

STIR and Tensorflow

Philipp Windischhofer*

Wednesday 19th July, 2017

Abstract

This document provides a bit more in-depth and technical information regarding the integration of Tensorflow into STIR. It should give enough hints and tips such as to ease further development and extension of the ideas presented here. See also the presentation slides that accompany this report, at `../pres/`.

1 The overall setup

STIR stands for Software for Tomographic Image Reconstruction. As such, it provides a wealth of functionality, different reconstruction algorithms and many options to tune them. As far as reconstruction goes, there are two main classes, *analytic* and *iterative* algorithms. While the former are computationally relatively cheap and, in principle, exact, they turn out to be very susceptible to noise in the raw data and therefore the latter category is widely used in practice. The principle of such a reconstruction is as follows. The PET scanner consists of an array of scintillatoin crystals arranged in a more- or less cylindrical geometry. However, it is not single crystals that are relevant for our purposes, but pairs of crystals, where each pair makes up a *line of response* (LOR). Every time both crystals of a certain detector pair register an event within a certain *coincidence time window*, this is interpreted as coming from an electron-positron annihilation that happened somewhere on the line connecting the two crystals (that is, the line of response) and the LOR gets assigned this hit.

The inputs to the reconstruction algorithm are precisely these counts assigned to each possible LOR. Note that this form of the data may not be directly provided by the detector, but could also only be formed in a stage of preprocessing, e.g. by taking the raw *listmode* data (that lists each hit in each detector separately, along with energy and timing information) and then extracting coincident hits.

It is then the goal of the reconstruction algorithm is then to solve the inverse problem of finding the activity distribution in the sample that lead to the observed counts / LOR pattern. Iterative algorithms perform this job by extremizing a likelihood function. Since the counts per LOR follow a Poissonian distribution, a possible candidate for such a likelihood function simply is

$$P(\mathbf{p}|f) = \prod_{j=1}^{N_{LOR}} \frac{e^{-\bar{p}_j}}{p_j!} (\bar{p}_j)^{p_j}. \quad (1)$$

Here, $\mathbf{p} = (p_1, \dots, p_{N_{LOR}})$ is a vector of the measured counts per LOR, and \bar{p}_j are the counts expected *if* the true tracer distribution is given by f :

$$\bar{p}_j = \int_{LOR} d\mathbf{r} f(\mathbf{r}). \quad (2)$$

This task of computing the expected counts per LOR from a (conjectured) image f is termed *forward projection*. In an iterative reconstruction, starting from an initial assumption for f , it gets updated in every iteration in such a way as to maximize $P(\mathbf{b}|f)$.

The forward projection occupies a large fraction of the total computation time required for the whole reconstruction: STIR, in its standard configuration, spends over 90% of its time for forward projection. Therefore, there is a huge potential for considerably reducing the total reconstruction time, if the forward projection task can be sped up.

*philipp.windischhofer@cern.ch, philipp.windischhofer@gmail.com

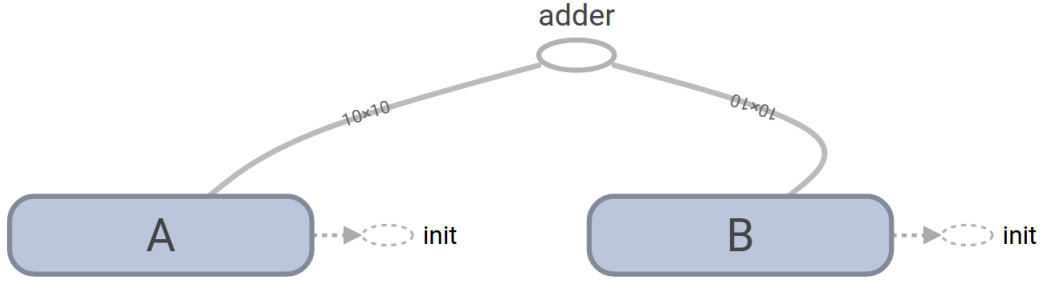


Figure 1: A simple computational graph. It takes two Tensors as input, each with a size of 10×10 , and adds the two.

More precisely, the bulk of computation time is used to compute the integral for $\overline{p_j}$, which, in a discrete voxel array into which the image is usually partitioned, takes the form

$$\overline{p_j} = \sum_{\text{LOR}} \text{LOI}(k, l, m) f(k, l, m), \quad (3)$$

where the tuples (k, l, m) are the indices of the voxels through which the LOR passes, and LOI denotes the *length of intersection* of the LOR through the voxel (k, l, m) .

