Applying Machine Learning to Particle Detectors

Alexander Ercolani
University of Connecticut, Department of Physics

Abstract

Particle detectors are instruments used to collect data on high energy particle interactions. These interactions, called events, produce large numbers of particles, and in turn, a large amount of data. The raw data produced by the detectors must be processed so researchers can determine what to save for analysis and what to discard. However, the calculations used when recreating these events take time and a large amount of computing power. To combat this, machine learning can learn patterns to perform the same calculations more efficiently. This project explores the applications of machine learning on data from the GlueX Detector.

Table of Contents

1. Introduction

Particle accelerators are used in labs around the world to perform high energy particle collisions. These high energy collisions are capable of producing a wide range of particles. The final state of the particles, as recorded by a particle detector, can be used to reconstruct the physics events which occurred, allowing researchers to study the underlying physics of the most basic building blocks of the universe. These experiments produce large quantities of data and it is the job of built-in triggers to determine which data is worth keeping in order to save on storage space, the time it takes to write the data to storage, and the time it takes to perform reconstructions of the physics events. Some labs, such as CERN, have begun to develop machine learning algorithms capable of acting as triggers [1]. The GlueX experiment at the Thomas Jefferson National Accelerator Facility (JLab) has not yet deployed machine learning triggers. This paper explores the implementation of neural networks using data from GlueX simulations to study the potential use of machine learning for fast event reconstruction in a level-3 trigger.

1.1 Gluex Experiment

The Continuous Electron Beam Accelerator Facility (CEBAF) at JLab accelerates electrons around a racetrack-shaped path through two parallel linear accelerators. The electrons from CEBAF reach 12 GeV of energy before they enter Hall D. Hall D is the hall which houses the GlueX experiment. Upstream of Hall D the electrons pass through the Tagger Hall that houses a diamond radiator and tagging spectrometer creating a 9 GeV photon beam from the process of bremsstrahlung. This photon beam leaves the tagger hall, and enters Hall D where it interacts in a liquid hydrogen proton target in the main experimental hall (see Fig. 1.). When the beam collides with the target a photoproduction reaction occurs, producing a scattered proton

and a system of mesons. These particles then move through the detector which is composed of components capable of tracking charged and neutral particles. The reconstructed events recorded by the GlueX detector are then utilized to map the spectrum of hybrid mesons [2].

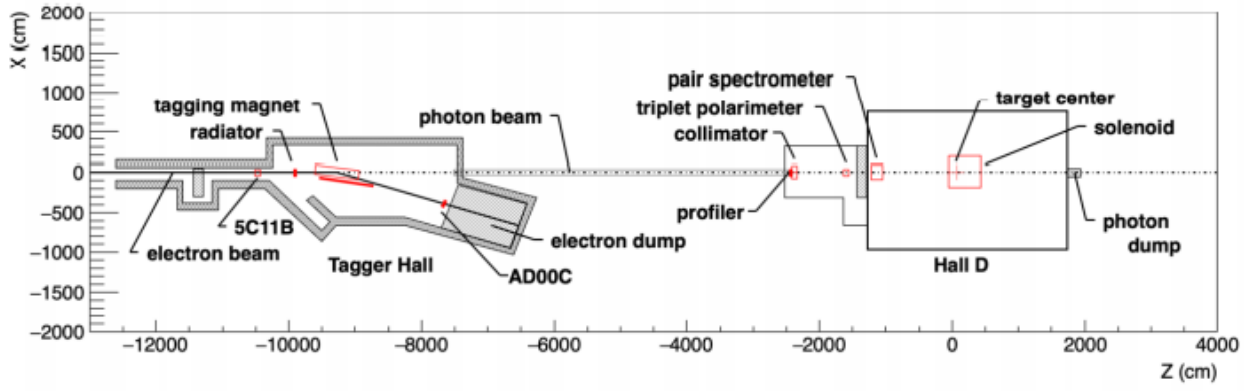

Figure 1: Schematic of Hall D. On the left is the Tagger Hall where the electron beam is used to produce photons through bremsstrahlung. On the right is the Hall D experimental hall where the target and GlueX detector are located [2].

