4-2019

Machine Learning and Electron Track Reconstruction for the MOLLER Experiment

Mary Robinson

Recommended Citation

Robinson, Mary, "Machine Learning and Electron Track Reconstruction for the MOLLER Experiment" (2019). *Undergraduate Honors Theses.* Paper 1325.
https://scholarworks.wm.edu/honorstheses/1325

Machine Learning and Electron Track Reconstruction for the MOLLER Experiment

A thesis submitted in partial fulfillment of the requirement for the degree of Bachelor of Science with Honors in Physics from the College of William and Mary in Virginia,

by

Mary Robinson

Accepted for: Honors

[illegible]
Advisor: Prof. David Armstrong

[illegible]
Prof. Seth Aubin

Pierre R. Clare
Prof. Pierre Clare

Department of Physics
College of William and Mary
Williamsburg, Virginia
May 2019

Contents

Acknowledgments **iii**

List of Figures **vii**

Abstract **v**

1 Introduction **1**

2 The MOLLER Experiment **3**

2.1 Theoretical Motivation 3

2.2 Experimental Setup 6

3 Machine Learning Overview **10**

3.1 Basic Structure of Machine Learning Algorithms 11

3.1.1 Training, Testing, and Validation Datasets 12

3.1.2 Loss and Optimization 13

3.2 Neural Networks 16

3.2.1 Epochs and Batch Size 17

3.2.2 LSTM Networks 18

4 Overview of Simulated Data **20**

5 Track Reconstruction Network **25**

5.1 The Baseline Network 25

5.2 Results 28

5.2.1 Impact of Network Parameters 29

6 Conclusion and Outlook **36**

A Data Processing Function **37**

B Track Reconstruction Network **39**

C The Adam Optimizer **41**

Acknowledgments

I would like to thank David Armstrong, Wouter Deconinck, and the rest of the parity group at William and Mary for their assistance with this project.

To David: Thank you for your guidance and help with this project over the past year. I learned a ton from you and I really enjoyed having the opportunity to work with you.

And, finally, to Alex: thank you for your endless kindness, patience, and support. You are wonderful.

List of Figures

2.1 **Helicity and parity violation** - Consider a particle with momentum **p** and helicity S. The top left image shows a right-handed particle; the top right image shows a left-handed particle. The bottom image shows what happens to the right-handed particle when it undergoes a parity transformation. . . 5

2.2 **Planned layout of the MOLLER experiment** - The electron beam (starting in the bottom right corner of the figure) will travel through the liquid hydrogen target, pass through the toroid system, then enter the detector system [1]. 7

2.3 **Planned layout of the detector system** - The electron beam is shown entering the detector system from the left. It passes through the GEM detectors, then enters the quartz assembly. Pion detectors are located downstream of the quartz assembly [1]. 8

2.4 **The GEM detector** - This is an image of the surface of a GEM taken with an electron microscope. The sheet is 50 μm thick and each of the holes is 70 μm in diameter [5]. 8

3.1 **Basic structure of a machine learning algorithm** - The input is fed into a model, which then predicts the output. The model predictions are compared to the actual output and the error (or loss) is computed. The model parameters are updated according to the value of the error. This cycle of making predictions, calculating the error, and updating the model parameters continues until the error is minimized. At this point, the model is said to be optimal. (Figure adapted from [7]) 12

3.2 Choose a starting point. 15

3.3 Compute the gradient of the loss function at the starting point. 15

3.4 Move in the direction of the negative gradient. 15

3.5 Repeat steps 1-3 until the loss is minimized. 16

3.6 **Structure of a neural network** - A neural network is a collection of variables (called "nodes") which are arranged in layers, similar to the neurons in a biological nervous system. The nodes of each layer are formed from combinations of the nodes in the preceding layers [10]. 16

3.7 **Repeating unit in a LSTM network** - The structure of a LSTM network allows the algorithm to "remember" information over time. Information from preceding units is passed into the current unit and used to update the so-called "cell state", which is passed on to the next unit. (Adapted from [12]). 18

4.1 **Virtual tracking and main detector planes** - This is the arrangement of the virtual tracking and main detector planes in the GEANT 4 simulation for the MOLLER experiment. The simulated electron beam travels from the left to the right along the z axis. The locations of the virtual tracking planes and main detector planes correspond to the planned locations of the GEM and quartz detectors for the experiment (see Figure 2.3). 20

4.2 **Dimensions of the virtual detectors** - Each of the virtual detectors is a circle with a radius of two meters. 21

