are shown in Listing 3. (The methods of the Conv layer are omitted for brevity.)

```

1 def train_dataset(self, dataset, nepochs, local_batch_size,
2                   val_split, loss, metrics, optimizer, lr_schedulers):
3     # ...
4     for epoch in range(nepochs):
5         train_batch_generator, val_batch_generator = \
6             dataset.get_train_val_generator(local_batch_size, self.rank,
7                                           self.nprocs, val_split)
8         # ...
9         for X_batch, Y_batch, batch_size in train_batch_generator:
10            train_batch_loss = \
11                self.__train_batch(X_batch, Y_batch,
12                                  batch_size, loss_func, metrics_funcs, optimizer,
13                                  lr_schedulers)
14            train_total_loss, train_batch_count, status = \
15                self.__update_running_average(train_batch_loss, train_total_loss,
16                                             train_batch_count, batch_size, loss_metrics)
17            # ...

```

**Listing 1** Method `train_dataset()` of the `Model` class.

```

1 def __train_batch(self, X_batch, Y_batch, batch_size, loss_func,
2                  metrics_funcs, optimizer, lr_schedulers):
3     # Forward propagation (FP)
4     x = X_batch
5     for l in range(1, len(self.layers)):
6         x = self.layers[l].forward(x)
7
8     # Gradient computation (GC)
9     loss, dx = loss_func(x, Y_batch)
10    for l in range(len(self.layers)-1, 0, -1):
11        dx = self.layers[l].backward(dx)
12
13    # Weight update (WU)
14    for l in range(len(self.layers)-1, 0, -1):
15        self.layers[l].update_weights(optimizer)

```

**Listing 2** Method `train_batch()` of the `Model` class.

### 3.2 Exploiting DP in PyDTNN

In the DP version of the training process, the batch has to be distributed among the processes (cluster nodes), while the model (defined by the values of weights and biases) needs to be replicated in all the processes [3].

```

1 class Layer():
2     # ... Other methods omitted for brevity
3     def update_weights(self, optimizer):
4         optimizer.update(self)
5
6 class FC(Layer):
7     # ... Other methods omitted for brevity
8     def forward(self, x):
9         self.x = x
10        y = x @ self.weights + self.biases
11        return y
12
13    def backward(self, dy):
14        self.dw = self.x.T @ dy
15        self.db = dy.sum(axis=0)
16        dx = dy @ self.weights.T
17        return dx

```

**Listing 3** Main training methods of the Layer and FC classes.

**Distributed batch** In the application, the user specifies the dataset object and the batch size, passing these values to the `train_dataset()` function. During the parallel execution, `batch_size` is the dimension (number of samples) of the local batch that each process will tackle. The dimension of the global batch is then roughly obtained as the product between the size of the local batches and the number of processes.

```

1 def batch_generator(self, generator, local_batch_size=64, rank=0, nprocs=1,
2     shuffle=True):
3     batch_size = local_batch_size * nprocs
4     for X_data, Y_data in X_train_data, Y_train_data:
5         nsamples = X_data.shape[0]
6         np.random.shuffle(np.arange(nsamples))
7         # ...
8         # Generate batches
9         for batch_num in range(0, end_for, batch_size):
10            start = batch_num + rank * local_batch_size
11            end = batch_num + (rank+1) * local_batch_size
12            indices = s[start:end]
13            X_local_batch = X_data[indices,...]
14            Y_local_batch = Y_data[indices,...]
15            yield (X_local_batch, Y_local_batch, batch_size)
16        # ...

```

**Listing 4** Batch generator method of the Dataset class.

When the `train_dataset()` function is executed in parallel, all processes receive the full dataset (global dataset containing all samples). Listing 4 shows the `batch_generator()` coroutine that serves as a data generator for the training loop in line 9 of Listing 1. There, each process selects its subset (or local batch) depending on its rank. While a true distribution of the samples provides a more scalable solution, in our design, we have prioritized simplicity over efficiency.

