A multithreaded global tracking algorithm based on directed acyclic graphs and machine learning

A1. Authors

Marcel Kunze
Heidelberg University
marcel.kunze@uni-heidelberg.de

A2. Team in the competition

Competition Name: TrackML Throughput Phase

Team Name: cloudkitchen

Leaderboard Place: 3

A3. Summary

The algorithm has been designed to make use of artificial neural networks for pattern recognition on the basis of spatial coordinates together with simple geometrical information such as directional cosines or a helix score calculation. Typical patterns to be investigated are hit pairs and triples that might seed candidate tracks. The training of the networks was accomplished by presentation of typically 5 million ground truth patterns over 500 epochs. The hit data are sorted into voxels organized in directed acyclic graphs (DAG) in order to allow for fast track propagation. In addition to the spatial hit data the DAGs hold information about the network model to apply, and a z vertex estimate derived from ground truth. As they combine the data with the corresponding methods the DAGs form a nice foundation to define tasks that can be run in parallel very efficiently in a multithreaded environment. There are two sets of graphs: One set covers detector slices wrt. the z-axis, the other covers phi/theta tiles. Each set would work perfectly well by itself, but a clever combination of the two yields the best CodaLab score: The first set is used to seed the pair finder while the other drives the triple finder. Prior to the execution of the model the DAGs have been trained with tracking ground truth of typically 15-25 sample events, yielding a good balance between graph traversal time and accuracy.
The path and track finding is based on inward/outward triple prolongation in combination with outlier density approximation as proposed by J.S. Wind in the Kaggle accuracy phase. Running two threads the execution time is on average about 7 seconds per event at 93% accuracy in a docker environment, consuming 700 MB RAM.

A4. High-level Description

The tracking model has been designed and implemented as a standard C++11 shared library. It may be run using the *main* C++ driver program, or it may be loaded into the python runtime environment using *ctypes*. The architecture comprises a *Tracker* class for data housekeeping and steering, as well as a *Reconstruction* class to implement the algorithms. The data are organized in the *Graph* class that has been designed as a STL-like header file. The neural networks are handled by the *XMLP* class. The *Trainer* class inherits from *Tracker*: it takes care of neural network training. While the training is based on the Neural Network Objects [2] and the ROOT toolkit [3] there is no dependency of the tracking shared library to

external packages. Persistency of graphs and neural networks has been achieved by streaming of the objects to corresponding text files.
The program consists of 5 parts: Setup – pair finder – triple finder – path finder – track assignment. The setup reads all configuration data and initializes the neural networks and graphs prior to processing the first event. Parts 2-5 run as threads in parallel for each event, followed by a final serial track assignment to join the partial results into a common solution. The program implements multithreading by instantiating *NTHREADS Reconstruction* objects and managing a set of suiting tasks by use of a thread safe stack. The tasks correspond to graphs that hold the corresponding hit data and a set of neural networks to classify the data. While the event is being processed each thread pops a task from the stack and takes care of its execution. Once the stack is empty and all tasks are finished the first thread continues and joins the partial results into the final assignment of hits to tracks. The track assignment is written to a result file and handed over to the Python frame that delivers it to the CodaLab platform.

A5. Scientific details

The model generally uses a cylinder coordinate system (theta,phi,r_t) to describe the hit data. A library of track patterns has been organized in directed acyclic graphs (DAG) of space elements such that any element has potential following elements. As such by graph traversal potential paths can be derived. In principal the resolution could be chosen on a very fine-granular detector cell level. Although this would yield very accurate results, the resulting graphs tend to grow very large and get slow. For this reason, a two-dimensional *graphHash* function has been defined to identify a spatial phi/theta segment for any hit:

