# De-Noising Drift Chambers in CLAS12 using Convolutional Auto Encoders

Polykarpos Thomadakis[a,1,2], Angelos Angelopoulos[a,1], Gagik Gavalian[b,1], Nikos Chrisochoides[a]

[a] *CRTC, Department of Computer Science, Old Dominion University, Norfolk, VA, USA*

[b] *Jefferson Lab, Newport News, VA, USA*

**Abstract**

Modern Nuclear Physics experimental setups run experiments with higher beam intensity resulting in increased noise in detector components used for particle track reconstruction. Increased uncorrelated signals (noise) result in decreased particle reconstruction efficiency. In this paper, we investigate the usage of Machine Learning, specifically Convolutional Neural Network Auto-Encoders (CAE), for de-noising raw hits from drift chambers in the CLAS12 detector. To the best of our knowledge, this is the first time CAE is employed to perform such an operation in this field.

During the de-noising phase, it is important to remove as much noise as possible while retaining the valid hits to avoid losing crucial information about the experiment. We show that using CAE, it is possible to remove noise hits while retaining up to 94% of valid tracks for a beam current of 110 $nA$ while for lower beam currents ($45 - 55$ $nA$), we get up to 98% efficiency. Studies on experimental conditions with increasing noise show that CAE performs better than conventional tracking algorithms in isolating hits belonging to tracks. Specifically, the de-noising algorithm results in tracking efficiency improvements greater than 15%, in real data production procedures with nominal conditions, and up to two times better efficiency in synthetically generated data with high luminosity conditions ($90 - 110$ $nA$), indicating that machine learning can lead to significantly shorter times for conducting physics experiments.



[1] Authors contributed equally.

[2] Correspoding author, *pthom001.odu.edu*

## 1. Introduction

With the evolution in detector technologies and electronic components used in the Nuclear Physics field, experimental setups become larger and more complex. Faster electronics enable experiments to run with higher beam intensity, providing more interactions per time and more particles per interaction. Modern Nuclear Physics experiments include many interconnected detector systems that detect results of interacting particles to analyze and test physics models. The increased beam intensities present a challenge to the particle detectors because of the higher amount of noise and uncorrelated signals. Higher noise levels lead to a more challenging particle reconstruction process by increasing the number of combinatorics to analyze and background signals to eliminate before detector data processing. On the other hand, increasing the beam intensity in an experiment can provide physics outcomes faster but only if combined with a highly efficient track reconstruction process. Thus, a method that provides efficient tracking under high luminosity conditions can significantly reduce the amount of time required to conduct physics experiments.

The recent developments in Artificial Intelligence (AI) field provide tools that can be used in physics data processing in the place of conventional procedural algorithms to improve data processing accuracy and speed. In this article, we developed AI methods (neural networks) for de-noising data from particle tracking detectors to improve track reconstruction efficiency. This development was targeted at CLAS12 [1] Drift Chambers (DC) detector ([2]) at Jefferson Lab, Virginia. The developed neural network was used to pre-process data for standard experimental running conditions and showed significant improvements in track reconstruction efficiency ($> 15\%$). The studies were extended to synthetically generated data emulating much higher beam intensity running conditions to investigate how neural networks can perform on future experiments. Results show that in high beam intensity experimental conditions AI-assisted methods outperform conventional algorithms in removing uncorrelated signals (noise) and provide a significant increase in track reconstruction efficiency of up to 80%. In summary, we show that (i) CAEs can be an efficient method to remove excess noise hits from raw data with very low risk of losing valid hits in the process, (ii) they provide good tracking efficiency in high beam conditions which can be improved marginally with only a small penalty in their noise removing capabilities, (iii) CAEs can perform substantially better than conventional algorithms with an increasing difference in performance as the beam current increases. Moreover, as a by-product of this effort, we have developed a software tool that can easily be reused for similar studies on other detectors and different computing platforms.

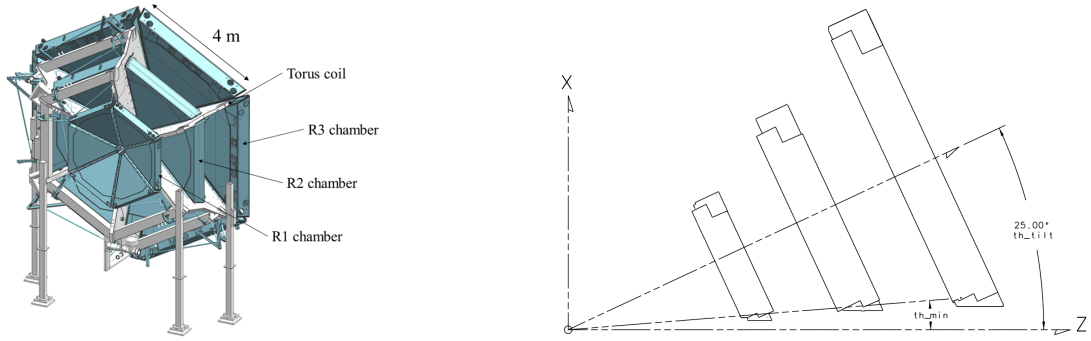
The rest of the paper is organized as follows: Section 2 presents some background for this work, providing more details about the CLAS12 detector, and the drift chambers, and briefly introduces how Convolutional Neural Networks work. Section 3 presents a brief introduction to auto-encoder neural networks and related work. Section 4 presents the proposed method, different CAE architectures that we experimented with for this study, and how the input and output datasets are processed. Section 5 describes the portable and easy-to-use software we developed to perform the experiments for the different architectures and parameters presented in this paper. Section 6 contains the performance evaluation of all tested neural

network architectures on nominal conditions (45 nA of beam current) for both track reconstruction and noise removal efficiency. In section 7, we conduct a study about the effect of increased beam current (up to 110nA) on the performance of the proposed method. Section 8 shows a performance comparison between our proposed method and the conventional algorithm on nominal and high luminosity conditions. Finally, section 9 initiates a discussion about the significance of our results and section 10 concludes this paper.

## 2. Background

### *2.1. CLAS12 Detector*

The CEBAF Large Acceptance Spectrometer at 12 GeV (CLAS12 [1]) is located at Hall B, one of the experimental halls at the Jefferson Lab in Newport News, VA, serving a variety of physics experiments with different running conditions.




(a) (b)

Figure 1: View of CLAS12 detector showing Drift Chambers and Toroidal Magnet (on the left). Side view of one of the sectors of Drift Chambers (on the right), consisting of three "regions" and covering 60° azimuthal range.

The forward part of CLAS12 is built around a superconducting toroidal magnet (Figure 1a). The six coils of the toroid divide the detector azimuthally into six sectors. Each sector contains three multi-layer drift chambers [2] (DC) for reconstructing the trajectories of charged particles originating from a fixed target. Figure 1b depicts the cross-section of one of the sectors of drift chambers covering an azimuthal angular range of 60°. One sector is composed of three drift chambers (called "regions"), each consisting of two sections (called "superlayers").

Particles originating from the target within polar angular range from 5° to 40° leave signals on all six super-layers and can then be reconstructed by a tracking algorithm. The track reconstruction algorithm for CLAS12 [3] works in two stages, hit-based tracking, and time-based tracking. In the hit-based tracking stage, uncorrelated noise hits in Drift Chambers are identified and removed by the Simple Noise Removal (SNR) algorithm. The remaining hits are then grouped into clusters within the wire layers of a given DC super-layer. To reduce tracking inefficiencies attributed to wire malfunctions or intrinsic inefficiencies it is acceptable for two layers to be missing within a super-layer when forming a cluster; the remaining wire layers are sufficient to determine the cluster's shape and find the track trajectory. After

identifying clusters in each super-layer and forming track candidates with combinations of all of them (one from each super-layer), an initial fit to the track candidates is performed to determine if the cluster combination forms a "good" track. Tracks identified as good continue to the next stage (called time-based tracking), where timing information from hits is applied in order to refine track fitting.

### *2.1.1. Drift Chambers*

The Drift Chambers in CLAS12 are used for tracking charged particles. Each sector is composed of three drift chambers (called "regions") with each region consisting of two sections (called "super-layers"). Each super-layer has six layers of wires (12 wire planes in each "region") with those residing on the two adjacent super-layers oriented at $\pm 6^{\circ}$ stereo angles. Each layer of wires has 112 hexagonal cells spanning a range from about $5^{\circ}$ to $40^{\circ}$ in polar angle.