The remainder of this report will be concerned with describing the implementation of an algorithm for the computation of the LOI (or of the above sum, respectively).

2 Setting up Tensorflow

The general line of attack in this matter will be to exploit the massive parallel computing power of modern GPUs, and thus try to perform a large fraction of the required operations on a GPU. Several approaches to this problem already exist that use a CUDA implementation. This, however, ties the implementation to a certain type of GPU.

2.1 What is Tensorflow?

In order to regain as much flexibility as possible, the implementation presented here makes use of *Tensorflow*. Tensorflow is an open-source library developed and maintained by Google that supports large-scale numerical calculations. Although heavily used in the machine learning community, its basic architecture makes it suited also for much more general applications.

Computations within Tensorflow are realized in terms of data-flow *graphs* (see Figure 1 for an simple example). The (directed) edges of the graph transport data (in the form of **Tensors**, the primitive data type of Tensorflow, which are arrays of numbers that can have arbitrary dimensions) between the vertices, which symbolize the operations that are run on the data.

The definition and construction of this graph is completely decoupled from its *execution*. In particular (and this will also be the approach pursued for this work), the graph can be constructed and saved to disk by a simple Python script, while the actual execution of the graph (and the collection of the results) is performed from within the C++ environment of STIR. Tensorflow also allows for great flexibility in terms of the execution of the graph: once its definition has been completed, it can run either on a GPU, on a CPU (with any number of threads), or even on a cluster-like infrastructure with multiple worker nodes.

2.2 Preparing Tensorflow and STIR

In order to be able to use the C++ API of Tensorflow from within STIR, a few preparatory steps are necessary (due to Google relying on its in-house build tool `Bazel` for the compilation of all Tensorflow-related projects, while STIR makes use of `CMake`). All of the following steps and tests have been performed on PCETH128.

Thus, to be able to use Tensorflow with STIR, we first need to build a shared library, against which the STIR OSMAPSL executable can be linked. To this end, follow the instructions here¹. In Step 2, the external Protobuf and Eigen libraries have been installed locally and *not* added as external dependencies. Step 3 does not need to be performed, as the required changes to the `CMake`-files of STIR have already been made and included into the codebase (see below).

The Tensorflow-enabled version of STIR, including the modified source- and makefiles is available from a GIT repository². After the Tensorflow library has been compiled according to the above steps and installed into the default directory `/usr/local/`, STIR should compile by issuing a simple `make OSMAPSL` command. In case Tensorflow is installed elsewhere, the installation path needs to be updated in the file `STIR_SRC_DIRECTORY/src/cmake/FindTensorFlow.cmake`.

Note that the Makefiles will be *overwritten* if the `configure` script is invoked again. In case this should become necessary (e.g. because additional features of STIR are required, such as openMP support), the Makefiles will need to be adapted again (this is because Tensorflow has not yet been integrated into the `configuration` script and therefore cannot be chosen as a separate installation dependency there).

2.2.1 Changes made to the Makefiles

- Tensorflow requires the C++11 standard, but the rest of STIR has been written in compliance with C++98. Compiling all of STIR in C++11 is however not possible, because of backward-incompatibility of some newly introduced keywords. However, it is possible to only compile the `recon_buildblock`, of which Tensorflow forms a part, in C++11. This is effected by adding the lines

```
set (dir_LIB_SOURCES ${dir}_LIB_SOURCES)
target_link_libraries(recon_buildblock ${PROJECT_LIBRARIES})
```

to `STIR_SRC_DIRECTORY/src/recon_buildblock/CMakeLists.txt`. The second one instructs the compiler to link this buildblock against the Tensorflow library.

- Also the top-level Makefile `STIR_SRC_DIRECTORY/CMakeLists.txt` needs to be modified following the integration of Tensorflow and the required dependencies *Protobuf* and *Eigen*. For them to be included, add

```
message(STATUS "module path: ${CMAKE_MODULE_PATH}")

### put here new libraries for tensorflow support
# for Eigen
find_package(Eigen REQUIRED)

# for Protobuf
find_package(Protobuf REQUIRED)

# for Tensorflow
find_package(TensorFlow REQUIRED)

set(PROJECT_INCLUDE_DIRS ${PROJECT_INCLUDE_DIRS} ${TensorFlow_INCLUDE_DIRS}
    ${Eigen_INCLUDE_DIRS} ${Protobuf_INCLUDE_DIRS})
```

¹<https://github.com/cjweeks/tensorflow-cmake>

²<https://github.com/philippwindischhofer/STIR/tree/stir-tf>

```

set (PROJECT_LIBRARIES $ {PROJECT_LIBRARIES} ${TensorFlow_LIBRARIES})
set (PROJECT_DEPENDENCIES ${PROJECT_DEPENDENCIES} Eigen Protobuf)

include_directories (${PROJECT_INCLUDE_DIRS})

```

to this file.

