Tarantella

Release 0.8.0

Peter Labus, Alexandra Carpen-Amarie, Martin Kuehn

Nov 17, 2022

OVERVIEW

1 Why Tarantella? 3
1.1 Goals . . . 3
1.2 Performance Results . . . 3

2 Distributed Data Parallel Training 7
2.1 The general idea . . . 7
2.2 Distribution of mini-batches . . . 7
2.3 Overlapping communication with computation . . . 7
2.4 Tensor Fusion . . . 8
2.5 Model initialization and loading . . . 8

3 Distributed Datasets 9

4 Points to Consider 11
4.1 Global versus local batch size . . . 11
4.2 Batch normalization layers . . . 11
4.3 Managing individual devices . . . 12

5 Installation 13
5.1 Installing dependencies . . . 13
5.2 SSH key-based authentication . . . 15
5.3 Building Tarantella from source . . . 15
5.4 [Optional] Building and running tests . . . 16
5.5 [Optional] Building documentation . . . 16

6 Quick Start 17
6.1 Code example: LeNet-5 on MNIST . . . 17
6.2 Executing your model with `tarantella` . . . 19
6.3 Save and load Tarantella models . . . 21
6.4 Using distributed datasets . . . 23
6.5 Callbacks . . . 23
6.6 Important points . . . 25

7 Tutorials 27
7.1 Prerequisites . . . 27
7.2 ResNet-50 . . . 28
7.3 Transformers . . . 31

8 Advanced Topics 37
8.1 GASPI ranks . . . 37
8.2 Using local batch sizes . . . 38

8.3 Setting tensor fusion threshold . . . 38
8.4 Performance aspects . . . 39
8.5 Python Interpreter . . . 39
8.6 Reproducibility . . . 39

9 Frequently Asked Questions (FAQ) **41**

10 Bug Reports **45**

11 Feature Requests **47**

12 Contributing **49**

13 Contact **51**

14 License **53**

Bibliography **67**

TARANTELLA

Tarantella is an open-source, distributed Deep Learning framework built on top of TensorFlow, providing scalable Deep Neural Network training on CPU and GPU compute clusters.

Tarantella is easy-to-use, allows to re-use existing TensorFlow models, and does not require any knowledge of parallel computing.

CHAPTER

ONE

WHY TARANTELLA?

Tarantella is an open-source Deep Learning framework that focuses on providing fast, scalable and efficient training of Deep Neural Networks (DNNs) on High Performance Computing (HPC) clusters.

1.1 Goals

Tarantella is designed to meet the following goals:

```
Tarantella...

  1. ...provides strong scalability
  2. ...is easy to use
  3. ...follows a synchronous training scheme
  4. ...integrates well with existing models
  5. ...provides support for GPU and CPU systems
```

Tarantella provides close to linear speed-up for the training of common Deep Learning architectures, thus considerably reducing the required time-to-accuracy in many Deep Learning workflows. To make this capability accessible to as many users as possible, Tarantella's interface is designed such that its use does not require any expertise in HPC or parallel computing.

To allow integrating Tarantella into any TensorFlow-based Deep Learning workflow, we put special emphasis on strictly following the synchronous optimization scheme used to train DNNs. This guarantees that results obtained in serial execution can be reproduced when using distributed training (cf. however *these guidelines*), so that computation can be scaled up at any point in time without losing reproducibility of the results.

Furthermore, we made sure that existing TensorFlow models can be made ready for distributed training with minimal effort (follow the *Quick Start guide* to learn more). Tarantella supports distributed training on GPU and pure CPU clusters, independently of the hardware vendors.

1.2 Performance Results

To investigate the scalability of Tarantella distributed training with respect to the number of devices used, we performed several experiments across multiple machines and models used in the fields of computer vision and natural language processing.

We show below some of the results we obtained when training two state-of-the-art models in parallel with Tarantella on two types of machines: the HPC-DA cluster of the Technical University of Dresden is a machine designed for data science workloads, equipped with *6* GPUs per node, while SuperMUC-NG from the Leibniz Supercomputing Centre is a typical HPC machine suitable for CPU-intensive simulations. The hardware details of the two machines used in our experiments are shown below.

Cluster	Hardware specifications per node
HPC-DA	• 6 x NVIDIA VOLTA V100 GPU with 32GB HBM2 • 2 x IBM Power9 CPU (22 cores @2.80 GHz) • NVLINK bandwidth 150 GB/s between GPUs and host • 2 x 100 Gbit/s Infiniband interconnect between nodes
SuperMUC-NG	• 2 x Intel Skylake Xeon Platinum 8174 CPU (48 cores @3.10 GHz) • 100 Gbit/s OmniPath network

First we look at the speedups that Tarantella can achieve when scaling up the number of devices for the ResNet-50 model trained with the ImageNet dataset. ResNet-50 is one of the most studied deep neural networks for computer vision tasks, featuring over *23 million* trainable parameters.

More specifically, Figure 1 illustrates the runtime per epoch on the *HPC-DA* cluster, when using up to *96* GPUs. Figure 2 showcases the same experiment performed on CPUs on the *SuperMUC-NG* machine, showing that training ResNet-50 distributedly scales on up to *256* processes. Compared to the baseline single-device runtime of the ResNet-50 model using TensorFlow 2.2, Tarantella succeeds in training the model **62x faster** on the CPU cluster and **57x faster** on the GPUs.

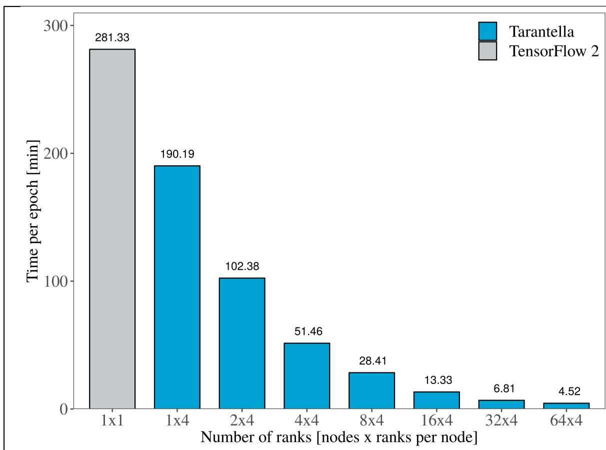

Fig. 1: Figure 1. Training Resnet-50 on CPU nodes

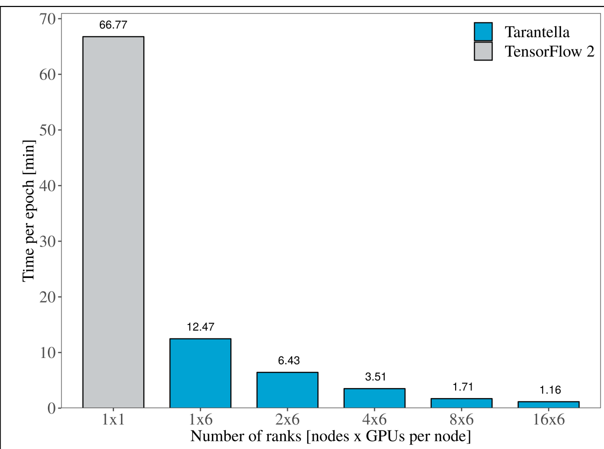

Fig. 2: Figure 2. Training Resnet-50 on GPUs

The Transformer is another widely-popular model that originated in the field of natural language processing (NLP). With more than *200 million* parameters, training the transformer (big) model heavily relies on data paralellism to achieve reasonable training times. We show that Tarantella distributed training also scales when using the Transformer for a translation task trained on the WMT14 English-German Translation dataset.

Figure 3 gives an insight of the time savings that Tarantella-based training can attain on a GPU machine such as the *HPC-DA* cluster, reaching a **34x speedup** for one epoch on *96* devices.

To find out more about training such models with Tarantella, take a look at our *tutorials*.

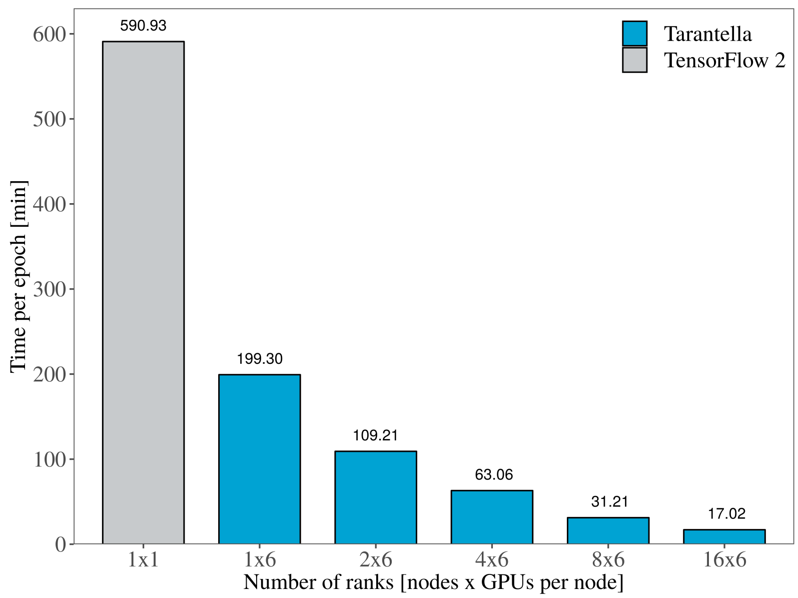

Fig. 3: Figure 3. Training the Transformer (big) on GPUs

CHAPTER

TWO

DISTRIBUTED DATA PARALLEL TRAINING

The following section explains the parallelization strategy Tarantella uses to provide distributed training. A full understanding thereof is, however, not required to be able to use the software. Please note the *points to consider* to achieve best performance and reproducibility.

2.1 The general idea

In order to parallelize the training of DNNs, different, complementary strategies are available. The conceptually simplest and most efficient one is called *data parallelism*. This strategy is already in use when deploying batched optimizers, such as stochastic gradient descent (SGD) or ADAM. In this case, input samples are grouped together in so-called mini-batches and are processed in parallel.

2.2 Distribution of mini-batches

Tarantella extends this scheme by splitting each mini-batch into a number of micro-batches, which are then executed on different devices (e.g., GPUs). In order to do this, the DNN is replicated on each device, which then processes part of the data independently of the other devices. During the *backpropagation* pass, partial results need to be accumulated via a so-called allreduce collective operation.

2.3 Overlapping communication with computation

Tarantella implements this communication scheme using the Global Address Space Programming Interface (GASPI). This allows in particular to overlap the communication needed to execute *allreduce* operations with the computation done in the *backpropagation* part of the DNN training. This is done by starting *allreduce* operations as soon as the required local incoming gradients are available, while continuing with *backpropagation* calculations at the same time. The final, accumulated gradients are only expected once the entire *backpropagation* is completed. This drastically mitigates the communication overhead introduced by the need to synchronize the different devices, and leads to higher scalability.

2.4 Tensor Fusion

The granularity at which Tarantella executes *allreduce* operations can be varied from one *allreduce* per layer (finest granularity) to one *allreduce* per iteration (coarsest granularity). Using coarser granularities, i.e., *fusing* gradient tensors, can lead to better bandwidth utilization, thus potentially increasing performance. *Tensor Fusion* is set up before the first iteration of training and incurs no additional communication overhead. Tarantella enables *Tensor Fusion* by default, but its granularity can be adjusted by the user, cf. *here*.

2.5 Model initialization and loading

In order to guarantee that all devices have the same copy of the DNN when training is initiated, the model needs to be communicated from one device to all the others. This is done in Tarantella via the use of a so-called broadcast operation. This scheme applies both when the weights of a DNN are initialized randomly, or loaded from a checkpoint. As Tarantella provides this functionality automatically, the user does not have to take care of it.

CHAPTER

THREE

DISTRIBUTED DATASETS

In order to process micro-batches independently on each device and to obtain the same results as in serial execution, the input data of each mini-batch has to be split and distributed among all devices.

Tarantella automatically takes care of this through the use of distributed datasets. The user simply provides Tarantella with a `tf.data.Dataset` that is batched with the mini-batch size. Tarantella will then automatically distribute the input data by splitting the mini-batch into individual micro-batches. Splitting is done at the level of samples (as opposed to e.g., files) to ensure *reproducibility* of serial results.