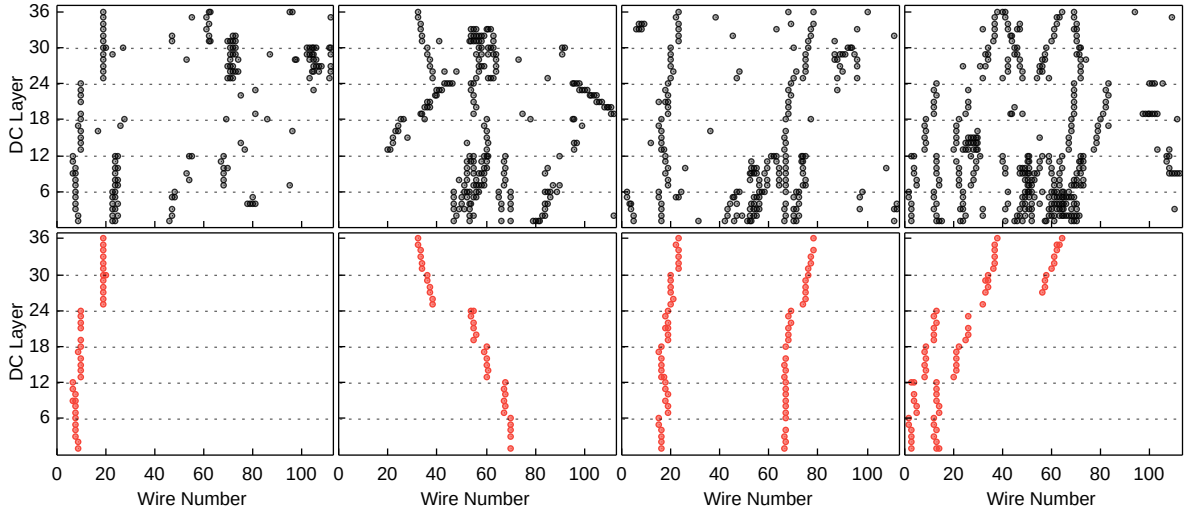



Figure 2: Example events from drift chambers. Points represent hits in drift chambers with wire number (112 wires per layer) on the x-axis and layer number (total of 36) on the y-axis.The gray point are all the hit wires in the event, and red point hits that are associated with reconstructed tracks.

Figure 2 presents some example events shown for one sector at a time. The top row depicts raw hits detected on a sector while the bottom depicts only those belonging to identified tracks. Hits present in the top row but missing on the bottom comprise the noise hits of the event. Particles passing through each super-layer of chambers leave signal in one or two wires in each layer. Wires with signals on each super-layer close to each other are grouped into segments. A track candidate is formed from 6 segments (one per super-layer) in each sector and then further validated by fitting. The efficiency of track reconstruction relies on cleanly identifying segments in each super-layer. With increased noise which arises from running with high beam intensity, removal of noise hits with conventional algorithms becomes less efficient, and with loss of segment the tracking efficiency suffers. With this work we will study how Convolutional Auto-Encoders can improve the segment finding algorithm by removing uncorrelated hits from noisy raw data.

*2.2. Convolutional Neural Networks*

Convolutional Neural Networks (CNNs) are a type of neural networks known to perform better on data where spatial locality is important (e.g. images). A common CNN consists of Convolutional Layers, Pooling Layers and a Fully Connected Layer. Convolutional layers are the most important layers in a CNN, consisting of an input, a number of kernels and an output (aka feature map).

- The input to a CNN layer is an N-dimensional array (for example, a colored image with height, width and rgb values) given as an input to the network or produced by a previous layer.
- A kernel is a 2-dimensional array of weights, usually 3x3, that is used to apply convolution on the input. The kernel is shifted on the input, based on a stride, and on each shift a dot product is applied between the respective area of the input and the kernel to produce a single element of the output (also known as feature map). The process continues until the whole input has been processed and the whole matrix of the feature map has been produced. The weights of the kernel are the parameters learned by the neural network and are adjusted between iterations through the process of backpropagation and gradient decent.
- The result produced by applying convolution on the input using the provided kernel is the output (feature map) of the layer. An activation function is finally applied on the output array (e.g. ReLU [4]) to introduce non-linearity in the model.

The size of the output is affected by 3 parameters: the *number of kernels*, the *stride* used during convolution and the type of *padding*. The *number of kernels* determines the depth of the output, e.g. n kernels would generate n outputs, thus an output of depth n. By increasing the depth of the output we can extract more features through the extra number of weights to be trained. As mentioned before, the *stride* determines the step that the kernel moves over the input to apply dot product. A stride larger than one would generate an output of size smaller than the input. *Padding* can be either valid or same. A valid padding adds no padding to the input which will result in an output of smaller size for kernels greater than 1x1. This occurs because the kernel will be shifted on the input and apply dot-product fewer times than the size of the input. In other words, when a row or column of the kernel exceeds either side of the input the dot-product will be aborted. In same padding the input is padded by zeros to ensure that the output will be of the same size as the input.

Examples of the convolution process between a 5x5 array and 3x3 kernel(s) for valid and same padding and strides of 1x1 and 2x2 are shown in figure 3. When valid padding is used the kernel can only be applied 3 times horizontally with a stride of 1x1 before moving to the next row. This process is repeated until no more steps are left to be taken both horizontally and vertically for a total of 9 applications of the dot-product (3x3 output). When using a stride of 2x2, the kernel is shifted by 2 elements at a time horizontally and 2 elements vertically when a row is completed which allows the application of only 4 dot-products in total (2x2 output). The exact same process is followed when same padding is applied with the only difference being the introduction of more elements on the edges of the input array to ensure that the output of the convolution (assuming stride 1) will be of the same size as

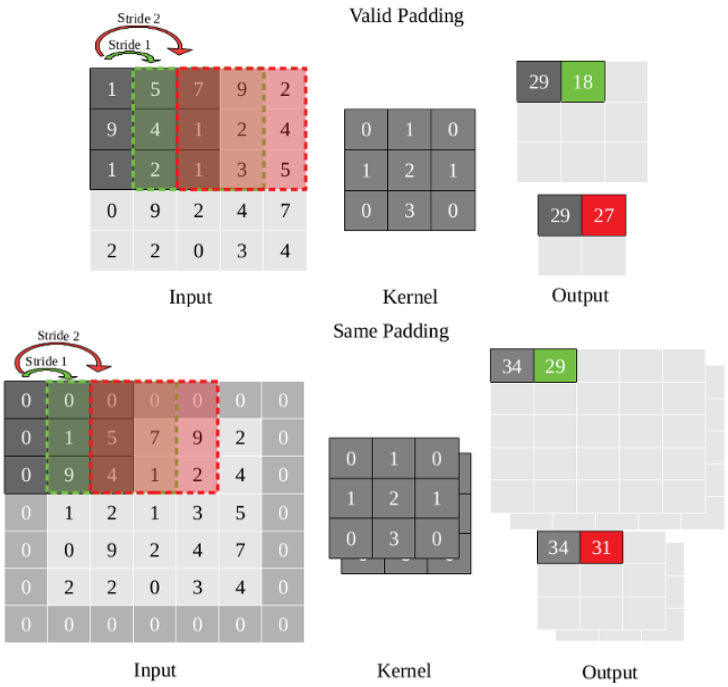



Figure 3: Examples of convolution between a 5x5 array and one (top) or two (bottom) 3x3 kernel(s). The figures show the convolution process with a stride of 1x1 (green square area) and a stride of 2x2 (red square area). The resulting outputs are generated by applying dot product between the kernel and the respective are of the input, starting from the gray area and shifting the kernel based on the stride ( the output for stride 1x1 is generated by shifting the kernel one step at a time, like the green area, while for stride 2x2 by shifting it two steps at a time, like the red square area). The top figure represents the sizes of the outputs when using valid padding while the bottom shows the output for same padding. The sizes of the outputs are affected by both the stride and the padding because those affect how many times the kernel can shift on the input and apply the dot-product.

the input. In both cases the number of kernels equals the depth of the output.

Pooling layers perform downsampling by sweeping a 2-dimensional filter over their input and applying some aggregation function to generate the output. The process is very similar to convolution with the difference being that instead of a dot-product an aggregation function is applied (see fig. 4 for an example). The aggregation can either be the maximum or the average value of values in the area under the filter , hence the names max pooling or average pooling respectively. This process is used to reduce the amount of parameters to be learned as well as to prevent overfitting.

The fully connected layer is a feed-forward network residing at the end of the CNN.

## 3. Related Work

To the best of our knowledge, there is no prior work that has utilized AI to denoise data from hits coming from drift chambers. However, there is prior work on other fields that have used auto-encoders for different applications, including image denoising which inspired our work. Auto-encoders[5] are a type of Artificial Neural Network employed to learn efficient

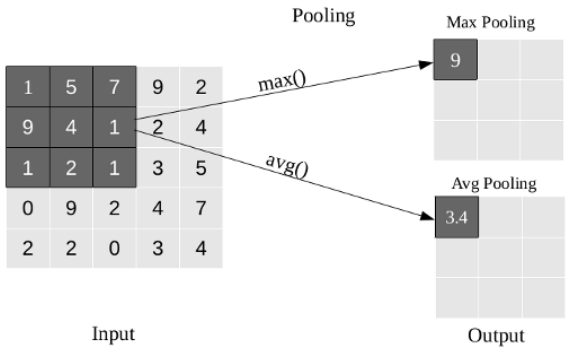



Figure 4: Example of the pooling operations on a 5x5 array using a 3x3 pooling array. The 3x3 array is shifted up the input, just like in convolution, but instead of dot-product an aggregation function is applied. Since the pooling array is of size larger than 1x1 the resulting array size is smaller than the input. Based on the type of aggregation used, the pooling is called max pooling or average pooling respectively.

data encodings. In the simplest case, an auto-encoder is trained to recreate input data to its output. This process is performed in order to extract a representation of the data (*code*), in an unsupervised manner. This representation can then be used to recreate the input on the output, using the decoder.