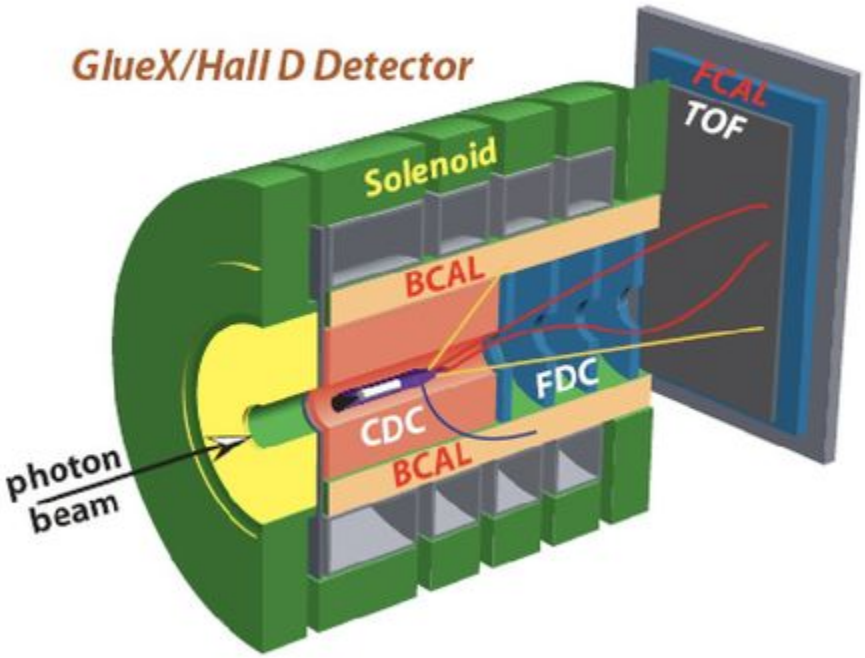

Figure 2: Cut-away view, showing half of the GlueX detector [3].

1.2 GlueX Data Collection

The signals created as particles interact with the detector are monitored by fast electronics circuits that determine when to trigger the capture of what is happening in the detector during a brief instant following an interaction in the target. Each trigger causes thousands of simultaneous signals from the detector subsystems to be digitized for a period of up to one microsecond, and saved in a data structure known as a physics event. One physics event produces approximately 15-20 kilobytes of raw data [2]. Overall, the detector collects about 600 megabytes of data per second; the phase 1 run of GlueX, which concluded in 2018, recorded a total of 3 petabytes of data. The data is constantly monitored during the collection process to determine which runs have viable physics events to analyse. The level-1 trigger uses a simple energy threshold to determine if the physics event is worth keeping and currently accepts a relatively large amount of physics events, only removing events which are entirely background radiation. The goal of the trigger is to limit data collection to a rate which can be written to tape in real time [4].

The full raw data that passes the initial triggers is stored on site. While being stored, a quality check is performed by reconstructing the first five files of a run's data. This check has the purpose of determining what fraction of the physics events is worth fully reconstructing and converting from raw data on tape to the Reconstructed Events Storage (REST) [2]. Later, full reconstructions are performed both on- and offsite using large computer farms. The next phase of the GlueX experiment is looking to add a level-3 software trigger that will similarly reconstruct events prior to them being written to tape, hoping to reduce the volume of data to be stored and later reconstructed by filtering out bad quality data [5].