4.3 **Distribution of electron hits in virtual detectors** - These histograms show the distribution of electron hits in (a) the four virtual tracking planes and (b) the main detector plane over all 100,000 events. The majority of the hits are located in ring(s) at a radius of approximately one meter. These are (primarily) the Møller electrons. The two-ring structure in figure (a) occurs because the 4 meter separation between the first two tracking planes and the last two (see figures 4.1 and 4.4) has been flattened into one dimension. . . 22

4.4 **Distribution of electron hits in virtual detectors (3-dimensions)** - This scatter plot shows the distribution of electron hits in all five virtual detector planes over all 100,000 events. The two leftmost planes are the first two tracking detectors; the three on the right are the final two tracking detectors followed by the main detector. Again, the highest concentration of hits in each detector occurs in a ring at a ~ 1 m radius. 23

4.5 **Example of an electron track** - The electron travelled from left to right in the positive z direction. Each circle marks the location (the (x, y, z) position, a.k.a. the *hit*) where the electron passed through a virtual detector plane. For simplicity, we only consider electrons which pass through all five virtual planes. The *track* is formed by connecting the five hits to form a path. . . . 23

5.1 **Accuracy of baseline network** - Most of the model's predictions differed from the actual output values by about 0.2 mm. 29

5.2 **Impact of batch size on efficiency** - As the batch size increases, the time taken to train the model decreases rapidly. 30

5.3 **Impact of number of epochs on efficiency** - As the number of epochs increases, the time taken to train the model increases linearly. 31

5.4 **Impact of LSTM dimensionality on efficiency** - As the dimensionality of the LSTM layer increases, the time taken to train the model increases. . 31

5.5 **Impact of batch size on accuracy** - As batch size increases, δ increases overall. This increase does not, however, appear to be consistent. 32

5.6 **Impact of number of epochs on accuracy** - As the number of epochs increases, the value of δ decreases rapidly, which means the accuracy increases. 33

5.7 **Impact of LSTM dimensionality on accuracy** - We were unable to determine a clear relationship between the number of LSTM dimensions and the value of δ 33

5.8 **Fluctuation of δ** - Over ten repeated trials with every network parameter held constant, there was a maximum fluctuation of approximately 1 mm in the value of δ 34

5.9 **Impact of detector resolution on model accuracy** - The circles are the data from shifting the training set, and the triangles are the data from shifting the testing set. At each point, δ is higher when the training set is shifted than when the testing set is shifted. In both cases, δ increases with increasing detector resolution. 35

Abstract

This report presents the development of a machine learning algorithm (a neural network) for the purpose of track reconstruction in the MOLLER experiment. The MOLLER experiment is a collaboration at Jefferson Lab which plans on measuring the parity-violating asymmetry in high energy electron-electron (Møller) scattering. This measurement is an important test of the Standard Model and could potentially serve as evidence for new physics beyond the Standard Model.

Reconstruction of electron trajectories provides important kinematic data about the electrons which is necessary for the asymmetry measurement. As such, the MOLLER experiment requires a track reconstruction tool which can efficiently handle large amounts of data. We created a recurrent neural network which models the trajectories of electrons in the detector system for MOLLER. We trained the network using data from the GEANT 4 Monte carlo simulation for the experiment.

At this stage, our network is able to correctly predict the locations where an electron hits the main detector given that electron's previous positions in the detector system. We describe various studies we conducted to improve the accuracy and efficiency of our network, as well as present suggestions for any potential continuations of this project.

Chapter 1

Introduction

Motivation for this project arose due to a desire for an efficient method of track reconstruction for the upcoming MOLLER experiment at Jefferson Lab. Kinematic data for the electrons is crucial to the interpretation of the parity-violating asymmetry; in order to understand the electron kinematics, we need to know the trajectories of the electrons as they pass through the detector system. Since, like many modern physics experiments, MOLLER will generate enormous amounts of data, the collaboration requires a tool which can handle large datasets while performing both quickly and accurately.

So, cue machine learning — specifically, neural networks. The use of these algorithms has greatly increased in popularity in recent years. Though they may seem like something out of a science-fiction story — a black box which magically spits out the answers you want without any clear explanation as to *how*, exactly, it does so — neural networks are, in actuality, nothing more than a method of optimization. The advantage of neural networks is that they excel at pattern recognition and they are intended to be used in situations with a large volume of data. Track reconstruction seemed like a natural application for such a tool. In addition to this, neural networks require little human input compared to current methods of track reconstruction. Implementing a neural network as a replacement for, or as a supplement to, the current algorithms could potentially highlight previously unknown inaccuracies and biases which could, ultimately, lead to better results.

