Machine learning for surface prediction in ACTS

Benjamin Huth[1], *Andreas* Salzburger[2], and *Tilo* Wettig[1]

[1]Department of Physics, University of Regensburg, 93040 Regensburg, Germany
[2]CERN, Esplanade des Particules 1, 1211 Meyrin, Switzerland

Abstract. We present an ongoing R&D activity for machine-learning-assisted navigation through detectors to be used for track reconstruction. We investigate different approaches of training neural networks for surface prediction and compare their results. This work is carried out in the context of the ACTS tracking toolkit.

1 Introduction

Track reconstruction is a challenging part of event reconstruction in HEP experiments. It is usually CPU intensive and often prone to edge cases that need carefully crafted solutions for efficient and reliable results. One main task during track reconstruction is the transport of the track parameterization (i.e., the representation of the trajectory, either in free or in surface-bound coordinates) to another point in the detector. While this transport may have analytic solutions for certain special cases, in the general case it needs the numerical solution of the equations of motion through the detector setup. Such a general solution is implemented as a fourth-order Runge-Kutta numerical integrator in the ACTS tracking toolkit [1]. In a real-life experiment, however, the material inside the detector causes additional changes and uncertainties in the track propagation, manifested in a change of momentum due to energy-loss effects and to additional noise terms in the covariance matrix caused by multiple Coulomb scattering and energy-loss straggling. In order to adequately apply such correction terms, the integrator must be aware of its position in the detector geometry. For this task a navigation module is required.

The environment in which this navigation takes place is built up from individual surfaces, which are either sensitive surfaces or volume boundary surfaces. The navigation problem consists of predicting the next surface the track will cross in the detector geometry.

2 Motivation: limitations of the current ACTS navigator

A modern particle detector has thousands of individual measurement surfaces, e.g., silicon sensors in the pixel and/or strip detectors. In order to allow for acceptable timing performance when navigating through the detector, a selection of candidate surfaces has to be done. In ACTS this is realized through a dedicated navigator module that suggests the most likely next candidate to the numerical integrator, which then performs the solution of the differential equation of motion. The navigator module builds upon a hierarchical structure of volumes, layers and grid structures in which the candidate surfaces are filled and from which they are retrieved when the track is in local proximity.

While such an approach has been proven to be very effective, it comes at a price: geometry building and presorting is highly tailored to a specific detector layout and needs a certain amount of tuning to find the sweet spot between timing and navigation performance. The code for this kind of navigation is highly branched and complex, as it tries to follow a logical hierarchy tree through the detector. Attempts to apply the same code, e.g., on accelerated hardware have failed so far for several reasons: the geometry classes for guiding through the detector are highly polymorphic, and dynamic branches make large-scale parallelization difficult. In addition, unconventional detector layouts require dedicated modules that might not exist yet within the ACTS toolkit. Finally, this solution is not free from errors.

3 ML-assisted navigation

The navigation module is not required to give an exact prediction of the next surface to be intersected by the trajectory. Rather, it provides an inclusive list of candidate surfaces which are then attempted to be intersected. In the following, a machine-learning (ML) approach to finding these candidate surfaces is presented. This approach has the potential to overcome the limitations described in Sec. 2.

3.1 Training data

The underlying idea of ML-assisted navigation is that the necessary information to navigate through the detector can be learned from tracks simulated in this specific detector setup. In principle a navigator is needed beforehand to be able to generate these data. In the case of the results presented in this paper, the stable and well-tested infrastructure of ACTS is used for this.

Even in the absence of an existing navigation module, an ML-assisted navigation can be applied: the navigation problem is inherently deterministic, and a brute-force trial-and-error navigator will — together with the cross-check through the numerical integration — provide a precise solution to the problem. This is computationally very inefficient but only needs to be done once to train a navigator, which will then be much more efficient afterwards.

3.2 Representing surfaces

The basic entities in this context are *surfaces*. The first question that arises is how a surface can be represented in the training data so that a neural network can process it properly. In ACTS each surface has a unique identifier, `Acts::GeometryId` (in fact just a wrapper around an integer). This integer contains information about the surface that can be decoded by applying certain bit masks to the integer. However, using the numerical value of this integer in computations is not sensible because the results of operations such as addition or multiplication have no interpretation. Rather, this integer represents a *categorical variable*.

There exist different approaches to handle categorical variables in neural networks. One way to pass categorical values to a neural network is to use a *one-hot encoding*. This means that each possible category is represented by an entry in a d-dimensional vector, where d is the number of categories. The entry of the correct category is set to 1, all others to 0. However, since there are several thousand surfaces, the high dimension of the feature space would cause both algorithmic problems (*curse of dimensionality*) and computational inefficiencies (because the data are extremely sparse). Furthermore, one-hot-encoded values only allow one to distinguish categories but are not able to represent more complex relations between them.