**Replicated model** Before the training commences, PyDTNN sets the same seeds in all processes to generate the same initial random weights and biases (i.e., the replicated model) at each process. During training, PyDTNN then ensures that all processes perform a coordinated update of the model, as described next, to maintain the inter-process coherence of the NN model.

```

1 def __train_batch(self, X_batch, Y_batch, batch_size, loss_func,
2                   metrics_funcs, optimizer, lr_schedulers):
3     # Forward propagation (FP)
4     x = X_batch
5     for l in range(1, len(self.layers)):
6         x = self.layers[l].forward(x)
7
8     # Gradient computation (GC)
9     loss, dx = loss_func(x, Y_batch)
10    for l in range(len(self.layers)-1, 0, -1):
11        dx = self.layers[l].backward(dx)
12
13    # Weight update (WU)
14    for l in range(len(self.layers)-1, 0, -1):
15        self.layers[l].reduce_weights_sync(self.comm)
16        self.layers[l].update_weights(optimizer)

```

**Listing 5** Method `__train_batch()` of the `Model` class for distributed training.

The distributed training of a batch is illustrated in Listing 5. A direct comparison of this code with its non-distributed counterpart, in Listing 2, shows the same actions for the forward pass and gradient computation (initial part of the codes). The implicit difference between the distributed and “sequential” versions in these parts is that, in the former, each process acts on the local part of the batch, while the latter operates with the full batch (as described earlier in this subsection).

In contrast, the comparison between the weight updates in the sequential and distributed training codes shows a couple of new routines in the latter case. (Compare lines 14–15 in Listing 2 and lines 14–16 in Listing 5.) Concretely, in the distributed code (1) each process computes its local contribution to the weight updates, according to the information in the local batch that it has processed; and (2) all the contributions are reduced next, before accumulating them into the global (replicated) weights. This is, respectively, achieved in the distributed case via two functions calls: (1) `backward()` and (2) `reduce_weights_sync()`. The latter function performs a reshape (linearization) of the data structures, followed by the reduction, and completes the process by undoing the reshape; see the code in Listing 6.

```

1 class Layer():
2     # ...
3     def reduce_weights_sync(self, comm):
4         if comm and self.weights.size > 0:
5             dwb = np.concatenate((self.dw.flatten(), self.db.flatten()))
6             comm.Allreduce(MPI.IN_PLACE, dwb, op=MPI.SUM)
7             self.dw = dwb[:self.weights.size].reshape(self.weights.shape)
8             self.db = dwb[self.weights.size:].reshape(self.biases.shape)

```

**Listing 6** Reduction of local contributions to weights via the MPI Allreduce collective.

## 4 PyDTNN amiable user interface

The PyDTNN framework exposes a Keras-like user interface in order to flatten the entry learning curve. This decision pursues to help the novice user as well as motivate the more DL expert to start an interaction with the framework as there is no need to learn yet-another-interface.

Listing 7 presents the instructions necessary to define a representative convolutional neural network: VGG11 [9] for the CIFAR-10 dataset. This code illustrates the basic interaction cycle with the PyDTNN interface, which is composed of four steps where the user: (1) defines each (individual) layer of the model; (2) extracts the dataset for training (or inference) from the corresponding file(s); (3) sets a few basic training parameters such as the learning rate, the number of epochs to train, and the batch size; and, finally, (4) invokes the training (or inference) routine. Similarly, Listing 8 shows the code necessary to define the ResNet-32 network [10] for the same dataset. In this case, to permit the construction of the identity shortcut-connections required by the ResNet-32 model, PyDTNN includes the special `AdditionBlock` layer (see lines 10–18) which processes the different paths contained in it to finally perform an element-wise sum (during the forward pass) of the activations obtained at the last layer in each of the paths.