1.3 Forward Drift Chamber

The Forward Drift Chamber (FDC) is a section of the GlueX detector designed to perform charged particle tracking. The FDC can be seen, represented as four packages, in Figure 2. Within each package are two modules, each consisting of three cathode chambers (see Fig. 3.). A single cathode chamber is made of three layers, a layer of anode wires between two layers of cathode strips. The middle wired layer consists of 96 anode wires spaced between 97 field wires which are read using Time-to-Digital Converters (TDCs). The cathode planes contain 192 cathode strips which are read using flash Analog to Digital Converters (ADCs). The two cathode layers are rotated 75° and 105° with respect to the wire layer and 30° relative to each other. The cathode chambers as a whole are rotated 60° relative to the chamber upstream. The FDC is capable of detecting particles emerging from the target in a range from 1° up to 20° in the upstream modules and 10° in the downstream modules. The data produced by the FDC is used to reconstruct the three dimensional tracks of charged particles.

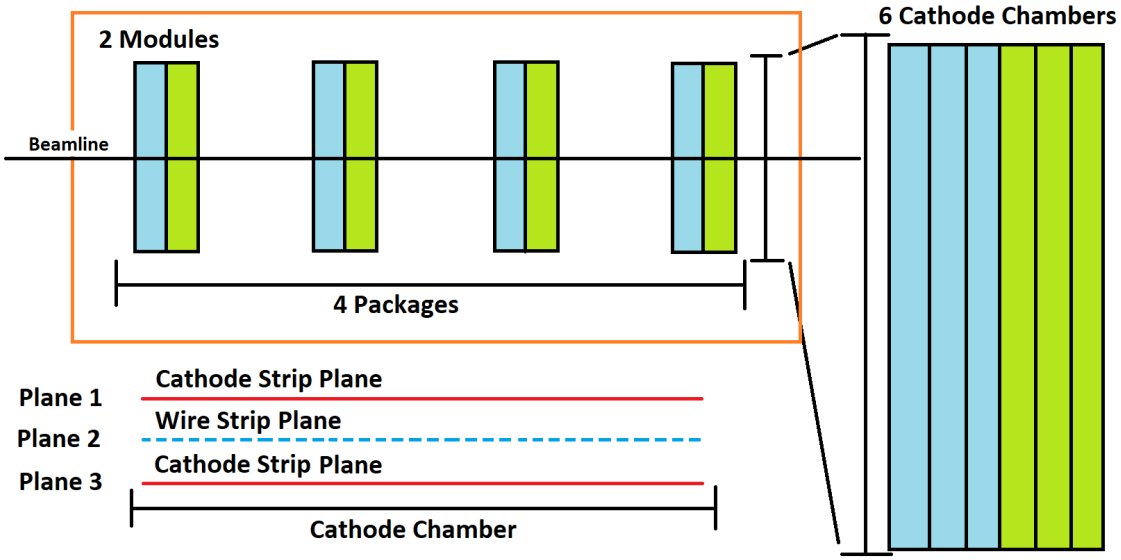

Figure 3: Conceptual diagram of the Forward Drift Chamber (FDC) located in the GlueX detector. In the orange box is a depiction of the four packages of the FDC along the beamline. The packages, seen blown up on the right, are cylindrical with two modules each, each module containing three cathode chambers. The cathode chambers contain two cathode strip planes on either side of a wire plane, seen at the bottom of the image.

2. Machine Learning

Machine learning is an area of computer science which uses computer algorithms to learn patterns from sets of data. These algorithms use training data to create a model and the model is then capable of making predictions on unseen data. Two common types of problems that can be solved using machine learning are regression and classification. Regression is tasked with taking an input of features and outputting a floating point number. Classification, on the other hand, takes a similar input of features and is tasked with determining which predetermined class the features are representative of.

Regression employs methods such as ordinary least squares which determines learned parameters by minimizing the sum of the squares of the difference between modeled behavior and expected behavior, as determined by a set of training data. Alternatively, for classification problems there exists support vector machines (SVMs) which attempt to find learned parameters

which are representative of a decision boundary with margins of error. The way SVMs learn is by trying to maximize the margins on either side of the decision boundary such that the boundary separates the classes in a way which puts them on either side of the boundary and as far away from it as possible. Neural networks are a more general form of machine learning that are capable of performing both regression and classification, as well as providing the building blocks for more abstract machine learning algorithms, such as convolutional neural networks.