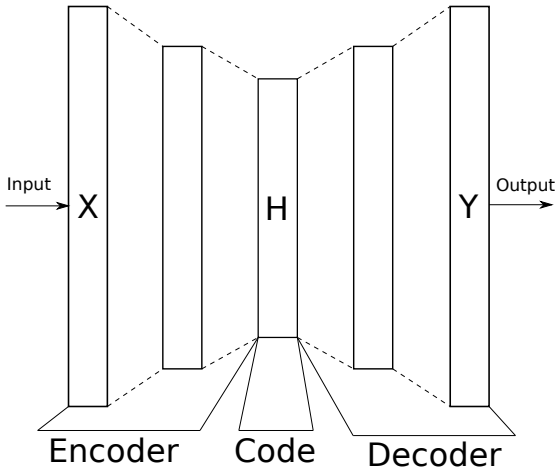



Figure 5: Schematic representation of an encoder with 3 hidden layers.

Auto-encoders consist of three components (see Fig. 5):

1. The encoder that encodes the input data X
2. The decoder that reconstructs the encoded data in the output Y
3. The hidden layer H that learns the encoding of the input data as defined by the encoder and is used by the decoder to reconstruct the input

The process followed by an auto-encoder is defined as the following transition:

$$H = enc(X)$$

$$Y = dec(enc(X))$$

$$Y = dec(H)$$

where *enc* and *dec* are the functions applied by the encoder and the decoder, respectively.

Using an auto-encoder where the hidden layer is of the same or larger size as the input and output layers could cause the network to over-fit the data and learn the identity function. To prevent such a case an auto-encoder needs to be augmented with some technique to avoid over-fitting (i.e. avoid losing generalizability). The applied techniques need to help the network find the optimum in the trade-off between bias and variance. In other words, the auto-encoder needs to be capable of reconstructing each specific input efficiently, while also generalizing well enough to represent the key characteristics of the data as a whole. Some of the techniques that can be employed to prevent over-fitting while allowing the model to generalize are:

1. Employing regularization *(Regularized Auto-encoders)*
2. Using a network architecture that creates a hidden layer of lower dimensions compared to the input and output *(Under-complete Auto-encoders)*

Regularization in the context of auto-encoders has been implemented in different ways. Some of the regularized auto-encoders that have been employed extensively in the past include:

1. *Sparse* auto-encoders[6], where an additional sparsity penalty is imposed, forcing the network to deactivate several neurons of the hidden layer based on the input data. As a result, the active code generated is still lower in dimension even though its number of neurons are equal to or larger than those of the input/output.
2. *Contractive* auto-encoders[7], where the goal is to make the reconstruction process less sensitive to small variations in the data. This is achieved by imposing a penalty that helps to carve a representation that better captures the local directions of variations dictated by the data, corresponding to a lower-dimensional non-linear manifold, while being more invariant to the vast majority of directions orthogonal to the manifold.
3. *De-noising* auto-encoders[8], where the network is given a corrupted input and is trained to reconstruct the uncorrupted input. During this process, the network learns the important aspects of the data while filtering out the noise.
4. *Variational* auto-encoders[9] are different from other auto-encoders, providing a statistical manner to describe the samples of the data-set in latent space. Therefore, in a variational auto-encoder, the encoder outputs a probability distribution in the code instead of a single value.

For the under-complete auto-encoders, the network encodes data in a lower dimension by transitioning from layers with a high to a low number of neurons. The reverse sequence of layers is then applied to recreate the input layer on the output. Throughout this sequence of layers, the model has to maintain its ability to reconstruct an accurate copy of the input. To achieve that, it has to learn and store information that better characterizes the data in a smaller set of neurons. Thus, the code learned in this process comprises a compressed representation of the data.

In the special case that an under-complete auto-encoder consists of a single hidden layer and only linear activation functions, it produces the same representations as Principal Component Analysis[10]. In other words, an auto-encoder generalizes PCA to non-linear data, one of its first applications in the literature. Other applications where auto-encoders have been applied extensively are Dimensionality Reduction[11], Information Retrieval[12], Anomaly Detection[13] and Image Processing[8].

Auto-encoders can either be implemented as Multi-Layer Perceptrons (MLP) [14], by setting the number of neurons per layer appropriately or as convolutional neural networks (CNN) [15]. An under-complete CNN auto-encoder is very similar to an under-complete MLP auto-encoder. Its hidden layers' dimensionality gradually decreases to a point where the code is generated. Then the dimensionality gradually increases until it matches that of the input. However, in CNN's the dimensionality of each hidden layer is not explicitly set (like the number of neurons of each layer in MLPs), instead, it is derived by the sizes, padding, and strides of the kernels as well as the pooling layers (if any).

## 4. Method

### *4.1. Data Representation*

Before we can experiment with machine learning for our needs, we have to define a representation for the detector data. As described in Section 2, each sector of the CLAS12 detector has three regions, each with two super-layers that consist of six layers of 112 hexagonal cells, for a total of 4032 ($3 \times 2 \times 6 \times 112$) cells. Each cell reports a value of 1 if a hit was detected or 0 otherwise. For our initial attempt with Multilayer Perceptron Autoencoders, each data point consisted of these 4032 values as features, without any pre-processing. For the CAE network, we reshaped the data point features into a $36 \times 112$ structure that better represents the spatial locality of the cells. This representation casts our problem to an image denoising application where convolutional autoencoders have been shown to perform well [8]. Specifically, the data generated by the detector were treated as a 2D black and white image, where a black pixel (value 1) indicates a sensor activation and a white pixel (value 0) implies no sensor activation. For training the neural network we used experimental data. The input to the network was comprised of an image that contains all hits in one sector and as an output image created from hits that belong to reconstructed track by conventional tracking algorithms. In the training and validation sample there were mixture of events of one, two or three tracks in the output sample.

### *4.2. Neural Network Architectures*

Several different CAE architectures were considered as shown in Table 1, all of them under-complete. The set of models presented in this table are variations of models we found to be working relatively well for our problem after a more wide experimentation with different neural network architectures. The models exhibit variations in the number of layers, the type of pooling layers as well as the approach to modify the dimensionality of the hidden layers (e.g. through kernel strides versus pooling).

Model 0 is a regular CNN autoencoder. The encoder of model 0 consists of two convolutional layers, each followed by a max-pooling layer. The decoder, which follows just after the

max-pooling layer, consists of two convolutional layers followed by two upsampling layers (upsampling layers perform the reverse operation of pooling layers, they increase the size of their input by duplicating rows/columns). The shrinking in dimensions, in this case, is performed by the max-pooling layers, while the convolution kernels are using same padding to keep the dimensionality of their outputs intact. Models 0a - 0f follow similar architectures with variations, including stacked convolutional layers (0a-0f) instead of single (i.e. two back-to-back convolutional layers without any pooling layer between them), average pooling instead of max pooling (0c), and different numbers of layers (0e). Models 1 & 2 decrease the dimensions of the hidden layers by using only the convolutional layers. They achieve that by employing strides with a dimensionality larger than 1x1. The upscaling (increase of the size of the input) that was performed by the upsampling layers in the previous models is achived here by deconvolution layers (deconvolution kernels perform a transpose convolution operation, the specifics about this operation is outside the scope of this work).

| Model | Architecture |
|---|---|
| 0 | C48(4x6) ; MP(2,2) ; C48(4,6) ; MP(2,2) ; C48(4,6) ; US(2,2) ; C48(4,6) ;US(2,2) ; C1(4x6) |
| 0a | C48(5x4) ; MP(2,2) ; 2*C48(4x3) ; MP(3,2) ; 2*C48(4,3) ; US(3,2) ; 2*C48(5x4) ; US(2,2) ; C1(5x4) |
| 0b | 2*C54(4x3) ; MP(2,2) ; 2*C54(4x3) ; MP(3,2) ; 2*C54(4x3) ; US(3,2) ; 2*C54(4,3) ; US(2,2) ; C1(4x3) |
| 0c | C48(5x4) ; AP(2,2) ; C48(4x3) ; AP(3,2) ; C48(4,3) ; US(3,2) ; C48(5x4) ; US(2,2) ; C1(5x4) |
| 0d | 2*C54(4x3) ; MP(2,2) ; 2*C54(4x3) ; MP(3,2) ; 2*C54(4x3) ; US(3,2) ; 2*C54(4,3) ; US(2,2) ; 2*C54(4x3) ; C1(4x3) |
| 0e | 2*C128(4x3) ; MP(3,2) ; 2*C128(4x3) ; US(3,2) ; 2*C128(4x3) ; C1(4x3) |
| 0f | 2*C28(4x3) ; MP(2,2) ; 2*C28(3x3) ; MP(3,2) ; 2*C28(3x3) ; US(3,2) ; 2*C28(3x3) ; US(2,2) ; 2*C28(4x3) ; C1(4x3) |
| 1 | C64(k:6x6,s:6x1) ; C64(k:2x2,s:2x1) ; DC64(k:2x3,s:2x1) ; DC1(k:6x6,s:6x1) |
| 2 | C64(k:3x3,s:3x1) ; C64(k:2x2,s:2x1) ; MP(1,2) ; C64(k:2x2) ; US(1,2) ; DC64(k:2x2,s:2x1) ; DC1(k:3x3,s:3x1) |

Table 1: Architectures of the models. "C#" stands for "Convolution 2D with # feature maps", "DC#" stands for "Deconvolution 2D with # feature maps", "MP" stands for "Max Pooling 2D", "AP" stands for "Average Pooling 2D", and "US" stands for "Up-Sampling 2D". For convolutions with stride (models 1,2), "k" stands for kernel size and "s" for stride size.