An alternative approach that is capable of representing such relations is the use of a technique called *embedding*. Each category is mapped to a point in an n-dimensional space. In this space, the relative positions of these points can encode relations between the categories. This technique was successfully applied to different fields in recent years [2]. In the case of surfaces, either an n-dimensional embedding can be trained or simply the 3D center coordinates of the surfaces can be used. Other options to generate embeddings are combinations of the above-mentioned approaches or the application of techniques such as *metric learning*. However, these were not considered in the work presented here.

Representing each surface as a point in an embedding space works fine for most surfaces in the detector. However, if a surface has a large spatial extent, tracks crossing it can have a wide variety of target surfaces, and it may be more difficult to predict the next surface accurately. In the tested setup, this applies mainly to the *beam-pipe surface*, an innermost cylinder that represents the actual beam containment and extends along the entire detector. In the training data, this one surface represents all track origins (while in reality, the tracks originate on the beam line). A possible optimization here is to split this surface into different sections along the beam-pipe axis, and generate an embedding for each of them. One can then even further split the track origins depending on the polar angle of their unit direction, $\phi = \text{arctan2}(d_x, d_y)$. The splitting in the z-direction is performed such that the number of track origins per bin is roughly the same (as a consequence, their boundaries are not equidistant on the z-axis). The beam-pipe split will be denoted as a tuple of two integers in the following: (z-split, ϕ-split). For certain models the use of beam-pipe splitting can improve the prediction performance at the beam pipe significantly.

3.3 Predicting the next surface

The problem setup for predicting the next surface is the following. The starting point is always on a surface, and at this point all information about the current track (position, momentum, charge) is available.[1] Given this information, the next surface the track will cross must be predicted. In the following, two different approaches to this problem are considered:

1. **Predict target directly.** The current surface $\vec{s}_i$ (represented as a vector in the embedding space) and the track parameters $\vec{p}_i$ are fed into the neural network that then predicts a point $\vec{s}_f$ in the embedding space:

$$(\vec{s}_i, \vec{p}_i) \rightarrow \vec{s}_f \, . \tag{1}$$

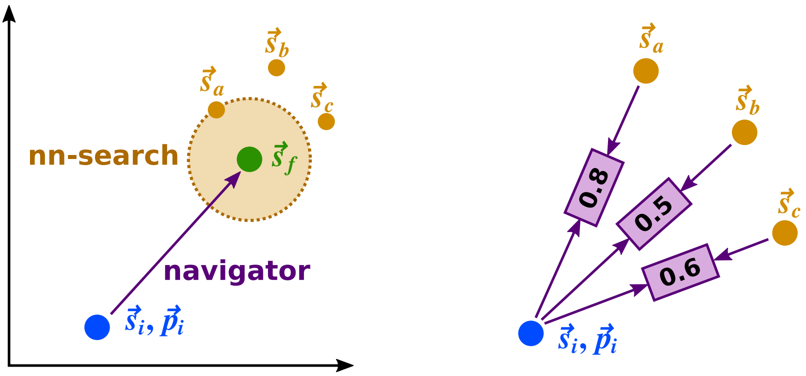
   Since this point will hardly ever match exactly any representation of another surface, it is resolved to k candidate surfaces with a *k-nearest-neighbor search* (KNN search) in the embedding space (see Fig. 1a).

   This approach has the advantage that it can predict the target without any candidates known beforehand. However, a nearest-neighbor search is a computationally very expensive operation.

2. **Predict score of surface pair:** In this case, a fixed set of candidate target surfaces is given for each starting surface. Together with the starting surface they effectively form a graph of all considered paths through the detector. The representation of the starting surface $\vec{s}_i$, the representation of a potential target surface $\vec{s}_f$ and the track parameters $\vec{p}_i$ are fed into the network. The result is a number between 0 and 1 representing how likely the network considers the tested candidate surface to be the next surface:

$$(\vec{s}_i, \vec{s}_f, \vec{p}_i) \rightarrow \text{score} \in (0, 1) \, . \tag{2}$$

[1] Note that the prediction does not attempt to estimate the full track state at the next surface. This must be done by the integration component of the particular propagator implementation.



(a) Target-prediction approach: Given the embedding $\vec{s}_i$ of the starting surface and the track parameters $\vec{p}_i$, a target embedding $\vec{s}_f$ is predicted which is then resolved to an actual surface by nearest-neighbor search.