During the creation of the model, the user can specify the distinct features of the layers. For example, for an FC layer, the user indicates the number of neurons and the activation function. In comparison, a CONV layer requires a larger number of parameters: the number and shape of the filters, the padding and stride factors for the filter application, and the activation function.

```

1 # 1) Define the model
2 v11 = Model()
3 # Layer 0: Input
4 v11.add( Input(shape=(3,32,32)) )
5 # Layer 1: Conv2D
6 v11.add( Conv2D(nfilters=64, filter_shape=(3,3), padding=1, stride=1,
7               activation="relu") )
8 v11.add( MaxPool2D(pool_shape=(2,2), stride=2) )
9 # Layers 3-16 omitted for brevity
10 # Layer 16: Output (FC)
11 v11.add( FC(shape=(10), activation="softmax") )
12
13 # 2) Retrieve CIFAR-10 as training data and labels
14 x, y = read_dataset("CIFAR10")
15
16 # 3) Initialize the training parameters (learning rate, number of epochs, and
17     batch size)
18 lr, n_epochs, bs = 0.1, 100, 64
19
20 # 4) Define the optimizer and train the network (parallel distributed)
21 opt = SGD(learning_rate = lr)
22 v11.train(x, y, n_epochs, bs, opt,
23         loss="categorical_crossentropy",
24         metrics="accuracy")

```

**Listing 7** Sample code for creating and training the VGG11 model using PyDTNN.



```

1 r32 = Model()
2 r32.add( Input(shape=(3, 32, 32)) )
3 r32.add( Conv2D(nfilters=16, filter_shape=(3, 3), stride=1, padding=1) )
4 r32.add( BatchNormalization() )
5
6 layout = [ [16, 5, 1], [32, 5, 2], [64, 5, 2] ] # Layout of ResNet-32
7 for n, b, s in layout:
8     for r in range(res_blocks):
9         if r > 0: s = 1
10        r32.add( AdditionBlock(
11            [ Conv2D(nfilters=n, filter_shape=(3, 3), stride=s, padding=1),
12              BatchNormalization(), Relu(),
13              Conv2D(nfilters=n, filter_shape=(3, 3), stride=1, padding=1),
14              BatchNormalization()
15            ], [
16              Conv2D(nfilters=n, filter_shape=(1, 1), stride=s),
17              BatchNormalization()
18            ] if s != 1 else [] ) )
19        r32.add( Relu() )
20
21 r32.add( AveragePool2D(pool_shape=(0,0)) ) # Global average pooling 2D
22 r32.add( Flatten() )
23 r32.add( FC(shape=(64,)) ); r32.add( BatchNormalization() ); r32.add( Relu() )
24 r32.add( FC(shape=(10,), activation="softmax" ) )

```

**Listing 8** Sample code for creating the ResNet-32 model using PyDTNN.

In addition, to specify a parallel execution, the user only has to invoke `mpirun` as, for example, in:

```

mpirun -iface ib0 -np 12 -ppn 1 python -u benchmarks_CNN.py \
--model vgg11 --dataset cifar10 --batch_size 64 --num_epochs 100

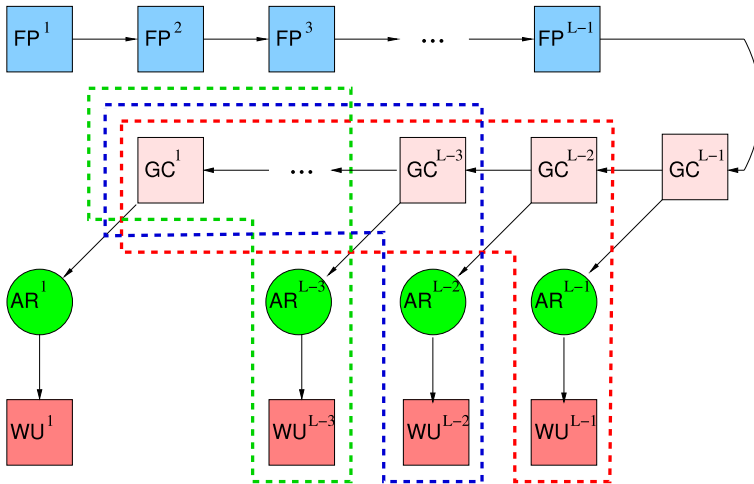
```

In this example, the `mpirun` command launches the DP training of the VGG11 model using 12 processes (`-np 12`), each mapped onto a cluster node (`-ppn 1`), and configured to use the Infiniband network interface (`-iface ib0`). The script `benchmarks_CNN.py` is a utility from PyDTNN whose parameters specify the model to be trained (`--model vgg11`), the dataset (`--dataset cifar10`), the batch size (`--batch_size 64`), and the number of epochs to execute (`--num_epochs 100`), among other options.