In this report, we describe the creation and implementation of a neural network which models the trajectories of electrons in a simulated version of the MOLLER experiment. Given tracking data for a set of electrons, the model is able to predict the positions where each electron hits the main detector. This allows us to completely determine the path that each electron takes as it travels through the (virtual) detector system. Ultimately we hope that this project provides a good indication of the viability of neural networks as a tool for track reconstruction and for use in physics in general.

Chapter 2

The MOLLER Experiment

The MOLLER (Measurement Of A Lepton Lepton Electroweak Reaction) project at Thomas Jefferson National Laboratory is an upcoming experiment which plans on measuring and interpreting the parity-violating asymmetry (A_{PV}) in polarized electron-electron (i.e. Møller) scattering. A_{PV} is predicted to be (approximately) 33 parts-per-billion; the collaboration's goal is to measure this value with a precision of 0.7 parts-per-billion. The experiment will be using Jefferson Lab's recently upgraded 11 GeV beam, with the hope that the measurement of the asymmetry will improve upon a previous measurement of the same value (performed at SLAC National Accelerator Laboratory) by a factor of five [1].

2.1 Theoretical Motivation

The experiment that the MOLLER collaboration will be conducting is a scattering experiment. Specifically, they will be scattering a beam of longitudinally polarized electrons off of atomic electrons in a liquid hydrogen target. This process of electron-electron scattering is called Møller scattering, and it is denoted as

$$e^- e^- \longrightarrow e^- e^-$$

.

Imagine taking a cubic chunk of electrons out of both the beam and the target. Suppose the side length of the cube from the beam is ℓ_B and the side length of the cube from the target is ℓ_T. The cube of particles from the beam will travel with velocity v toward the cube from the target; denote the cross-sectional area of overlap of the cubes by A. In addition, both the beam and the target will have a certain number of particles per unit volume. Call these quantities ρ_B (the density of the beam) and ρ_T (the density of the target). We can assume these densities to be constant, since the size of the beam and the target will be much larger than the range of the interaction between the electrons in either [2].

Given these values along with the number N of scattering events that occur, we can define [2]

$$\sigma = \frac{N}{\rho_B \ell_B \rho_T \ell_T A}$$

This quantity is called the *cross section*. It has units of area and is a measure of the likelihood of the interaction between the electrons taking place. The cross section is a property of the particles themselves; it does not depend on, say, the intensity of the beam in any given scattering experiment. As such, it is incredibly important to the study of the behavior of any elementary particles, because it allows results from different experiments to be compared [2].

In Møller scattering, both electromagnetic and weak effects are present. The cross section is proportional to the sum of the contributions from both of these interactions:

$$\sigma \sim \text{EM Contribution} + \text{Weak Contribution} \tag{2.1}$$

When electrons scatter, their momentum changes by some amount q. The electromagnetic interaction is mediated by the (massless) photon, and its contribution to the cross section is inversely proportional to q. The weak interaction, on the other hand, is mediated by the Z boson. Unlike the photon, the Z boson is massive; its mass M_Z is approximately 90 GeV, which is over eight times the 11 GeV beam energy that the MOLLER collaboration plans on using. The contribution of the weak interaction to the cross section is proportional to

$$\frac{1}{M_Z^2}\left(\frac{-1}{1+\frac{q^2}{M_Z^2}}\right) \tag{2.2}$$

Since M_Z is so large, $\frac{1}{M_Z^2} << 1$. Also, the change in momentum $q < M_Z$, so $\frac{q^2}{M_Z^2} < 1$, as well. As a result, the weak contribution is dominated by the electromagnetic contribution. How, then, are we to study the weak interaction experimentally?

The answer to this question is a phenomenon known as *parity violation*, which is related to the helicity of a particle. A particle's helicity is the sign of the projection of its spin onto its momentum vector. If helicity is positive (i.e. the projection of the spin vector is in the same direction as the particle's momentum vector), then the particle is said to be "right-handed". If the helicity is negative, then the particle is said to be "left-handed". The parity transformation P flips the direction of the particle's momentum, but leaves its spin unchanged [4].