2.1 Neural network

Neural networks are a type of machine learning algorithm that attempts to replicate the structure of neurons in the brain to make predictions on data. The most basic form of a neural network is referred to as both a feed-forward neural network and a multilayer perceptron [6]; the structure consists of layers of artificial neurons with weighted connections between layers (see Fig. 4.). Every neuron from a given layer has a connection to each of the neurons in the next layer. Each connection has a weight associated with it and each neuron has a bias, which combine to make up the learned parameters for the model. There is no limit to the number of layers in a neural network. When there are more than two layers the middle layers are referred to as hidden layers as they do not directly touch the input features or the final output and thus remain “hidden” to the user.

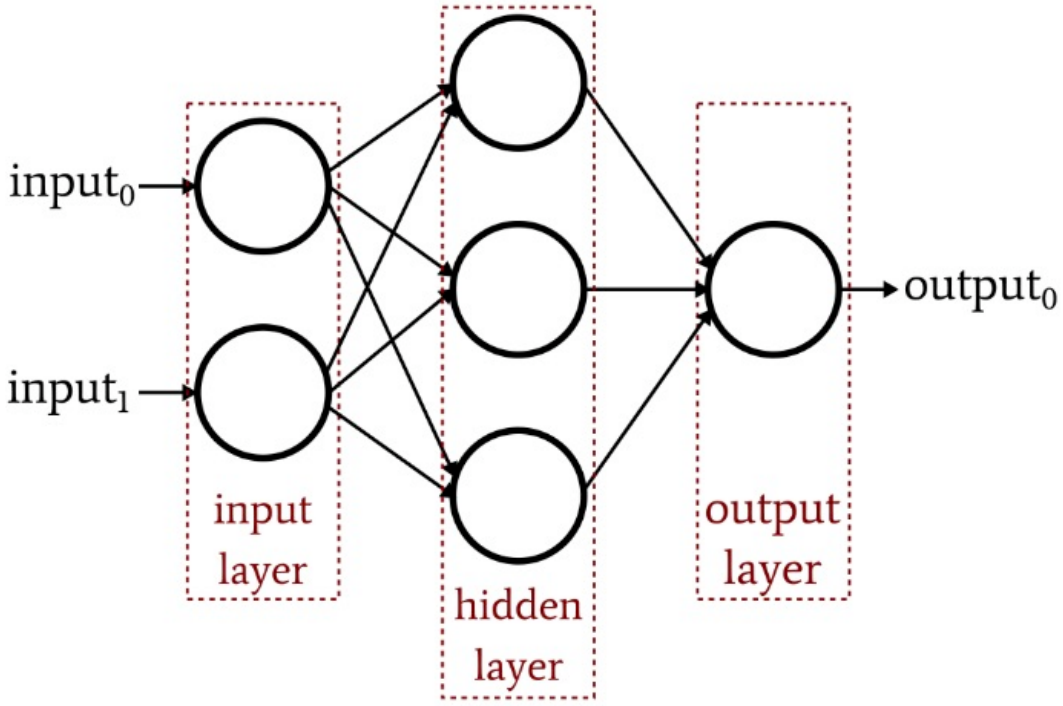

Figure 4: Depiction of a feed-forward neural network with two inputs, one output, and a single hidden layer with three neurons [6].

A node in a neural network is an artificial neuron which simulates neurons in the brain by activating when a specific input threshold is met. To do this, each neuron has an activation function which takes as input the dot product of the corresponding weight matrix with the matrix of outputs from the previous layer while adding the bias (see Fig. 5.) [7]. There are multiple common activation functions, with the most popular being Rectified Linear Unit (ReLU), a function which outputs zero for negative inputs and scales linearly for positive inputs (see Fig. 6.).