To guarantee reproducibility, it is also important that shuffling of samples is done in the same way on all devices. Tarantella does this using either the `seed` provided by the user, or a specific default seed. Please refer to the *Quick Start* for more details.

CHAPTER

FOUR

POINTS TO CONSIDER

4.1 Global versus local batch size

As explained above, when using data parallelism, there exists a *mini-batch size* (in the following also called global batch size or simply batch size) as well as a *micro-batch size* (also called local batch size). The former represents the number of samples that is averaged over in the loss function of the optimizer, and is equivalent to the (mini-)batch size used in non-distributed training. The latter is the number of samples that is processed locally by each of the devices per iteration.

Note: In Tarantella, the user always specifies the **global batch size**.

Using a strictly synchronous optimization scheme, and by carefully handling the data distribution, **Tarantella guarantees the reproducibility of DNN training results independently of the number of devices used**, as long as all hyperparameters (such as global batch size and learning rate) are kept constant.[1]

However, to achieve best performance for certain DNN operators (*Conv2D*, *Dense*, etc.) it is often advisable to *keep the local batch size constant*, and scale the global batch size with the number of devices used. This, in turn, will force you to adjust other hyperparameters, such as the learning rate, in order to converge to a comparable test accuracy, as observed for instance in [Shallue].

In practice, the use of a learning rate schedule with initial *warm up* and a *linear learning rate scaling* [Goyal], as it is described *here*, often suffices.

Tip: For best performance, scale the batch size with the number of devices used, and *adapt the learning rate schedule*.

4.2 Batch normalization layers

The issue of global versus local batch size particularly affects the layers that calculate (and learn) statistics over entire batches. A well-known example of this type of layer is batch normalization.

Caution: Tarantella always calculates batch statistics over **local batches**.

As a consequence, the training results for DNNs with batch-normalization layers **will not be identical when changing the number of devices, even if the global batch size stays the same.** At the moment, this can be circumvented by using normalization layers that do *not* average over entire batches, such as instance normalization [Ulyanov].

[1] This is strictly true, only when all randomness in TensorFlow is seeded or switched off, as explained in the *advanced topics*

Averaging over *local* batches instead of global batches should in practice have only minor influence on the quality of the final test accuracy. Note however, the extreme case of very small *local* batch sizes.

> **Caution:** Avoid using `BatchNormalization` layers when the global batch size divided by the number of devices used is *smaller than 16*. A warning is issued when this occurs.

In such cases, the local batches that are used to collect statistics are too small to obtain meaningful results. This will likely reduce the benefits of batch normalization, cf. for instance [Yang] and [Uppal]. In this case, please consider increasing the global batch size, or reducing the number of devices used.

4.3 Managing individual devices

Although Tarantella's user interface abstracts away most of the details of parallel programming, it is sometimes useful to be able to control Python code execution at device level. This can be achieved using the GASPI concept of a `rank`. Details on how to do this can be found in the *advanced topics*.

References

CHAPTER FIVE

INSTALLATION

Tarantella needs to be built from source. Since Tarantella is built on top of TensorFlow, you will require a recent version of it. Additionally, you will need an installation of the open-source communication libraries GaspiCxx and GPI-2, which Tarantella uses to implement distributed training.

Lastly, you will need pybind11, which is required for Python and C++ inter-communication.

In the following we will look at the required steps in detail.

5.1 Installing dependencies

5.1.1 Compiler and build system

Tarantella can be built using a recent gcc compiler with support for C++17 (starting with `gcc 7.4.0`). You will also need the build tool CMake (from version `3.12`).

5.1.2 Installing TensorFlow

First you will need to install TensorFlow. Supported versions start at `Tensorflow 2.4`, and they can be installed in a conda environment using pip, as recommended on the TensorFlow website.

In order to do that, first install conda on your system. Then, create and activate an environment for Tarantella:

```
conda create -n tarantella
conda activate tarantella
```

Now, you can install the latest supported TensorFlow version with:

```
conda install python=3.9
pip install --upgrade tensorflow==2.9.*
```

Tarantella requires at least Python `3.7`. Make sure the selected version also matches the TensorFlow requirements.

5.1.3 Installing pybind11

The next dependency you will need to install is pybind11, which is available through pip and conda. We recommend installing `pybind11` via conda:

```
conda install pybind11 -c conda-forge
```

5.1.4 Installing GPI-2

Next, you will need to download, compile and install the GPI-2 library. GPI-2 is an API for high-performance, asynchronous communication for large scale applications, based on the GASPI (Global Address Space Programming Interface) standard.

The currently supported versions start with `1.5`, and they need to be built with position independent flags (`-fPIC`). To download the required version, clone the GPI-2 git repository and checkout the latest `tag`:

```
git clone https://github.com/cc-hpc-itwm/GPI-2.git
cd GPI-2
git fetch --tags
git checkout -b v1.5.1 v1.5.1
```

Now, use autotools to configure and compile the code:

```
./autogen.sh
export GPI2_INSTALLATION_PATH=/your/gpi2/installation/path
CFLAGS="-fPIC" CPPFLAGS="-fPIC" ./configure --with-ethernet --prefix=${GPI2_INSTALLATION_
↪PATH}
make -j$(nproc)
```

where `${GPI2_INSTALLATION_PATH}` needs to be replaced with the path where you want to install GPI-2. Note the `--with-ethernet` option, which will use standard TCP sockets for communication. This is the correct option for laptops and workstations.

In case you want to use Infiniband, replace the above option with `--with-infiniband`. Now you are ready to install GPI-2 with:

```
make install
export PATH=${GPI2_INSTALLATION_PATH}/bin:$PATH
```

where the last two commands make the library visible to your system. If required, GPI-2 can be removed from the target directory by using `make uninstall`.

5.1.5 Installing GaspiCxx

GaspiCxx is a C++ abstraction layer built on top of the GPI-2 library, designed to provide easy-to-use point-to-point and collective communication primitives. Tarantella's communication layer is based on GaspiCxx and its PyGPI API for Python. Currently we support GaspiCxx version v1.2.0.

To install GaspiCxx and PyGPI, first download the latest release from the git repository:

```
git clone https://github.com/cc-hpc-itwm/GaspiCxx.git
cd GaspiCxx
git fetch --tags
git checkout -b v1.2.0 v1.2.0
```

GaspiCxx requires an already installed version of GPI-2, which should be detected at configuration time (as long as `${GPI2_INSTALLATION_PATH}/bin` is added to the current `${PATH}` as shown *above*).

Compile and install the library as follows, making sure the previously created conda environment is activated:

```
conda activate tarantella

mkdir build && cd build
export GASPICXX_INSTALLATION_PATH=/your/gaspicxx/installation/path
cmake -DBUILD_PYTHON_BINDINGS=ON    \
      -DBUILD_SHARED_LIBS=ON        \
      -DCMAKE_INSTALL_PREFIX=${GASPICXX_INSTALLATION_PATH} ../
make -j$(nproc) install
```

where `${GASPICXX_INSTALLATION_PATH}` needs to be set to the path where you want to install the library.

5.2 SSH key-based authentication

In order to use Tarantella on a cluster, make sure you can ssh between nodes without password. For details, refer to the *FAQ section*. In particular, to test Tarantella on your local machine, make sure you can ssh to `localhost` without password.

5.3 Building Tarantella from source

With all dependencies installed, we can now download, configure and compile Tarantella. To download the source code, simply clone the GitHub repository:

```
git clone https://github.com/cc-hpc-itwm/tarantella.git
cd tarantella
git checkout tags/v0.9.0 -b v0.9.0
```

Next, we need to configure the build system using CMake. For a standard out-of-source build, we create a separate `build` folder and run `cmake` in it:

```
conda activate tarantella

cd tarantella
mkdir build && cd build
export TARANTELLA_INSTALLATION_PATH=/your/installation/path
cmake -DCMAKE_INSTALL_PREFIX=${TARANTELLA_INSTALLATION_PATH} \
      -DCMAKE_PREFIX_PATH=${GASPICXX_INSTALLATION_PATH} ../
```

This will configure your installation to use the previously installed GPI-2 and GaspiCxx libraries. To install Tarantella on a cluster equipped with Infiniband capabilities, make sure that GPI-2 is installed with Infiniband support as shown *here*.

Now, we can compile and install Tarantella to `TARANTELLA_INSTALLATION_PATH`:

```
make -j$(nproc) install
export PATH=${TARANTELLA_INSTALLATION_PATH}/bin:${PATH}
```

5.4 [Optional] Building and running tests

In order to build Tarantella with tests, you will also need to install Boost (for C++ tests), and pytest (for Python tests). Additionally, the PyYAML and NetworkX libraries are required by some tests.

To install boost with the required *devel*-packages, under Ubuntu you can use

```
sudo apt install libboost-all-dev
```

while in Fedora you can use

```
sudo dnf install boost boost-devel
```

The other dependencies can be installed in the existing conda environment:

```
pip install -U pytest
pip install PyYAML==3.13
conda install networkx
```

After having installed these libraries, make sure to configure Tarantella with testing switched on:

```
cd tarantella
mkdir build && cd build
export LD_LIBRARY_PATH=`pwd`:${LD_LIBRARY_PATH}
export LD_LIBRARY_PATH=${GPI2_INSTALLATION_PATH}/lib64:${LD_LIBRARY_PATH}
export LD_LIBRARY_PATH=${GASPICXX_INSTALLATION_PATH}/lib:${LD_LIBRARY_PATH}

export PYTHONPATH=`pwd`:${PYTHONPATH}
export PYTHONPATH=${GASPICXX_INSTALLATION_PATH}/lib:${PYTHONPATH}

cmake -DENABLE_TESTING=ON ../
```

Now you can compile Tarantella and run its tests in the `build` directory:

```
make -j$(nproc)
ctest
```

5.5 [Optional] Building documentation

If you would like to build the documentation locally, run the following `cmake` command

```
cmake -DCMAKE_INSTALL_PREFIX=${TARANTELLA_INSTALLATION_PATH} -DBUILD_DOCS=ON ..
```

before compiling. This requires you to have Sphinx installed:

```
pip install -U sphinx
```

CHAPTER

SIX

QUICK START

This section explains how to get started using Tarantella to distributedly train an existing TensorFlow model. First, we will examine what changes have to be made to your code, before executing it on the command line with `tarantella`. Finally, we will present the features Tarantella currently supports and what important points need to be taken into account when using the framework.

6.1 Code example: LeNet-5 on MNIST

After having *built and installed* Tarantella we are ready to add distributed training support to an existing TensorFlow model. We will first illustrate all the necessary steps, using the well-known example of **LeNet-5** on the **MNIST** dataset. Although this is not necessarily a good use case to take full advantage of Tarantella's capabilities, it will allow you to simply copy-paste the code snippets and try them out, even on your laptop.

Let's get started!

```
import tensorflow as tf
from tensorflow import keras

# Initialize Tarantella (before doing anything else)
import tarantella as tnt

# Skip function implementations for brevity
[...]

args = parse_args()

# Create Tarantella model from a `keras.Model`
model = tnt.Model(lenet5_model_generator())

# Compile Tarantella model (as with Keras)
model.compile(optimizer = keras.optimizers.SGD(learning_rate=args.learning_rate),
              loss = keras.losses.SparseCategoricalCrossentropy(),
              metrics = [keras.metrics.SparseCategoricalAccuracy()])

# Load MNIST dataset (as with Keras)
shuffle_seed = 42
(x_train, y_train), (x_val, y_val), (x_test, y_test) = \
      mnist_as_np_arrays(args.train_size, args.val_size, args.test_size)

train_dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))
```

(continues on next page)

(continued from previous page)

```
train_dataset = train_dataset.shuffle(len(x_train), shuffle_seed)
train_dataset = train_dataset.batch(args.batch_size)
train_dataset = train_dataset.prefetch(tf.data.experimental.AUTOTUNE)

test_dataset = tf.data.Dataset.from_tensor_slices((x_test, y_test))
test_dataset = test_dataset.batch(args.batch_size)

# Train Tarantella model (as with Keras)
model.fit(train_dataset,
          epochs = args.number_epochs,
          verbose = 1)

# Evaluate Tarantella model (as with Keras)
model.evaluate(test_dataset, verbose = 1)
```

As you can see from the marked lines in the code snippet, you only need to add *two lines of code* to train LeNet-5 distributedly using Tarantella! Let us go through the code in some more detail, in order to understand what is going on.