3 Creating the graphs

The generation of Tensorflow graphs is completely independent of their execution. The graphs that have been tested together with the full STIR-environment are contained in `STIR_SRC_DIRECTORY/src/tensorflow`. They are generated by a Python script (simply because the Python API of Tensorflow is better documented and more stable compared to the C++ API). Additional scripts implementing various other algorithms that have only been used for testing purposes can be found in an independent repository (see Section 6 below).

3.1 Implemented algorithm

The algorithm that has been found to give the best performance in terms of speedup is a version of *ray marching* and uses an iterative procedure to find an estimate of the sum in Equation 3. However, it does not perform the computation of the LOI on the level of the whole LOR, but only on a voxel-by-voxel basis.

It takes as inputs a list of points $\{\mathbf{p}_i\}$ and unit vectors $\{\mathbf{v}_i\}$. It then computes the LOI of the LOR whose direction is given by \mathbf{v}_i , in the voxel in which the point \mathbf{p}_i lies. Thus, the algorithm returns a list of LOIs (which is of the same length as $\{\mathbf{v}_i\}$ and $\{\mathbf{p}_i\}$).

With this in mind, an LOR is now no longer defined solely by its start- and stop-position, but also by how the points along the LOR get selected, which are then used to compute the total LOI. If one chooses these points such that each voxel through which the LOR passes contains exactly one point, then the algorithm will return the same answer (up to the deviations introduced by it being an iterative, and thus not exact, algorithm) as the method by Siddon, which is used by STIR by default. On the other hand, separating the notions of an LOR and of the points that define it offers more flexibility: one can sample an LOR by a certain (fixed) number of points along it, spaced equidistantly or randomly, and can thus trade very precisely the quality of the reconstructed image against the reconstruction time. In a next step, one might not simply choose points that lie *precisely* on the LOR, but select them following a certain probability distribution peaked along the original LOR. This would effectively convert the LOR into a TOR (tube of response) and sample also voxels which do not directly lie on the line-of-sight of the crystal pair. Given the finite size of each crystal, this is closer to reality than the other, idealized treatment.

Even more importantly, partitioning a LOR into a (potentially fixed) number of points means that the Tensorflow graph can be designed such that it accepts (and operates on) a fixed number of points (*batchsize*). This allows it to use the GPU very efficiently, since *the same operations* are applied on a set of input data with a constant size.

More precisely, given a single point and a single unit vector to set the LOR direction, the algorithm

performs the following steps (in pseudocode):

Data: point coordinates \mathbf{p} , direction (unit) vector \mathbf{v}

Result: LOI through voxel that contains \mathbf{p} , given the LOR with direction \mathbf{v}
compute the position of \mathbf{p} within the voxel that contains it;

for a fixed number of iterations **do**

 evaluate the signed distance function at the current location and store its value;
 update the current location by moving along the direction of \mathbf{v} with a step size given by the SDF;
 add the current step size to a running total;

end

the LOI is given (to the accuracy prescribed by the chosen number of iterations) by the final value of the running sum;

Algorithm 1: Version of ray marching used to compute approximately the LOI through one voxel.

The *signed distance function* (SDF) is used to reduce the number of iterations needed to reach a certain accuracy level. In principle, it would be enough to use a fixed step size and iterate as long as the distance to the boundary of the voxel is nonzero. The SDF is defined to return a value that is *always less* than the actual, physical distance from the current position to the boundary of the voxel. In this way, by choosing a step size that is equal to this distance, the updated position is guaranteed to remain within the voxel volume, but, depending on the quality of the SDF, the number of necessary iterations is reduced (since the step size is adjusted dynamically).