All models presented use Rectified Linear Unit [4] activation function for all layers except the last layer where Sigmoid[16] activation is used. The sigmoid layer in the end restricts the network outputs in the range of [0,1]. In order to assign a pixel to a class of black or white we initially set a cutoff threshold at 0.50. In section 7.2 we experiment with different values for this threshold and present its impact on our results. We use Nesterov momentum Adam (Nadam) [17] as the optimizer and binary cross-entropy as the loss function for all models.

The architecture of one of the models (0b) is shown in Figure 6. The input to the network is a single image of dimension 36x112. The model then creates feature maps of dimensions 36x112, 18x56, and finally encoding the input in feature maps of dimension 6x28. This down-sampling process is then reversed to produce an output of dimension 36x112 that represents the input data without the noise. Two consecutive convolution layers with 4x3 kernels are used before max pooling and up-sampling layers.

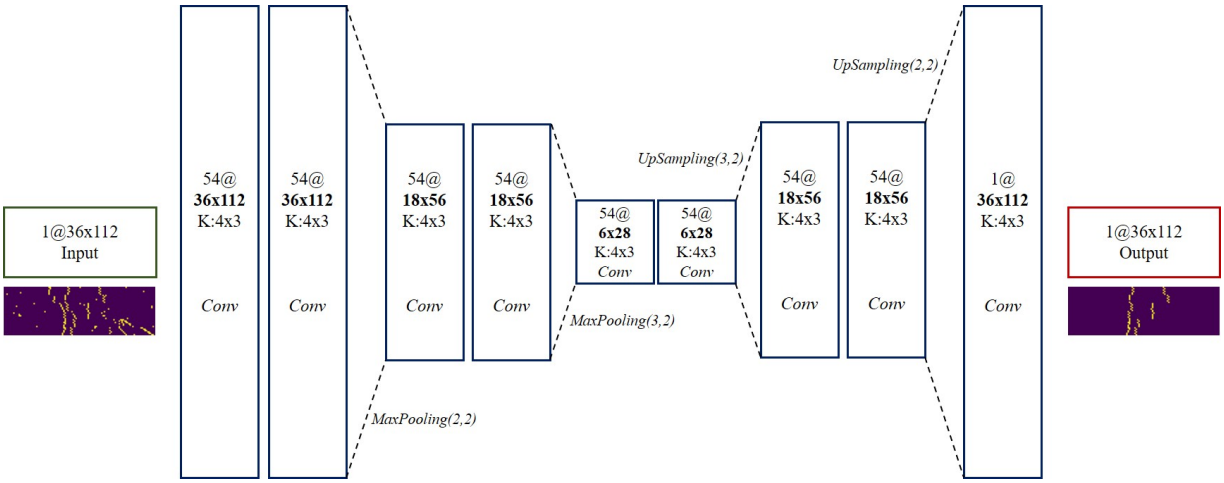



Figure 6: Schematic representation of model 0b from Table 1

### *4.3. Data Post-processing*

The de-noising operation may remove some valid hits, misidentifying them as noise, which can cause small gaps in the particle tracks. To mitigate this issue, a simple hit reconstruction algorithm can be employed. The algorithm iterates each hit present in the de-noised data and marks any hits within a radius of one sensor around it in the raw(noisy) data. All marked hits are then added to the de-noised data. This process recovers many of the missing hits but may also introduce additional noise. However, the improved efficiency it generates by far outweighs the impact of the introduced noise. We plan to introduce such a step in the future, however, in this work all results presented do not include it. The reason for this is that timing cuts are necessary in order to reinstate a hit, and at the hit based level here we do not have access to time of the hit.

## 5. 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 Python programming language to accelerate the development process and make the software easily maintainable, reusable, and extensible. We use TensorFlow [18] with keras[19] as a frontend for the creation, training, and validation of machine learning models. By employing TensorFlow, our machine learning operations are optimized to run efficiently on both CPUs and GPUs, achieving high performance. Since our datasets are sparse, we use the svmlight data format [20] which is designed for labeled data with sparse features. To load and store datasets we make use of the scikit-learn [21] library. scikit-learn is also utilized to generate validation datasets by randomly picking data entries from the training dataset. To enable easy distribution of the software, we provide a YAML[22] file that contains the required dependencies that can be provided to software like Anaconda [23] to handle all software dependencies.

[3]Code & data as well as a more detailed guide available at: `https://github.com/gavalian/clas12ai/tree/master/path_denoise_2d/`

The software consists of three separate operations, `train`, `test` and `predict`. Each operations 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 (explained later), and the directory where the trained model shall be stored. In addition to the trained model, the `train` subcommand will also generate a graph presenting the training and validation error for each training epoch, which allows the user to quickly check if the created model is under/overfitting the data. Optionally, the user can also set the number of training epochs to go through (default: 10), the batch size (default: 16) and a separate validation file (default: 10% of the training set).
The `test` subcommand performs the respective testing operation; Accepted arguments include the trained model to use, the dataset to test it on, the directory to store the performance results (all required), and the batch size (optional). When testing completes, it generates a report for the user with the accuracy results, the percentage of noise removed, etc., as well as the histograms presented later in the paper. It also prints random examples comparing the raw input (noisy data), the clean data (ground truth), and the reconstructed input (denoised data). The results are both printed on the screen and stored in a CSV-format file to allow for further investigation into the data.
The `predict` subcommand is what will be used once the performance evaluation has been completed and the model architecture and the related parameters have been chosen. It receives the same arguments as `test` and outputs a new svmlight-format file that contains each of the original raw(noisy) data followed by the reconstructed(denoised) data generated by them.

Extending the framework with new models is simple. A new model architecture can be implemented by subclassing a built-in abstract class and implement two methods; namely the `build()`, and `preprocess()` methods. The `build()` method is where the architecture of the new model is described using Keras notations, compiled, and returned to the framework. The `preprocess()` method is provided so that the user can perform any processing that might be needed for the specific model, application, and dataset. For example, to test CNN auto-encoders the input needs first to be transformed to a 36x112x1 tensor to fit CNN's expected input format ($width \times height \times channels$).

## 6. Model Comparison Results

### *6.1. Experimental Setup*

For all the experiments presented in the following sections we used a single NVIDIA Tesla V100 GPU, Tensorflow 2.0 and CUDA 10.2.

### *6.2. Results*

In this section we present the training and testing process as well as performance results of the machine learning models described in section 4 . For each model presented in Table 1 we experimented using real data generated by the CLAS12 drift chambers running on nominal conditions (i.e. 45nA of beam current). All models were trained on the same

dataset of 60000 samples, consisting of events with single or multiple tracks. Training data used is from experimental reconstructed data, where input sample is simply an image of all hits in the drift chamber sector, and output image is all hits belonging to track (or tracks) that were reconstructed by conventional tracking algorithm (after de-noising the hits with standard SNR algorithm). Testing results presented are gathered by running the generated models on a different dataset of 60000 samples.

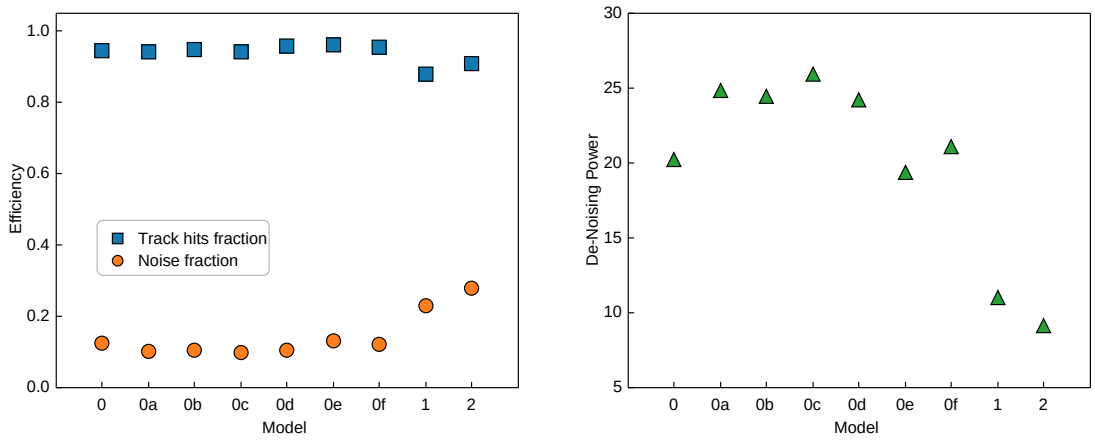



Figure 7: Model efficiency comparison. The efficiency of track trajectory hits reconstruction (blue squares) and remaining noise fraction after de-noising (orange circles) are plotted as a function of model (left panel). The noise reduction power (green triangle), defined as fraction of noise hits in the raw sample divided by the fraction of noise hits in the reconstructed image (right panel).