First we need to import the Tarantella library:

```
import tarantella as tnt
```

Importing the Tarantella package will initialize the library and set up the communication infrastructure. Note that this should be done before executing any other code.

Next, we need to wrap the `keras.Model` object, generated by `lenet5_model_generator()`, into a `tnt.Model` object:

```
model = tnt.Model(lenet5_model_generator())
```

That's it!

All the necessary steps to distribute training and datasets will now be automatically handled by Tarantella. In particular, we still run `model.compile` on the new `model` to generate a compute graph, just as we would have done with a typical Keras model.

Next, we load the MNIST data for training and testing, and create `tf.data.Dataset` s from it. Note that we `batch` the dataset for training. This will guarantee that Tarantella is able to distribute the data later on in the correct way. Also note that the `batch_size` used here, is the same as for the original model, that is the *global* batch size. For details concerning local and global batch sizes have a look *here*.

Now we are able to train our `model` using `model.fit`, in the same familiar way used by the standard Keras interface. Note, however, that Tarantella is taking care of the proper distribution of the `train_dataset` in the background. All the possibilities of how to feed datasets to Tarantella are explained in more detail below. Lastly, we can evaluate the final accuracy of our `model` on the `test_dataset` using `model.evaluate`.

To test and run Tarantella in the next section, you can find a full version of the above example here.

6.2 Executing your model with `tarantella`

Next, let's execute our model distributedly using `tarantella` on the command line. Make sure to add the path to your installed *GaspiCxx* library to `LD_LIBRARY_PATH`:

```
export LD_LIBRARY_PATH=${GASPICXX_INSTALLATION_PATH}/lib:${LD_LIBRARY_PATH}
```

The simplest way to run the model is by passing its Python script to `tarantella`:

```
tarantella -- model.py
```

This will execute our model distributedly on a single node, using all the available GPUs. In case no GPUs can be found, `tarantella` will be executed in serial mode on the CPU, and a `WARNING` message will be issued. In case there are available GPUs, but we want to execute `tarantella` on CPUs nonetheless, we can add the `--no-gpu` option.

```
tarantella --no-gpu -- model.py
```

We can also set command line parameters for the python script `model.py`, which have to succeed the name of the script:

```
tarantella --no-gpu -- model.py --batch_size=64 --learning_rate=0.01
```

On a single node, we can also explicitly specify the number of TensorFlow instances we want to use. This is done with the `-n` option:

```
tarantella -n 4 -- model.py --batch_size=64
```

Here, `tarantella` will try to execute distributedly on 4 GPUs. If there are not enough GPUs available, `tarantella` will print a `WARNING` and run 4 instances of TensorFlow on the CPU instead. If there are no GPUs installed or the `--no-gpu` option is used, `tarantella` will not print a `WARNING`.

Next, let's run `tarantella` on multiple nodes. In order to do this, we need to provide `tarantella` with a `hostfile` that contains the `hostname` s of the nodes that we want to use:

```
$ cat hostfile
name_of_node_1
name_of_node_2
```

With this `hostfile` we can run `tarantella` on multiple nodes:

```
tarantella --hostfile hostfile -- model.py
```

In this case, `tarantella` uses *all* GPUs it can find. If no GPUs are available, `tarantella` will start *one* TensorFlow instance per node on the CPUs, and will issue a `WARNING` message. Again, this can be disabled by explicitly using the `--no-gpu` option.

As before, you can specify the number of GPUs/CPUs used per node explicitly with the option `--n-per-node <number>`:

```
tarantella --hostfile hostfile --n-per-node 4 --no-gpu -- model.py --batch_size=64
```

In this example, `tarantella` would execute 4 instances of TensorFlow on the CPUs of each node specified in `hostfile`.

Caution: `tarantella` requires all the names in the `hostfile` be **unique**, and all nodes be **homogeneous** (same number and type of CPUs and GPUs).

In addition, `tarantella` can be run with different levels of logging output. The log-levels that are available are `INFO`, `WARNING`, `DEBUG` and `ERROR`, and can be set with `--log-level`:

```
tarantella --hostfile hostfile --log-level INFO -- model.py
```

By default, `tarantella` will log on the *master rank* only. This can be changed by using the `--log-on-all-devices` option, which will print log messages for each *rank* individually.

Similarly, by default `tarantella` will print outputs from functions like `fit`, `evaluate` and `predict`, as well as callbacks only on the master rank. Sometimes, it might be useful to print outputs from all devices (e.g., for debugging), which can be switched on with the `--output-on-all-devices` option.

`tarantella` relies on GPI-2's tools for starting processes on multiple nodes (i.e., `gaspi_run`). To properly configure an execution, it will take care of exporting relevant environment variables (such as `PYTHONPATH`) for each process, and of generating an execution script from the user inputs. Details of this process can be monitored using the `--dry-run` option.

To add your own environment variables, add `-x ENV_VAR_NAME=VALUE` to your `tarantella` command. This option will ensure the environment variable `ENV_VAR_NAME` is exported on all ranks before executing the code. An example is shown below:

```
tarantella --hostfile hostfile -x DATASET=/scratch/data TF_CPP_MIN_LOG_LEVEL=1 -- model.
↪py
```

Both `DATASET` and `TF_CPP_MIN_LOG_LEVEL` will be exported as environment variables before executing `model.py`, in the same order they were specified to the command line.

Additionally, you can overwrite the *Tensor Fusion* threshold `tarantella` uses with `--fusion-threshold FUSION_THRESHOLD_KB` (cf. *here* and *here*), and set any number of environment variables, most notably `TNT_TENSORBOARD_ON_ALL_DEVICES`, as explained *here*.

To terminate a running `tarantella` instance, execute another `tarantella` command that specifies the `--cleanup` option in addition to the name of the program you want to interrupt.

```
tarantella --hostfile hostfile --cleanup -- model.py
```

The above command will stop the `model.py` execution on all the nodes provided in `hostfile`. You can also enable the `--force` flag to immediately terminate unresponsive processes.

Note: Any running `tarantella` execution can be terminated by using `Ctrl+c`, regardless of whether it was started on a single node or on multiple hosts.

6.3 Save and load Tarantella models

Storing and loading your trained `tnt.Model` is very simple.

Tarantella supports all the different ways in which you can load and store a `keras.Model` (for a guide look for instance here). In particular, you can:

- save the whole model (including the architecture, the weights and the state of the optimizer)
- save the model's architecture/configuration only
- save the model's weights only

6.3.1 Whole-model saving and loading

Saving the entire model including the architecture, weights and optimizer can be done via

```
model = ...  # get `tnt.Model`
model.save('path/to/location')
```

Alternatively, you could use `tnt.models.save_model('path/to/location')`, which works on both `keras.Model` s and `tnt.Model` s.

You can then load your model back using

```
import tarantella as tnt
model = tnt.models.load_model('path/to/location')
```

which will return an instance of `tnt.Model`.

If the saved model was previously compiled, `load_model` will also return a compiled model. Alternatively, you can deliberately load the model in an uncompiled state by passing the `compile = False` flag to `load_model`.

6.3.2 Architecture saving and loading

If you only want to save the configuration (that is the architecture) of your model (in memory), you can use one of the following functions:

- `tnt.Model.get_config`
- `tnt.Model.to_json`
- `tnt.Model.to_yaml` [supported up to TF 2.6]

The architecture without its original weights and optimizer can then be restored using:

- `tnt.models.model_from_config` / `tnt.Model.from_config`
- `tnt.models.model_from_json`
- `tnt.models.model_from_yaml` [supported up to TF 2.6]

respectively. Here is an example:

```
import tarantella as tnt
model = ...  # get `tnt.Model`
config = model.get_config()
new_model = tnt.models.model_from_config(config)
```

The same can be achieved through cloning:

```
import tarantella as tnt
model = ...  # get `tnt.Model`
new_model = tnt.models.clone_model(model)
```

6.3.3 Weights saving and loading

Storing and loading the weights of a model to/from memory can be done using the functions `tnt.Model.get_weights` and `tnt.Model.set_weights`, respectively. Saving and loading weights to/from disk is done using the functions `tnt.Model.save_weights` and `tnt.Model.load_weights`, respectively.

Here is an example how this can be used to restore a model:

```
import tarantella as tnt
model = ...  # get `tnt.Model`
config = model.get_config()
weights = model.get_weights()

# initialize a new model with original model's weights
new_model = tnt.models.model_from_config(config)
new_model.set_weights(weights)
```

6.3.4 Checkpointing via callbacks

Apart from saving and loading models manually, Tarantella also supports checkpointing via Keras' `ModelCheckpoint` callback, as it is described for instance here.

```
import tensorflow as tf
import tarantella as tnt

model = ...  # get `tnt.Model`

checkpoint_path = 'path/to/checkpoint/location'
model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
  filepath=checkpoint_path, monitor='val_acc', verbose=1, save_best_only=False,
  save_weights_only=False, mode='auto', save_freq='epoch', options=None)

model.fit(train_dataset,
          validation_data = val_dataset,
          epochs = 2,
          callbacks = [model_checkpoint_callback])
```

Note: All saving to the filesystem (including `tnt.Model.save` and `tnt.Model.save_weights`) by Tarantella will only be done on the master rank.

This is the default and will yield correct behavior when you are using a distributed filesystem. If you wish to explicitly save on all devices you can pass `tnt_save_all_devices = True` to `tnt.Model.save`, `tnt.Model.save_weights` and `tnt.models.save_model`.

6.4 Using distributed datasets

This section explains how to use Tarantella's distributed datasets.

The recommended way in which to provide your dataset to Tarantella is by passing a *batched* `tf.data.Dataset` to `tnt.Model.fit`. In order to do this, create a `Dataset` and apply the `batch` transformation using the (global) batch size to it. However, do not provide a value to `batch_size` in `tnt.Model.fit`, which would lead to double batching, and thus modified shapes for the input data.

Tarantella can distribute any `tf.data.Dataset`, regardless of the number and type of transformations that have been applied to it.

Note: When using the `dataset.shuffle` transformation without a `seed`, Tarantella will use a fixed default `seed`.

This guarantees that the input data is shuffled the same way on all devices, when no `seed` is given, which is necessary for consistency. However, when a random `seed` is provided by the user, Tarantella will use that one instead.

Tarantella also supports batched and unbatched `Dataset` s in `tnt.Model.fit` when setting the `tnt_micro_batch_size` argument. This can be useful to maximize performance in multi-node executions, as explained *here*. Keep in mind however, that Tarantella still expects the `Dataset` to be batched with the global batch size, and that the micro-batch size has to be consistent with the global batch size.[1] This is why it is recommended to use an unbatched `Dataset` when setting `tnt_micro_batch_size` explicitly.

Tarantella does not support any other way to feed data to `fit` at the moment. In particular, Numpy arrays, TensorFlow tensors and generators are not supported.

Tarantella's automatic data distribution can be switched off by passing `tnt_distribute_dataset = False` in `tnt.Model.fit`, in which case Tarantella will issue an `INFO` message. If a validation dataset is passed to `tnt.Model.fit`, it should also be batched with the global batch size. You can similarly switch off its automatic micro-batching mechanism by setting `tnt_distribute_validation_dataset = False`.

6.5 Callbacks

Tarantella fully supports all pre-defined Keras callbacks:

- `tf.keras.callbacks.CSVLogger`
- `tf.keras.callbacks.EarlyStopping`
- `tf.keras.callbacks.History`
- `tf.keras.callbacks.LearningRateScheduler`
- `tf.keras.callbacks.ModelCheckpoint`
- `tf.keras.callbacks.ProgbarLogger`
- `tf.keras.callbacks.ReduceLROnPlateau`
- `tf.keras.callbacks.RemoteMonitor`
- `tf.keras.callbacks.TensorBoard`
- `tf.keras.callbacks.TerminateOnNaN`

[1] That is, the global batch size must equal the micro batch size times the number of devices used.

