Charged Particle Reconstruction in CLAS12 using Machine Learning

Polykarpos Thomadakis[a,1], Kevin Garner[a], Gagik Gavalian[b], Nikos Chrisochoides[a]

[a] *CRTC, Department of Computer Science, Old Dominion University, Norfolk, VA, USA*
[b] *Jefferson Lab, Newport News, VA, USA*

Abstract

In this work, we present studies of track parameter reconstruction from raw information in CLAS12 detector's Drift Chambers, using Machine Learning (ML). We study the resolution of tracks reconstructed with different types of ML models/algorithms, including Multi-Layer Perceptron (MLP), Extremely Randomized Trees (ERT) and Gradient Boosting Trees (GBT) using simulated data. The resulting ML model is capable of reconstructing track parameters (particle momentum, and polar and azimuthal angles) with accuracy similar to Hit Based (HB) tracking code, but 150 times faster. Moreover, physics reactions can be identified using the particles reconstructed by the neural network in real-time (with a rate of about 34 kHz) during experimental data collection. The developed model can be used in numerous applications, such as triggering specific physics reactions in real-time, detector performance monitoring, and real-time detector calibration.

1. Introduction

The mission of Nuclear Physics (NP) is to explore and understand all forms of nuclear matter. Experiments in nuclear physics use large accelerators that collide particles at nearly the speed of light to study the structure of nuclei, and nuclear astrophysics, and to produce short-lived forms of matter for investigation. Modern experiments use high-intensity beams incident on targets that produce tens of thousands of interactions per second. At high luminosities, with increased noise in tracking detectors, conventional algorithms are hindered by increased combinatorics when trying to find and identify tracks. This leads to inferior detector data reconstruction speed and a decrease in track-finding efficiency. Machine Learning (ML) methods can be used to improve track-finding efficiency both for track identification [1, 2] and noise rejection applications [3, 4], while combining the two can lead to increased tracking efficiency.

1.1. CLAS12 Detector

The CLAS12 detector, located in experimental Hall-B at Jefferson lab, is designed to study electro-induced nuclear and hadronic reactions by providing efficient detection of charged and neutral particles over a large fraction of the full solid angle. The detector

[1]Correspoding author *pthom001@odu.edu*

is based on a combination of a six-coil torus magnet and a high-field solenoid magnet. The combined magnetic field provides a large coverage in both azimuthal and polar angles; a schematic view is presented in Figure 1.

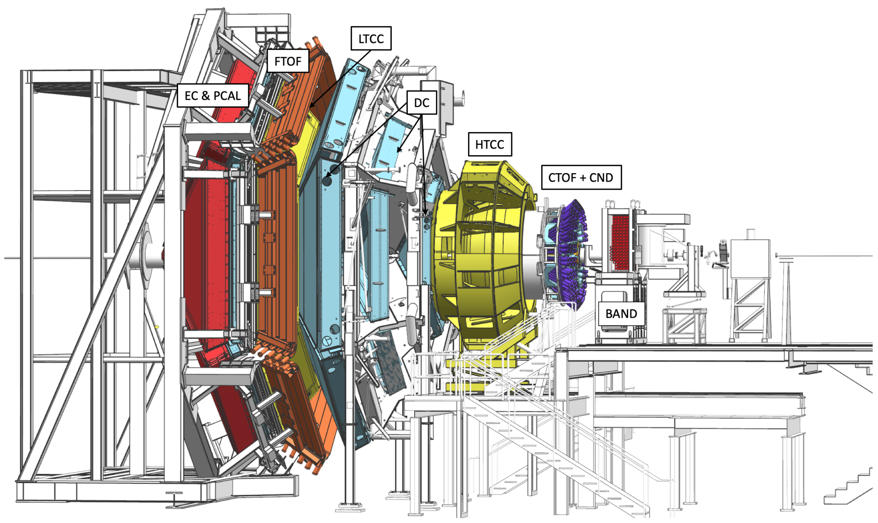
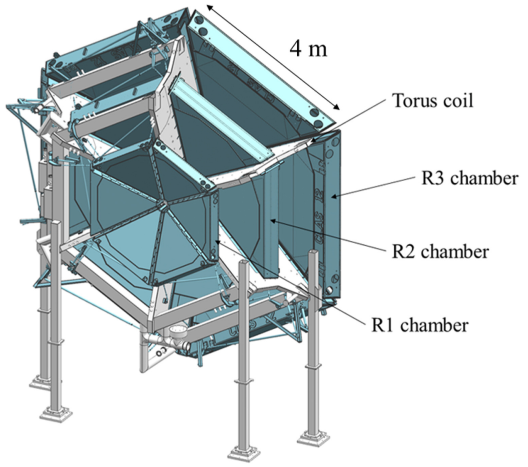


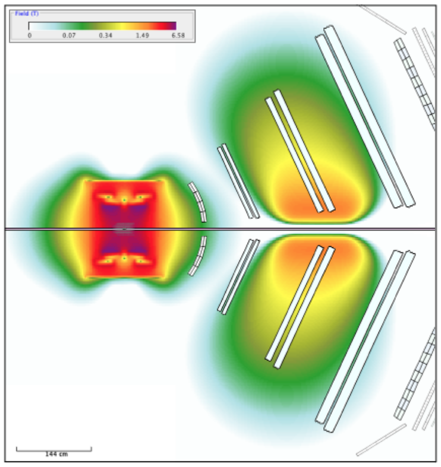
Figure 1: The CLAS12 detector in the Hall B beamline (figure reproduced from [5], coauthored by Dr. Gavalian). The electron beam enters from the right and impinges on the production target located in the center of the solenoid magnet shown at the right (upstream) end of CLAS12, where other detector components are also visible. Scattered electrons and forward-going particles are detected in the Forward Detector (FD), consisting of the High Threshold Cherenkov Counter (HTCC) (yellow), followed by the torus magnet (gray), the drift chamber tracking system (light blue), and time-of-flight scintillation counters (brown), and electromagnetic calorimeters (red).

The CLAS12 detector is divided into two parts: the Central Detector, built around the solenoid magnet, and the Forward Detector, built around the torus magnet. In the forward direction, Drift Chambers (DC) are used to track charged particles.

The six coils of the torus magnet mechanically support the forward tracking system, consisting of three independent DCs in each of the six sectors of the torus magnet. Each of the six DC sectors has a total of 36 layers with 112 sense wires, arranged in 3 regions (R1, R2, and R3) of 12 layers. In each of the six torus sectors, the DCs are arranged identically. As displayed in Figure 2a, the R1 chambers are located at the entrance to the torus magnetic field region. The R2 chambers are located inside the magnet where the magnetic field is close to its maximum, and the R3 chambers are placed in a low magnetic field space just downstream of the torus magnet. This arrangement provides independent and redundant tracking in each of the six torus sectors. Each of the 3 regions consists of 6 layers (called a super layer) with wires strung at a stereo angle of $+6^\circ$ with respect to the sector midplane and 6 layers (a second super layer) with wires strung at a stereo angle of -6° with respect to the sector midplane. This stereo view enables excellent resolution in the most important polar angle (laboratory scattering angle), and good resolution in the less critical azimuthal scattering angle. For details of the DC construction and performance, see [6]. The combined magnetic field can be seen in Figure 2b, where the slice view is shown. The particles originating from the interaction of electron beam with hydrogen target, located in the middle of the solenoid magnet (Figure 2b), are tracked in the forward direction using



(a)



(b)

Figure 2: (a) Drift chamber system in the CLAS12 forward tracking system from the design model. The small-size R1 chambers are located just in front of the torus magnet coils (gray shade). The medium-size R2 chambers are sandwiched between the coils of the magnet, and the large-size R3 chambers are located just downstream of the magnet. (b) Combined solenoid and torus magnetic fields. The color code shows the total magnetic field of both the solenoid and torus at full current. The open boxes indicate the locations and dimensions of the active detector elements (figures reproduced from [5], coauthored by Dr. Gavalian).

drift chambers. Particles traveling through each super-layer leave a signal in each of the wire layers. Wire hits in each super-layer are combined into clusters, then, using six clusters (one from each super-layer), the tracking code constructs track candidates. These track candidates are fitted in the early stage of track reconstruction to derive the track parameters, such as momentum and polar and azimuthal angle, based only on the positions of the hits (clusters), called hit-based tracking. At later stages of track reconstruction, timing information from clusters is used to further refine measured track parameters. The initial determination of track parameters based on hits is very important for determining event start time at the target since it leads to more refined time-based tracking. Finding the right combinations of clusters and the subsequent track parameter extraction is computationally intensive and takes about 40% of the total track reconstruction code. In CLAS12 reconstruction software, the hit-based tracking takes about $380 - 420\ ms$ per event.