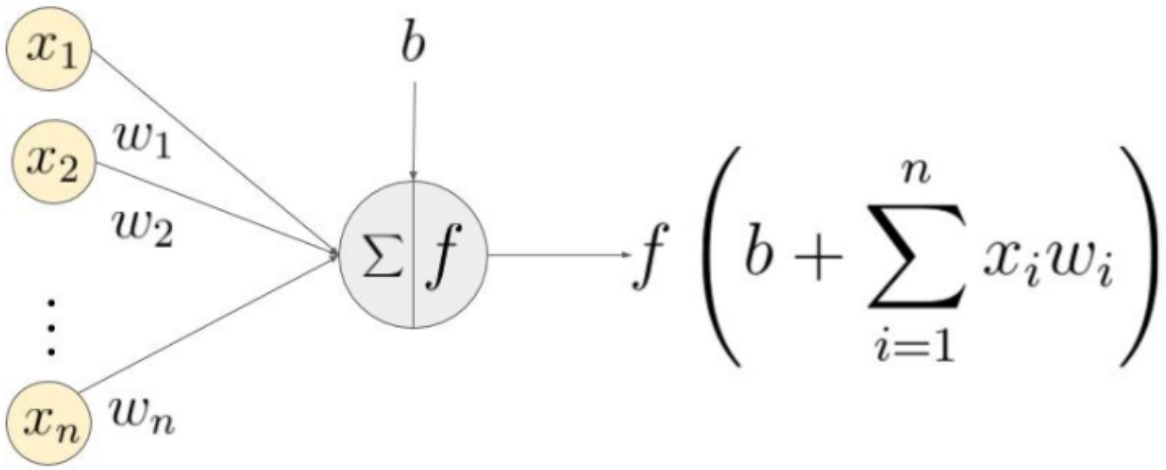

Figure 5: Image depicting the activation process for an artificial neuron in a neural network [7].

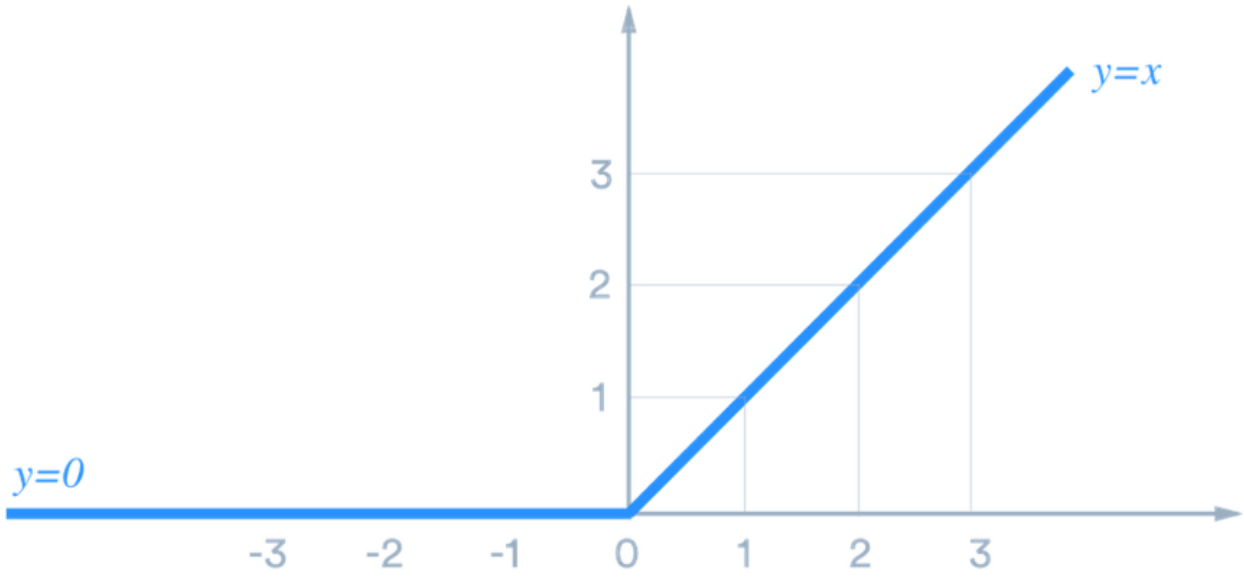

Figure 6: Graph of the Rectified Linear Unit (ReLU) activation function [8].