All of these callbacks are implemented in such a way that the device-local, micro-batch information is accumulated over all devices. This leads to the same callback behavior as in a serial execution (using the full batch). That is, users do not need to make any modifications to their code when using Keras callbacks with Tarantella.

However, when using the TensorBoard callback, by default, Tarantella will only collect device-local information *on one device*. If you want to collect the local information on all devices use the environment variable `TNT_TENSORBOARD_ON_ALL_DEVICES`:

```
TNT_TENSORBOARD_ON_ALL_DEVICES=true tarantella -- model.py
```

Note: The explicit addition of `BaseLogger` callbacks is not supported in Tarantella.

6.5.1 Custom Callbacks

Any custom Keras callback can be used in a distributed fashion with Tarantella. To this end, define your own custom Keras callback as explained in the Writing Custom Callbacks guide.

Next, all you need to do is wrap the `keras_callback` into a `tnt.keras.callbacks.Callback` object and simply add it to the list of callbacks provided in the model training or inference methods:

```
class CustomCallback(keras.callbacks.Callback):
  def on_train_begin(self, logs = None):
    keys = list(logs.keys())
    print("Starting training; got log keys: {}".format(keys))
  ...

keras_callback = CustomCallback()

tnt_callback = tnt.keras.callbacks.Callback(keras_callback,
                                            aggregate_logs = True,
                                            run_on_all_ranks = True)
model.fit(train_dataset,
          epochs = 2,
          callbacks = [tnt_callback])
```

The execution of a `tnt.keras.callbacks.Callback` can be configured through the following parameters:

- `run_on_all_ranks` - defines whether the callback will be run on all devices or just the master rank (defaults to `True`). While most callbacks need to collect data from all the used devices, there are cases when this behavior is not desirable (e.g., a profiling callback might only need to measure timings on the master rank).
- `aggregate_logs` - specifies whether the logs need to be aggregated from all devices (defaults to `True`). For instance, `loss` values have to be aggregated across all micro-batches to provide the relevant batch-level information. Conversely, logs counting the number of `iterations` do not require aggregation, as the iteration counter is identical on all participating devices.

The `keras.callbacks.Callback` object can also be directly passed (without the wrapper) to the list of callbacks provided to the `model.fit` function. In this case, the `tnt.keras.callbacks.Callback` object is automatically created with the default parameter values.

6.5.2 Lambda Callbacks

A `LambdaCallback` allows users to create simple custom callbacks using a lambda function. To use this feature in Tarantella, create a Keras lambda callback as explained in the TensorFlow guide.

Then, wrap the callback object into a `tnt.keras.callbacks.Callback` as shown in the previous section.

```
# Print the batch number at the beginning of every batch.
batch_print_callback = LambdaCallback(on_batch_begin = lambda batch,logs: print(batch))

# Run the callback on the master rank only
tnt_print_callback = tnt.keras.callbacks.Callback(batch_print_callback,
                                                   aggregate_logs = False,
                                                   run_on_all_ranks = False)
```

6.5.3 Rank-Local Callbacks

There are cases when user-defined callbacks do not require distributed processing, such as callbacks that print information or measure runtimes. To configure a callback to run only on the *master rank*, wrap it as a `tnt.keras.callbacks.Callback` and set the constructor parameters as follows:

```
class MyCustomCallback(keras.callbacks.Callback):
  ...

keras_callback = MyCustomCallback()
tnt_callback = tnt.keras.callbacks.Callback(keras_callback,
                                             aggregate_logs = False,
                                             run_on_all_ranks = False)
```

Note that callbacks running on a single rank will only have access to local data corresponding to that rank. For instance, even though the models are identical on all ranks, a logging callback that displays metrics will only be aware of locally collected metrics, that is, metrics generated based on the micro-batches that the rank has processed.

6.6 Important points

There is a number of points you should be aware of when using Tarantella.

Note: Tarantella does not support custom training loops.

Instead of using custom training loops, please use `Model.fit(...)`.

Note: Tarantella supports all TensorFlow optimizers with the exception of `tf.keras.optimizers.Ftrl`.

Since the `Ftrl` optimizer does not use batches, it is not supported in Tarantella.

CHAPTER

SEVEN

TUTORIALS

This section delves into more advanced usage of Tarantella with the help of state-of-the-art models for two widely-used applications in Deep Learning:

- Image classification: ResNet-50
- Machine translation: Transformer

The image classification model architectures are imported through the tf.keras.applications module, available in recent TensorFlow releases.

The Transformer model presented in this tutorial is adapted from the TensorFlow Model Garden. While the model implementations and hyperparameters are unchanged to preserve compatibility with the TensorFlow official models, we provide simplified training schemes that allow for a seamless transition from basic serial training to distributed data parallelism using Tarantella.

7.1 Prerequisites

The model can be downloaded from the Tnt Models repository.

```
cd /your/models/path
git clone https://github.com/cc-hpc-itwm/tarantella_models

cd tarantella_models/src
export TNT_MODELS_PATH=`pwd`
```

This tutorial assumes the following dependencies are installed:

- TensorFlow 2.9.1
- Tarantella 0.9.0

For a step-by-step installation, follow the *Installation* guide.

7.2 ResNet-50

Deep Residual Networks (ResNets) represented a breakthrough in the field of computer vision, enabling deeper and more complex deep convolutional networks. Introduced in [He], ResNet-50 has become a standard model for image classification tasks, and has been shown to scale to very large number of nodes in data parallel training [Goyal].

7.2.1 Run Resnet-50 with Tarantella

Before running the model, we need to add it to the existing PYTHONPATH.

```
export PYTHONPATH=${TNT_MODELS_PATH}:${PYTHONPATH}
```

Furthermore, the ImageNet dataset needs to be installed and available on all the nodes that we want to use for training. TensorFlow provides convenience scripts to download datasets, in their datasets package that is installed as a dependency for the TensorFlow Model Garden. Install ImageNet to your local machine as described here.

```
export TNT_DATASETS_PATH=/path/to/downloaded/datasets

python -m tensorflow_datasets.scripts.download_and_prepare \
--datasets=imagenet2012 --data_dir=${TNT_DATASETS_PATH}
```

Let's assume we have access to two nodes (saved in hostfile) equipped with 4 GPUs each. We can now simply run the ResNet-50 as follows:

```
tarantella --hostfile ./hostfile --devices-per-node 4 \
-- ${TNT_MODELS_PATH}/models/image_classification/train_imagenet_main.py --data_dir=$
↪{TNT_DATASETS_PATH} \
                                                                         --model_
↪arch=resnet50 \
                                                                         --strategy=data␣
↪\
                                                                         --batch_
↪size=512 \
                                                                         --train_
↪epochs=90 \
                                                                         --epochs_
↪between_evals=10
```

The above command will train a ResNet-50 models on the 8 devices available in parallel for 90 epochs, as suggested in [Goyal] to achieve convergence. The --val_freq parameter specifies the frequency of evaluations of the *validation dataset* performed in between training epochs.

Note the --batch_size parameter, which specifies the global batch size used in training.

7.2.2 Implementation overview

We will now look closer into the implementation of the ResNet-50 training scheme. The main training steps reside in the `models/image_classification/train_imagenet_main.py` file.

The most important step in enabling data parallelism with Tarantella is to wrap the Keras model into a Tarantella model that uses data parallelism for speeding up training.

This is summarized below for the *ResNet50* model:

```
model = tf.keras.applications.resnet50.ResNet50(include_top=True, weights=None,␣
↪classes=1000,
                                                input_shape=(224, 224, 3), input_
↪tensor=None,
                                                pooling=None, classifier_activation=
↪'softmax')
...
if args.distribute == ParallelMethods.TNT:
  model = tnt.Model(model,
                    parallel_strategy = tnt.ParallelStrategy.DATA)
```

Next, the training procedure can simply be written down as it would be for a standard, TensorFlow-only model. No further changes are required to train the model in a distributed manner.

In particular, the ImageNet dataset is loaded and preprocessed as follows:

```
train_input_dataset = load_dataset(dataset_type='train',
                                   data_dir=args.data_dir, num_samples = args.train_num_
↪samples,
                                   batch_size=args.batch_size, dtype=tf.float32,
                                   drop_remainder=args.drop_remainder,
                                   shuffle_seed=args.shuffle_seed)
```

The load_dataset function reads the input files in `data_dir`, loads the training samples, and processes them into TensorFlow datasets.

The user only needs to pass the global `batch_size` value, and the Tarantella framework will ensure that the dataset is properly distributed among devices, such that:

- each device will process an independent set of samples
- each device will group the samples into micro batches, where the micro-batch size will be roughly equal to `batch_size / num_devices`. If the batch size is not a multiple of the number of ranks, the remainder samples will be equally distributed among the participating ranks, such that some ranks will use a micro-batch of `(batch_size / num_devices) + 1`.
- each device will apply the same set of transformations to its input samples as specified in the `load_dataset` function.

The advantage of using the *automatic dataset distribution* mechanism of Tarantella is that users can reason about their I/O pipeline without taking care of the details about how to distribute it.

Before starting the training, the model is compiled using a standard Keras optimizer and loss.

```
model.compile('optimizer' : tf.keras.optimizers.SGD(learning_rate=lr_schedule,␣
↪momentum=0.9),
              'loss' : tf.keras.losses.SparseCategoricalCrossentropy(),
              'metrics' : [tf.keras.metrics.SparseCategoricalAccuracy()])
```

We provide flags to enable the most commonly used Keras `callbacks`, such as the `TensorBoard` profiler, which can simply be passed to the `fit` function of the Tarantella model.

```
callbacks.append(tf.keras.callbacks.TensorBoard(log_dir = flags_obj.model_dir,
                                                profile_batch = 2))
```

If model checkpointing is required, it can be enabled through the `ModelCheckpoint` callback as usual (cf. *checkpointing models with Tarantella*).

```
callbacks.append(tf.keras.callbacks.ModelCheckpoint(ckpt_full_path, save_weights_
→only=True))
```

There is no need for any further changes to proceed with distributed training:

```
history = model.fit(train_dataset,
                    validation_data = val_dataset,
                    validation_freq=args.val_freq,
                    epochs=args.train_epochs,
                    callbacks=callbacks,
                    verbose=args.verbose)
```

7.2.3 Advanced topics

Scaling the batch size

Increasing the batch size provides a simple means to achieve significant training time speed-ups, as it leads to perfect scaling with respect to the steps required to achieve the target accuracy (up to some dataset- and model- dependent critical size, after which further increasing the batch size only leads to diminishing returns) [Shallue].

This observation, together with the fact that small local batch sizes decrease the efficiency of DNN operators, represent the basis for a standard technique in data parallelism: *using a fixed micro batch size and scaling the global batch size with the number of devices*.

Tarantella provides multiple mechanisms to set the batch size, as presented in the *Quick Start guide*.

In the case of ResNet-50, we specify the global batch size as a command line parameter, and let the framework divide the dataset into microbatches.

Scaling the learning rate

To be able to reach the same target accuracy when scaling the global batch size up, other hyperparameters need to be carefully tuned [Shallue]. In particular, adjusting the learning rate is essential for achieving convergence at large batch sizes. [Goyal] proposes to *scale the learning rate up linearly with the batch size* (and thus with the number of devices).

The scaled-up learning rate is set up at the begining of training, after which the learning rate evolves over the training steps based on a so-called *learning rate schedule*.

In our ResNet-50 example, we use a ExpDecayWithWarmupSchedule.

Another type of schedule that we have implemented is the PiecewiseConstantDecayWithWarmup schedule, which is similar to the schedule introduced by [Goyal].

In both schedules, when training starts, the learning rate is initialized to a large value that allows to explore more of the search space. The learning rate will then decay the closer the algorithm gets to convergence.

The initial learning rate in the *ExpDecayWithWarmupSchedule* is scaled linearly with the number of devices used as follows:

```
initial_learning_rate = base_learning_rate * num_ranks
```

Learning rate warm-up

Whereas scaling up the learning rate with the batch size is necessary, a large learning rate might degrade the stability of the optimization algorithm, especially in early training. A technique to mitigate this limitation is to *warm-up* the learning rate during the first epochs, particularly when using large batches [Goyal].