1.2. Motivation

The CLAS12 reconstruction software already uses Neural Networks to identify tracks from all combinations of 6 clusters [2]. The track candidate identification neural network runs at speeds of 2 ms per event in a single thread and provides rates close to 30 kHz. In this article, we use Machine Learning (ML) to reconstruct particle track parameters from hits in the drift chamber. Combining the track-finding neural network, mentioned above, with a track parameter extraction network will allow experiments to reconstruct particles in real-time (at rates comparable with experimental data collection rates). The track parameter extraction network is using clusters reconstructed in each super-layer of the drift chambers. Examples of tracks are shown in Figure 3, where cluster mean positions for each super-layer are plotted for each example track.

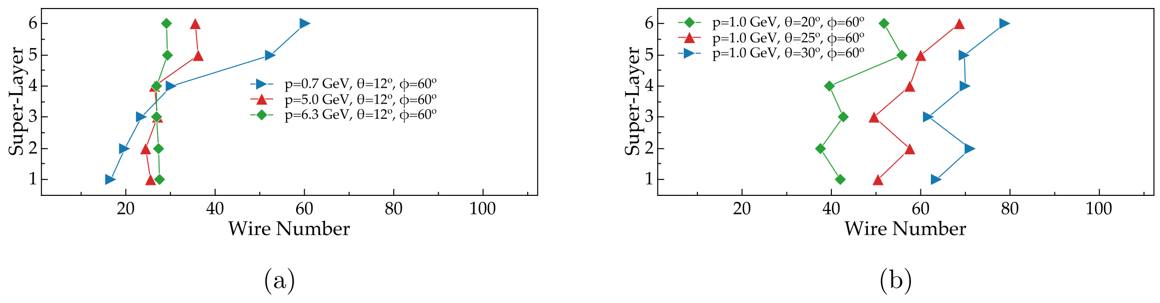


Figure 3: Examples of tracks in sector one of the drift chambers. Average wire position for cluster is plotted vs super-layer number for the tracks. a) tracks with different momenta but same angles ($\theta = 12^\circ$ and $\phi = 60^\circ$). b) tracks for different polar angle with same momentum and azimuthal angle ($p = 1\ GeV$ and $\phi = 60^\circ$).

In Figure 3a, tracks with different momenta but with same angles ($\theta = 12^\circ$ and $\phi = 60^\circ$) are shown. The particles with different momenta passing first through the solenoid magnet enter the first region of drift chambers in different locations. Traveling through the entire length of drift chambers and passing through a toroidal magnetic field, particles with different momenta change their trajectory as can be seen in the figure. In Figure 3b, particles with the same momenta and azimuthal angle ($p = 1\ GeV$ and $\phi = 60^\circ$) are shown for different polar angle values. As it is evident from the figures, each track has a unique set of clusters in drift chambers. We used a neural network to learn the output track parameters based on the provided input cluster positions and studied different types of machine learning models and hyperparameters to determine which network provides the best parameter inference in the least amount of time.

2. Data Description

Data for these studies are produced using Geant [7], based on CLAS12 detector emulation software, GEMC [8]. GEMC is a broadly used sofware for all studies perfomed in CLAS12 [9, 10, 11] and is actively maintained to accurately simulate high energy physics signals. The generated data are reconstructed using CLAS12 data reconstruction software [12] and are then normalized and used to train and test the ML models against reconstructed track parameters. This process allows us to generate datasets large enough to train a machine learning model optimally, while assuring that the key features and patters learned from training can accurately represent any new experiment in the existing settings.

The missing mass distribution for $H(e^-, e^-\pi^+)X$ can be seen in Figure 4, where the stages of reconstruction software are shown. In Figure 4a, the missing mass distribution is shown for simulated events, before processing them with GEMC. The missing nucleon peak is clearly visible at mass value $m_n = 0.938$. In Figure 4b, the missing mass distribution is shown after CLAS12 software reconstruction of charged particles (e^- and π^+), using only hits in drift chambers (called Hit Based Tracking). The peak is wider because the particle

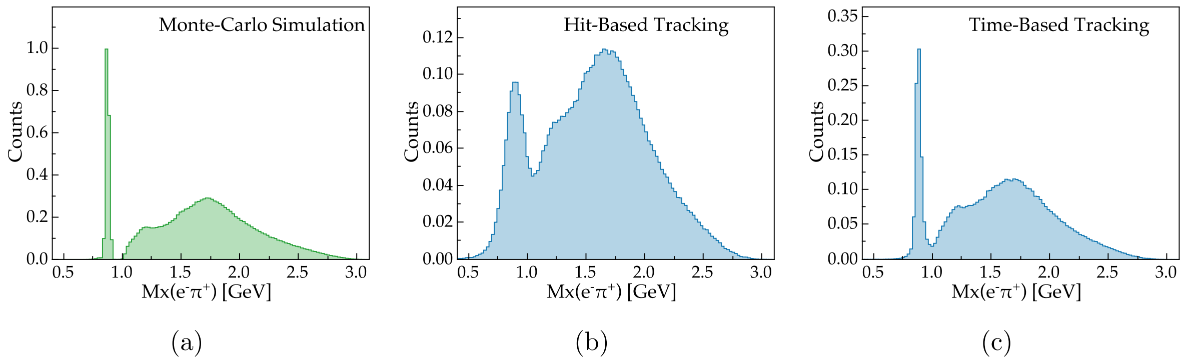


Figure 4: Missing mass distribution of $e^-\pi^+$ for generated $H(e^-, e^-\pi^+)X$ events, calculated using a) particle parameters that were generated by physics generator (Pythia), b) particle parameters reconstructed by Hit Based Tracking (HB) algorithm and c) using particle parameters from Time Based Tracking (TB) reconstruction algorithm.

passing through detector components experiences scattering, which results in energy loss and the reconstructed final momentum not being the same as the original momentum of the particle scattered off the target. The second stage of track reconstruction is time-based tracking where track parameters (i.e. momentum, polar and azimuthal angle) are further refined by fitting track candidates with Kalman-Filter, using timing information from hits that the track candidate contains. The final missing mass distribution after the time-based tracking with improved momentum resolution is shown in Figure 4c.

2.1. Training Dataset

The training dataset [2] consists of 6 input numbers (nodes), representing the mean wire position for a cluster in each super-layer of the drift chambers, and three output nodes, representing the particle parameters: momentum (p), polar angle (θ), and azimuthal angle (ϕ). The reconstructed track parameters for training are taken from the results of time-based tracking. The results from the ML models are then compared to distributions from hit-based tracking. For our studies, we experimented with three ML models, namely Multi-Layer Perceptron (MLP), Extremely Randomized Trees (ERT), and Gradient Boosting Trees (GBT). For all models, we evaluated different hyperparameters and present the ones producing the best results. To evaluate model performance, we constructed missing mass distributions using inferred track parameters and compared them to missing mass distributions from hit-based tracking of CLAS12 reconstruction software.