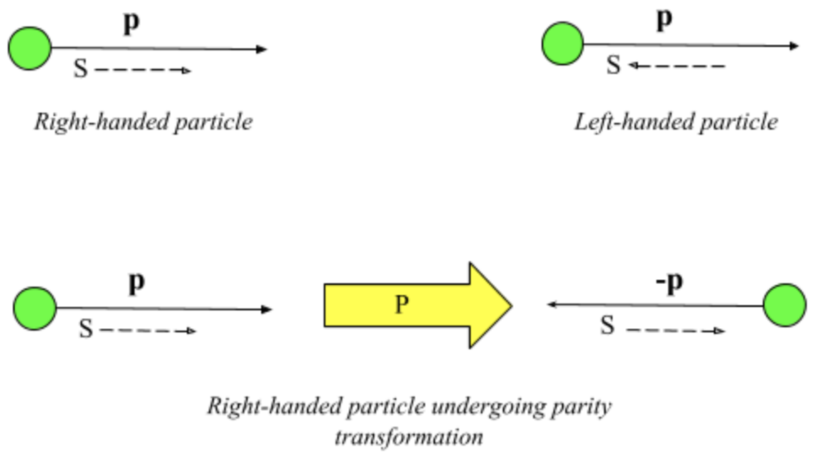


Figure 2.1: Helicity and parity violation - Consider a particle with momentum **p** and helicity S. The top left image shows a right-handed particle; the top right image shows a left-handed particle. The bottom image shows what happens to the right-handed particle when it undergoes a parity transformation.

Parity is a symmetry in the electromagnetic interaction. This means that the electromagnetic interaction is the same before and after a parity transformation. The weak interaction, however, is not symmetric. According to the Standard Model, the weak force only affects left-handed particles. Because of this, in a scattering experiment such as MOLLER, the cross section for right-handed electrons (denoted by σ_R) and the cross section for left-handed electrons (σ_L) are not equal. As a result, we can define [1] [3]

$$A_{PV} \equiv \frac{\sigma_R - \sigma_L}{\sigma_R + \sigma_L}, \tag{2.3}$$

a quantity called the *parity-violating asymmetry.* Parity is easy to implement experimentally, so A_{PV} can be measured which, in turn, allows us to probe the weak interaction and test the Standard Model.

Testing the Standard Model is the true motivation behind the MOLLER experiment. The Standard Model gives a highly accurate (better than 2 parts-per-billion) prediction for the value of the asymmetry, so if the value that MOLLER measures *disagrees* with the Standard Model prediction, it will be strong evidence for the existence of physics beyond the Standard Model (BSM). On the other hand, if MOLLER measures a value which does agree with the Standard Model prediction, it will place a restriction on potential BSM theories.

2.2 Experimental Setup

The proposed setup for the MOLLER experiment consists of the following components [1]:

1. the polarized electron beam, which will be 75 μA with 80% longitudinal polarization;
2. a polarimetry system, which will continuously measure the polarization of the electron beam;
3. a 1.5 m liquid hydrogen target;
4. a precision collimation system, to reduce background noise in the data;

5. a toroidal spectrometer to focus the Møller scattered electrons; and, finally,

6. the detector system, which will consist of:

   (a) four planes of gas electron multiplier (GEM) detectors to track the positions of individual particles,

   (b) the main quartz detector assembly, to measure A_{PV}, and

   (c) pion detectors.

The layout and relative sizes of these components are shown in Figure 2.2.