In our ResNet-50 example, the *ExpDecayWithWarmupSchedule* schedule starts with a small value for the learning rate, which then increases at every step (i.e., iteration), for a number of initial warmup_steps.

The `warmup_steps` value defaults to the number of iterations of the first five epochs, matching the schedule proposed by [Goyal]. After the `warmup_steps` are done, the learning rate value should reach the *scaled initial learning rate* introduced above.

```
def warmup():
  # Learning rate increases linearly per step.
  multiplier = self.warmup_rate * (step / self.warmup_steps)
  return tf.multiply(self.initial_learning_rate, multiplier)
```

7.3 Transformers

The Transformer is a Deep Neural Network widely used in the field of natural language processing (NLP), in particular for tasks such as machine translation. It was first proposed by [Vaswani].

7.3.1 Prerequisites

In the following we will assume that TensorFlow was installed in a `conda` environment called `tarantella`.

The Transformer model architecture can be obtained from the TensorFlow official Model Garden:

```
conda activate tarantella
pip install tf-models-official==2.9.1
```

7.3.2 Run the Transformer with Tarantella

The Tranformer training scheme can be found here, and has to be added to the existing `PYTHONPATH`:

```
export PYTHONPATH=${TNT_MODELS_PATH}/models/transformer:${PYTHONPATH}
```

We will follow the training procedure presented in [Vaswani], where the authors show results for training the *big* variant of the Transformer model on a machine translation dataset called WMT14.

To install the dataset, we will use the Tensorflow `datasets` package, which should have been already installed in your `conda` environment as a dependency for the TensorFlow Model Garden, and download the English-German dataset to match the results by [Vaswani]. Detailed instructions on how to obtain the dataset are provided in the TensorFlow documentation.

Now we can start training. Once again, let's assume we have access to two nodes (specified in `hostfile`) equipped with 4 GPUs each.

```
export WMT14_PATH=/path/to/the/installed/dataset

tarantella --hostfile ./hostfile --devices-per-node 4 \
-- ${TNT_MODELS_PATH}/models/transformer/transformer_tnt.py \
                     --data_dir=${WMT14_PATH} \
                     --vocab_file=${WMT14_PATH}/vocab.ende.32768 \
                     --bleu_ref=${WMT14_PATH}/newstest2014.de \
                     --bleu_source=${WMT14_PATH}/newstest2014.en \
                     --param_set=big \
                     --train_epochs=30 \
                     --epochs_between_evals=30 \
                     --batch_size=32736
```

The above command will select the `big` model implementation and train it on the 8 specified devices in a distributed fashion. To reach the target accuracy, [Vaswani] specifies that the model needs to be trained for `30` epochs.

The Transformer requires access to a vocabulary file, which contains all the tokens derived from the dataset. This is provided as the `vocab_file` parameter and is part of the pre-processed dataset.

After training, one round of evaluation is conducted using the `newstest2014` dataset to translate English sentences into German. The frequency of evaluation rounds can be changed by updating the *epochs_between_evals* parameter.

7.3.3 Implementation overview

The Transformer model itself is implemented and imported from the TensorFlow Model Garden. The training procedure and dataset loading and pre-processing do not require extensive changes to work with Tarantella. However, we provide a simplified version to highlight the usage of Tarantella with Keras training loops.

Thus, the Keras transformer model is created in TransformerTntTask class. Two different versions of the model are used, one for training (wrapped into a Tarantella model), and one for inference (serial Keras model).

```
self.train_model = create_model(internal_model, self.params, is_train = True)
# Enable distributed training
self.train_model = tnt.Model(self.train_model)

# The inference model is wrapped as a different Keras model that does not use labels
self.predict_model = create_model(internal_model, self.params, is_train = False)
```

To illustrate alternatives in the use of Tarantella, we distribute the data manually here, data_pipeline.py file, as explained in the *manually-distributed datasets* section. Alternatively, automatic dataset distribution could be used, as explained in the *Quick Start*.

To be able to manually split the dataset across ranks, we need access to **rank IDs** and the **total number of ranks**, which are then passed to the IO pipeline.

The *Advanced Topics* section explains the API Tarantella exposes to access ranks.

```
train_ds = data_pipeline.train_input_fn(self.params,
                                        shuffle_seed = 42,
                                        num_ranks = tnt.get_size(),
                                        rank = tnt.get_rank())
```

Here, the `data_pipeline.train_input_fn` reads in the dataset and applies a series of transformations to convert it into a batched set of sentences.

Next, the user can also create callbacks, which can then be simply passed on to the training function.

```
callbacks.append(tf.keras.callbacks.TensorBoard(log_dir=self.flags_obj.model_dir))
```

Finally, we can call `model.fit` to start distributed training on all devices:

```
history = model.fit(train_ds,
                    tnt_distribute_dataset = False,
                    epochs=self.params["train_epochs"],
                    callbacks=callbacks,
                    verbose=1)
```

In the following sections we will show how we modify the `fit` loop to allow for a customized evaluation of the trained model.

7.3.4 Important points

Customized behavior based on rank

Although all ranks participating in data parallel training use identical replicas of the same model and make progress in sync, there are cases when certain tasks should be executed on a specific rank (or group or ranks). To this end, Tarantella provides a number of functions to identify the rank ID and allow users to add customized behavior based on rank, as decribed in this *section*.

In the case of the Transformer model, we want to use the rank information to perform several tasks:

- print logging messages

```
if tnt.is_master_rank():
  logging.info("Start train")
```

- distribute datasets manually among participating devices
- execute other models, such as a modified, serial version of the Tarantella model for *inference*
- enable certain callbacks only on one rank (e.g., profiling callbacks)

```
if self.flags_obj.enable_time_history:
  time_callback = keras_utils.TimeHistory(self.params["batch_size"],
                                          self.params["num_sentences"],
                                          logdir = None)
  tnt_time_callback = tnt.keras.callbacks.Callback(time_callback,
                                                   aggregate_logs = False,
                                                   run_on_all_ranks = False)
  callbacks.append(tnt_time_callback)
```

Such callbacks only collect local data corresponding to the specific rank where they are executed. In this example, the *TimeHistory* callback will measure timings only on the `master_rank`. While iteration and epoch runtimes should be the same on all ranks (as all ranks train in sync), other metrics such as accuracy will only be computed based on the local data available to the rank.

A callback that should be executed on a single rank has to be wrapped within a `tnt.keras.callbacks.Callback`, to explicitely disable distributed execution (as described in the *callbacks guide*).

Using manually-distributed datasets

Typically, it is the task of the framework to automatically handle batched datasets, such that each rank only processes its share of the data, as explained in the *Quick Start guide*.

However, there are complex scenarios when the user might prefer to manually build the dataset slices corresponding to each rank. Tarantella allows the user to disable the automatic distribution mechanism by passing `tnt_distribute_dataset = False` to the `model.fit` function.

This is how it is done in the case of the Transformer:

```
history = self.train_model.fit(train_ds,
                               callbacks = callbacks,
                               tnt_distribute_dataset = False,
                               initial_epoch = epoch,
                               epochs = epoch + min(self.params["epochs_between_evals"],
                                                    self.params["train_epochs"]-epoch),
                               verbose = 2)
```

Also note the use of `initial_epoch` and `epochs`. This combination of parameters is necessary to allow evaluation rounds in between training epochs, when a validation dataset cannot be simply passed to `model.fit`. In particular, our transformer implementation features a different model for inference, as described *below*.

Now that automatic distribution is disabled, let us take a look at how to split the dataset manually among devices. The input data processing is implemented in data_pipeline.py.

In the case of the Transformer model, the global `batch_size` stands for the total number of input tokens processed in a single iteration. However, as the training is performed in (fixed-sized) sentences, our global `batch_size` used for training will be in fact the number of sentences comprised in such a batch.

Furthermore, we need to divide the number of sentences across ranks, such that each rank can work on a separated shard of `micro_batch_size` sentences. Finally, the dataset itself needs to be batched using the `micro_batch_size` and each device instructed to select its own shard:

```
number_batch_sentences = batch_size // max_length

micro_batch_size = number_batch_sentences // num_ranks

# Batch the sentences and select only the shard (subset)
# corresponding to the current rank
dataset = dataset.padded_batch(micro_batch_size,
                               ([max_length], [max_length]),
                               drop_remainder=True)
dataset = dataset.shard(num_ranks, rank)
```

Mixing Keras and Tarantella models

An essential aspect of the Transformer model is that it operates on slightly different model versions during training and inference. While in training the model works on encoded tokens, inference requires translation to and from plain text. Thus, the model needs to use modified input and output layers for each of these tasks.

To illustrate the way a Tarantella model can work alongside a typical Keras model, we only execute the training phase on the Transformer within a (distributed) Tarantella model.

Take a look at the model creation function. It builds two different Keras models depending on whether training is enabled or not, both of them based on the same *internal model* (i.e., using the same learned weights).

Now, when initializing our Transformer task, we only wrap one of the models as a `tnt.Model`:

```
# Transformer model used both as Tarantella model (in training) and as a serial
# model for inference
internal_model = transformer.Transformer(self.params, name="transformer_v2")

# The train model includes an additional logits layer and a customized loss
self.train_model = create_model(internal_model, self.params, is_train = True)
# Enable distributed training
self.train_model = tnt.Model(self.train_model)

# The inference model is wrapped as a different Keras model that does not use labels
self.predict_model = create_model(internal_model, self.params, is_train = False)
```

Training can now proceed as usual, by only calling the `fit` method on our `train_model`. We can however design our training loop to stop every `epochs_between_evals` epochs, evaluate the training accuracy using the serial `predict_model`, and then continue from where it left off.

```
for epoch in range(0, self.params["train_epochs"], self.params["epochs_between_evals"]):
  # as our dataset is distributed manually, disable the automatic Tarantella distribution
  history = self.train_model.fit(train_ds,
                                 callbacks = callbacks,
                                 tnt_distribute_dataset = False,
                                 initial_epoch = epoch,
                                 epochs = epoch + min(self.params["epochs_between_evals
↪"],
                                                      self.params["train_epochs"]-epoch),
                                 verbose = 2)

  if tnt.is_master_rank():
    eval_stats = self.eval()
```

The `self.eval()` method performs the translation on the test dataset using the standard Keras `predict_model`.

```
def eval(self):
  ...
  uncased_score, cased_score = transformer_main.evaluate_and_log_bleu(
                                          self.predict_model,
                                          self.params,
                                          self.flags_obj.bleu_source,
                                          self.flags_obj.bleu_ref,
                                          self.flags_obj.vocab_file)
```

A validation dataset can be provided in the form of a pair of input files specified at the command line as `bleu_source` and `bleu_ref`. If the validation dataset exists, the evaluation method will compute and log the corresponding BLEU scores (both case-sensitive and case-insensitive) serially.

CHAPTER

EIGHT

ADVANCED TOPICS

This guide covers a number of advanced topics, such as performance, reproducibility and user customization.

8.1 GASPI ranks

To distribute the DNN training, Tarantella starts multiple processes on different devices. These processes will be assigned different IDs by the GPI-2 communication library, in order to organize communication and synchronization between the different devices. These IDs are called *ranks*. Usually, Tarantella abstracts away the concept of *ranks*, in such a way that Tarantella's user interface is essentially the same as Keras' user interface.

However, sometimes it is useful, to execute a specific part of code only on one or a subgroup of all ranks. In particular, one sometimes wants to execute a code block on the device that started `tarantella`, the so-called *master rank*.

To access ranks, Tarantella provides the following functions

- `tnt.get_rank()`
- `tnt.get_size()`
- `tnt.get_master_rank()`
- `tnt.is_master_rank()`

`tnt.get_rank()` returns the ID of the local rank. `tnt.get_size()` returns the total number of ranks. `tnt.get_master_rank()` and `tnt.is_master_rank()` return the ID of the master rank and a boolean for whether the local rank is the master rank or not, respectively.

Here is a simple example, where using the master rank can be useful to print notifications only once to `stdout`:

```
if tnt.is_master_rank():
  print("Printing from the master rank")
```

More usage examples can be found in the *Tutorials* section.

8.2 Using local batch sizes

As it has been stated in the *points to consider*, when using Tarantella the user always specifies the *global* batch size. This has the advantage that the optimization process during the training of a DNN, and in particular the loss function do not depend on the number of devices used during execution.