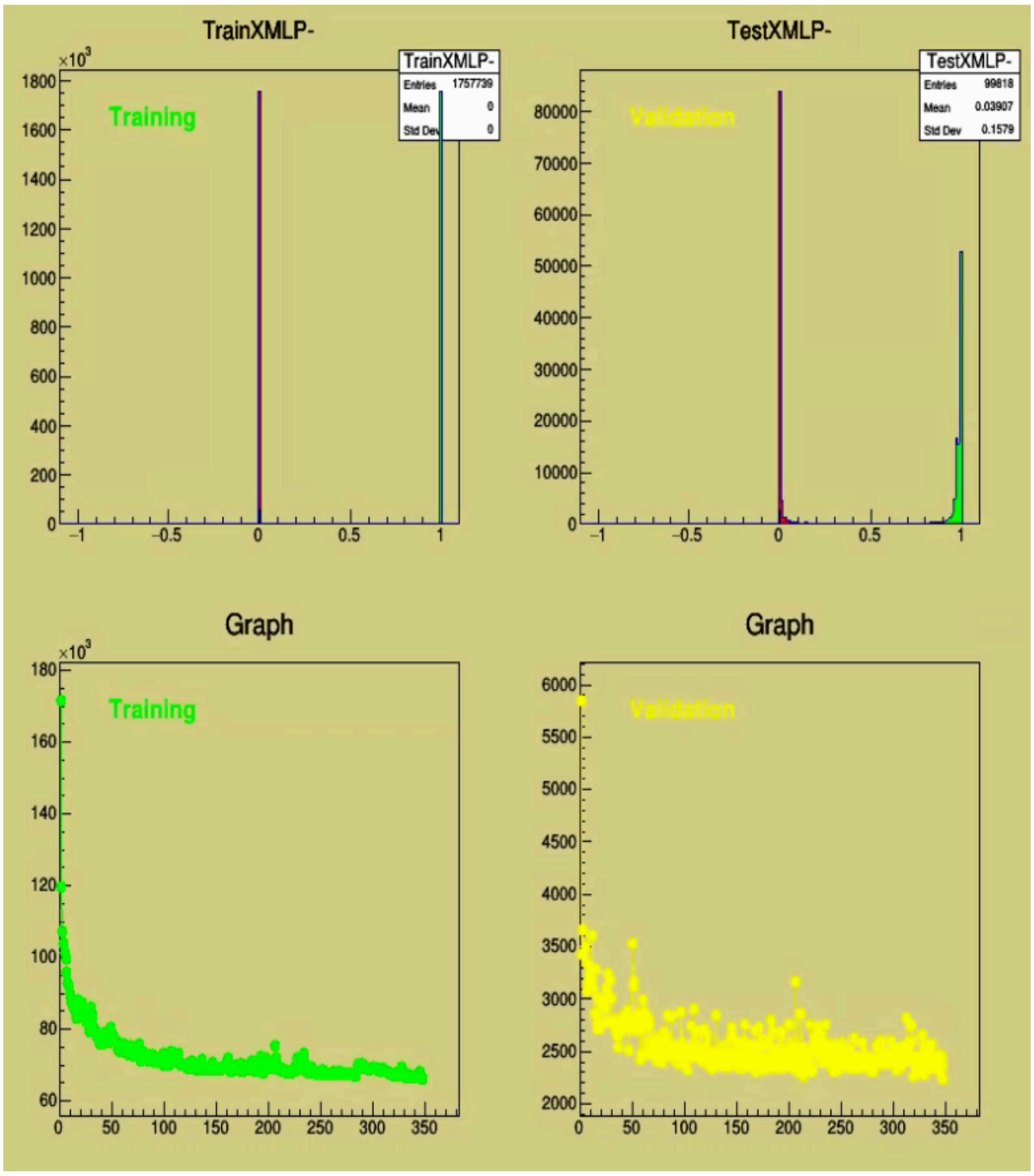
```
  int i1 = (int) (phires*0.15*(M_PI+phi));
  int i2 = (int) (theres*0.1*(5-theta));
```

where 'theta' actually corresponds to asinh(z/r_t) to flatten the distribution [4]. The constants *phires* and *theres* define the granularity of the spatial tesselation. It turned out by tuning that a setting of 12 tiles in phi and 14 tiles in theta yielded the best compromise of accuracy vs. speed (i.e. CodaLab score). In order to execute fast, each tile has its dedicated graph (168 in total). The graphs have been trained with ground truth tracking data (typically 15-25 events), taking about a minute.
In addition, a *voxel* hash function has been defined to identify a hit and its correspondence to a spatial segment:

```
  long long index = (((long) i1<<32) | (((long)i2<<24) | (l<<16) | m;
```

where i1 and i2 are the corresponding *graphHash* values and l and m are the layer and module numbers of the hit, respectively.
There are two sets of graphs: One set covers the two detector hemispheres wrt. the z-axis, the other covers the phi/theta tiles. The first set is used to seed the pair finder, the other is used to drive the triple finder. Each set would work perfectly well by itself, but a clever combination of the two yields the best CodaLab score.
The pair finder utilizes 2 neural networks, XMLP1 and XMLP2. XMLP1 is an 8-15-5-1 multilayer perceptron that has been trained with the ground truth cylinder coordinates of the two hits in addition with the two directional cosines of the hits towards the vertex.

XMLP2 is a 9-15-5-1 multilayer perceptron that in addition takes a helix score as an input, assuming a z vertex in addition to the pair. Both networks behave very well in any direction. As it consists of fewer nodes and it does not need a helix calculation, XMLP1 executes a little bit faster than XMLP2 at the expense of a few percent lower accuracy in the central section of the detector system. The final setup therefore combines XMLP1 for the forward/backward section (“discs”) with XMLP2 for the central section (“tubes”). In average there are about 500,000 pair combinations accepted at a cut on the output of 0.15 (XMLP1) and 0.55 (XMLP2) at an overall tracking score of 99.4%. The list of pairs is then submitted to the triple finder. The triple finder uses a 10-15-5-1 multilayer perceptron that has been trained with 3 cylindrical hit coordinates plus an additional helix score (XMLP3). In average it accepts about 320,000 combinations per event with a tracking score around 97%. The error rate presenting 100,000 validation patterns reaches about 6-8% for XMLP1/XMLP2 and around 2% for XMLP3, respectively. The following picture shows the training of XMLP3 after 350 epochs of 3.5 million patterns (upper row: training and validation output function, lower row: loss function for training and validation data sets):



The track finding and assignment is based on inward/outward triple prolongation in combination with outlier density approximation as proposed by J.S. Wind in the Kaggle accuracy phase [1]. It takes care of joining the graph results and yields an accuracy of about 93%. This part is necessarily executed as a serial task.