(b) Score-prediction approach: The embedding $\vec{s}_i$ of the starting surface, the embedding of a possible target surface (e.g., $\vec{s}_a$) and the track parameters $\vec{p}_i$ are used to predict the probability that this candidate target is the actual target.

Figure 1: Visualization of the two surface-prediction methods.

Then all candidates are sorted by score and used for navigation (see Fig. 1b).

The advantage of this method is that it works with a fixed set of surface candidates. Since no nearest-neighbor search needs to be performed, the method is computationally less expensive. On the other hand, the method can only find surfaces that are present in the graph. Hence it can happen that the correct surface cannot be found, in which case a fallback solution must be implemented. If this happens rarely enough, the score-prediction method is a good option.

4 Training process and test results

In this section, the setup for the different approaches is described and their results are presented and discussed.

4.1 Training-data generation, model architecture and software environment

The data used for training, validating and testing the navigation models are generated using the ACTS propagation example (settings: a flat distribution in momentum between 0.1 and 100 GeV and angular uniform distributions within detector acceptance) with a custom logger to record the necessary data. All results shown in this paper are produced with the data of roughly 500 thousand simulated tracks in ACTS' *generic detector* that has also been used for the Tracking Machine Learning Challenge (TrackML) [3]. Two thirds of these data are used for training, and the remaining third is used for evaluation after the training. The data are split on a track basis, so only complete tracks are present in the training and evaluation datasets. The pre-trained embeddings (see next subsection) were fitted using a set of 128 thousand tracks.

All presented models are simple fully-connected feed-forward neural networks. As input features, surface embedding and track parameters are used as described in Sec. 3.3. The track

	# of layers	Units per layer	Embedding	BP split (z, ϕ)	loss
Target pred.	3	500	10D, pre-trained	(40,1)	MSE
Score pred.	4	750	3D, real space	(400,16)	binary crossentropy

Table 1: Hyperparameters of the models presented. All models where trained with the help of the *Adam* optimizer and a learning rate of 0.001. *BP split* is short for beam-pipe split in this context.

parameters handed over to the models are the three components d_x, d_y, d_z of the unit direction and the ratio q/p of charge and magnitude of momentum. The hyperparameters used are listed in Table 1.

The training and evaluation environment is based on Tensorflow/Keras 2.3.0 [4, 5] on top of Python 3.8.6. All code used to generate these results can be found at [6].

4.2 Pre-training of embeddings

When training an embedding, there are two choices: one can train the embedding as part of the final navigation model, or one can train it beforehand and use the embedding later as input to the model. The first approach has the obvious advantage that the embedding can be fitted better to the specific problem but has the disadvantage of combining two different training objectives in one model. Whereas the navigation model should generalize as well as possible beyond the training data, the embedding should be fitted as well as possible to the training data, with no need for generalization. Because of these considerations and insights from numerical experiments, pre-trained embeddings were used for all results presented.

Within the Keras framework, embeddings of the form $n \in [0, N) \rightarrow \vec{v} \in \mathbb{R}^d$ (where n is an integer identifier for a specific surface and N is the number of surfaces) can be easily created and trained using the provided embedding layer. As training data, pairs of surface identifiers n_1 and n_2 with the corresponding truth information were used: 0 if they are not connected by any track in the simulation data, 1 if they are. To map the two resulting embedding vectors to a single scalar, a dot product $\vec{v}_1 \cdot \vec{v}_2$ is used. The dot product is then shifted and scaled by a dense layer of size 1 to match the output range $(0, 1)$. Afterwards, the embedding layer can be used to preprocess the data before feeding them into the navigation model. The embeddings used for this work are trained for 20 thousand epochs with simulated data consisting of 128 thousand tracks.

4.3 Baseline comparison: weighted-graph navigator

To allow for better interpretation of performance results, the performance of a very basic navigation model is used for comparison. This model always predicts the paths based on their number of occurrences in the training data (“weighted-graph navigator”). For example, the path that occurs most is always the first prediction. Even though it is only used as a performance baseline here, this approach could be used for a “first guess” in a real implementation, in order to save computing time in some cases. However, in a more realistic detector setup the performance of this approach is expected to deteriorate since there will be more paths available.