We used randomly selected simulated data to develop the models since the geometry used in the simulation is the same for all six sectors and the ML model can be trained for one sector and applied to all. We specifically trained each model on data from sector 2 (i.e. training set, $\approx$ 100k rows) and then proceeded to evaluate them using the data from all remaining sectors (i.e. testing set, $\approx$ 100k rows). In an experimental setup, detector components are usually shifted slightly from their intended ideal positions and track reconstruction

[2]Data available at: `https://github.com/gavalian/artin/tree/main/projects/pe/py`

software must take these shifts into account to accurately reconstruct track parameters. The extension of this method to experimental data will require training the model for each sector individually to account for slight uncertainties in detector positioning. We must note that normalization was applied to both the input and output features when building/testing each model. Their performance improved significantly compared to utilizing the raw input/output values (regardless of how the models' attributes were tuned).

3. Models Description and Performance Evaluation

In this section, we present the utilization and evaluation of three machine learning algorithms to deal with the challenge of track parameter reconstruction, namely Multi-layer Perceptron (MLP), Extermely Randomized Trees (ERT) and Gradient Boosting Trees (GBT). The three algorithms were chosen based on our previous work in applying machine learning on CLAS12 data for different applications [1, 3], as well as, an extensive experimentation with several other algorithms, including other tree-based algorithms, convolutional and graph neural networks. We evaluated the aforementioned algorithms on the dataset presented above, tuned their hyper-parameters using grid-search cross-validation and present the ones that achieved the best results below.

3.1. Multi-Layer Perceptron

A multi-layer perceptron (MLP) [13] regressor neural network is a supervised learning algorithm that trains on a dataset (given a set of features and a target) to learn a function

$$f(\cdot) : R^m \rightarrow R^o$$

, where m is the number of input dimensions and o is the number of dimensions for the output.

MLP can learn a non-linear function approximator for regression given that it may contain one or more non-linear layers (called hidden layers) between the input and output layers. Each layer contains a set of neurons (where the input layer's neurons represent the input features) that transform the previous layer's values based on a weighted linear summation and an activation function. The activation function simply governs the degree to which an output signal of a neuron is affected by the weight, before moving to the next layer. This forward pass on the network is part of the stochastic gradient descent algorithm. This includes the use of backpropagation, a method used to calculate an error (after comparing the network's output with the expected value) that is propagated through the layers of the network to update the weights associated with each neuron (according to how much they contributed to the error). The goal is to allow this backpropagation to occur over numerous training passes (or epochs) so that the MLP model can create an accurate function approximator based on the data.

There are several parameters that can be tuned, which can drastically effect the end accuracy of the model (including, but not limited to, the batch size, learning rate, number of hidden layers and neurons per layer, activation functions utilized, and number of epochs used for training). During a training epoch, the error calculated from each individual example in the dataset can be used to update the weights of the neurons either immediately (before training with the next example in the same epoch) or saved to update the weights later after a specific batch size of examples have been tested. The former tends to result in a less accurate model while the latter results in more stability with its predictions. That being said, the batch size should be relatively small for large datasets. Otherwise, the weights would not be updated often enough, which would result in the model learning too slowly to become sufficiently accurate by the end of training. An additional parameter that directly affects how much a weight is changed, based on the calculated error, is called the learning rate.

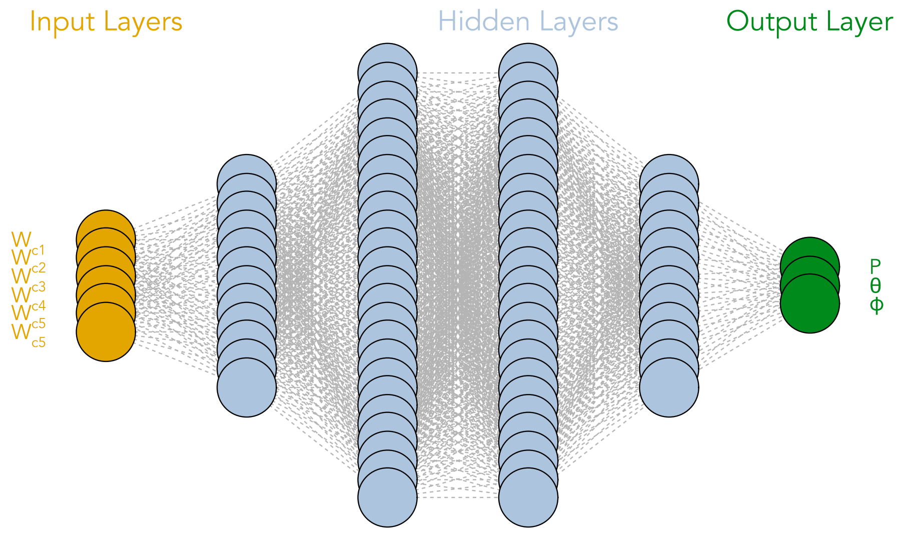
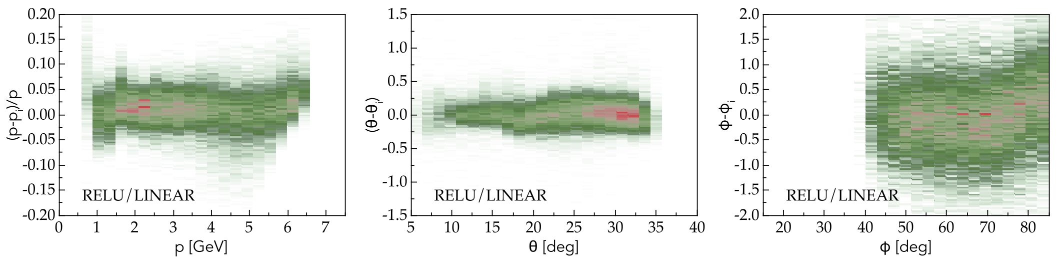


Figure 5: MLP Network Architecture with an input layer of 6 neurons, corresponding to the mean wire position of a cluster in each super-layer of the drift chambers, four hidden layers (with 12, 24, 24, and 12 neurons respectively), and an output layer of 3 neurons for the output particle parameters

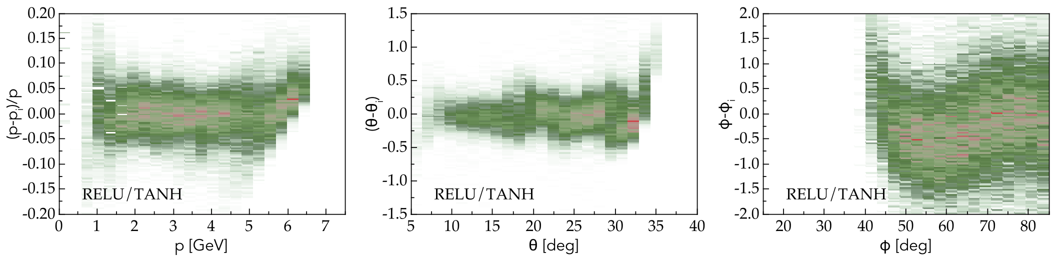
Activation Function	Mean Squared Error	Score
RELU/LINEAR	2.116297e-04	0.94453174
RELU/TANH	2.806255e-04	0.93888646
RELU/SIGMOID	2.514560e-04	0.94529152
TANH/TANH	2.117480e-04	0.94845265
TANH/LINEAR	1.991078e-04	0.94859838
SIGMOID/LINEAR	5.823259e-04	0.91438031
SIGMOID/SIGMOID	6.433758e-03	0.68839836