## 5 Extensibility of PyDTNN

To illustrate the possibilities and ease of customizing PyDTNN, we next describe a couple of extensions of the baseline implementation.

**Overlapping communications with computation** Let us start by considering the dependencies between the major operations in a forward-backward pass, displayed in Fig. 2. On the one hand, there exist strict dependencies between the Gradient computations of “consecutive” layers since  $GC^{l-1}$  depends on  $GC^l$ . On the other hand, the corresponding reduction communication and weight update are decoupled so that, once  $GC^l$  is available, the exchange  $AR^l$  and the update  $WU^l$  can proceed in parallel with  $GC^{l-1}$ ,  $GC^{l-2}$ , ...,  $GC^1$ . As corresponds to a synchronous variant of the training, the



**Fig. 2** Data dependencies in the training. The colored boxes correspond to the computational stages: FP, GC, and WU; the circles denote Allreduce (AR) exchanges; and the arrows indicate dependencies. The colored dashed lines mark operations which can be overlapped

update  $WU^l$  for the samples in a batch must be completed before these weights can participate in the forward pass  $FP^l$  with the next batch of samples.

Listing 5 shows the code that is executed by PyDTNN for the distributed training procedure. Lines 9–11 calculate GC (per layer); in line 15, the call to `allreduce_weights()` synchronizes the weight matrices in all processes; and line 16 completes the backward pass by updating the local weights.

The goal of the following exercise is to illustrate how to transform the baseline version of PyDTNN into a variant where the communications are overlapped with other Gradient computations. This can be achieved by using the non-blocking version of the MPI routine for the global reduction with a synchronization point (in the form of an invocation to the MPI routine `wait`) before the corresponding weight update. Listing 9 shows the changes that have to be introduced in the original code of the PyDTNN library in order to overlap computation and communication during the training process. As in the previous example, lines 9–11 compute the GC stage, and this is followed by the invocation to `reduce_weights_async()`. The main difference is that this function employs the non-blocking primitive `Iallreduce` instead of its blocking counterpart `Allreduce`. The non-blocking variant allows overlapping the communication with the computation of other GC stages; see Fig. 2. Besides, to ensure the communication completion in due time, an MPI wait function, `wait_allreduce_async()`, is added before the weight update.

```

1 def __train_batch(self, X_batch, Y_batch, batch_size, loss_func,
2                   metrics_funcs, optimizer, lr_schedulers):
3     # Forward propagation (FP)
4     x = X_batch
5     for l in range(1, len(self.layers)):
6         x = self.layers[l].forward(x)
7
8     # Gradient computation (GC)
9     loss, dx = loss_func(x, Y_batch)
10    for l in range(len(self.layers)-1, 0, -1):
11        dx = self.layers[l].backward(dx)
12        self.layers[l].reduce_weights_async(self.comm)
13
14    # Weight update (WU)
15    for l in range(len(self.layers)-1, 0, -1):
16        self.layers[l].wait_allreduce_async(self.comm)
17        self.layers[l].update_weights(optimizer)

```

**Listing 9** Method `__train_batch()` of the `Model` class for distributed training with overlapped communication and computation.