2.2 Optimization

For neural networks to be able to make predictions on unseen data it is necessary to first train them. Neural networks are trained using gradient descent to optimize a loss/cost function. The cost function provides a cost during training which applies a numerical value to how far away from the expected value the model output value is. Gradient descent takes the gradient of

the cost function with respect to the weights and biases, determining a “direction” to travel in towards a local minimum. This direction is always the negative of the gradient because the gradient always points in the direction of steepest ascent; this implies the negative of the gradient points towards the direction of steepest descent. The reason gradient descent is used is because it only requires the existence of the first derivative of the cost function and therefore does not require the expensive computations to calculate the hessian. Alternatives are to use Newton’s method to find the direction to move towards; this method models the cost function as a quadratic and uses both the hessian and gradient to calculate both the direction of travel and the step size. Newton's method has the benefit of reaching the minimum in fewer iterations, however, the computational costs typically outweigh the benefits, leaving gradient descent as the most efficient optimization method..

Being able to determine the direction towards a local minimum of the cost function allows the network to determine how to change the learned parameters to perform more effectively. To efficiently perform the optimization, a process called back propagation is used. Back propagation is the process of calculating the gradient starting from the output layer and moving towards the input layer; this is done because subsequent layers are dependent on each other and by computing the gradient starting from the output layer we can avoid recalculating the same values multiple times. During training, a validation data set is used to monitor the model performance and ensure it does not suffer from overfitting. The model does not update its parameters after predicting on a validation set, allowing the data to continuously represent new or unseen data. Small training data sets and imbalanced training data can lead to overfitting, resulting in a model that only works on the data it was trained with. Combating this requires

stopping the training when the validation loss begins to plateau or rise as the training loss continues to decrease. An alternative is increasing the amount of training data.

3. Implementation

3.1 Model

The neural network in this project was implemented using the Tensorflow python library with the Keras API. The model uses six layers, four of which are hidden. The input layer has 482 features corresponding to the 96 anode wires, 384 cathode strips, the module number, and the module layer number. Thus, the network takes raw information from a single cathode chamber. The raw data from the wires and cathode strips provides information on the particle as it interacts with the detector, while the module number and module layer number allow the model to account for how the chamber is rotated. The four hidden layers contain 256, 128, 64, and 32 nodes, respectively. The output layer of the model is composed of two nodes. A mean squared error loss function is used for regression, training the model to determine the xy-coordinate of the particle as it passes through a given chamber. The code which creates the architecture for the model is shown below. Each layer utilizes the relu activation function and the model consists of densely connected layers.

```
def create_model_1():
      return tf.keras.models.Sequential([
      layers.Dense(482, activation = 'relu', input_shape=(482,)),
      layers.Dense(256, activation = 'relu'),
      layers.Dense(128, activation= 'relu'),
      layers.Dense(64, activation = 'relu'),
      layers.Dense(32, activation = 'relu'),
      layers.Dense(2)
      ])
```