Table 1: Comparison of Activation Function Results for MLP

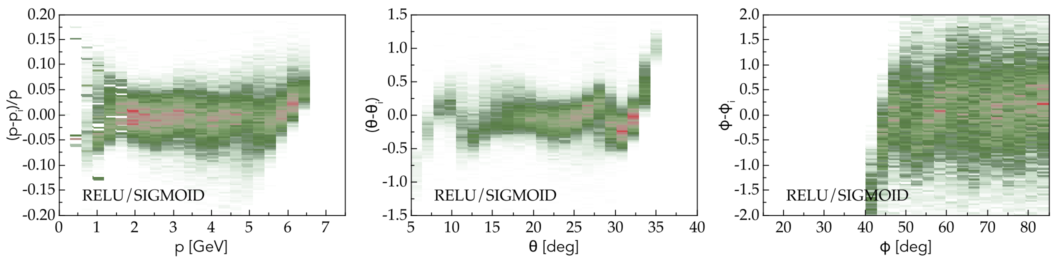
Due to the complexity of the problem at hand, the MLP network for our application was configured with four hidden layers. The input layer contained 6 neurons while the subsequent layers contained 12, 24, 24, 12, respectively, and the output layer contained 3 (for each of the aforementioned output particle parameters). This architecture can be seen



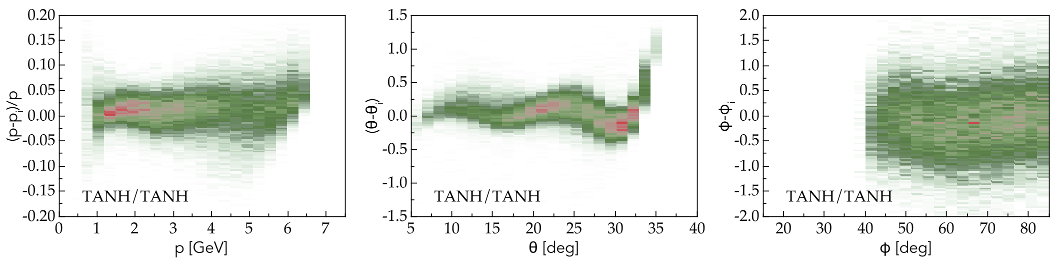
(a) Hidden layers: *ReLU*. Output layer: *linear*.



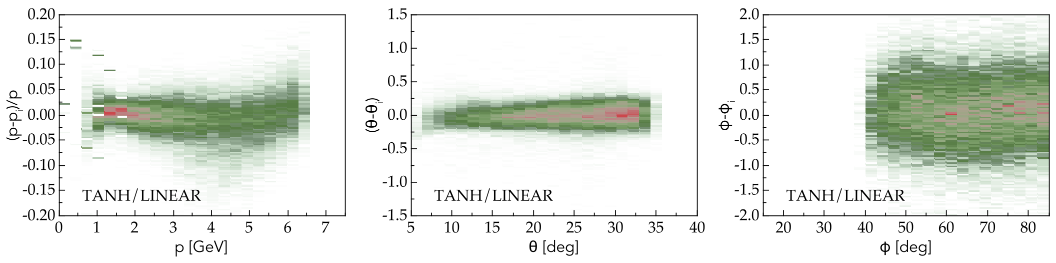
(b) Hidden layers: *ReLU*. Output layer: *tanh*.



(c) Hidden layers: *ReLU*. Output layer: *sigmoid*.

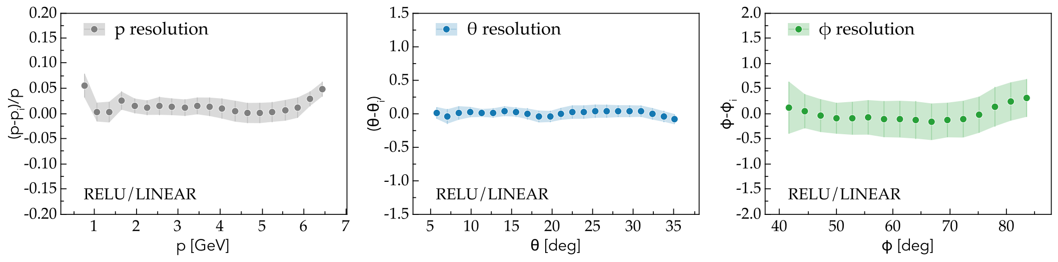


(d) Hidden layers: *tanh*. Output layer: *tanh*.

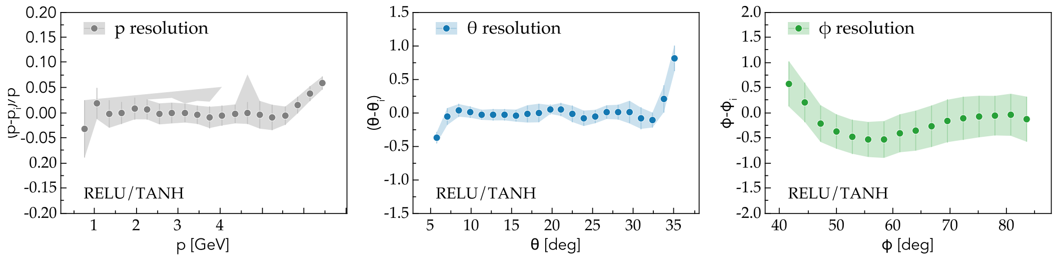


(e) Hidden layers: *tanh*. Output layer: *linear*.

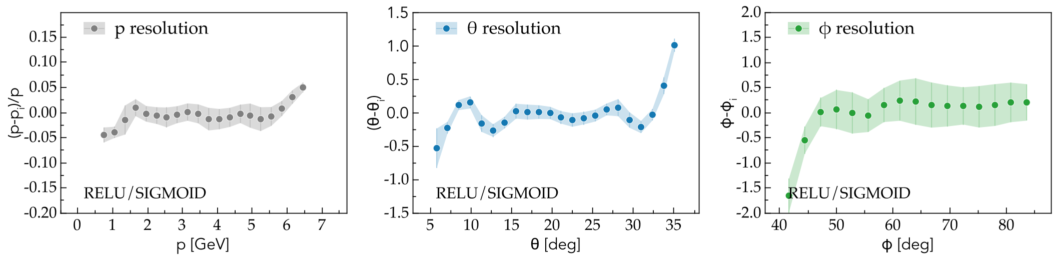
Figure 6: Reconstructed resolution for track parameters, p(left), θ(middle), ϕ(right), using the MLP with different combinations of activation functions for the hidden and output layers.



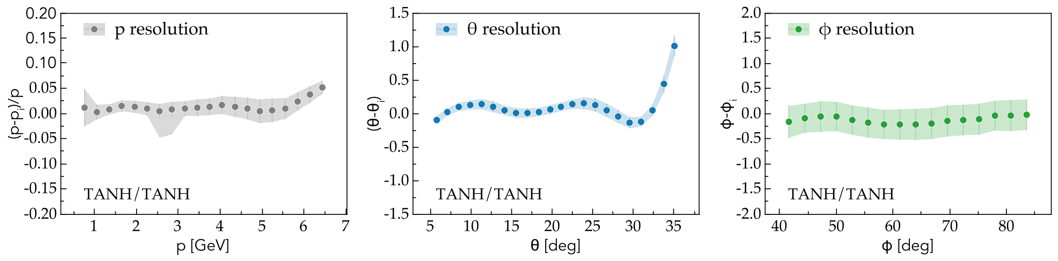
(a) Hidden layers: *ReLU*. Output layer: *linear*.