**Customizing the arithmetic precision** An additional example of the PyDTNN extensibility is presented in Listing 10. There, we demonstrate how to customize the precision for the reduction of the weights in the backward process using, in this particular case, two different datatypes: FP32 (`comp_dtype`) and FP16 (`comm_dtype`). This function employs FP32 for the arithmetic (line 15) but transforms the data from FP32 to FP16 for communication (lines 18 and 26). The purpose of this modification is to reduce the number of bytes transferred while maintaining the precision of the local arithmetic.

```

1 def Mixed_Allreduce(sbuf, comm_dtype, op_reduce, root, comm):
2     # Get rank and communicator size
3     rank, size = comm.Get_rank(), comm.Get_size()
4
5     # Get computation dtype
6     comp_dtype = sbuf.dtype
7
8     # Allocate reception buffer as comm_dtype
9     rbuf = np.empty_like(sbuf, dtype=comm_dtype)
10
11     if rank == root:
12         # Reduction code
13         acc = np.zeros_like(rbuf, dtype=comp_dtype)
14         for r in range(size):
15             if r != root:
16                 comm.Recv([rbuf, rbuf.size*rbuf.itemsize, MPI.CHAR],
17                           source=r, tag=RED)
18                 # Convert from comm_dtype to comp_dtype for computation
19                 acc = op_reduce(acc, rbuf.astype(comp_dtype))
20
21             # Convert from comp_dtype to comm_dtype for communication
22             rbuf = acc.astype(comm_dtype)
23
24         # Broadcast code
25         for r in range(size):
26             if r != rank:
27                 comm.Send([rbuf, rbuf.size*rbuf.itemsize, MPI.CHAR],
28                           dest=r, tag=BCAST)
29     else:
30         # Convert from comp_dtype to comm_dtype for communication
31         sbuf = sbuf.astype(comm_dtype)
32         comm.Send([sbuf, sbuf.size*sbuf.itemsize, MPI.CHAR],
33                 dest=root, tag=RED)
34         comm.Recv([rbuf, rbuf.size*rbuf.itemsize, MPI.CHAR],
35                 source=root, tag=BCAST)
36
37     # Convert from comm_dtype to comp_dtype
38     return rbuf.astype(comp_dtype)

```

**Listing 10** Implementation of mixed-precision Allreduce algorithm in PyDTNN.

**Blocking the convolution operators** A significant part of the computational cost of CNNs is due to the application of convolutions. A general, flexible, and high-performance approach to deal with this type of operators, in a convolutional layer, is to process the layer input tensor (activations) via the im2col transform [4], followed by an invocation to a general matrix multiplication (Gemm) kernel to multiply the weight matrix with the output of the im2col transform [4, 5]. Unfortunately, applying this transform results in a very large matrix, which may exhaust the memory of the system. In particular, the im2col transform expands the layer input tensor into an augmented matrix that is  $k_h \times k_w$  times larger, where  $k_h/k_w$  denotes the height/width of the filter layer.

```

1 def forward(self, x):
2     # ...
3     self.x_cols = im2col_cython(x, self.kh, self.kw, self.vpadding,
4                               self.hpadding, self.vstride, self.hstride)
5
6     w_cols = self.weights.reshape(self.co, -1)
7     res = self.matmul(w_cols, self.x_cols)
8     # ...

```

**Listing 11** Implementation of a convolution layer via the im2col transform (implemented as an external method in Cython) in PyDTNN.

Listing 11 shows how the convolution operator (appearing in a convolutional layer in the forward pass) is applied in PyDTNN by first invoking an external method, for efficiency implemented in Cython (lines 3–4), to then perform the necessary matrix multiplication (line 7).

To reduce memory consumption, we can perform an alternative segmented application of the im2col transform, as shown in Listing 12. There, the im2col transform is calculated in chunks of size `chunk_size` (see lines 11–13), requiring only a matrix that is `batch_size / chunk_size` times smaller than that used in the approach of Listing 11. In line 17, each of the im2col chunks (`x_cols`) is multiplied by the reshaped weights (`w_cols`) to obtain the corresponding portion of the output tensor (`y_cols`).

```

1 def forward(self, x):
2     # ...
3     # Reshaped weight tensor of shape (co, kh * kw)
4     w_cols = self.weights.reshape(self.co, -1)
5     # Output tensor of shape (co, ho * wo * b)
6     y_cols = np.empty(self.co, self.ho * self.wo * self.batch_size)
7
8     for s in range(0, self.batch_size, self.chunk_size):
9         # Calculate end index in x of shape (b, ci, hi, wi)
10        e = min(s + self.chunk_size, self.batch_size)
11        x_cols = im2col_cython(x[s:e,...], self.kh, self.kw,
12                               self.vpadding, self.hpaddng,
13                               self.vstride, self.hstride)
14
15        # Calculate start/end indices in res of shape (co, ho * wo * b)
16        s_, e_ = s * self.ho * self.wo, e * self.ho * self.wo
17        y_cols[:,s_:e_] = self.matmul(w_cols, x_cols)
18        # ...