Figure 7 shows the evaluation results for all 9 models. The distribution of hit reconstruction efficiency (blue rectangles) presents the fraction of valid hits that were retained in the inferred image after applying the de-noising CAE over those belonging to a track in the raw input (i.e. shows the effectiveness of each model in retaining hits belonging to a track). The noise level (orange circles, left panel) is the hits in the reconstructed image that do not belong to a track as a fraction of the hits belonging to a track. In the original data sample, the average noise level is 250%. By applying the de-noiser, the noise level was reduced to 10%-27%,10 to 25 times less noise. Table 2 presents more detailed performance results of the models with respect to noise removal. Based on these results, we define the noise reduction power metric as the ratio of noise before and after de-noising is applied. The de-noising power as a function of the model can be seen in Figure 7 right panel (green triangles).

| Metric (%) | 0 | 0a | 0b | 0c | 0d | 0e | 0f | 1 | 2 |
|---|---|---|---|---|---|---|---|---|---|
| Noise Mean Before De-noising | 252 | 252 | 252 | 252 | 252 | 252 | 252 | 252 | 252 |
| Noise RMS Before De-noising | 163 | 163 | 163 | 163 | 163 | 163 | 163 | 163 | 163 |
| Noise Mean After De-noising | 12 | 10.1 | 10.3 | 9.9 | 10.4 | 13 | 12 | 23 | 27.5 |
| Noise RMS After De-noising | 24 | 22 | 21 | 21 | 22 | 25 | 23 | 30 | 32 |
| Noise Removed After De-noising | 95 | 96 | 96 | 96.1 | 95.7 | 94.7 | 95.2 | 90.9 | 89 |

Table 2: Evaluation results for all models with respect to denoise efficiency. Noise level in the first four rows is the hits in the de-noised image that do not belong to a track as a fraction of the hits belonging to a track. The last row presents the percentage of noise that was removed by the autoencoder models.

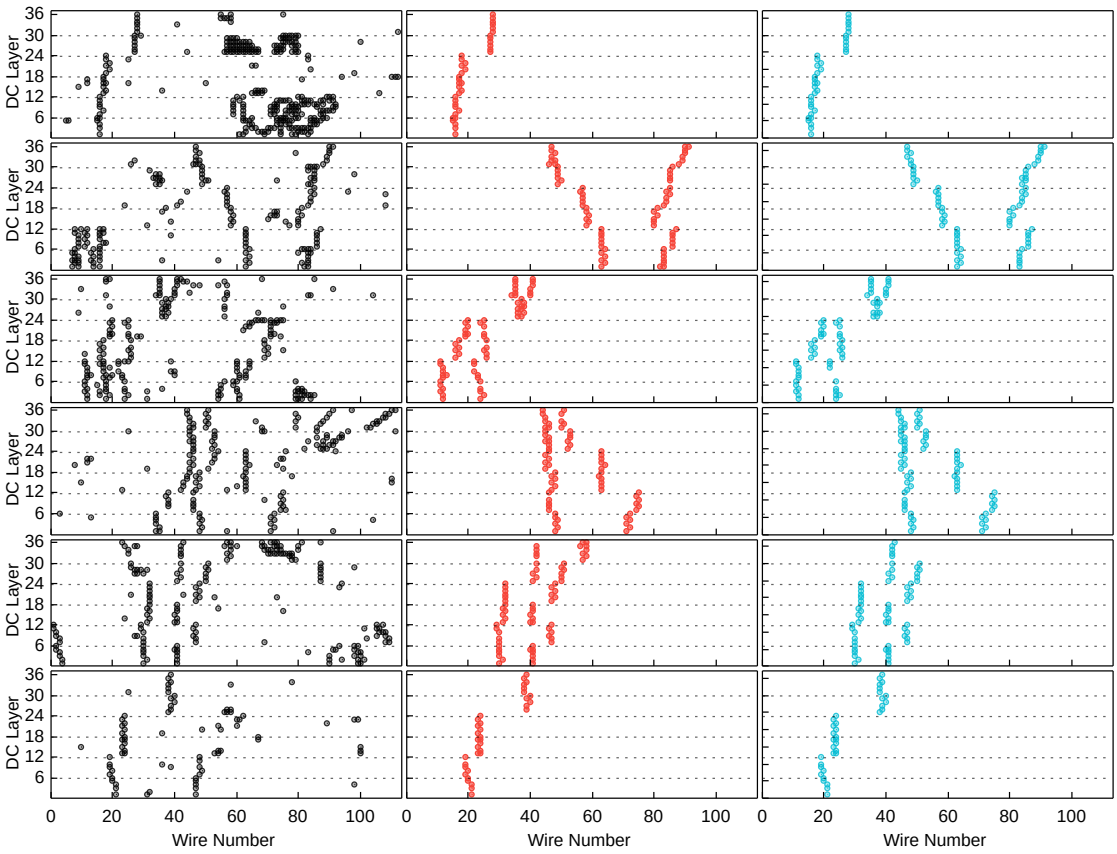



Figure 8: Examples of the performance of the CNN denoising autoencoder. The first column shows the raw (noisy) input given to the network, the middle one shows the noise-free image and the last column presents the result reconstructed by the autoencoder. Dashed lines represent boundaries of drift chamber's super-layers.

From these results, it is clear that CAEs are able to de-noise and reconstruct given input accurately, including the case when there are multiple tracks in a single input. The best model, considering a combination of low noise fraction, high track hits fraction and high denoising power, (0b, see Figure 6) achieved a mean valid hit detection accuracy of over 95.5% with the mean amount of noise not exceeding 11.05%. 0b achieved very similar denoising efficiency to 0a and 0c, however, it performs better in track hits accuracy so we choose it as our best model. We should note though that since the performance of the models is very close to each other the ranking can change between different training sessions due to the randomness that is introduced from the learning process. Figure 8 shows examples of input together with the expected, clean output and the model predictions. Models 1 & 2 produced the worst results which could be attributed to the fact that whole columns of the input are skipped when using a stride of 2 and since our features are mostly presented in the vertical dimension, a lot of them could be lost. The training process for the best model requires approximately 17 minutes to complete on our hardware setup using the aforementioned dataset of 60000 samples. De-noising a single event requires on average 250 $\mu s$ in our experimental setup, however, when running as part of the reconstruction process on JLab's computing facility which only consists of CPUs, it takes approximately 90ms per

event. The tracking code itself takes 520 ms per event (but this is background dependent), however, we noticed that de-noised files run faster which means that we will gain some of the lost time back from 90ms. More detailed benchmarks will be done once the de-noiser is fully integrated into the workflow.

## 7. Systematic Studies

### *7.1. Luminosity Studies*

In section 6 we showed that CAEs are very efficient in de-noising data from CLAS12 drift chambers, removing up to 96% of the initial background hits while retaining more than 95% of track related ones. Next, we will study how increased noise (by-product of high luminosity) can affect the accuracy of our de-noising auto-encoder. We used experimental data with 45 $nA$ to train the best network we developed (model 0b) and all our studies are performed using this model. Higher luminosity testing samples were produced using a standard background merging software [24] developed for CLAS12. The generation of these testing samples is performed in the following steps:

1. An initial set of events is taken from low luminosity (low beam current) experimental data where the noise level is minimal.
2. The low luminosity run is then merged with background generated from experimental data on different beam current runs.

As a result we produced data samples corresponding to 45 $nA$, 50 $nA$, 55 $nA$, 90 $nA$, 100 $nA$ and 110 $nA$ for our studies.

The testing samples were then analyzed with CAE to extract track hits reconstruction efficiency and background hits removal efficiency.