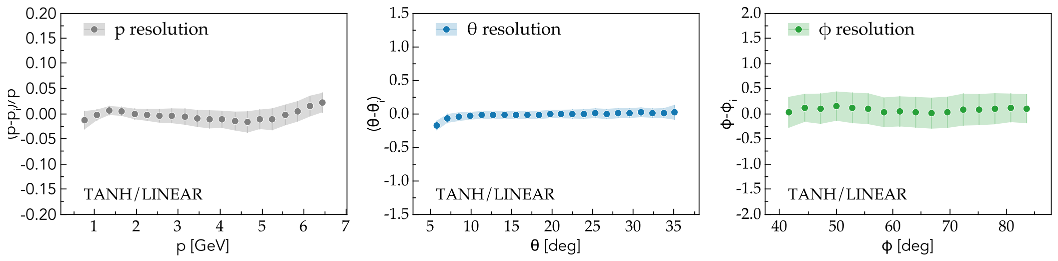
(b) Hidden layers: *ReLU*. Output layer: *tanh*.



(c) Hidden layers: *ReLU*. Output layer: *sigmoid*.



(d) Hidden layers: *tanh*. Output layer: *tanh*.



(e) Hidden layers: *tanh*. Output layer: *linear*.

Figure 7: Gaussian fit of the reconstructed resolution mean for track parameters, p(left), θ(middle), ϕ(right), using the MLP with different combinations of activation functions for the hidden and output layers. The sigmas of the fit are set as error bars for each point

in figure 5. Several combinations of activation functions were tested, where one function would be utilized for the input layer and all hidden layers, and another function would be utilized specifically for the output layer. Each configuration's performance was evaluated by measuring its parameter estimation efficiency as a function of parameters. For example, if the true (original) value of the momentum parameter is p_i for track i and the predicted value is p, an inference resolution can be calculated as $(p - p_i)/p$. Figure 6 shows the inference calculated for each output parameter using different combinations of activation functions. Figure 7 gives a gaussian fitted over each of the plots in figure 6, showing mean inferences with the sigmas of the fits set as error bars for each point. Table 1 shows the mean squared error and score given by each combination of activation functions after testing the trained models.

The number of epochs used for training were 1500 and the batch size utilized was 64. The solver used for weight optimization was "Adam", a stochastic gradient descent method that is designed based on adaptive estimates of lower-order moments [14]. We utilized a learning rate of 0.0001 and a beta 1 value of 0.99. This beta 1 parameter of the optimizer affects the exponential decay rate for the first moment estimates (allowing larger changes to be made to the weights during the initial steps of training and later reduced to be fine-tuned). Finally, the AMSGrad variant of the algorithm was applied (which optimizes the scaling of gradient updates when calculating the square roots of exponential moving averages of squared past gradients) [15]. This optimization is effective when training networks with larger output spaces (more than one parameter). The default values of the remaining parameters of the "Adam" optimizer in Tensorflow's Keras API were used.

3.2. Extremely Randomized Trees

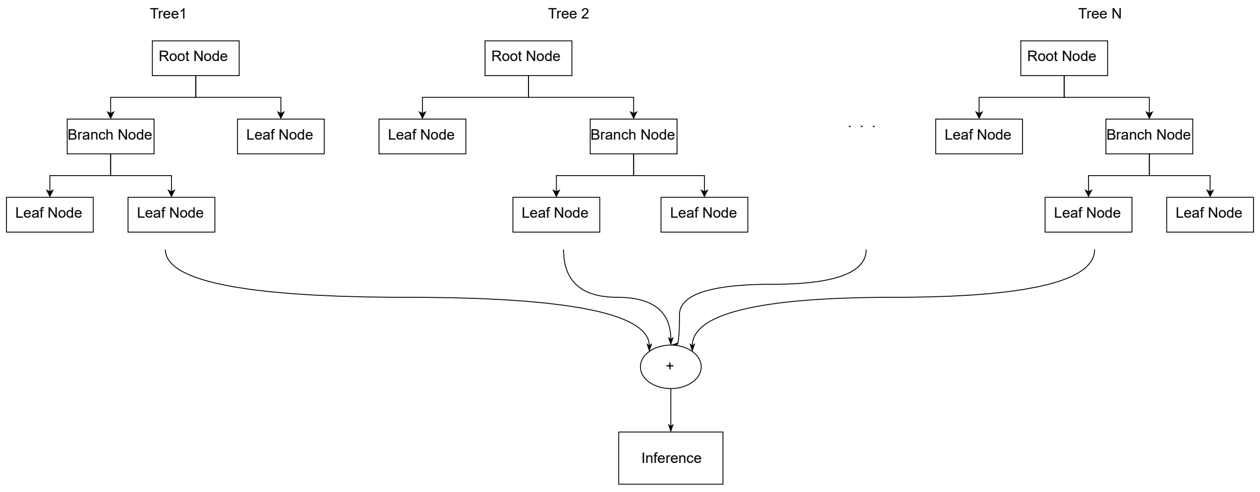


Figure 8: An example of the ERT algorithm. An ensemble of decision trees is formed in parallel from subsets of the dataset. Inference is performed by aggregating the decisions of all trees.

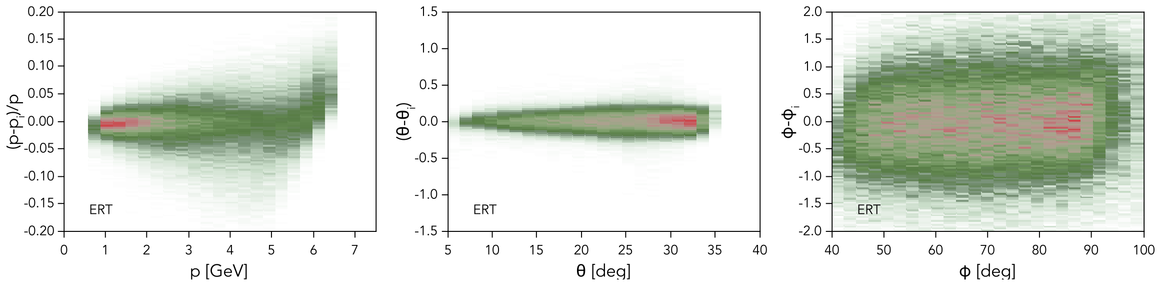
Extremely Randomized Trees (ERT) [16] is a supervised learning algorithm that constitutes an ensemble of randomly generated decision trees [17]. A decision tree is formed by recursively dividing the dataset into subsets. The splitting criteria are formed by the algorithm based on the classification features and such that the derived subsets are optimal. The recursive splitting process completes when the derived subset consists only of elements of the same class or when splitting does not add any extra value.

However, this algorithm introduces a high probability of overfitting the training data, in other words, creating a model that cannot generalize on new data. For example, it could generate a tree with a leaf for each example in the training set which would not be reusable when used on a different dataset. To face such issues, one can use multiple decision trees formed by random subsets of the dataset. The randomness introduced by generating new decision trees in this way dramatically improves the prediction power of the model. A method that follows this approach by picking random subsets of the input dataset with replacement is called random forests [18]. Extremely Randomized Trees is an extension of the random forests method that incorporates more randomness by randomly selecting the splitting criteria instead of choosing the best split. Moreover, this method forms the different random trees by using subsets of the input dataset without replacement. The final prediction of this type of model, where multiple trees are involved, is produced by taking the average of the predictions of all random trees (Fig. 8).