```

**Listing 12** Implementation of a convolution layer via the segmented im2col transform (implemented as an external method in Cython) in PyDTNN.

## 6 Efficiency of two-level parallel PyDTNN

As argued earlier, PyDTNN exploits two levels of parallelism: inter-node and intra-node, with the second one being extracted via the invocation to multithreaded routines, much like other frameworks for distributed DL. In any case, we want to emphasize that PyDTNN is designed as a tool to rapidly prototype ideas, not as a DL solution to compete in performance with modern DL frameworks.

In the following evaluation, we expose and motivate the performance gap between PyDTNN and TensorFlow (TF, version 2.2.0) using the native Keras backend enhanced with Horovod (version 0.20.3). For this evaluation, we train the AlexNet, VGG11, and ResNet-32 models (on the CIFAR-10 dataset) inspecting three metrics: (1) total execution time; (2) number of epochs for convergence; and (3) speed-up with respect to the baseline execution. All the experiments were carried out on a cluster consisting of eight nodes, each equipped with two Intel Xeon Gold 5120 CPU (Skylake) processors with 14 cores each (28 cores in total), 190 GiB of DDR4 RAM, and connected via a Mellanox EDR Infiniband switch. Regarding the software, we leveraged Intel Python v3.7.4 and NumPy v1.17.4 linked against Intel

**Table 1** Execution time (in kiloseconds) and number of epochs (#E) for the training of AlexNet, VGG11, and ResNet-32 on CIFAR-10 using TF and PyDTNN, with a threshold convergence validation accuracy of 70% using different number of processes in DP (#P) and threads per process (#T)

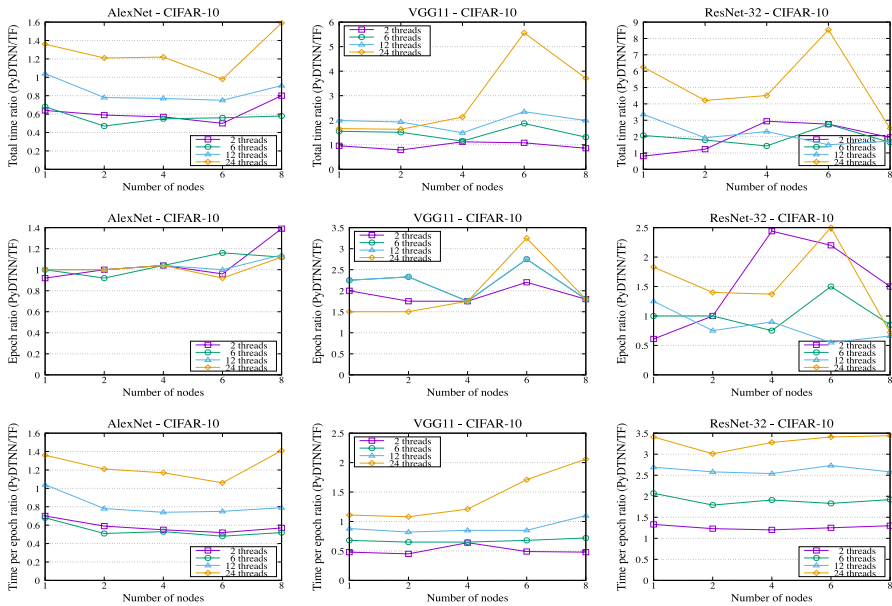
#P	#T	AlexNet				VGG11				ResNet-32			
		TF		PyDTNN		TF		PyDTNN		TF		PyDTNN	
		Time	#E	Time	#E	Time	#E	Time	#E	Time	#E	Time	#E
1	2	7.15	25	4.62	23	5.68	4	5.46	8	12.78	13	10.47	8
	6	4.12	23	2.81	23	2.38	4	3.69	9	2.56	6	5.31	6
	12	2.49	25	2.60	25	1.54	4	3.08	9	2.55	8	8.60	10
	24	1.86	25	2.54	25	1.20	4	2.00	6	1.57	6	9.82	11
2	2	4.49	25	2.66	25	3.16	4	2.53	7	3.16	6	3.89	6
	6	3.20	25	1.52	23	0.99	3	1.51	7	1.74	7	3.13	7
	12	1.66	23	1.30	23	0.70	3	1.36	7	1.31	8	2.55	6
	24	1.17	25	1.42	25	0.74	4	1.21	6	0.72	5	3.05	7
4	2	2.47	25	1.43	26	1.69	4	1.90	7	2.44	9	7.22	22
	6	1.59	25	0.88	26	0.70	4	0.80	7	1.48	12	2.13	9
	12	0.98	25	0.76	26	0.49	4	0.73	7	1.00	11	2.33	10
	24	0.66	25	0.81	26	0.40	4	0.86	7	1.15	16	5.20	22
6	2	1.90	27	0.96	26	1.38	5	1.50	11	1.84	10	5.09	22
	6	1.17	24	0.66	28	0.49	4	0.91	11	1.19	14	3.28	21
	12	0.73	26	0.55	26	0.35	4	0.83	11	1.16	20	1.75	11
	24	0.56	28	0.55	26	0.28	4	1.59	13	0.47	10	4.08	25
8	2	1.22	23	0.98	32	1.09	5	0.94	9	1.89	14	3.70	21
	6	0.85	24	0.50	27	0.47	5	0.61	9	1.30	21	2.14	18
	12	0.58	27	0.53	31	0.33	5	0.66	9	0.80	18	1.39	12
	24	0.39	24	0.62	27	0.27	5	1.02	9	0.78	22	1.97	16