Prior to plugging the raw data into the network, the data is normalized, scaling the inputs to values between zero and one. The normalization serves the purpose of limiting the size of input values so the weights remain a reasonable size; by ensuring that all the inputs are similarly scaled, the inputs interact with the activation functions based primarily on their learned weights rather than their size relative to other inputs. Without normalization the weights may be excessively large or small to compensate for the difference in scale between inputs. Below is the code for the data formatting. It can be seen that the output array, with the local name ‘batch’, has 482 elements. The last two elements are the chamber layer and module number as described previously. The first 96 elements are the anode wires and the next 384 elements are the two sets of 192 cathode strips. For both the anode wires and the cathode strips they are one-hot encoded, meaning they are labeled with a one if they experience particle interaction and zero otherwise. This method of encoding the data performs the same function as normalization, however, some information is lost regarding the particles timing and energy. In the context of determining the position of the particle in the chamber, the lost information may have small effect on accuracy, but does inhibit the model's predictive power significantly.

```
def format_data(batch_size=10):
      batch = []
      anodes = []
      cathodes = []
      wires = []
      i = 0

      for r in hddm_s.istream(file_name):
      for chamber in r.getFdcChambers():
            data = [0]*482
            data[480] = chamber.layer
            data[481] = chamber.module
            for wire in chamber.getFdcAnodeWires():
```

```
                data[wire.wire] =  1
                anodes.append(wire.wire)

          for strip in chamber.getFdcCathodeStrips():
                if strip.plane == 1:
                data[strip.strip + 96-1] = 1
                cathodes.append((1,strip.strip))
                elif strip.plane == 3:
                data[strip.strip + 96 + 192-1] = 1
                cathodes.append((3,strip.strip))

          batch.append(np.array(data))
          wires.append((anodes,cathodes))
          anodes = []
          cathodes = []
          i+=1

          if i%100 ==0:
                print("data index:", i)
          if i%batch_size == 0:
                break
     if (i%batch_size == 0) and (i != 0):
                break
     return np.array(batch), wires
```

3.2 Training Data

The model was trained on 1,000,000 simulated single-track FDC events. The diameter of the FDC is 1m, thus for 1,000,000 uniformly distributed events one expects to have a data set with approximately two hits per square millimeter. The data used for training is generated using Monte Carlo simulations of the GlueX experiment. The data consists of 1,000,000 single proton events with no secondary particles. If an event sends a proton through the FDC it may travel through more than one chamber, creating multiple FDC hits which can be used for training. Simulated data gives the advantage of it being fully reconstructed with labels. As well, by using

simulated data we have the ability to test the capability of machine learning to perform reconstruction on charged particle tracks without having to worry about noise and multiple particle events; while this limits the use of the model it provides a baseline prior to the development of more sophisticated models.

4. Model Performance

4.1 Charged Particle Tracking

The trained model is capable of determining the location of a proton hit through a given chamber of the FDC within approximately five square milimeters of the actual hit. An example output from the model is seen in Figure 7 where the red ‘x’ marks the predicted hit location and the yellow square represents the actual hit location recorded during the simulation. This output is indicative of the potential for neural networks to discern high level information from raw data from the detector. The model is not accurate enough to perform full track reconstruction on the level of the current implementation, however, with training on larger data sets the model can only become more accurate.

4.2 Timing

The total reconstruction time for a charged track in the forward region of the detector is 125 milliseconds on a present-day 2GHz Intel processor. The amount of time that this model takes to make a prediction is 27.3 microseconds, a factor of approximately 5000 speed up. It should be noted that the reconstruction algorithms are designed to reconstruct multiple charged particle tracks at one time while this model is designed for single proton tracks; however, structurally, neural networks allow for a high amount of parallelization, and a more sophisticated model which is capable of reconstructing multiple particle tracks is not expected to have

significantly increased computation time compare to the current model. As is, the model performs all of its calculations serially and only makes a prediction on one chamber, but, it is important to note that in a live environment the same model would be run in parallel among the chambers while current software reconstructions are purely performed with serial computations. With a hardware implementation of the neural network in a graphics processor (GPU) or field-programmable gate array (FPGA), directly connected to the detector hardware, further increases in speed would be expected.