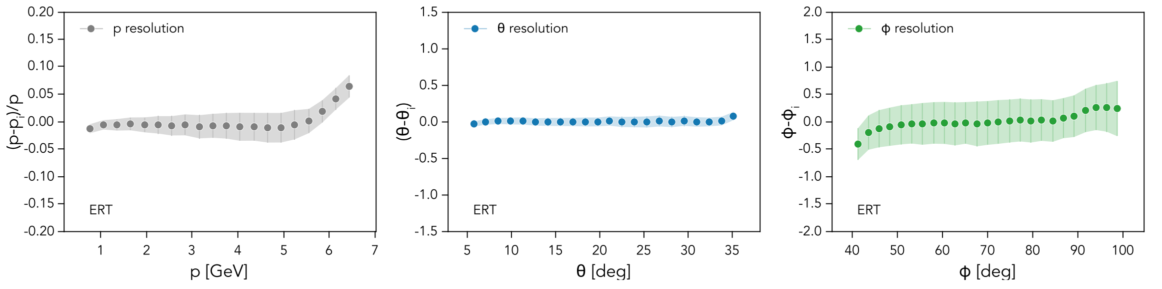
For our application, we use a model with three hundred estimators (decision trees), with no limit in the number of features to be considered when splitting. In addition, we used the information gain (entropy) split quality criterion. The rest of the parameters were kept at their default values.

3.3. Gradient Boosting Trees

Gradient Boosting Trees (GBT) [19] is a supervised learning algorithm that, like Extremely Randomized Trees, constitutes an ensemble of decision trees. The main difference



(a) Reconstructed resolution.



(b) Gaussian fit of the reconstructed resolution mean. Sigmas are set as error bars.

Figure 9: Reconstruction performance of parameters, p(left), θ(middle), ϕ(right), using the Extremely Randomized Trees model.

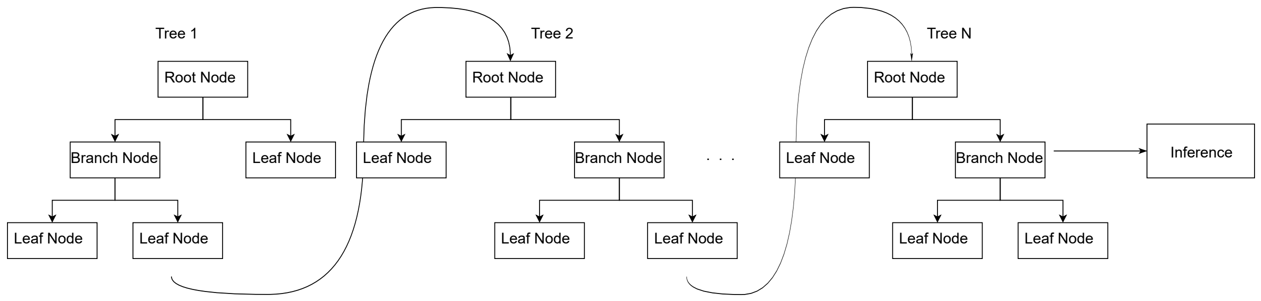
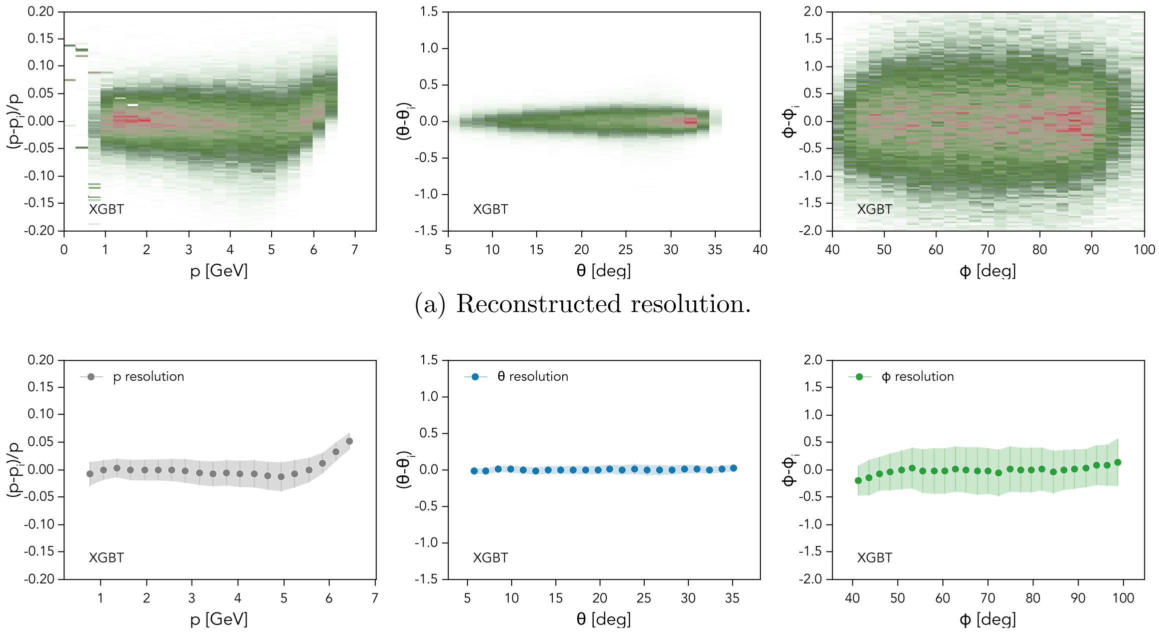


Figure 10: An example of the GBT algorithm. An ensemble of trees is formed sequentially, with each new tree optimizing the residuals of the previous one. Inference is performed by incrementally combining the results of individual trees in the order they were created.

between the two models is how they combine decision trees. In contrast to ERT which builds decision trees in parallel by fitting different subsamples of the dataset, GBT builds new trees sequentially and incrementally (Fig. 10). The goal of each new tree is to minimize the error of the previous tree; thus, each tree fits the residuals from the previous one (hence the name gradient boosting). Another difference is regarding how inference is run. While ERT outputs a decision by aggregating the predictions of all trees at the end of the process (by averaging or majority vote), GBT combines results incrementally (e.g., by iterating the created trees sequentially and aggregating). Finally, because new trees are fit to improve errors of previous ones, GBT is more prone to overfitting when excessively increasing the number of trees. On the other hand, trees in ERT are independent, thus, increasing their number does not lead to overfitting.

The GBT network was trained on simulated sample data to extract the performance of reconstructing track parameters and compare them to other networks (shown in Figure 11). We used grid-search cross-validation to tune the parameters for the model used in these experiments. We found that the following parameters produced the best results: *learning rate:* 0.1, *max depth:* 6, *max delta step:* no restriction, *gamma:* no restriction, *min child weight:* 7, *number of parallel trees:* 4, *number of boosting iterations:* 1300.



(a) Reconstructed resolution.

(b) Gaussian fit of the reconstructed resolution mean. Sigmas are set as error bars.

Figure 11: Reconstruction performance of parameters, p(left), θ(middle), ϕ(right), using the Gradient Boosting Trees.

3.4. Software Implementation

To conduct this study, we implemented a framework that simplifies the process of training, validation, and prediction. Our software[3] is written in the Python programming language to accelerate the development process and make the software easily maintainable, reusable, and extensible. We use TensorFlow 2 [20] for the creation, training, and validation of the MLP machine learning model while scikit-learn [21] was used for the ERT model. It should be noted that Tensorflow 2 was selected to build the MLP model rather than scikit-learn due to the fact that Tensorflow's Keras API [22] gives more control over a model's architecture. More specifically, scikit-learn allows only one activation function to be utilized in an MLP model (across all layers). One cannot use a different activation function per layer (or modify the output activation function, as in this experiment). Tensorflow's Keras API gives this flexibility. For the gradient boosting decision trees model, XGBoost, we used the official open-source software [23].