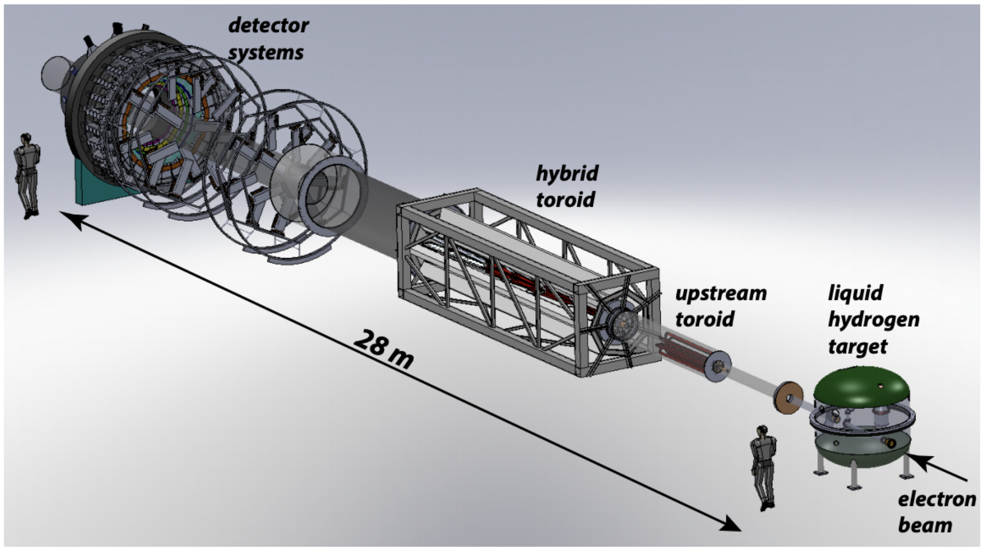


Figure 2.2: Planned layout of the MOLLER experiment - The electron beam (starting in the bottom right corner of the figure) will travel through the liquid hydrogen target, pass through the toroid system, then enter the detector system [1].

Since this project deals with track reconstruction, we are mainly concerned with the detector system — specifically, the four planes of GEM detectors. The planned arrangement of the detector system can be seen in Figure 2.3.

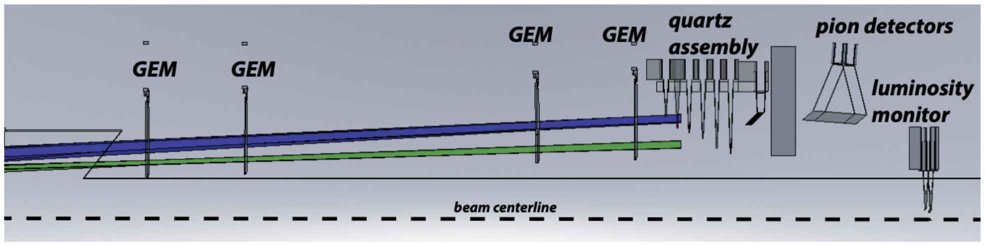


Figure 2.3: Planned layout of the detector system - The electron beam is shown entering the detector system from the left. It passes through the GEM detectors, then enters the quartz assembly. Pion detectors are located downstream of the quartz assembly [1].

As we have already stated, the purpose of the GEM detectors is to track the positions of the particles which enter the detector system. Each GEM consists of a thin piece of plastic which is coated in a layer of metal. The sheet is pierced by an array of tiny, regularly spaced holes. The entire structure is filled with a gas. Applying a voltage across the sheet results in a high electric field in the holes. When a scattered electron passes through one of these holes, it ionizes the gas molecules, resulting in an "avalanche" electrons — in other words, the scattered electron "multiplies" into a cascade of many electrons. The electrons in this avalanche then pass through the multiplication region and are collected. This data allows us to see the position where each scattered electron passes through the detector.

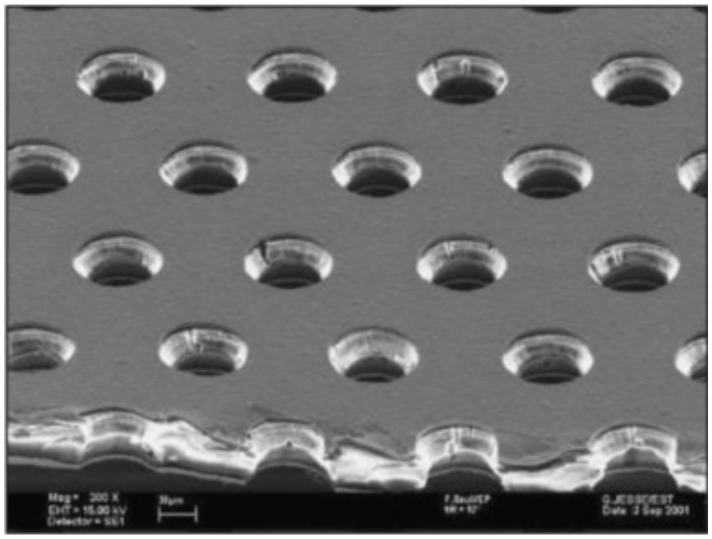


Figure 2.4: The GEM detector - This is an image of the surface of a GEM taken with an electron microscope. The sheet is 50 μm thick and each of the holes is 70 μm in diameter [5].

MOLLER will have a series of four GEM detectors. Ideally, each Møller scattered electron will pass through each of the four GEMs before entering the main detector. This will give a position measurement for each electron at four different locations in the detector system. If we fit these position measurements for any given electron with a path, we will know the trajectory that it followed.