However, when the number of devices becomes very large, the (device-local) micro-batch size might become so small, that DNN kernel implementations are less efficient, resulting in overall performance degradation. This is why it is in practice often advisable to scale the global batch size with the number of nodes. This will often lead to linear speedups in terms of the time to accuracy when increasing the number of devices used, at least up to some *critical batch size*, cf. [Shallue] and [McCandlish]. Changing the batch size of the optimizer will however also imply the need to adapt the learning rate schedule.

For details, cf. for instance the *ResNet-50 tutorial*.

If you decide to scale the batch size with the number of nodes, Tarantella provides two different ways to achieve this easily. The first option is to multiply the local batch size (for instance passed via a command-line parameter) with the number of devices used, batch your dataset with it, and call `fit` on it:

```
micro_batch_size = args.micro_batch_size
batch_size = tnt.get_size() * micro_batch_size
train_dataset = train_dataset.batch(batch_size)
tnt_model.fit(train_dataset)
```

As a second option you can also pass the local batch size directly to the `tnt_micro_batch_size` parameter in fit, and leave your dataset unbatched:

```
micro_batch_size = args.micro_batch_size
tnt_model.fit(train_dataset,
              tnt_micro_batch_size = micro_batch_size)
```

This parameter is also available in `evaluate` and `predict`. In addition, `fit` also supports setting the validation set micro batch size in a similar way with `tnt_validation_micro_batch_size`. For more information, please also read *using distributed datasets*.

8.3 Setting tensor fusion threshold

Tarantella automatically uses *Tensor Fusion* with a default threshold of 32kB. This threshold specifies the minimal size of local buffers in *allreduce* communication operations used to accumulate partial gradients during *backpropagation*.

Note that the threshold value implies a trade-off between the potential to utilize network bandwidth, and the overlap of computation and communication during *backpropagation*. The larger the threshold, the more bandwidth-bound the *allreduce* algorithm will get, but the less potential there will be to overlap its execution with kernel computations. Also note that the ideal threshold value will generally depend on the number of nodes used.

To change the default value, you can pass a threshold value in kB to `tarantella`:

```
tarantella --hostfile hostfile --fusion-threshold=<FUSION_THRESHOLD_KB> -- model.py
```

8.4 Performance aspects

To increase execution performance on CPUs, it is often desirable to bind processes to physical cores or groups of cores in order to improve data locality and reduce context switching.

Tarantella provides two command-line flags to enable rank pinning to physical sockets. They rely on the numactl utility to detect existing NUMA domains and pin processes to them.

Tarantella pinning flags allow users to:

- pin each Tarantella process deployed on a host to a separate socket (through the `--pin-to-socket` flag)
- pin memory alocation for each Tarantella process to the socket memory (through the `--pin-memory-to-socket` flag).

Using only `--pin-to-socket` will result in memory being only preferentially allocated on the socket memory, but potentially using memory from other NUMA domains when necessary.

The example below illustrates the usage of the `--pin-to-socket` and `--pin-memory-to-socket` flags to start two Tarantella ranks on each host listed in `hostfile`, each of them pinned to a different socket.

```
tarantella --hostfile hostfile --npernode 2 --pin-to-socket -- model.py
```

8.5 Python Interpreter

The `tarantella` CLI can be used as generic tool for executing code on multiple devices simultaneously. While usually the executed program is a Python file, Tarantella uses the Python interpreter it finds in the current `$PATH`. Changing the interpretor can be easily achieved by using the `--python-interpreter` flag:

```
tarantella --hostfile hostfile --npernode 2 --python-interpreter=/path/to/python --␣
↪model.py
```

Additionally, the user can also execute binary files that do not require any Python support by simply passing an empty string to the `--python-interpreter` flag.

A typical use case for the interpreter is to enable the usage of other tools that can only be enabled from the command line, such as checking for memory leaks in a parallel program with `valgrind`

```
tarantella -n 2 --python-interpreter="valgrind  --leak-check=yes \
                                      --track-origins=yes --tool=memcheck \
                                      python" \
           -- model.py
```

8.6 Reproducibility

Reproducibility is a very important prerequisite to obtain meaningful results in scientific computing and research. Unfortunately, using stochastic algorithms, pseudo random generators and having to deal with the pitfalls of floating-point arithmetics, it is particularly difficult to achieve reproducibility in Deep Learning research.

In order to be able to reproduce results obtained with TensorFlow, when running in a multi-node/multi-device setting with Tarantella, one needs to meet at least the following requirements:

- set the random seed with `tf.random.set_seed(seed)`
- set the environment variable `os.environ['TF_DETERMINISTIC_OPS'] = '1'`

- set the environment variable `os.environ['TF_CUDNN_DETERMINISTIC'] = '1'`
- set the random seed when using layers such as `keras.layers.Dropout`
- set the shuffle seeds when using `tf.data.Dataset` with `shuffle(seed=seed)` and `list_files(seed=seed)`
- set the `deterministic` parameter to `True` in `Dataset` transformations such as `interleave` and `map`

Additionally, Python-specific random generators might need to be seeded, in particular:

- `random.seed(seed)`
- `numpy.random.seed(seed)`
- `os.environ['PYTHONHASHSEED'] = str(seed)`

For more details, take a look at a more in-depth study of non-determinism sources in TensorFlow.

CHAPTER NINE

FREQUENTLY ASKED QUESTIONS (FAQ)

This is a list of frequently asked questions about Tarantella. Please feel free to *suggest new ones*!

Question

How can I ssh to `localhost` without password?

In order to run Tarantella programs, you will need to be able to ssh to `localhost` without password. In order to do that generate `ssh` keys first:

```
cd ~/.ssh
ssh-keygen
```

Make sure not to overwrite existing keys. When asked for a passphrase, `Enter passphrase (empty for no passphrase):`, simply leave empty and return with enter. Also take specific care to set correct user rights on all files in `.ssh`, cf. for instance here. Next, append the public key to the `authorized_keys` file:

```
cat id_rsa.pub >> authorized_keys
```

Now, install and start an ssh server, e.g., openssh-server on Fedora. More details can be found for instance here.

Question

I get an execution error `GPI library initialization incorrect environment vars` when trying to run my script. What shall I do?

Most likely you are running your program with `python my_script.py` or `./my_script.py`. Please make sure to execute your code with `tarantella -- my_script.py` instead.

Question

I get an execution error `GPI library initialization general error`. What shall I do?

This error occurs when the GPI-2 library tries to connect to a previously used socket, which is not yet released. Try to re-run your code after a short while so that the port becomes available again.

Question

The execution seems to stall. What shall I do?

Please use the `tarantella --cleanup` command to kill any processes that might be still running from a previous (aborted) call to `tarantella` as shown *here*. Note that you can also interrupt a running `tarantella` instance (distributed on multiple nodes) by using `Ctrl+c`.

Question

When trying to build Tarantella, CMake cannot find pybind11:

`Could not find a package configuration file provided by "pybind11" with any of the following names: [...]`

What shall I do?

This error occurs when pybind11 is installed using pip. Please use conda instead, as recommended in the *installation guide*.

Question

When trying to build Tarantella, CMake does not detect the Python interpreter from the active conda environment. What shall I do?

You will need to manually add the path to the conda environment's `bin` directory to your `PATH`. You will also need to specify the path to the python library on the command line when configuring Tarantella:

```
PATH_TO_CONDA_ENV=/path/to/conda/env
export PATH=${PATH_TO_CONDA_ENV}/bin:${PATH}
cmake -DPYTHON_EXECUTABLE=${PATH_TO_CONDA_ENV}/bin/python \
      -DPYTHON_LIBRARY=${PATH_TO_CONDA_ENV}/lib ../
```

Question

Why do I get runtime errors when I compile Tarantella using *clang*?

Currently, Tarantella can be built properly only by using *gcc*.

The *clang* compiler relies on a different standard library (*libc++* instead of *libstdc++* that is used by *gcc*).

However, the TensorFlow pip/conda packages for Linux are compiled using *gcc*. The *tnt_tfops* library in Tarantella is linked against Tensorflow, which leads to linking errors at runtime if the two libraries expect a different standard library implementation.

Question

I get *undefined symbol* errors in the *libtnt-tfops.so* library at runtime. What can I do?

Such errors might be due to a TensorFlow version mismatch between Tarantella and the loaded Conda environment. Make sure to use the same Conda environment that was active when compiling Tarantella.

Question

Why does loading a Tarantella or Keras model from YAML fail?

Make sure to have the *PyYAML* Python package installed in your environment, using version *3.13* or below. Newer versions of *PyYAML* do not work with TensorFlow model loading.

```
pip install PyYAML==3.13
```

Question

Can I install Tarantella on MacOS?

Tarantella is only supported on Linux systems, as its GPI-2 dependency is built on top of a Linux kernel API called *epoll*.

CHAPTER

TEN

BUG REPORTS

To report a bug please open an issue on GitHub.

When opening an issue, please make sure you include as much information as possible about the issue. Please consider providing at least the following points:

- What version of Tarantella you are using
- What linux distribution you are using (e.g., Linux Ubuntu 20.04)
- What kind of system you are experiencing the issue on (type and number of nodes, network interconnect, etc.)
- What did you expect to see and what have you seen instead
- What exact steps are needed to reproduce the issue

CHAPTER

ELEVEN

FEATURE REQUESTS

For contributions other than modifications to the source code, as for example suggestions of a feature or enhancement, please open an issue on GitHub with the label `Feature`.

When providing a feature request, please consider providing at least the following information:

- What is the current behavior of the software and how does the feature improve it
- Who would benefit from the feature
- Is there a relevant reference or academic paper describing the feature
- Are you willing to contribute to and/or maintain the feature

CHAPTER

TWELVE

CONTRIBUTING

Thank you for considering to contribute to Tarantella.

There are many ways to contribute to Tarantella. This includes sharing DNN models distributed through Tarantella, providing suggestions on improving the documentation, as well as contributing with changes to the Tarantella code base. Even by simply providing suggestions on how we can *improve Tarantella* and help spreading the word about it are great ways to contribute and make Tarantella better software.

If you want to contribute to Tarantella with changes to its code, please open a pull request on GitHub.

CHAPTER

THIRTEEN

CONTACT

In case you have any feature request, or want to report a bug please follow *these instructions*.

If you consider contributing to Tarantella, please follow the instructions *here*.