To enable easy distribution of the software, we provide a YAML[24] file that contains the required dependencies that can be provided to software like Anaconda [25] to handle all

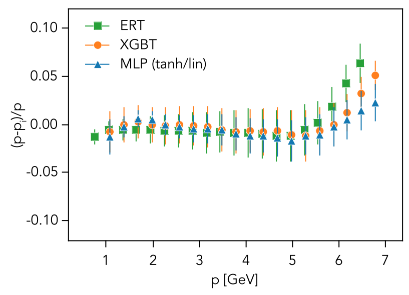
[3]Code and data available at: `https://github.com/gavalian/artin/tree/main/projects/pe/py`

software dependencies. Our software consists of two separate operations, `train` and `test`. Each operation is implemented as a subcommand that accepts its own parameters (similar to how git commands work: `git commit` ⟨`args`⟩, `git branch` ⟨`args`⟩, etc.).
The `train` subcommand, as shown from its name, is used to train a new model. It takes a set of required and optional arguments; the required ones include the dataset to use for training, the model architecture to be used (MLP, ERT, GBT), and the path to store the trained model. Optionally, the user can also set a separate validation file.

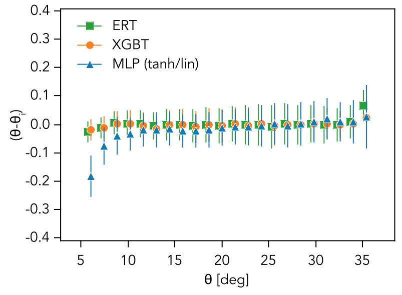
The `test` subcommand performs the respective evaluation operation; accepted arguments include the trained model to use, the dataset to test it on, and the directory to store the performance results (all required). When testing completes, it generates a report for the user with the performance results.

4. Results

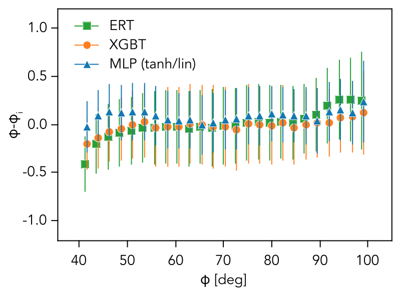
This section presents studies with several neural network architectures to reconstruct charged particle momentum and direction using raw data from CLAS12 drift chambers. The precision of reconstructed parameters is summarized in Figure 12. The Multi-Layer perceptron performs better in momentum reconstruction precision along the generated range, while ERT and XGBT underperform in the high momentum range. The direction of the particle (polar and azimuthal angle) is also reconstructed better with MLP, mostly flat across the entire range.



(a)



(b)



(c)

Figure 12: Machine Learning models performance comparison for Multi-Layer Perceptron (MLP), Extremely Randomized Trees (ERT), and Gradient Boosted Trees (XGBT) on testing dataset, with respect to (a) momentum, (b) polar angle and (c) azimuthal angle.

The inference times were measured for each of the networks (presented in Table 2). All models perform similarly with MLP being a little faster on inference times and capable of processing events with 22 kHz rate (calculated using 2.5 tracks on average in each physics event). When used with a track classifier network, accounting for the time spent on clustering the data and running classification on all possible combinations of track candidates, we estimated that the full track reconstruction can be achieved in real-time with a rate of 32 kHz, faster than data acquisition speed. The real-time track reconstruction rate is calculated by taking into account the clustering time from drift chamber reconstruction and track candidate classification time using the current implementation of track reconstruction using artificial intelligence [2].

Model	Training Time (ms)	Inference Time (ms)
Multi-Layer Perceptron	0.015	0.018
Extremely Randomized Trees	0.057	0.019
Gradient Boosted Trees	0.087	0.027

Table 2: Measured time for training and inference per sample for different network models.

The developed models were used to analyze simulated data and calculate missing mass distribution, which was compared with the results of conventional hit-based tracking. The comparison of track parameters, such as momentum, polar angle, and azimuthal angle, are shown in Figure 13 for simulated reaction $H(e^-, e^-\pi^+)X$ for both methods. The particle parameters distributions presented for both positively- (top row) and negatively- (bottom row) charged particles match those reconstructed by conventional hit-based tracking algorithms very well.

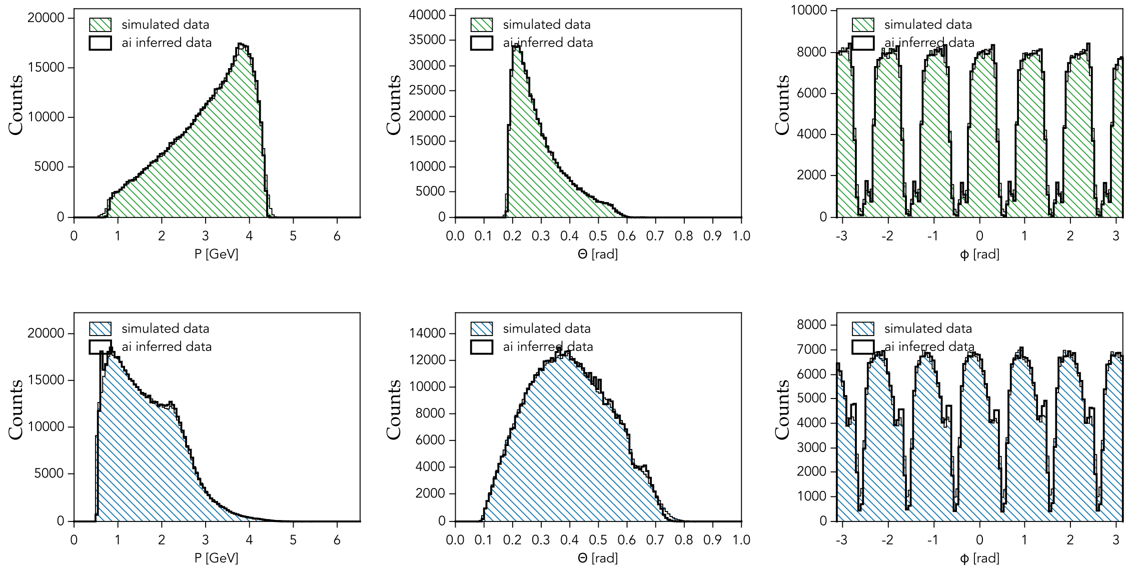
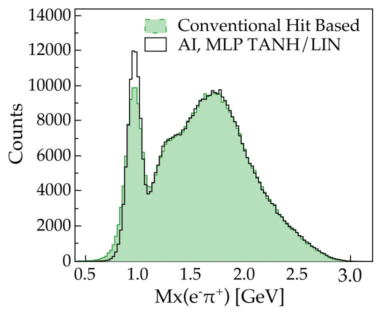
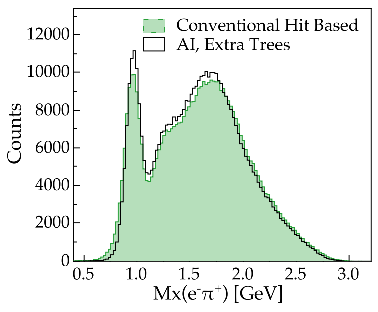


Figure 13: Particle parameters, i.e. momentum (left), polar (middle) and azimuthal (right) angle, for positive (bottom row) and negative (top row) compared for traditional Hit Based tracking and ML inferred tracks.