The optimizer was SGD with a learning rate (LR)  $\eta = 10^{-3}$  for AlexNet and  $10^{-2}$  for ResNet-32, momentum set to 0.9; and Adam with a LR  $\eta = 10^{-4}$  for the VGG11. For the DP execution, the LR was tuned using the linear scale rule  $LR=LR_{base} \times p$ , where  $p$  is the number of processes. The batch size  $b$  was set to  $64 \times p$  for AlexNet and VGG11; and  $128 \times p$  for ResNet-32

MKL 2020.0 Update 1 from the Intel Composer XE 2020 package. We also used MPI4Py v3.0.3 linked against the Intel MPI library from the same Intel package.

Table 1 reports the training costs (in kiloseconds) and the number of epochs for PyDTNN and TF(+Horovod), for various numbers of MPI ranks (or processes) and threads per process. Each process is bound to a single node and each thread to a core inside the node. These values correspond to the actual execution time for each framework when training AlexNet, VGG11, and ResNet-32, on the CIFAR-10 dataset, till a validation accuracy threshold of 70% is achieved.

The first result in Table 1 that catches our attention is the difference between the number of epochs that the two frameworks require for reaching the convergence threshold for the VGG11 and ResNet-32 models; in contrast, for the AlexNet model, both frameworks need approximately the same number of epochs. This factor is crucial to explain the distinct performance of the frameworks. To gain insights into the



**Fig. 3** Total time, time per epoch, and number of epochs ratio PyDTNN/TF (top, middle and bottom rows, respectively) for AlexNet, VGG11, and ResNet-32 on CIFAR-10 when varying number of nodes (processes) and threads per process, with a threshold convergence validation accuracy of 70%

computational behavior of both models, Fig. 3 illustrates the differences between the two frameworks by comparing the global execution time, the number of epochs, and the execution time per epoch for the same DL models and number of processes/threads configurations. In the figure, the ratios are computed by dividing the corresponding value for PyDTNN by that of TF. Thus, a value higher than 1 means that TF outperforms PyDTNN, while a result lower than 1 indicates the opposite case.