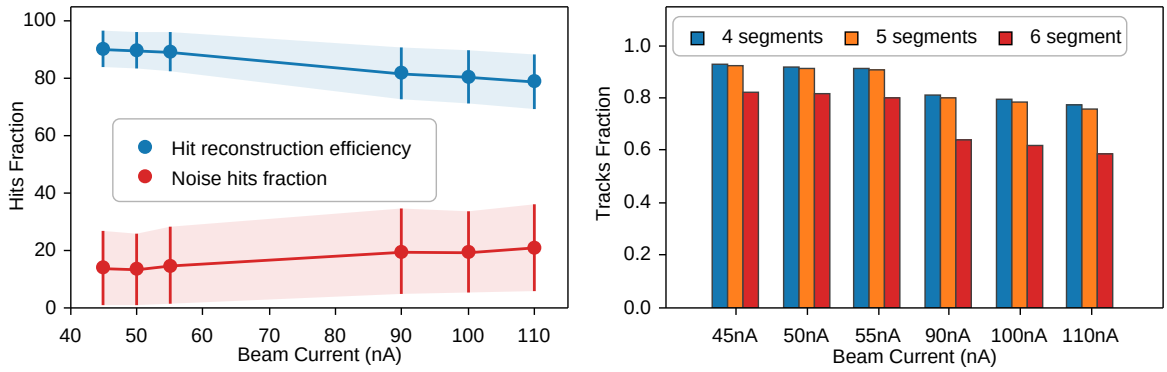



Figure 9: Reconstructed hits fraction for real signal and background hits (left) as a function of initial beam current. The fraction of tracks reconstructed with 4,5 and 6 segments fully recovered from de-noised data sample (right).

The results are shown in Figure 9. The blue dots of the figure on the left panel depict the fraction of valid hits reconstructed as a function of beam current. The red dots show the fraction of noise hits present in the de-noised data as a function of beam current. As shown

in the figure, network performance drops slowly with the luminosity resulting in average efficiency of about 76% for 110 $nA$ beam current. Sample images for different luminosity can be seen in Figure 10. One sample from each luminosity (background) setting is presented in each row, along with the clean event (the middle column) that represents only hits that belong to a reconstructed track and the auto-encoder reconstructed image (right column). As seen in the example images, there are a few traces of background still present in the reconstructed image, however, these will be eliminated by the tracking algorithm during track candidate selection. It is worth noting that more than 90% of the initial background is removed by the neural network, simplifying the clustering process significantly.

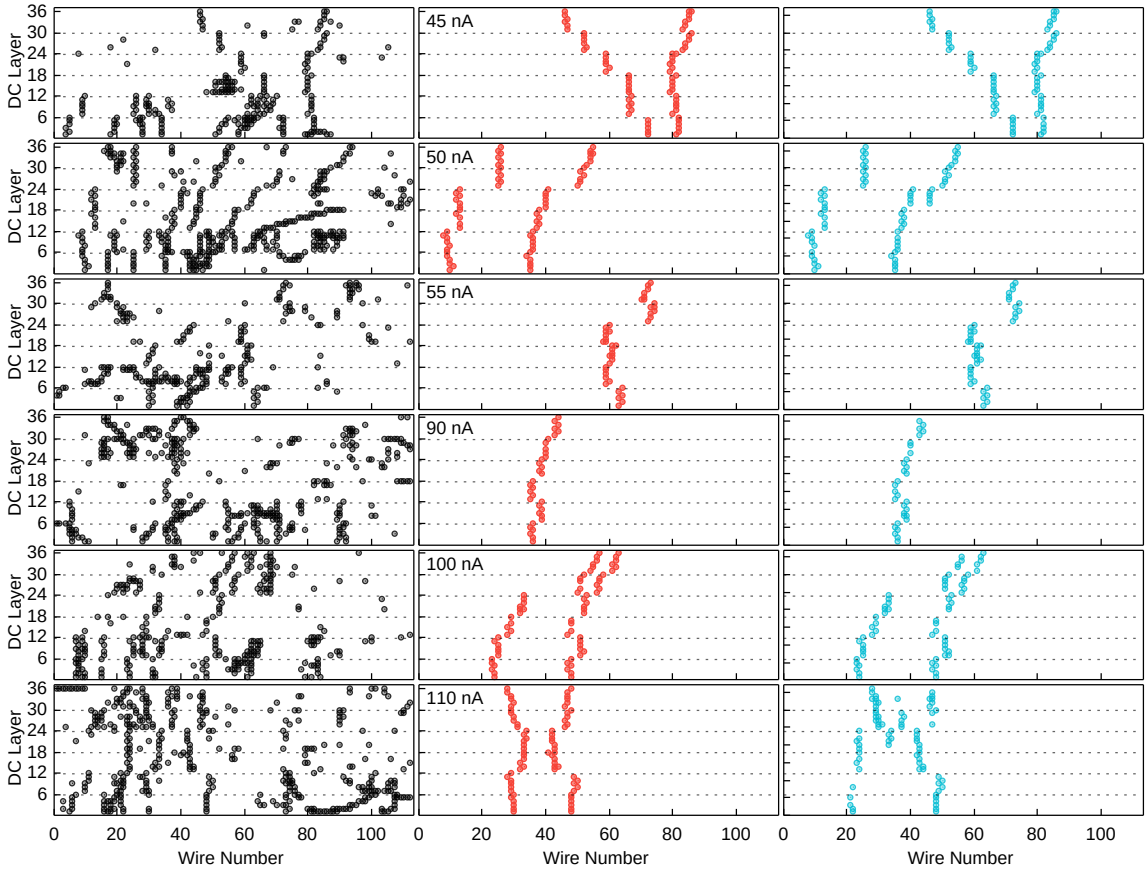



Figure 10: Examples of network de-noising with raw data on the left column, the ground truth image in the middle and reconstructed image on the right. The rows represent one example from each background setting corresponding to 45, 50, 55, 90, 100 and 110 nA respectively. Dashed lines represent boundaries of drift chamber's super-layers.

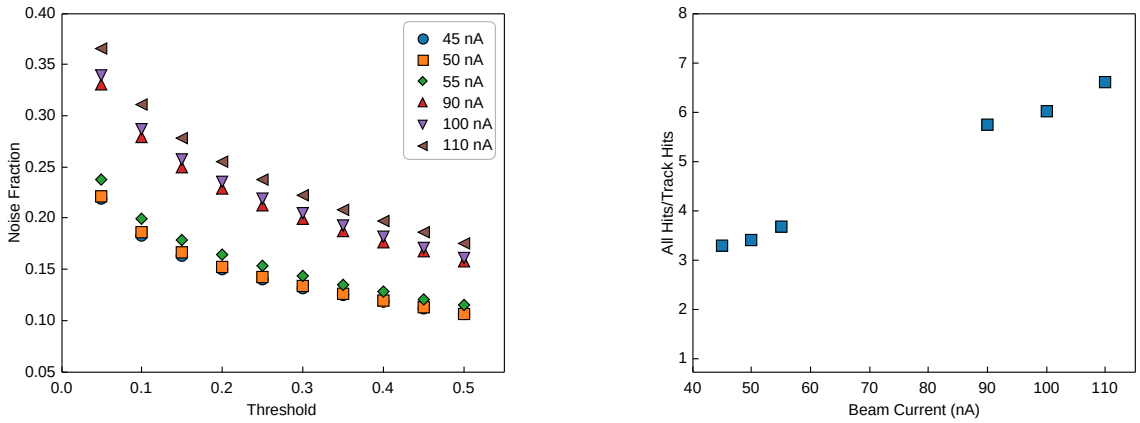
Even though hit reconstruction efficiency indicates the reconstruction efficiency of our method, it does not fully describe its capability to reconstruct full tracks. To quantify its track reconstruction efficiency, we need to measure the network's ability to reconstruct enough hits that would lead to full track reconstruction. In the reconstruction procedure of CLAS12, a track can be fully reconstructed if segments of the track are identified in any five or four consecutive super-layers. By employing a separate neural network, the missing segment positions can be predicted based on the existing 5 (or 4) segments and recovered from the raw

input data. A detailed description of this procedure is presented in [25] and [26].

Given that tracks can be recovered from incomplete segments reconstructed, we will determine the fraction of events that have retained 4, 5, or 6 segments by the CAE to evaluate its track reconstruction efficiency. Before that, we should note that the segment recovery efficiency is also subject to some other considerations. Segments ideally consist of hits in all six layers of a super-layer. However, if hits in 2 or more layers are recovered, there is enough information about the position and direction (or angle relative to wire planes) of the segment to recover the rest of the hits of the segment from raw hits data. With all these considered, we analyzed the output images of the CAE to measure how many segments are reconstructed for each event. The results are shown in Figure 9 (right) where track reconstruction efficiency is presented as a function of beam current. Tracks where at least four of their segments are recovered can be fully reconstructed, resulting in a track reconstruction efficiency of more than 75% under running conditions of 110 $nA$.

### *7.2. Reconstruction Threshold Studies*

As mentioned in section 4 we used a cutoff threshold of 0.5 to assign each of the output values of the CAE to a class of no hit detection (values $< 0.5$) or hit detection (values $\geq 0.5$). In the following studies, we investigate whether modifying this threshold can improve the de-noising and track reconstruction efficiency. We rerun our luminosity studies with varying cut-off thresholds ranging from 0.05 to 0.5 with a step of 0.05. The resulting distributions can be seen in Figure 11a, where the noise level is presented as a function of the cut-off threshold for different beam currents.