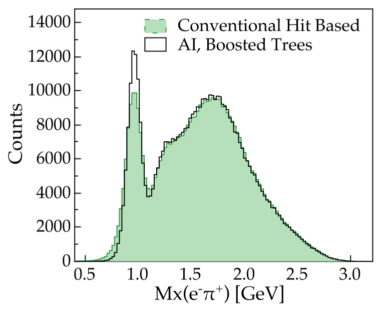
Further, the missing mass distributions for $H(e^-, e^-\pi^+)X$ are calculated for conventional and AI tracking, and results are presented in Figure 14. The filled histograms show distribution from the conventional algorithm while the solid line histograms are from AI track reconstruction. It is evident that in all three cases (for all networks), the AI inferred track parameters result in much better missing mass distribution resolution. The Gradient Boosting Trees network performs the best while MLP comes a close second. As stated previously, our goal is to reconstruct particle parameters and attain inference in the least amount of



(a) MLP



(b) ERT



(c) GBT

Figure 14: Missing mass distribution of $e^-\pi^+$, for generated $H(e^-, e^-\pi^+)X$ events, for different networks compared to the missing mass calculated using track parameters reconstructed by a conventional tracking algorithm at the hit based level.

time possible. While the MLP model performs faster in inference time, it would take a significantly longer training time in order for its inferred track parameters to provide a better missing mass distribution resolution. For example, the MLP model was also trained with 10,000 epochs rather than 1,500 and while the inference slightly improved, it still did not outperform the GBT model. Given the performance of the GBT model, we concluded that the trade-off between training time/hardware resources and equalized parameter estimation performance provided by the other models did not garner sufficient merit. Furthermore, gaining a slight improvement in inference does not warrant any extensive man labor required to fine tune each model to the highest possible optimizations (e.g. further testing different configurations of activation layers in the MLP model, number of neurons per layer, etc.). Highly accurate results are achieved in a short time using the model attributes selected. Overall, we conclude that the missing mass distribution calculated from the GBT model's track parameters yield the best results compared to the other models of our experiment.

5. Summary

In this work, we presented a machine learning approach to reconstructing charged track parameters in CLAS12 using only hit positions in drift chambers. The models were trained using simulated track parameters (i.e. momentum, polar and azimuthal angles). Multi-Layer Perceptron (MLP), Extremely Randomized Trees (ERT), and Gradient Boosting Trees (GBT) were used as test networks with GBT showing the best performance in track parameter inference. The tracks reconstructed by the ML models were used to calculate physics observables, namely missing mass of the $H(e^-, e^-\pi^+)X$ system, which was compared to missing mass calculated using tracks reconstructed by a conventional hit-based tracking algorithm.

It was shown that the reconstructed nucleon final state using ML inferred tracks has better resolution and the inference takes a fraction of the time (4 ms per event) compared to the conventional hit-based tracking algorithm (350 ms per event). The model achieves precision within 3-3.5 which is not enough to do physics analysis and to isolate rare physics reactions, however, it offers a plethora of other benefits.

This approach to data reconstruction has significant implications for detector monitoring when conducting nuclear physics experiments. A typical experiment in CLAS12 collects data at a rate of 11 kHz, and with a neural network capable of reconstructing real physics quantities in real-time (using a 48 core CPU), and allowing the monitoring of detector performance, this also opens a door to performing calibrations in real-time.

The reconstruction of physics observables in real-time has the potential to yield the following benefits:

- **Detector Monitoring:** Reconstructing tracks from separate sectors in the drift chamber can help to monitor the detector's health and make it easy to identify newly developed inefficient regions of tracking detectors.

- **Real-Time Detector Calibration:** The detector components can be initially calibrated during the experimental data collection without the need for extensive data processing after data taking for calibration purposes.

- **Improved Data Processing speed:** With initial track parameter reconstruction during data taking, the conventional algorithms can skip the process of hit-based tracking and rely on final, time-based tracking to improve the measured track parameters. This can significantly reduce data processing times since hit-based tracking currently comprises 35% of the total tracking time.

- **Physics Reaction Identification:** High-intensity experiments searching for rare reactions process a large amount of data to isolate required event topologies. Tagging the physics reactions in real-time during experimental data collection can significantly reduce data analysis and reaction search times.

These are only a few obvious applications where real-time track reconstruction can be used. The benefits of this kind of network in production mode can not be overstated.

6. Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Nuclear Physics under contract DE-AC05-06OR23177, and NSF grant no. CCF-1439079 and the Richard T. Cheng Endowment. This work was performed using the Turing and Wahab computing clusters at Old Dominion University.

References

[1] P. Thomadakis, A. Angelopoulos, G. Gavalian, and N. Chrisochoides, "Using machine learning for particle track identification in the clas12 detector," *Computer Physics Communications*, vol. 276, p. 108360, 2022.

[2] G. Gavalian, P. Thomadakis, A. Angelopoulos, N. Chrisochoides, R. De Vita, and V. Ziegler, "CLAS12 track reconstruction with artificial intelligence." `https://arxiv.org/abs/2202.06869`, 2022. arXiv:2202.06869.

[3] P. Thomadakis, A. Angelopoulos, G. Gavalian, and N. Chrisochoides, "De-noising drift chambers in clas12 using convolutional auto encoders," *Computer Physics Communications*, vol. 271, p. 108201, 2022.

[4] G. Gavalian, P. Thomadakis, A. Angelopoulos, and N. Chrisochoides, "Convolutional auto-encoders for drift chamber data de-noising for CLAS12." `https://arxiv.org/abs/2205.02616`, 2022. arXiv:2205.02616.

[5] V. Burkert *et al.*, "The CLAS12 Spectrometer at Jefferson Laboratory," *Nucl. Instrum. Meth. A*, vol. 959, p. 163419, 2020.

[6] M. Mestayer *et al.*, "The CLAS12 drift chamber system," *Nucl. Instrum. Meth. A*, vol. 959, p. 163518, 2020.

[7] S. Agostinelli *et al.*, "Geant4—a simulation toolkit," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 506, no. 3, pp. 250–303, 2003.

[8] M. Ungaro, "Geant based monte-carlo." `https://gemc.jlab.org/gemc/`, 2020.

[9] M. Khachatryan, A. Papadopoulou, A. Ashkenazi, F. Hauenstein, A. Nambrath, A. Hrnjic, L. Weinstein, O. Hen, E. Piasetzky, M. Betancourt, *et al.*, "Electron-beam energy reconstruction for neutrino oscillation measurements," *Nature*, vol. 599, no. 7886, pp. 565–570, 2021.

[10] P. Chatagnon *et al.*, "First measurement of timelike compton scattering," *Phys. Rev. Lett.*, vol. 127, p. 262501, Dec 2021.

[11] A. E. Alaoui, N. Baltzell, and K. Hafidi, "A rich detector for clas12 spectrometer," *Physics Procedia*, vol. 37, pp. 773–780, 2012. Proceedings of the 2nd International Conference on Technology and Instrumentation in Particle Physics (TIPP 2011).

[12] V. Ziegler *et al.*, "The CLAS12 software framework and event reconstruction," *Nucl. Instrum. Meth. A*, vol. 959, p. 163472, 2020.

[13] S. Haykin, *Neural networks: a comprehensive foundation.* Prentice Hall PTR, 1994.

[14] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *International Conference on Learning Representations*, 12 2014.

[15] S. J. Reddi, S. Kale, and S. Kumar, "On the convergence of adam and beyond." `https://arxiv.org/abs/1904.09237`, 2019. arXiv:1904.09237.

[16] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Mach. Learn.*, vol. 63, p. 3–42, Apr. 2006.

[17] J. R. Quinlan, "Induction of decision trees," *Mach. Learn.*, vol. 1, p. 81–106, Mar. 1986.

[18] T. K. Ho, "Random decision forests," in *Proceedings of 3rd international conference on document analysis and recognition*, vol. 1, pp. 278–282, IEEE, 1995.

[19] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, (New York, NY, USA), pp. 785–794, ACM, 2016.

[20] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283, 2016.

[21] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, *et al.*, "scikit-learn: Machine learning in python," *Journal of machine learning research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[22] F. Chollet *et al.*, "Keras." `https://keras.io`, 2015.

[23] T. Chen, "XGBoost: Scalable, portable and distributed gradient boosting." `https://github.com/dmlc/xgboost`. Online; Accessed: 2022-05-09.

[24] "The Official YAML Web Site." `https://yaml.org/`. Online; accessed 2021-04-29.

[25] "Anaconda Software Distribution." `https://docs.anaconda.com/`. Online; accessed 2021-04-29.