If you have any further questions or comments please email to support@tarantella.org

CHAPTER

FOURTEEN

LICENSE

```
                    GNU GENERAL PUBLIC LICENSE
                       Version 3, 29 June 2007

 Copyright (C) 2007 Free Software Foundation, Inc. <https://fsf.org/>
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

                            Preamble

  The GNU General Public License is a free, copyleft license for
software and other kinds of works.

  The licenses for most software and other practical works are designed
to take away your freedom to share and change the works.  By contrast,
the GNU General Public License is intended to guarantee your freedom to
share and change all versions of a program--to make sure it remains free
software for all its users.  We, the Free Software Foundation, use the
GNU General Public License for most of our software; it applies also to
any other work released this way by its authors.  You can apply it to
your programs, too.

  When we speak of free software, we are referring to freedom, not
price.  Our General Public Licenses are designed to make sure that you
have the freedom to distribute copies of free software (and charge for
them if you wish), that you receive source code or can get it if you
want it, that you can change the software or use pieces of it in new
free programs, and that you know you can do these things.

  To protect your rights, we need to prevent others from denying you
these rights or asking you to surrender the rights.  Therefore, you have
certain responsibilities if you distribute copies of the software, or if
you modify it: responsibilities to respect the freedom of others.

  For example, if you distribute copies of such a program, whether
gratis or for a fee, you must pass on to the recipients the same
freedoms that you received.  You must make sure that they, too, receive
or can get the source code.  And you must show them these terms so they
know their rights.

  Developers that use the GNU GPL protect your rights with two steps:
```

(continues on next page)

(continued from previous page)

```
(1) assert copyright on the software, and (2) offer you this License
giving you legal permission to copy, distribute and/or modify it.

  For the developers' and authors' protection, the GPL clearly explains
that there is no warranty for this free software.  For both users' and
authors' sake, the GPL requires that modified versions be marked as
changed, so that their problems will not be attributed erroneously to
authors of previous versions.

  Some devices are designed to deny users access to install or run
modified versions of the software inside them, although the manufacturer
can do so.  This is fundamentally incompatible with the aim of
protecting users' freedom to change the software.  The systematic
pattern of such abuse occurs in the area of products for individuals to
use, which is precisely where it is most unacceptable.  Therefore, we
have designed this version of the GPL to prohibit the practice for those
products.  If such problems arise substantially in other domains, we
stand ready to extend this provision to those domains in future versions
of the GPL, as needed to protect the freedom of users.

  Finally, every program is threatened constantly by software patents.
States should not allow patents to restrict development and use of
software on general-purpose computers, but in those that do, we wish to
avoid the special danger that patents applied to a free program could
make it effectively proprietary.  To prevent this, the GPL assures that
patents cannot be used to render the program non-free.

  The precise terms and conditions for copying, distribution and
modification follow.

                       TERMS AND CONDITIONS

  0. Definitions.

  "This License" refers to version 3 of the GNU General Public License.

  "Copyright" also means copyright-like laws that apply to other kinds of
works, such as semiconductor masks.

  "The Program" refers to any copyrightable work licensed under this
License.  Each licensee is addressed as "you".  "Licensees" and
"recipients" may be individuals or organizations.

  To "modify" a work means to copy from or adapt all or part of the work
in a fashion requiring copyright permission, other than the making of an
exact copy.  The resulting work is called a "modified version" of the
earlier work or a work "based on" the earlier work.

  A "covered work" means either the unmodified Program or a work based
on the Program.

  To "propagate" a work means to do anything with it that, without
```

(continues on next page)

(continued from previous page)

```
permission, would make you directly or secondarily liable for
infringement under applicable copyright law, except executing it on a
computer or modifying a private copy.  Propagation includes copying,
distribution (with or without modification), making available to the
public, and in some countries other activities as well.

  To "convey" a work means any kind of propagation that enables other
parties to make or receive copies.  Mere interaction with a user through
a computer network, with no transfer of a copy, is not conveying.

  An interactive user interface displays "Appropriate Legal Notices"
to the extent that it includes a convenient and prominently visible
feature that (1) displays an appropriate copyright notice, and (2)
tells the user that there is no warranty for the work (except to the
extent that warranties are provided), that licensees may convey the
work under this License, and how to view a copy of this License.  If
the interface presents a list of user commands or options, such as a
menu, a prominent item in the list meets this criterion.

  1. Source Code.

  The "source code" for a work means the preferred form of the work
for making modifications to it.  "Object code" means any non-source
form of a work.

  A "Standard Interface" means an interface that either is an official
standard defined by a recognized standards body, or, in the case of
interfaces specified for a particular programming language, one that
is widely used among developers working in that language.

  The "System Libraries" of an executable work include anything, other
than the work as a whole, that (a) is included in the normal form of
packaging a Major Component, but which is not part of that Major
Component, and (b) serves only to enable use of the work with that
Major Component, or to implement a Standard Interface for which an
implementation is available to the public in source code form.  A
"Major Component", in this context, means a major essential component
(kernel, window system, and so on) of the specific operating system
(if any) on which the executable work runs, or a compiler used to
produce the work, or an object code interpreter used to run it.

  The "Corresponding Source" for a work in object code form means all
the source code needed to generate, install, and (for an executable
work) run the object code and to modify the work, including scripts to
control those activities.  However, it does not include the work's
System Libraries, or general-purpose tools or generally available free
programs which are used unmodified in performing those activities but
which are not part of the work.  For example, Corresponding Source
includes interface definition files associated with source files for
the work, and the source code for shared libraries and dynamically
linked subprograms that the work is specifically designed to require,
such as by intimate data communication or control flow between those
```

(continues on next page)

(continued from previous page)

```
subprograms and other parts of the work.

  The Corresponding Source need not include anything that users
can regenerate automatically from other parts of the Corresponding
Source.

  The Corresponding Source for a work in source code form is that
same work.

  2. Basic Permissions.

  All rights granted under this License are granted for the term of
copyright on the Program, and are irrevocable provided the stated
conditions are met.  This License explicitly affirms your unlimited
permission to run the unmodified Program.  The output from running a
covered work is covered by this License only if the output, given its
content, constitutes a covered work.  This License acknowledges your
rights of fair use or other equivalent, as provided by copyright law.

  You may make, run and propagate covered works that you do not
convey, without conditions so long as your license otherwise remains
in force.  You may convey covered works to others for the sole purpose
of having them make modifications exclusively for you, or provide you
with facilities for running those works, provided that you comply with
the terms of this License in conveying all material for which you do
not control copyright.  Those thus making or running the covered works
for you must do so exclusively on your behalf, under your direction
and control, on terms that prohibit them from making any copies of
your copyrighted material outside their relationship with you.

  Conveying under any other circumstances is permitted solely under
the conditions stated below.  Sublicensing is not allowed; section 10
makes it unnecessary.

  3. Protecting Users' Legal Rights From Anti-Circumvention Law.

  No covered work shall be deemed part of an effective technological
measure under any applicable law fulfilling obligations under article
11 of the WIPO copyright treaty adopted on 20 December 1996, or
similar laws prohibiting or restricting circumvention of such
measures.

  When you convey a covered work, you waive any legal power to forbid
circumvention of technological measures to the extent such circumvention
is effected by exercising rights under this License with respect to
the covered work, and you disclaim any intention to limit operation or
modification of the work as a means of enforcing, against the work's
users, your or third parties' legal rights to forbid circumvention of
technological measures.

  4. Conveying Verbatim Copies.
```

(continues on next page)

(continued from previous page)

```
  You may convey verbatim copies of the Program's source code as you
receive it, in any medium, provided that you conspicuously and
appropriately publish on each copy an appropriate copyright notice;
keep intact all notices stating that this License and any
non-permissive terms added in accord with section 7 apply to the code;
keep intact all notices of the absence of any warranty; and give all
recipients a copy of this License along with the Program.

  You may charge any price or no price for each copy that you convey,
and you may offer support or warranty protection for a fee.

  5. Conveying Modified Source Versions.

  You may convey a work based on the Program, or the modifications to
produce it from the Program, in the form of source code under the
terms of section 4, provided that you also meet all of these conditions:

    a) The work must carry prominent notices stating that you modified
    it, and giving a relevant date.

    b) The work must carry prominent notices stating that it is
    released under this License and any conditions added under section
    7.  This requirement modifies the requirement in section 4 to
    "keep intact all notices".

    c) You must license the entire work, as a whole, under this
    License to anyone who comes into possession of a copy.  This
    License will therefore apply, along with any applicable section 7
    additional terms, to the whole of the work, and all its parts,
    regardless of how they are packaged.  This License gives no
    permission to license the work in any other way, but it does not
    invalidate such permission if you have separately received it.

    d) If the work has interactive user interfaces, each must display
    Appropriate Legal Notices; however, if the Program has interactive
    interfaces that do not display Appropriate Legal Notices, your
    work need not make them do so.

  A compilation of a covered work with other separate and independent
works, which are not by their nature extensions of the covered work,
and which are not combined with it such as to form a larger program,
in or on a volume of a storage or distribution medium, is called an
"aggregate" if the compilation and its resulting copyright are not
used to limit the access or legal rights of the compilation's users
beyond what the individual works permit.  Inclusion of a covered work
in an aggregate does not cause this License to apply to the other
parts of the aggregate.

  6. Conveying Non-Source Forms.

  You may convey a covered work in object code form under the terms
of sections 4 and 5, provided that you also convey the
```

(continues on next page)

(continued from previous page)

```
machine-readable Corresponding Source under the terms of this License,
in one of these ways:

    a) Convey the object code in, or embodied in, a physical product
    (including a physical distribution medium), accompanied by the
    Corresponding Source fixed on a durable physical medium
    customarily used for software interchange.

    b) Convey the object code in, or embodied in, a physical product
    (including a physical distribution medium), accompanied by a
    written offer, valid for at least three years and valid for as
    long as you offer spare parts or customer support for that product
    model, to give anyone who possesses the object code either (1) a
    copy of the Corresponding Source for all the software in the
    product that is covered by this License, on a durable physical
    medium customarily used for software interchange, for a price no
    more than your reasonable cost of physically performing this
    conveying of source, or (2) access to copy the
    Corresponding Source from a network server at no charge.

    c) Convey individual copies of the object code with a copy of the
    written offer to provide the Corresponding Source.  This
    alternative is allowed only occasionally and noncommercially, and
    only if you received the object code with such an offer, in accord
    with subsection 6b.

    d) Convey the object code by offering access from a designated
    place (gratis or for a charge), and offer equivalent access to the
    Corresponding Source in the same way through the same place at no
    further charge.  You need not require recipients to copy the
    Corresponding Source along with the object code.  If the place to
    copy the object code is a network server, the Corresponding Source
    may be on a different server (operated by you or a third party)
    that supports equivalent copying facilities, provided you maintain
    clear directions next to the object code saying where to find the
    Corresponding Source.  Regardless of what server hosts the
    Corresponding Source, you remain obligated to ensure that it is
    available for as long as needed to satisfy these requirements.

    e) Convey the object code using peer-to-peer transmission, provided
    you inform other peers where the object code and Corresponding
    Source of the work are being offered to the general public at no
    charge under subsection 6d.

  A separable portion of the object code, whose source code is excluded
from the Corresponding Source as a System Library, need not be
included in conveying the object code work.

  A "User Product" is either (1) a "consumer product", which means any
tangible personal property which is normally used for personal, family,
or household purposes, or (2) anything designed or sold for incorporation
into a dwelling.  In determining whether a product is a consumer product,
```

(continues on next page)

(continued from previous page)

```
doubtful cases shall be resolved in favor of coverage.  For a particular
product received by a particular user, "normally used" refers to a
typical or common use of that class of product, regardless of the status
of the particular user or of the way in which the particular user
actually uses, or expects or is expected to use, the product.  A product
is a consumer product regardless of whether the product has substantial
commercial, industrial or non-consumer uses, unless such uses represent
the only significant mode of use of the product.

  "Installation Information" for a User Product means any methods,
procedures, authorization keys, or other information required to install
and execute modified versions of a covered work in that User Product from
a modified version of its Corresponding Source.  The information must
suffice to ensure that the continued functioning of the modified object
code is in no case prevented or interfered with solely because
modification has been made.

  If you convey an object code work under this section in, or with, or
specifically for use in, a User Product, and the conveying occurs as
part of a transaction in which the right of possession and use of the
User Product is transferred to the recipient in perpetuity or for a
fixed term (regardless of how the transaction is characterized), the
Corresponding Source conveyed under this section must be accompanied
by the Installation Information.  But this requirement does not apply
if neither you nor any third party retains the ability to install
modified object code on the User Product (for example, the work has
been installed in ROM).

  The requirement to provide Installation Information does not include a
requirement to continue to provide support service, warranty, or updates
for a work that has been modified or installed by the recipient, or for
the User Product in which it has been modified or installed.  Access to a
network may be denied when the modification itself materially and
adversely affects the operation of the network or violates the rules and
protocols for communication across the network.

  Corresponding Source conveyed, and Installation Information provided,
in accord with this section must be in a format that is publicly
documented (and with an implementation available to the public in
source code form), and must require no special password or key for
unpacking, reading or copying.

  7. Additional Terms.

  "Additional permissions" are terms that supplement the terms of this
License by making exceptions from one or more of its conditions.
Additional permissions that are applicable to the entire Program shall
be treated as though they were included in this License, to the extent
that they are valid under applicable law.  If additional permissions
apply only to part of the Program, that part may be used separately
under those permissions, but the entire Program remains governed by
this License without regard to the additional permissions.
```

(continues on next page)