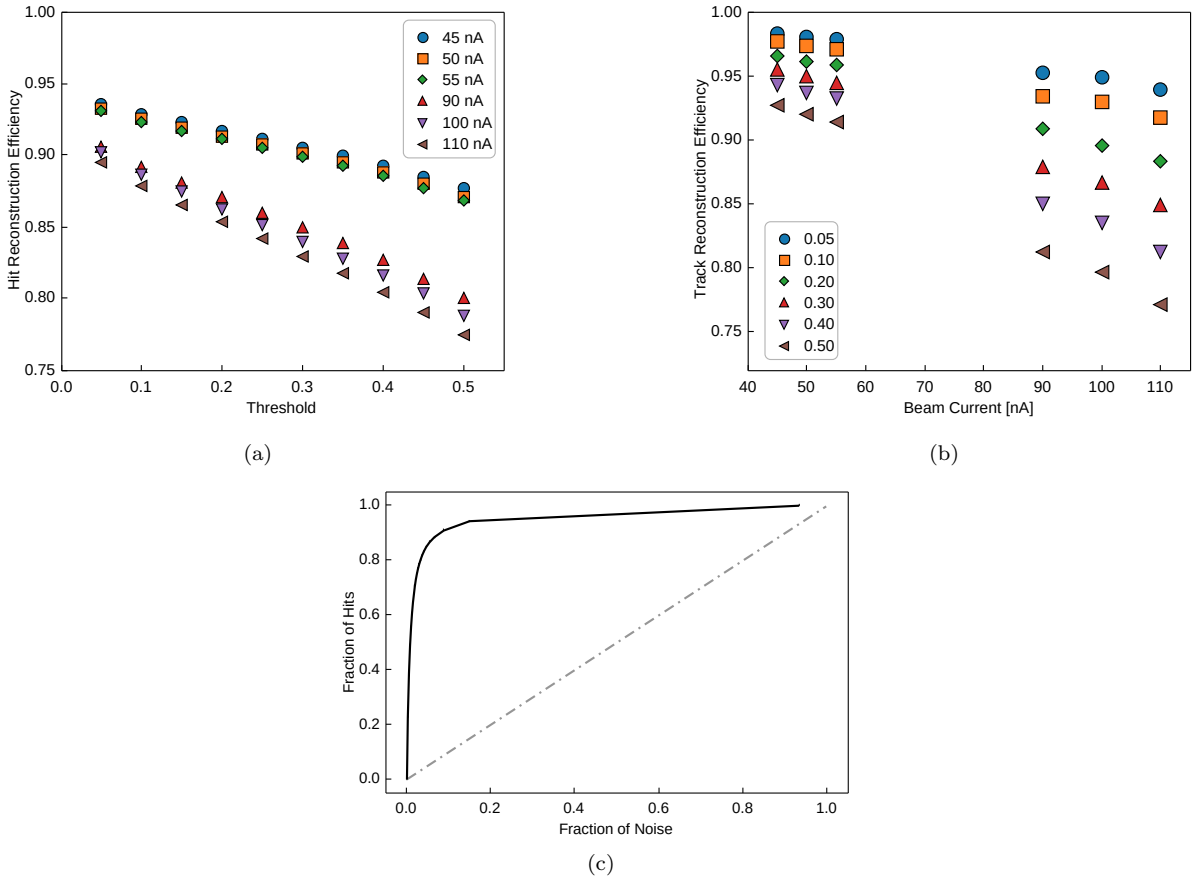



(a) (b)

Figure 11: Noise reduction efficiency for all beam currents as a function of cutoff threshold (left). The ratio of all hits in the event to the hits belonging to a reconstructed track as a function of beam current, before applying de-noising(right).

As expected, lowering the cut-off threshold for pixel reconstruction does increase the noise level, though the noise is still significantly lower than in the original noisy data. For comparison, Figure 11b shows the ratio of all hits to those that are part of a track for the initial

noisy data under high luminosity. For 110 nA, the ratio of noise in the raw data sample is about 6.5, while using the CAE with the lowest threshold of 0.05 yields a noise level of 0.37. Even though this change in the threshold affects the noise level slightly, it gives a significant improvement in hit and track reconstruction efficiency. Figure 12a shows the hit efficiency improvement as exhibited by lowering the cut-off threshold. For the highest beam current of 110 $nA$, the hit efficiency improves from 0.76 to 0.89. The same analysis is performed to measure the final track reconstruction efficiency after de-noising. The results are shown in Figure 12b. The same trend as with hit reconstruction efficiency is observed in track reconstruction efficiency. In this case, the improvement is even bigger with the track efficiency for 110 $nA$ increasing from 76% to 94%. Figure 12c presents the Receiver Operating Characteristic Curve (ROC)[27] curve produced by running model 0b on the 110 $nA$ dataset where true positive rate corresponds to the ratio of hits correctly identified and false positive rate to the noise that was wrongly maintained in the output. The curve confirms that our model accurately classifies hits and noise. In the next section, we show that the results obtained for high beam currents are significantly better than the standard tracking procedure for similar data.




(a) (b)

(c)

Figure 12: Hit reconstruction efficiency for all beam current data samples as a function of cutoff threshold (a). The track segment reconstruction efficiency as function of beam current for different cutoff thresholds (b). ROC curve for model 0b running on the 110 $nA$ dataset (c).

## 8. Studies with experimental data

From our systematic studies, we established that using a threshold of 0.05 for hit reconstruction was efficient in removing a significant fraction of noise hits while significantly improving the track hits reconstruction efficiency. The next step of our study was to use the presented CAE in the data reconstruction workflow of the CLAS12 detector. We used our CAE as a stand-alone module to de-noise raw data from drift chambers before passing them to the reconstruction process. We used two sets of data to measure the effect of de-noising on the reconstruction software. One data set was created by merging background to experimental data taken with beam current of 5 $nA$ and producing six data sets with experimental conditions corresponding to 45 $nA$, 50 $nA$, 55 $nA$, 90 $nA$, 110 $nA$ and 110 $nA$. The second data set was taken from experimental data with beam current 40 $nA$ and 50 $nA$. For all data sets the CAE was used to produce de-noised data set. Both data sets (original data and the de-noised data) were then processed with CLAS12 reconstruction software.

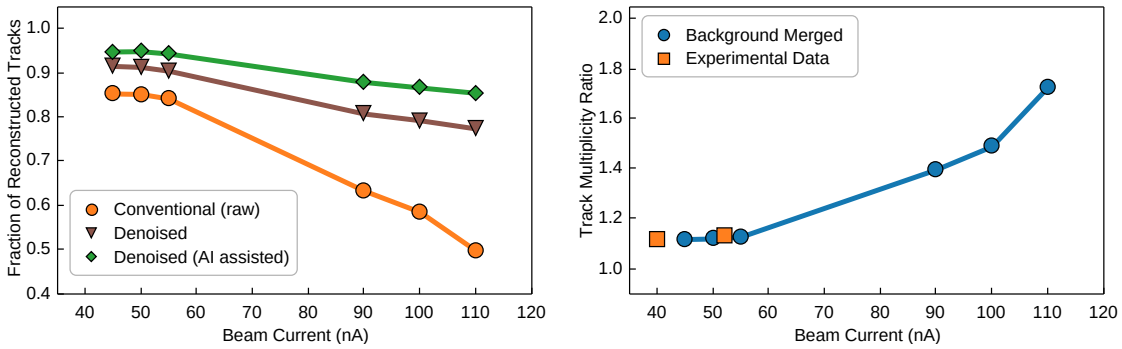



Figure 13: Track reconstruction efficiency (for 6 super-layer tracks) as a function of beam current for raw data (orange circles), for de-noised data (brown triangles), and for de-noised data running through AI assisted track reconstruction [26] (green diamonds) (left). Track reconstruction gain with de-noising algorithm, ratio of reconstructed 6 super-layer tracks with CAE to conventional raw data reconstruction for different beam current settings for background merged files (blue circles) and for real data (orange squares) (right).

The resulting files were analyzed on an event-by-event basis to measure the track reconstruction efficiency after de-noising. We isolated tracks from the original 5 $nA$ data (reconstructed with six super-layers) and examined whether the same tracks were reconstructed from the background merged file (both raw and de-noised version). Figure 13 (left panel) shows the fraction of tracks reconstructed from background-merged files for both data sets (raw and de-noised) as a function of beam current. The three different tests represent reconstruction using the CLAS12 tracking algorithm on raw (un-altered) data (orange circles), data pre-processed with the de-noising CAE (brown triangles), and finally, results from using the CLAS12 reconstruction, augmented with AI-assisted track identification software [26], on de-noised data. As shown in the figure, passing data through the de-noising CAE leads to a significant improvement in the fraction of tracks that can be recovered in high background conditions compared to using conventional methods. The main reason for this improvement is that in high background conditions the conventional algorithm fails to isolate clusters in each super-layer to have all track candidates considered for track composition. Using CAE

helps to reduce noise, thus enabling the conventional tracking algorithm to more efficiently reconstruct clusters and, as a result, reconstruct more tracks.