For a cube with side lengths (d_x, d_y, d_z) and where the origin is located at $(\frac{d_x}{2}, \frac{d_y}{2}, \frac{d_z}{2})$, a SDF is given by

$$\text{SDF}(x, y, z) = -\max\left(\left|x - \frac{d_x}{2}\right| - \frac{d_x}{2}, \left|y - \frac{d_y}{2}\right| - \frac{d_y}{2}, \left|z - \frac{d_z}{2}\right| - \frac{d_z}{2}\right) \quad (4)$$

4 Computing matrix elements with Tensorflow

`ProjMatrixElemsForOneBin`-objects are central for the forward projection as implemented into STIR. For a given crystal pair (LOR), these objects contain a list of the traversed voxels as well as the corresponding LOIs. In STIR, they are constructed by a call to `ProjMatrixByBin::get_proj_matrix_elems_for_one_bin(Bin bin)`, where `bin` contains information that specifies the crystal pair. Several subclasses derive from `ProjMatrixByBin` that differ by the algorithm which is used to actually find the traversed voxels and their LOIs. `ProjMatrixByBinUsingRayTracing` uses the classical Siddon's algorithm. A new class was introduced, `ProjMatrixByBinUsingRayTracingTF`, which makes use of Tensorflow and the ray marching algorithm described before to compute the `ProjMatrixByBin` elements.

From the point of view of the GPU, it is not beneficial to compute the matrix elements one by one. On the other hand, *pooling* requests for matrix elements (as issued by the higher-level STIR classes) into a queue and only starting the computation once enough requests have been accumulated makes much better use of the resources.

For this reason, `ProjMatrixByBinUsingRayTracingTF` provides the methods `schedule_matrix_elems_for_one` and `execute` that add a matrix element to the queue, and execute all stored requests, respectively. This, in turn, necessitated some small changes in the code of `ForwardProjectorByBinUsingProjMatrixByBin` which (if the Tensorflow mode is enabled) calls these methods.

In turn, `ProjMatrixByBinUsingRayTracingTF` relies on another new class, `TFRayTracer` which handles all low-level tasks and in general operates on the level of individual points, rather than LORs or matrix elements.

To enable Tensorflow for a reconstruction, it is sufficient to use `Matrix type := Ray Tracing TF` in the `*.par`-file. Using `Matrix type := Ray Tracing` instead will have STIR run in legacy mode without Tensorflow support.

5 Class overview

What follow is a list of STIR classes that play important roles as far as forward projection and the computation of matrix elements is concerned.

- **ProjMatrixByBin**: base class for computing projection matrix elements. The method `get_proj_matrix_elem` is called to compute matrix elements. If the requested matrix element (or another matrix element related to it by one of the symmetries of the scanner geometry) has already been computed and was stored in the cache, it is returned. Otherwise, it calls `calculate_proj_matrix_elems_for_one_bin` (implemented by the child class) to actually compute it.
- **ProjMatrixByBinUsingRayTracing**: derived from **ProjMatrixByBin**. For the given matrix element, it generates a number of LORs (set by the `number of rays in tangential direction to trace for each bin:=` parameter in the parameter file) and uses Siddon's algorithm to compute the LOIs for all voxels that are traversed by these. All of this information (voxel indices and LOIs) is contained in the **ProjMatrixElemsForOneBin** object that it returns.
- **RayTraceVoxelsOnCartesianGrid**: this contains the actual implementation of Siddon's algorithm
- **ProjMatrixElemsForOneBin**: object that represents a projection matrix element. It contains a vector of **ProjMatrixElemesForOneBinValue** objects, each of which stores a tuple of voxel indices as well as the LOI through this specific voxel.
- **ProjMatrixElemsForOneBinValue**: simple container class that holds a set of voxel indices as well as the LOI through this particular voxel.
- **ProjMatrixByBinUsingRayTracingTF**: newly introduced class that computes **ProjMatrixElemsForOneBin** objects by using the ray marching algorithm on the GPU. It in turn relies on **TFRayTracer** for the low-level tasks. To ensure compatibility, it also still implements `calculate_proj_matrix_elems_for_one_bin` but using this method is not the most efficient way to use the class. Instead, use `schedule_matrix_elems_for` and `execute` to add a certain matrix element to a queue, and compute all matrix elements in the queue, respectively. When calling `schedule_matrix_elems_for_one_bin`, the demanded LOR is sampled by a number of points which in turn are passed to **TFRayTracer** for execution. Once the result is available, the individual LOIs are again partitioned as assigned to the matrix elements they belong to.
- **TFRayTracer**: this is the low-level class that actually instantiates a Tensorflow session and loads the computational graph from a file. It operates on the level of individual points and maintains a queue of points that need to be treated. New points are added to this queue by calling the `schedulePoint` method, and the entire queue is executed by calling `execute`.

For more detailed descriptions of the functionality, consult the source files directly.


6 More Tensorflow

More Python scripts that were used for various tests (of algorithms different from the one explained in Section 3.1) are available in an independent repository³. Its contents is as follows.

³<https://gitlab.phys.ethz.ch/luster/tf-raytracing>

6.1 Repository overview

7 Futher information

 **Schematic:** Schematic data.