(continued from previous page)

```
  When you convey a copy of a covered work, you may at your option
remove any additional permissions from that copy, or from any part of
it.  (Additional permissions may be written to require their own
removal in certain cases when you modify the work.)  You may place
additional permissions on material, added by you to a covered work,
for which you have or can give appropriate copyright permission.

  Notwithstanding any other provision of this License, for material you
add to a covered work, you may (if authorized by the copyright holders of
that material) supplement the terms of this License with terms:

    a) Disclaiming warranty or limiting liability differently from the
    terms of sections 15 and 16 of this License; or

    b) Requiring preservation of specified reasonable legal notices or
    author attributions in that material or in the Appropriate Legal
    Notices displayed by works containing it; or

    c) Prohibiting misrepresentation of the origin of that material, or
    requiring that modified versions of such material be marked in
    reasonable ways as different from the original version; or

    d) Limiting the use for publicity purposes of names of licensors or
    authors of the material; or

    e) Declining to grant rights under trademark law for use of some
    trade names, trademarks, or service marks; or

    f) Requiring indemnification of licensors and authors of that
    material by anyone who conveys the material (or modified versions of
    it) with contractual assumptions of liability to the recipient, for
    any liability that these contractual assumptions directly impose on
    those licensors and authors.

  All other non-permissive additional terms are considered "further
restrictions" within the meaning of section 10.  If the Program as you
received it, or any part of it, contains a notice stating that it is
governed by this License along with a term that is a further
restriction, you may remove that term.  If a license document contains
a further restriction but permits relicensing or conveying under this
License, you may add to a covered work material governed by the terms
of that license document, provided that the further restriction does
not survive such relicensing or conveying.

  If you add terms to a covered work in accord with this section, you
must place, in the relevant source files, a statement of the
additional terms that apply to those files, or a notice indicating
where to find the applicable terms.

  Additional terms, permissive or non-permissive, may be stated in the
form of a separately written license, or stated as exceptions;
```

(continues on next page)

(continued from previous page)

```
the above requirements apply either way.

  8. Termination.

  You may not propagate or modify a covered work except as expressly
provided under this License.  Any attempt otherwise to propagate or
modify it is void, and will automatically terminate your rights under
this License (including any patent licenses granted under the third
paragraph of section 11).

  However, if you cease all violation of this License, then your
license from a particular copyright holder is reinstated (a)
provisionally, unless and until the copyright holder explicitly and
finally terminates your license, and (b) permanently, if the copyright
holder fails to notify you of the violation by some reasonable means
prior to 60 days after the cessation.

  Moreover, your license from a particular copyright holder is
reinstated permanently if the copyright holder notifies you of the
violation by some reasonable means, this is the first time you have
received notice of violation of this License (for any work) from that
copyright holder, and you cure the violation prior to 30 days after
your receipt of the notice.

  Termination of your rights under this section does not terminate the
licenses of parties who have received copies or rights from you under
this License.  If your rights have been terminated and not permanently
reinstated, you do not qualify to receive new licenses for the same
material under section 10.

  9. Acceptance Not Required for Having Copies.

  You are not required to accept this License in order to receive or
run a copy of the Program.  Ancillary propagation of a covered work
occurring solely as a consequence of using peer-to-peer transmission
to receive a copy likewise does not require acceptance.  However,
nothing other than this License grants you permission to propagate or
modify any covered work.  These actions infringe copyright if you do
not accept this License.  Therefore, by modifying or propagating a
covered work, you indicate your acceptance of this License to do so.

  10. Automatic Licensing of Downstream Recipients.

  Each time you convey a covered work, the recipient automatically
receives a license from the original licensors, to run, modify and
propagate that work, subject to this License.  You are not responsible
for enforcing compliance by third parties with this License.

  An "entity transaction" is a transaction transferring control of an
organization, or substantially all assets of one, or subdividing an
organization, or merging organizations.  If propagation of a covered
work results from an entity transaction, each party to that
```

(continues on next page)

(continued from previous page)

```
transaction who receives a copy of the work also receives whatever
licenses to the work the party's predecessor in interest had or could
give under the previous paragraph, plus a right to possession of the
Corresponding Source of the work from the predecessor in interest, if
the predecessor has it or can get it with reasonable efforts.

  You may not impose any further restrictions on the exercise of the
rights granted or affirmed under this License.  For example, you may
not impose a license fee, royalty, or other charge for exercise of
rights granted under this License, and you may not initiate litigation
(including a cross-claim or counterclaim in a lawsuit) alleging that
any patent claim is infringed by making, using, selling, offering for
sale, or importing the Program or any portion of it.

  11. Patents.

  A "contributor" is a copyright holder who authorizes use under this
License of the Program or a work on which the Program is based.  The
work thus licensed is called the contributor's "contributor version".

  A contributor's "essential patent claims" are all patent claims
owned or controlled by the contributor, whether already acquired or
hereafter acquired, that would be infringed by some manner, permitted
by this License, of making, using, or selling its contributor version,
but do not include claims that would be infringed only as a
consequence of further modification of the contributor version.  For
purposes of this definition, "control" includes the right to grant
patent sublicenses in a manner consistent with the requirements of
this License.

  Each contributor grants you a non-exclusive, worldwide, royalty-free
patent license under the contributor's essential patent claims, to
make, use, sell, offer for sale, import and otherwise run, modify and
propagate the contents of its contributor version.

  In the following three paragraphs, a "patent license" is any express
agreement or commitment, however denominated, not to enforce a patent
(such as an express permission to practice a patent or covenant not to
sue for patent infringement).  To "grant" such a patent license to a
party means to make such an agreement or commitment not to enforce a
patent against the party.

  If you convey a covered work, knowingly relying on a patent license,
and the Corresponding Source of the work is not available for anyone
to copy, free of charge and under the terms of this License, through a
publicly available network server or other readily accessible means,
then you must either (1) cause the Corresponding Source to be so
available, or (2) arrange to deprive yourself of the benefit of the
patent license for this particular work, or (3) arrange, in a manner
consistent with the requirements of this License, to extend the patent
license to downstream recipients.  "Knowingly relying" means you have
actual knowledge that, but for the patent license, your conveying the
```

(continues on next page)

(continued from previous page)

```
covered work in a country, or your recipient's use of the covered work
in a country, would infringe one or more identifiable patents in that
country that you have reason to believe are valid.

  If, pursuant to or in connection with a single transaction or
arrangement, you convey, or propagate by procuring conveyance of, a
covered work, and grant a patent license to some of the parties
receiving the covered work authorizing them to use, propagate, modify
or convey a specific copy of the covered work, then the patent license
you grant is automatically extended to all recipients of the covered
work and works based on it.

  A patent license is "discriminatory" if it does not include within
the scope of its coverage, prohibits the exercise of, or is
conditioned on the non-exercise of one or more of the rights that are
specifically granted under this License.  You may not convey a covered
work if you are a party to an arrangement with a third party that is
in the business of distributing software, under which you make payment
to the third party based on the extent of your activity of conveying
the work, and under which the third party grants, to any of the
parties who would receive the covered work from you, a discriminatory
patent license (a) in connection with copies of the covered work
conveyed by you (or copies made from those copies), or (b) primarily
for and in connection with specific products or compilations that
contain the covered work, unless you entered into that arrangement,
or that patent license was granted, prior to 28 March 2007.

  Nothing in this License shall be construed as excluding or limiting
any implied license or other defenses to infringement that may
otherwise be available to you under applicable patent law.

  12. No Surrender of Others' Freedom.

  If conditions are imposed on you (whether by court order, agreement or
otherwise) that contradict the conditions of this License, they do not
excuse you from the conditions of this License.  If you cannot convey a
covered work so as to satisfy simultaneously your obligations under this
License and any other pertinent obligations, then as a consequence you may
not convey it at all.  For example, if you agree to terms that obligate you
to collect a royalty for further conveying from those to whom you convey
the Program, the only way you could satisfy both those terms and this
License would be to refrain entirely from conveying the Program.

  13. Use with the GNU Affero General Public License.

  Notwithstanding any other provision of this License, you have
permission to link or combine any covered work with a work licensed
under version 3 of the GNU Affero General Public License into a single
combined work, and to convey the resulting work.  The terms of this
License will continue to apply to the part which is the covered work,
but the special requirements of the GNU Affero General Public License,
section 13, concerning interaction through a network will apply to the
```

(continues on next page)

(continued from previous page)

```
combination as such.

  14. Revised Versions of this License.

  The Free Software Foundation may publish revised and/or new versions of
the GNU General Public License from time to time.  Such new versions will
be similar in spirit to the present version, but may differ in detail to
address new problems or concerns.

  Each version is given a distinguishing version number.  If the
Program specifies that a certain numbered version of the GNU General
Public License "or any later version" applies to it, you have the
option of following the terms and conditions either of that numbered
version or of any later version published by the Free Software
Foundation.  If the Program does not specify a version number of the
GNU General Public License, you may choose any version ever published
by the Free Software Foundation.

  If the Program specifies that a proxy can decide which future
versions of the GNU General Public License can be used, that proxy's
public statement of acceptance of a version permanently authorizes you
to choose that version for the Program.

  Later license versions may give you additional or different
permissions.  However, no additional obligations are imposed on any
author or copyright holder as a result of your choosing to follow a
later version.

  15. Disclaimer of Warranty.

  THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY
APPLICABLE LAW.  EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT
HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY
OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE.  THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM
IS WITH YOU.  SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF
ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

  16. Limitation of Liability.

  IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING
WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS
THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY
GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE
USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF
DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD
PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS),
EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF
SUCH DAMAGES.

  17. Interpretation of Sections 15 and 16.
```

(continues on next page)

(continued from previous page)

```
  If the disclaimer of warranty and limitation of liability provided
above cannot be given local legal effect according to their terms,
reviewing courts shall apply local law that most closely approximates
an absolute waiver of all civil liability in connection with the
Program, unless a warranty or assumption of liability accompanies a
copy of the Program in return for a fee.

                     END OF TERMS AND CONDITIONS

            How to Apply These Terms to Your New Programs

  If you develop a new program, and you want it to be of the greatest
possible use to the public, the best way to achieve this is to make it
free software which everyone can redistribute and change under these terms.

  To do so, attach the following notices to the program.  It is safest
to attach them to the start of each source file to most effectively
state the exclusion of warranty; and each file should have at least
the "copyright" line and a pointer to where the full notice is found.

    <one line to give the program's name and a brief idea of what it does.>
    Copyright (C) <year>  <name of author>

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program.  If not, see <https://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

  If the program does terminal interaction, make it output a short
notice like this when it starts in an interactive mode:

    <program>  Copyright (C) <year>  <name of author>
    This program comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
    This is free software, and you are welcome to redistribute it
    under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate
parts of the General Public License.  Of course, your program's commands
might be different; for a GUI interface, you would use an "about box".

  You should also get your employer (if you work as a programmer) or school,
```

(continues on next page)

(continued from previous page)

```
if any, to sign a "copyright disclaimer" for the program, if necessary.
For more information on this, and how to apply and follow the GNU GPL, see
<https://www.gnu.org/licenses/>.

  The GNU General Public License does not permit incorporating your program
into proprietary programs.  If your program is a subroutine library, you
may consider it more useful to permit linking proprietary applications with
the library.  If this is what you want to do, use the GNU Lesser General
Public License instead of this License.  But first, please read
<https://www.gnu.org/licenses/why-not-lgpl.html>.
```

BIBLIOGRAPHY

[Shallue] Shallue, Christopher J., et al. "Measuring the effects of data parallelism on neural network training." arXiv preprint arXiv:1811.03600 (2018).

[Ulyanov] Ulyanov, Dmitry, Andrea Vedaldi, and Victor Lempitsky. "Instance normalization: The missing ingredient for fast stylization." arXiv preprint arXiv:1607.08022 (2016).

[Goyal] Goyal, Priya, et al. "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour." arXiv preprint arXiv:1706.02677 (2017).

[Yang] Yang, Greg, et al. "A mean field theory of batch normalization." arXiv preprint arXiv:1902.08129 (2019).

[Uppal] Curse of Batch Normalization (2020).

[McCandlish] McCandlish, Sam, et al. "An empirical model of large-batch training." arXiv preprint arXiv:1812.06162 (2018).

[He] He, Kaiming, et al. "Deep residual learning for image recognition." Proceedings of the IEEE conference on computer vision and pattern recognition. arXiv preprint arXiv:1512.03385 (2016).

[Vaswani] Vaswani, Ashish, et al. "Attention is all you need." Advances in neural information processing systems. arXiv preprint arXiv:1706.03762 (2017).