4.4 Target prediction

For this approach, a pre-trained embedding turned out to be much more effective than the use of the 3D surface coordinates. The use of small beam-pipe splits (e.g., (40,1)) slightly in-

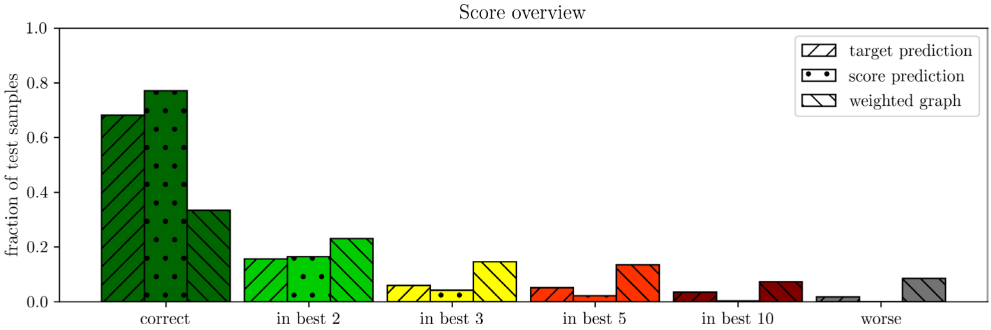
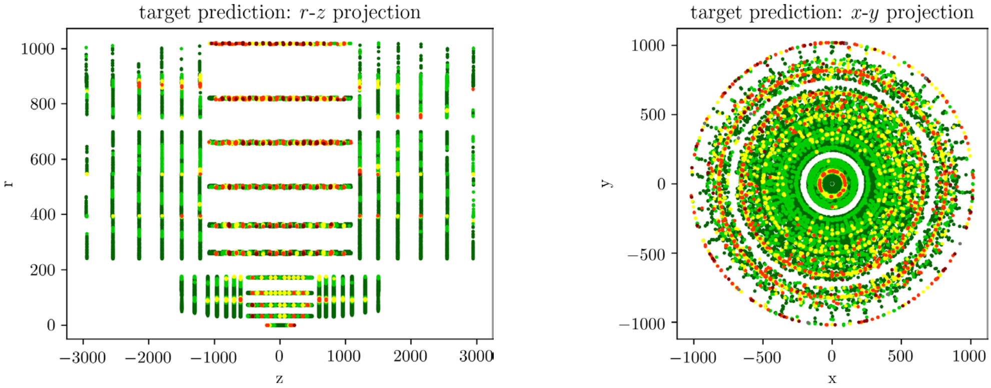
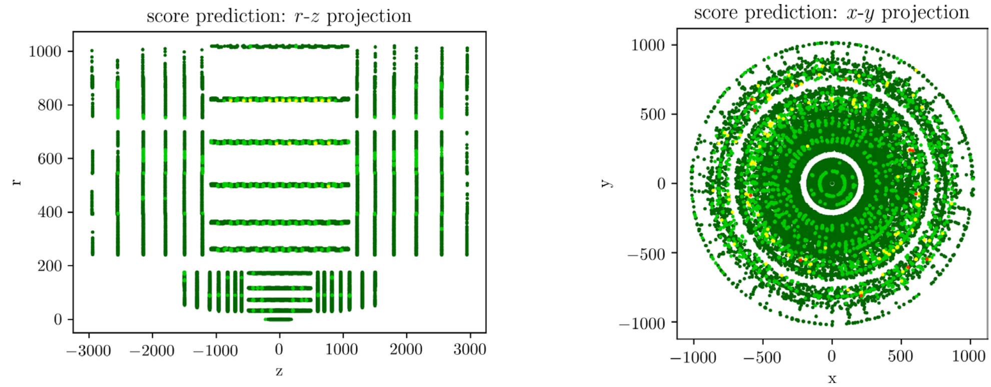


Figure 2: Score distributions in the generic detector for the two navigation models and a weighted-graph approach for a baseline comparison.



(a) Projection of the results of the target-prediction model onto the r-z plane (left) and the x-y plane (right). The red dots in the horizontal lines (left) match exactly the overlaps of the surfaces in this area.



(b) Same as Fig. 3a but for the score-prediction model. As in Fig. 3a (left plot), we can see score fluctuations which correlate with the overlaps. However, the overall performance is much better.

Figure 3: Projection of the score results onto the r-z and x-y planes. In each case, 100k random points where sampled and replaced by the average of all neighbors (in that plane) within a 5 mm radius. The scores are plotted with better scores in lower plotting layers. Hence better scorses could be hidden by worse scores. The color coding is as in Fig. 2.

creases the performance. However, even without a beam-pipe split a reasonable performance is obtained. Large beam-pipe splits (e.g., (400,16)) decrease the performance noticeably.