Methods of reconstructing the trajectories of particles from such data currently exist. This project asks whether or not using a neural network to model the electron trajectories in the detector system would be a viable alternative for the MOLLER collaboration. The following chapters describe the creation and implementation of such a neural network using data from a simulated version of the MOLLER experiment. In chapter 3, we give an overview of machine learning. In chapters 4 and 5, we explain the details of our track reconstruction neural network. Chapter 5 also presents data on the network's performance. We conclude the report with suggestions as to how this project can be continued.

Chapter 3

Machine Learning Overview

Machine learning is a field of artificial intelligence which focuses on methods that allow a computer to “learn” about a set of data without being explicitly programmed [6]. Generally, machine learning algorithms are used to identify patterns within large datasets. The algorithms identify a pattern by generating a model, then minimizing the difference between the values given by the model and the actual values in the dataset. Once such an algorithm has created a model, it can be fed new data which it has not been exposed to and used to make predictions about how well the new data fits into the pattern that the algorithm has identified [7].

The first step in developing a machine learning algorithm is to determine the type of prediction you wish to make. This is dependent on the nature of your data and dictates the type of algorithm you can use. An algorithm which makes a discrete prediction is called a *classification* algorithm. An example of a classification algorithm would be one which identifies an image of an animal as either a cat or or a dog. An algorithm which predicts continuous values, on the other hand, is said to be a *regression* algorithm. A machine learning algorithm which predicts the price of a stock is an instance of a regression algorithm [7].

3.1 Basic Structure of Machine Learning Algorithms

Regardless of algorithm type, all machine learning algorithms follow the same basic iterative process [7]:

Step 1: *Training data* is fed into the algorithm. Training data consists of an input dataset and an output dataset. Each input corresponds to an output, but, other than that correspondence, no other information about the data is provided to the algorithm. Put another way, the training data is a complete sample set of data from which the algorithm will attempt to determine the relationship between the input and the output.

Step 2: The algorithm creates a model to describe the training data. The model is a function of a set of parameters. As the training progresses, these parameters are updated with the goal of improving the accuracy of the model.

Step 3: The model uses the training input to compute an output. The model's output is compared to the actual training output which has been given to the algorithm and a quantity called the *loss* is calculated. Loss is a measure of how good or bad the model's prediction is. The lower the loss, the higher the accuracy of the model's prediction. If, for example, the model makes a completely accurate prediction, the value of the loss will be zero.

Step 4: Assuming the model's predictions are *not* perfect, the algorithm attempts to improve the model (i.e. decrease the loss) by updating the model parameters. This process is called *optimization*.

Step 5: The previous steps are repeated until the loss stops changing. At this point, the training is complete. The model can be given a set of unseen input data (i.e. data of the same structure as the training input set) and, ideally, will be able to accurately predict the correct output values. This new set of data is called the *testing data* and it is used to evaluate the model's performance.

This process is summarized in Figure 3.1:

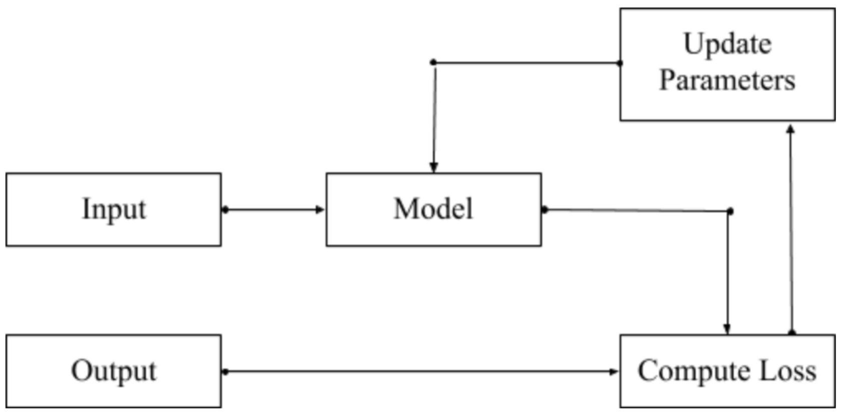


Figure 3.1: Basic structure of a machine learning algorithm - The input is fed into a model, which then predicts the output. The model predictions are compared to the actual output and the error (or loss) is computed. The model parameters are updated according to the value of the error. This cycle of making predictions, calculating the error, and updating the model parameters continues until the error is minimized. At this point, the model is said to be optimal. (Figure adapted from [7])

3.1.1 Training, Testing, and Validation Datasets

Suppose we have a set of data which describes a large group of dogs and cats. Suppose this dataset contains measurements of each animal's height, weight, body temperature, and life span. We wish to develop a machine learning algorithm which predicts whether a particular group of measurements describes a dog or a cat.