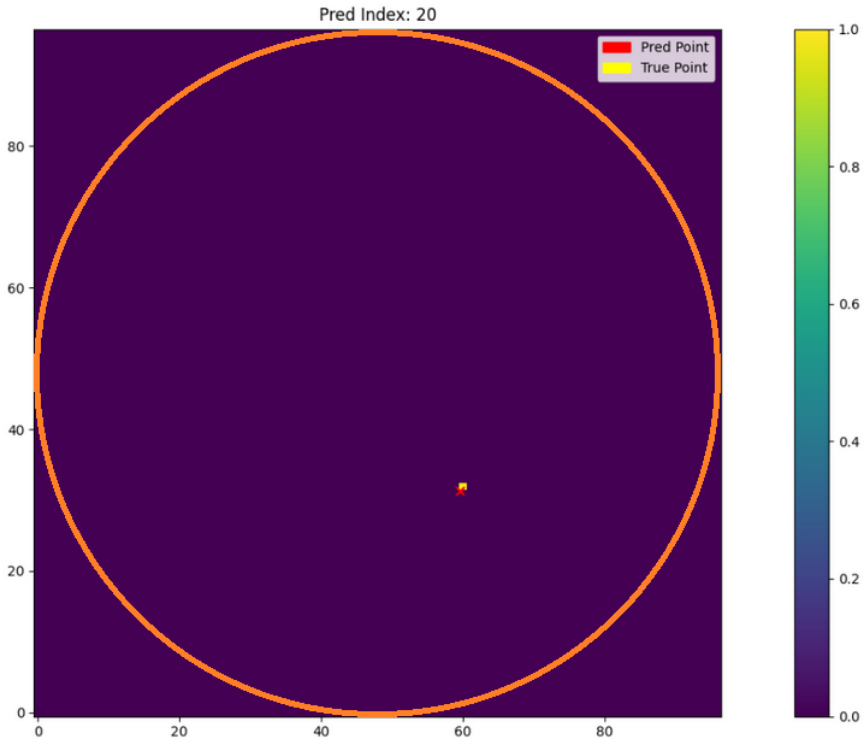

Figure 7: The hit location of a proton in a cathode chamber predicted by the trained model (seen as a red x) along with the actual location of the hit (seen as a yellow square).

5. Summary and Outlook

The results of the trained model indicate that neural networks can provide computationally efficient reconstruction algorithms. Currently a level-3 trigger requires a full reconstruction of the data to operate, but in the future a trained model can receive raw input and

make high-level inferences about the data. Due to their high parallelization, these models can be implemented as parallized hardware, quickly determining if an event is worth writing to tape in real time. There are multiple directions that this project can take in the future, one of which is training on reconstructed data to try and identify multiple particle hits in the detector, simultaneously reconstructing both tracks. The other direction is to use a software trigger to train the model to act as a level-3 trigger and discard noisy or incomplete events before they are saved. Overall, this project has shown that machine learning has the potential to improve particle detector research.

References

[1] J. Duarte, et al. "Fast inference of deep neural networks in FPGAs for particle physics". *Journal of Instrumentation* 13. 07(2018): P07027–P07027.

[2] Adhikari, S. et al. “The GlueX Beamline and Detector.” *arXiv: Instrumentation and Detectors* (2020): n. pag. (hall breakdown fig 1)

[3] The GlueX Collaboration, Gluex project overviews. (2021), from https://halldweb.jlab.org/wiki/index.php/GlueX_Project_Overviews

[4] “Level-1 Trigger.” *Level-1 Trigger - GlueXWiki*, The GlueX Collaboration, halldweb1.jlab.org/wiki/index.php/Level-1_Trigger.

[5] “Level-3 Trigger Strategy.” *Level-3 Trigger Strategy - GlueXWiki*, The GlueX Collaboration, halldweb.jlab.org/wiki/index.php/Level-3_Trigger_Strategy.

[6] Keim, Robert. “How to Train a Multilayer Perceptron Neural Network - Technical Articles.” *All About Circuits*, 26 Dec. 2019, www.allaboutcircuits.com/technical-articles/how-to-train-a-multilayer-perceptron-neural-network.

[7] abhigoku10. “Topic DL01: Activation Functions and Its Types in Artifical Neural Network.” *Medium*, Medium, 8 Apr. 2018,

abhigoku10.medium.com/activation-functions-and-its-types-in-artifical-neural-network-14511f3080a8.

[8] Sivarajkumar, Sonish. “ReLU - Most Popular Activation Function for Deep Neural Networks.” *Medium*, Medium, 15 May 2019, medium.com/@sonish.sivarajkumar/relu-most-popular-activation-function-for-deep-neural-networks-10160af37dda.