The performance metrics of the target-prediction navigator are evaluated with the help of a KNN search [7]. The ten nearest neighbors are sorted by their distance from the predicted point in the embedding space. If the closest neighbor is also the true target embedding, the prediction is considered to be correct and put it in the score category *correct*. If the true result is one of the other elements, that element is put in the corresponding category (the categories used are *in best 2*, *in best 3*, *in best 5*, *in best 10*). If the correct target does not occur in these 10 points, the prediction is considered to be failed (category *worse*). Results for the generic detector are shown in Figs. 2 and 3a.

The results show that the target-prediction navigator can reach good performance when using a large enough dataset for training. However, there always remains a small share of cases ($\sim 5\%$) where the predictions are more than 5 neighbors off. This happens mostly in the barrel sections of the short strip and the long strip and can be correlated to the overlaps of the surfaces in this area (see Fig. 3a, left plot). Further research is needed to show if this can be improved by, e.g., optimizing the embedding.

4.5 Score prediction

For the score prediction the most successful approach was to use the center coordinates of the surfaces as the embedding. Additionally, this approach benefits significantly from a fine-grained beam-pipe split. The performance metrics of the score-prediction navigator are evaluated by comparing the score of the true target to the score of the predicted target. If the true target has the highest score, the prediction is considered to be correct (category *correct*). If the candidate with the second-highest score is the true target, it belongs to category *in best 2*, etc. (as described in the previous subsection). If the graph only contains a few possible targets, the solution is guaranteed to be found in the, e.g., first 4 targets. This must be taken into account when interpreting the results.

The problem mentioned earlier, i.e., the possibility that the true target is not present in the graph, cannot occur in the evaluation since the graph is built from the entirety of the available data. In a real simulation environment this problem will occur, in which case a fallback solution must be implemented as mentioned above. Additionally, care must be taken when generating the graph from the data. Graphs generated from data sets like the one described above show non-negligible differences regarding the connections present in the graph. Further studies are needed to understand how to generate the graph in a way that it is less dependent on statistics.

The performance of the score-prediction approach, see Figs. 2 and 3b, is notably better than that of the target prediction, especially in the barrel regions. Most importantly, in nearly 100% of the cases the prediction lies within the first 5 candidates. This, together with the fact that no nearest-neighbor search must be performed to resolve the targets, makes it the preferred approach for ML-based navigation.

5 Implementation in ACTS

We provide a proof-of-concept implementation that comprises both discussed models, the score-prediction navigator and the target-prediction navigator, at [8]. To load the neural networks into ACTS, the ONNX API and runtime [9] is used.

6 Conclusion and Outlook

The results show that it is possible to use neural networks for navigation. Furthermore, these approaches can be applied to almost any detector geometry in combination with a trial-and-error navigator. It is expected that with more extensive hyperparameter optimization the results can improve further.

No detailed analysis regarding the runtime performance, the memory consumption and the computational overhead of the different approaches has been done yet. However, first tests show that the neural network inference with the current models is about three orders of magnitude slower than the straight-line intersection (which is used to verify a surface hit during integration). In order to make this approach computationally competitive, the runtime of the neural network inference must be improved substantially, e.g., by simplifying the network architecture or by large-scale parallelization. This is the subject of future work.

Regarding further research, not all potential model architectures have been tried for this task yet. For example, the potential of recurrent neural networks is yet completely unexplored. This will also be the subject of further research.

References

[1] X. Ai, C. Allaire, N. Calace, et al., *Acts project: v3.0.0* (2020), `https://doi.org/10.5281/zenodo.3937454`

[2] C. Guo, F. Berkhahn, CoRR **abs/1604.06737** (2016), `arXiv:1604.06737`

[3] S. Amrouche, L. Basara, P. Calafiura, et al., The Springer Series on Challenges in Machine Learning p. 231–264 (2019), `arXiv:1904.06778`

[4] F. Chollet et al., *Keras*, `https://keras.io` (2015)

[5] M. Abadi, A. Agarwal, P. Barham, et al., *TensorFlow: Large-scale machine learning on heterogeneous systems* (2015), `https://www.tensorflow.org/`

[6] B. Huth, *Source code repository for the models and the neccessary tools*, `https://github.com/benjaminhuth/acts_ml_navigator`

[7] F. Pedregosa, G. Varoquaux, A. Gramfort, et al., Journal of Machine Learning Research **12**, 2825 (2011)

[8] B. Huth, *Acts fork with ml navigator branch*, `https://github.com/benjaminhuth/acts/tree/feature/ml-navigator`

[9] Microsoft, *ONNX runtime (source code repository)*, `https://github.com/microsoft/onnxruntime`