First, we must determine what our input data and our output data look like. After our model has been trained, we want it to be able to make a prediction based on a set of measurements we provide it with. The set of measurements which we provide to the model is the input data. In this example, the input data are the height, weight, body temperature, and life span measurements for each of the animals. The output set reflects the information that we want our model to be able to predict. In this case, our output is either "dog" or "cat".

Now that we know what our input and output sets are, we can divide the dataset into a training set and a testing set. We take a certain percentage of inputs and their corresponding outputs to be the training set, and the remaining data to be the testing set. The relative sizes of the training and testing sets can vary, though generally the training set is larger, because this gives the model a wider sample of data to "learn" from.

As described in the previous section, the training set is used to create the model. Creating the model involves tuning the model parameters in response to loss calculations. In order to update these parameters, we take a subset of the training data to test the model on *while it is training* (as opposed to the testing set, which is used to evaluate the model's performance *after* the training has completed). We call this subset the *validation set* [8].

It is important to note that the only information which is actually given to the algorithm are the measurements themselves. The algorithm is not told which measurement is a weight and which measurement is a height. All it knows is that one set of numbers is an input and the other set is an output. This is what distinguishes machine learning from other tools which could be used to solve this same problem.

3.1.2 Loss and Optimization

Now consider any set D of training data that has been fed to a machine learning algorithm. This could be the dog and cat data from before, or any other dataset with which you wish to train a model. Each data point in D consists of an input x (which can be one value or a list of values) and its corresponding output y. Following each iteration of training, the model gives a set of predictions $\{p(x)\}$ of the output for each input x in D. Then, for each input x, the model's prediction $p(x)$ is compared to the actual value y of output corresponding to x. This comparison is called the loss.

The loss is simply an error measurement. It serves as an indication of how well the model is performing and tells the algorithm how it should update the model parameters to improve the accuracy of its predictions. There are many different methods of computing the loss; one of the common methods is computing mean squared error (MSE), which is defined as [7]

$$MSE = \frac{1}{N} \sum_{(x,y) \in D} (y - p(x))^2 \tag{3.1}$$

The value of the loss will change with the parameters of the model. The algorithm's goal is to update the model parameters in order to minimize the loss, a process known as *optimization*. Perhaps the most common method of optimization is the method of *gradient descent*. The simplest gradient descent algorithm does the following [7]:

1. First, a set of initial values is chosen for the model parameters. This model is used to make predictions from which we compute the loss. Call this initial loss the *starting point*.

2. The gradient of the loss function at the starting point is computed.

3. The model parameters are updated by moving along the loss function in the direction of the negative gradient.

4. The previous three steps are repeated until a set of parameters is found for which the loss is minimized.

The process of gradient descent is shown in Figures 3.2 - 3.5:

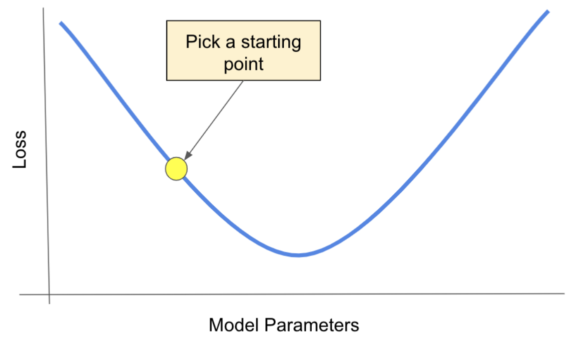


Figure 3.2: Choose a starting point.

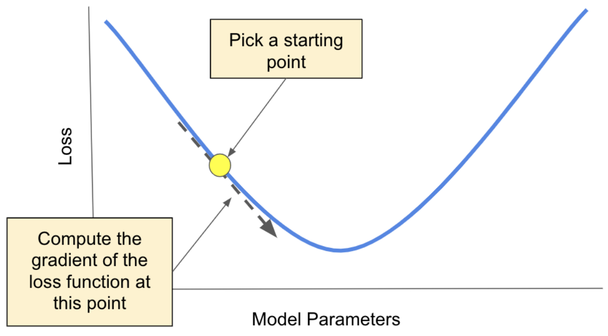


Figure 3.3: Compute the gradient of the loss function at the starting point.