The improvements of the CAE assisted tracking is shown in Figure 13 (right panel), which presents the ratio between tracks reconstructed with and without applying the CAE de-noising step. The track ratio presented is tracks reconstructed by conventional algorithm only, without AI assisted track candidate classification, just to emphasize that with conventional algorithm the gain from de-noising alone is very significant. As can be seen from the figure, CAE improves the track yield in normal experimental conditions (i.e. 40 $nA$-50$nA$) by about $12\% - 15\%$ while achieving a much larger improvement at higher background settings (about 75% at 110 $nA$). The improvement is also apparent on experimental data (without background merging), where we extract the ratio of reconstructed 6-super-layer tracks of CAE de-noised data over raw conventional reconstruction (Figure 13, right panel), for experimental data with 40 $nA$ and 50 $nA$. The experimental data with different beam currents show similar (consistent) improvements in track multiplicity.

## 9. Discussion

In this work, we developed methods for de-noising drift chamber data for the CLAS12 detector. The proposed Convolutional Auto-Encoders were trained on six-segment tracks, along with whole raw hits from experimental data, to remove noise hits and retain those belonging to a track. We developed this method as a stand-alone software that has to be run on raw data files before they are processed with standard CLAS12 reconstruction software. Applying this method as a preprocessing step presents some limitations on discovering its full potential in track reconstruction. One such limitation comes from the fact that not all pixels in each segment are reconstructed by CAE, leading to potential loss of efficiency. CLAS12 tracking software relies on having at least four layers (out of six) activated in each super-layer to form a cluster. If the CAE preserves three hits out of six, this cluster will not be reconstructed by CLAS12's software because our stand-alone program removes the hits from the list of potential signals from the raw data set. To overcome this limitation the CLAS12 tracking software has to be modified to take into account suggested hits from CAE and then try to perform clustering by retrieving nearby hits from raw data; such a modification will improve the segment reconstruction efficiency even further.

The results presented in this paper are preliminary and do not include all the improvements discussed above. However, they already show significant improvement in the number of tracks reconstructed by standard CLAS12 tracking code after applying de-noising to the data set. In normal experimental conditions (at 45nA), we get an improvement of $12 - 15\%$, in the number of tracks reconstructed, while in high luminosity conditions (above $100nA$), we observe improvements of more than 75%. These results have a significant impact on how experiments run. The length of an experiment (time allocated to run with the accelerator provided beam) is determined by the integrated luminosity requirements needed to produce physics results. The running conditions, i.e. target length and beam current, are chosen so that tracks are reconstructed with high efficiency. As can be seen from our results, track reconstruction efficiency with our current (not optimized) implementation at 110 $nA$ is higher than the standard reconstruction efficiency (without de-noising) at current experi-

mental running conditions ($45nA$ beam). By employing de-noising for track reconstruction algorithms, one can run experiments in a much shorter time to achieve the same statistical yield for the same physics outcome.

This project took about 4-5 months to complete. This includes finding the suitable Machine Learning algorithm to apply (CNN), inspired by image denoising applications, gathering the training and testing datasets, experimenting with different architectures, activation functions and error estimators, and finally experimenting with different thresholds. Finding the appropriate architecture was the most time consuming process as it required manually adjusting the parameters of the model and then training and testing until satisfactory results were obtained. We intend to continue our efforts to further improve our model as it is integrated in JLab's data processing pipeline.

## 10. Summary

In this work, we used Convolutional Auto Encoder neural networks to de-noise raw data from CLAS12 drift chambers. The Convolutional Auto Encoder showed very good performance in removing noise hits, reducing them by more than 90% while retaining 76% of the hits belonging to track trajectories. Further studies with threshold adjustment improved average track trajectory hit reconstruction to 94% with an insignificant increase in noise hits. The developed software was used to de-noise several sets of data which then ran through CLAS12's standard reconstruction software. We compared the reconstructed tracks yield to standard (non de-noised) data track yield, showing that the de-noising significantly improves track reconstruction efficiency for six-segment tracks. For high background conditions (such as for data $> 90nA$) the gains in track reconstruction are much higher, up to 1.8 times improvement for $110nA$. This software will be rigorously tested in CLAS12 data processing environment in parallel with standard reconstruction packages and will be implemented in the data processing workflow for assisting tracking algorithm in high background running conditions.

## 11. 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. The authors would like to thank Raffaella De Vita for help in processing data with CLAS12 reconstruction software. This work was performed using the Turing and Wahab computing clusters at Old Dominion University.

## References


[1] V. Burkert, et al., The CLAS12 Spectrometer at Jefferson Laboratory, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment 959 (2020) 163419. doi:10.1016/j.nima.2020.163419.

[2] M. Mestayer, et al., The CLAS12 drift chamber system, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment 959 (2020) 163518. doi:10.1016/j.nima.2020.163518.

[3] V. Ziegler, et al., The CLAS12 software framework and event reconstruction, Nucl. Instrum. Meth. A 959 (2020) 163472. doi:10.1016/j.nima.2020.163472.

[4] V. Nair, G. E. Hinton, Rectified linear units improve restricted boltzmann machines, in: Proceedings of the 27th International Conference on International Conference on Machine Learning, ICML'10, Omnipress, Madison, WI, USA, 2010, p. 807–814.

[5] D. E. Rumelhart, G. E. Hinton, R. J. Williams, Learning internal representations by error propagation, in: Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations, MIT Press, Cambridge, MA, USA, 1986, p. 318–362.

[6] A. Ng, et al., Sparse autoencoder, Stanford CS294A Lecture notes 72 (2011) (2011) 1–19.

[7] S. Rifai, P. Vincent, X. Muller, X. Glorot, Y. Bengio, Contractive auto-encoders: Explicit invariance during feature extraction, in: Proceedings of the 28th International Conference on International Conference on Machine Learning, ICML'11, Omnipress, Madison, WI, USA, 2011, p. 833–840.

[8] J. Xie, L. Xu, E. Chen, Image denoising and inpainting with deep neural networks, Advances in neural information processing systems 25 (2012) 341–349.

[9] D. P. Kingma, M. Welling, An introduction to variational autoencoders, Foundations and Trends® in Machine Learning 12 (4) (2019) 307–392. doi:10.1561/2200000056.

[10] I. T. Jolliffe, PRINCIPAL COMPONENT ANALYSIS: A BEGINNER'S GUIDE — I. Introduction and application, Weather 45 (10) (1990) 375–382. doi:10.1002/j.1477-8696.1990.tb05558.x.

[11] W. Wang, Y. Huang, Y. Wang, L. Wang, Generalized autoencoder: A neural network framework for dimensionality reduction, in: Proceedings of the IEEE conference on computer vision and pattern recognition workshops, 2014, pp. 490–497.

[12] A. Krizhevsky, G. E. Hinton, Using very deep autoencoders for content-based image retrieval., in: Proceedings of the European Symposium on Artificial Neural Networks, ESANN '11, Bruges, Belgium, 2011.

[13] M. Sakurada, T. Yairi, Anomaly detection using autoencoders with nonlinear dimensionality reduction, in: Proceedings of the MLSDA 2014 2nd Workshop on Machine Learning for Sensory Data Analysis, MLSDA'14, New York, NY, USA, 2014, p. 4–11. doi:10.1145/2689746.2689747.

[14] F. Murtagh, Multilayer perceptrons for classification and regression, Neurocomputing 2 (5) (1991) 183–197. doi:https://doi.org/10.1016/0925-2312(91)90023-5.

[15] Y. LeCunn, Y. Bengio, G. Hinton, Deep learning, Nature 521 (7553) (2015) 436–444.

[16] J. Han, C. Moraga, The influence of the sigmoid function parameters on the speed of backpropagation learning, in: Proceedings of the International Workshop on Artificial Neural Networks: From Natural to Artificial Neural Computation, IWANN '96, Springer-Verlag, Berlin, Heidelberg, 1995, p. 195–201.

[17] T. Dozat, Incorporating nesterov momentum into adam, in: International Conference on Learning Representations, Workshop Track, ICLR '16, Caribe Hilton, San Juan, Puerto Rico, 2016.

[18] 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), 2016, pp. 265–283.

[19] F. Chollet, et al., Keras (2015).
URL https://github.com/fchollet/keras

[20] C.-C. Chang, C.-J. Lin, Libsvm: A library for support vector machines, ACM Trans. Intell. Syst. Technol. 2 (3) (May 2011). doi:10.1145/1961189.1961199.

[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 12 (Oct) (2011) 2825–2830.

[22] The Official YAML Web Site.
URL https://yaml.org/

[23] Anaconda software distribution (2020).
URL https://docs.anaconda.com/

[24] S. Stepanyan, et al., CLAS12 FD charge particle reconstruction efficiency and the beam background merging, CLAS12-NOTE, 2020-005 (2020).

[25] G. Gavalian, Auto-encoders for track reconstruction in drift chambers for clas12 (2020). arXiv:2009.05144.

[26] G. Gavalian, P. Thomadakis, A. Angelopoulos, V. Ziegler, N. Chrisochoides, Using artificial intelligence for particle track identification in clas12 detector (2020). arXiv:2008.12860.

[27] A. P. Bradley, The use of the area under the roc curve in the evaluation of machine learning algorithms, Pattern Recogn. 30 (7) (1997) 1145–1159. doi:10.1016/S0031-3203(96)00142-2.
URL https://doi.org/10.1016/S0031-3203(96)00142-2