Focusing on the total execution time, we recognize that TF is more competitive than PyDTNN, except for AlexNet using 2/4 threads per process. These differences can be better explained by looking into the two other factors, number of epochs and execution time per epoch, as follows:

- Regarding the first factor, TF is in general more efficient as it achieves the same convergence threshold in a slightly smaller number of epochs than PyDTNN. We suspect these differences come from the distinct internal algorithmic implementations of both frameworks. In any case, we observe a considerable sensitivity of the number of epochs to training factors such as the number of nodes and threads per node, for both TF and PyDTNN.
- Concerning the execution time per epoch, we can observe that, for both AlexNet and VGG11 models using from 2 to 6 threads, PyDTNN is slightly more efficient than TF, while the opposite occurs for ResNet-32. This can be explained by the compute-bound nature of ResNet-32 over AlexNet and VGG11, which is better handled by TF with a large number of threads. A second observation about

this factor is that PyDTNN delivers fair scalability when increasing the number of processes. This is reasonably given that, in our experiments, the batch size is augmented linearly with the number of processes, leading to a good weak scaling ratio. In contrast, augmenting the number of threads/cores is done while maintaining the batch size and, therefore, the total training “workload” per epoch. In this scenario, the scalability of PyDTNN suffers. The ultimate reason for this is that PyDTNN relies on multi-threaded libraries for some of the most computationally demanding intra-node operations. However, there are many other parts of PyDTNN that simply rely on plain (sequential) Python code. As the number of threads is increased, by Amdahl’s Law, the contribution of these sequential parts to the overall execution time for these parts in PyDTNN becomes considerable and the degree of parallel efficiency decays.

## 7 General remarks

PyDTNN was started as an exercise to understand in detail distributed training of neural networks. While there exist several sophisticated DL frameworks for distributed training, in our experience, the ample functionality and high parallel performance of these frameworks come at the expense of considerable complexity, especially in the case of those packages that explicitly target distributed platforms such as clusters. For this reason, we designed our framework for distributed DL training that puts the focus on simplicity, at the expense of offering more limited functionality and sacrificing some of the (intra-node) parallel performance. This paper demonstrates that it is possible to offer a simple interface, together with a DNN training package that is easy to customize and can be very helpful to rapidly prototype ideas, offering fair parallel efficiency on a cluster.

**Acknowledgements** This work was supported by Project TIN2017-82972-R from the Spanish *Ministerio de Ciencia, Innovación y Universidades*. M. F. Dolz was supported by project CDEIGENT/2018/014 from the *Generalitat Valenciana*.

## References

1. Tal B-N, Torsten H (2019) Demystifying parallel and distributed deep learning: an in-depth concurrency analysis. *ACM Comput Surv* 52(4):65
2. Chan E, Heimlich M, Purkayastha A, van de Geijn R (2007) Collective communication: theory, practice, and experience. *Concurr Comput Pract Exp* 19(13):1749–1783
3. Kumar C, Sidd P, Patrice S (2006) High performance convolutional neural networks for document processing. In: *International workshop on frontiers in handwriting recognition*
4. He K, Zhang X, Ren S, Sun J (2016) Deep residual learning for image recognition. In: *2016 IEEE conference on computer vision and pattern recognition (CVPR)*, pp 770–778
5. Pouyanfar S et al (2018) A survey on deep learning: algorithms, techniques, and applications. *ACM Comput Surv* 51(5):92:1–92:36
6. Karen S, Andrew Z (2014) Very deep convolutional networks for large-scale image recognition. arXiv preprint [arxiv:1409.1556](https://arxiv.org/abs/1409.1556)
7. Sze V, Chen Y-H, Yang T-J, Emer JS (2017) Efficient processing of deep neural networks: a tutorial and survey. *Proc IEEE* 105(12):2295–2329



8. Aravind V, Andrew A, David G (2017) Parallel multi channel convolution using general matrix multiplication. In: 2017 IEEE 28th international conference on application-specific systems, architectures and processors (ASAP), pp 19–24
9. You Y, et al. (2018) Large-batch training for LSTM and beyond. Technical Report UCB/EECS-2018-138, Electrical Engineering and Computer Sciences, University of California at Berkeley
10. Yang Y, Igor G, Boris G (2017) Scaling SGD batch size to 32k for ImageNet training. arXiv preprint [arxiv:1708.03888](https://arxiv.org/abs/1708.03888)

**Publisher's Note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.