A6. Interesting findings

The training of the neural networks used pure cylinder coordinates in the first try. It turned out that the inputs could be folded in each direction due to the detector and event symmetry, thus considerably speeding up the training using less patterns at the same time. Technically this is most simply realized by use of the abs-function in combination with a ⍰/2 shift:
input 0.001*rz1:abs(abs(phi1)-1.57079632679):0.001*abs(z1):0.001*rz2:abs(abs(phi2)-1.57079632679):0.001*abs(z2):f0:f1:score*0.001

Conventional cuts on the vertex constraint can be used to lower the combinatorics to be processed by the neural networks. By simple geometrical calculation of the xy vertex and the z vertex the raw combinatorics to be classified by the neural networks can be decreased from more than 2,000,000 to about 1,000,000 combinations per event.

Larger events take longer to process and therefore tend to considerably lower the CodaLab score. By optimizing the cuts on the neural network outputs such that they are systematically increased with event size, large events can be sped up at a small loss of accuracy, improving the CodaLab score by 0.3%. The cuts of all neural networks have been hyper tuned to optimize the score with a granularity of 10,000 hits.

The training of the graphs needs only o(15-25) events to yield optimum results. If more events are used in the training, more accurate results may be achieved at a longer execution time (and hence a lower CodaLab score).

The accuracy of the result ca be improved by 0.2% if the graph tasks are organized such that each thread works on neighboring graphs. This is due to the fact that a path pre-assembly already happens on the thread level prior to the joining of the partial results at the end. In that way overlapping paths can already be merged on the parallel thread level thus relieving the serial task.

A7. Simple Features and Methods

The model can nicely be tuned wrt. accuracy and speed by simply changing the cuts on the neural network output nodes. The current network topology has been hyper tuned wrt. the optimum CodaLab score. The network layout might be enlarged and trained with more patterns to improve accuracy or it may be shrunk to improve speed. The same holds for the layout of the graphs.

A8. Model Training and Execution Time

The DAGs reside in the *graph* (tiles) and *paths* (slices) directories, respectively. The training of the DAGs is performed by

```
./makeGraph [ event number mode phires theres ]
```

- event: event to process (default: 21001)
- number: number of events in sequence (default: 1)
- mode: 1 for tile graphs, 3 for slice graphs
- phires: phi resolution (default: 12)
- theres: theta resolution (default: 14)

Graph generation takes about half a minute for 25 events. There is a script to prepare a suiting set of graphs:

```
./generate.sh 25 12 14
```

where the argument denotes the number of events to use for training and the phi and theta resolution, respectively. There are three neural networks for pattern recognition:

- XMLP1 for hit pair recognition (8-15-5-1 multilayer perceptron), good esp. in forward/backward detector region
- XMLP2 for hit pair recognition (9-15-5-1 multilayer perceptron), good esp. in central detector region
- XMLP3 for hit triple recognition (10-15-5-1 multilayer perceptron), good everywhere

The neural networks are trained with ntuples stored in ROOT files [3]. These are produced if the flag *TRAINFILE* is set in the programs. The training is being performed with the NetworkTrainer of the Neural Network Objects [2] by use of the corresponding training files:

```
NetworkTrainer train1.nno
NetworkTrainer train2.nno
NetworkTrainer train3.nno
```

The supervised training takes about half an hour for 500 epochs of 4-5 million patterns each (extracted from ca. 10 events). For each epoch the corresponding network files are stored in the *Networks* directory. Testing is generally done in a container environment using a 50 events data set (training_000021450_000021499). The model can be tested by running it in a docker container:

```
./rundocker.sh
```

The script needs to be modified to define the *INPUT_DATA* path to locate the data set.
The script in addition produces a zip-file of the repository to be submitted to CodaLab.
The model works with singularity containers as well:

```
./runsingularity.sh
```

A9. Outlook

Great care has been taken to avoid any low-level detector specific information in the core tracking algorithms in order to keep the algorithm as generic as possible. The neural networks are mainly trained with pure spatial information. As such the algorithms could be easily transferred to other environments.
The graph implementation furthermore offers a serialization function that allows to very quickly get a list of tracks from triples stored in a DAG by recursive graph traversal. This already works surprisingly well in an environment with a lower track density (up to a few hundred tracks).

A considerable speedup of the model is possible by use of more cores. First tests with high performance AWS ec2 instances indicate that a CodaLab score above 1.0 could be reached in a larger multithreading context.

A10. References

1. Johan Sokrates Wind, Fast and scalable tracking by outlier density estimation, August 2018, https://github.com/top-quarks/top-quarks/blob/master/top-quarks_documentation.pdf
2. Johannes Steffens, Marcel Kunze, Helmut Schmücker, Neural Network Objects, https://github.com/marcelkunze/rhonno
3. ROOT toolkit, https://root.cern.ch/
4. J.F. Puget, https://www.kaggle.com/c/trackml-particle-identification/discussion/63250