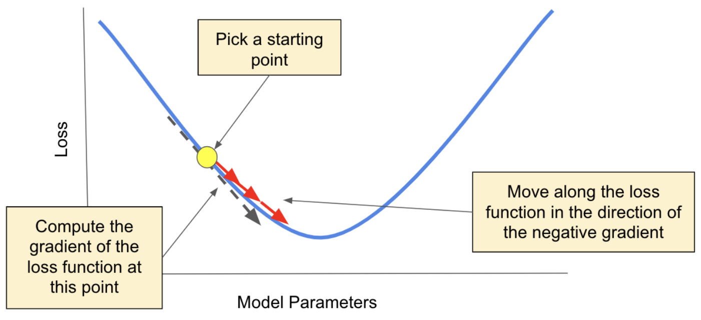

Pick a starting point
Loss
Compute the gradient of the loss function at this point
Move along the loss function in the direction of the negative gradient
Model Parameters

Figure 3.4: Move in the direction of the negative gradient.

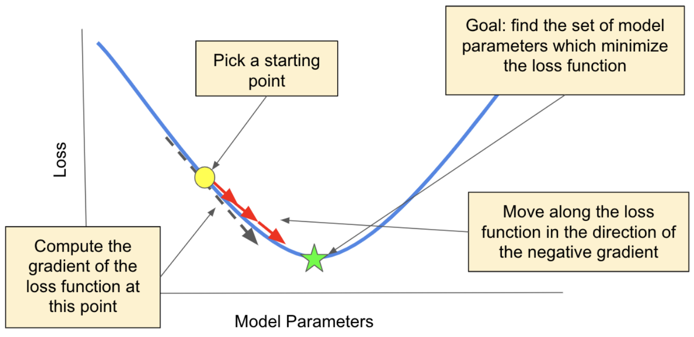

Goal: find the set of model parameters which minimize the loss function
Pick a starting point
Loss
Move along the loss function in the direction of the negative gradient
Compute the gradient of the loss function at this point
Model Parameters

Figure 3.5: Repeat steps 1-3 until the loss is minimized.

Traditional gradient descent algorithms have many weaknesses, namely that they tend to get stuck at false minima. Many variations of gradient descent algorithms have been developed to address such weaknesses [9]. Most machine learning algorithms use one of these variations for optimization.

3.2 Neural Networks

Neural networks are one of the most popular machine learning algorithms. The name "neural network" comes from the algorithm's basic structure, which mimics the structure of neurons in a biological nervous system:

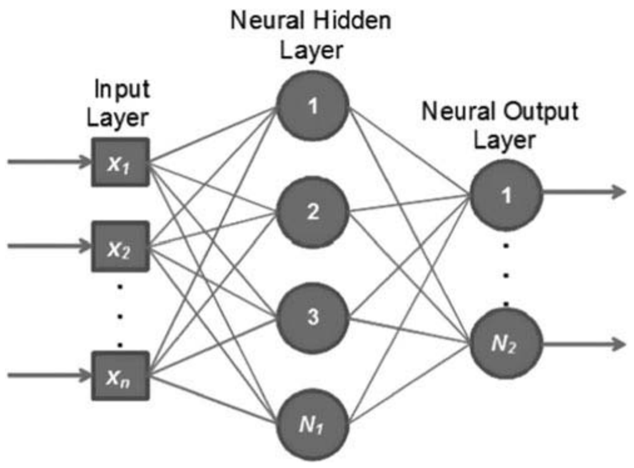


Figure 3.6: Structure of a neural network - A neural network is a collection of variables (called "nodes") which are arranged in layers, similar to the neurons in a biological nervous system. The nodes of each layer are formed from combinations of the nodes in the preceding layers [10].

There are many different types of neural networks which can be used to make various kinds of predictions. All of them work by combining the input values (which are arranged into the nodes shown in Figure 3.6) in order to determine the output values. The simplest neural network will consist only of an input layer and an output layer. Each output in this simple neural network will be given as a linear combination of the values in the input layer. In more complex networks, the input data passes through one or more "hidden" layers, where intermediate values are computed using various (not necessarily linear) functions. These intermediate values are then combined to compute the output.

3.2.1 Epochs and Batch Size

When developing a neural network, there are a few parameters which we can control. These parameters are [11]

(a) **Batch size**: During optimization, the network loops through the training dataset and computes the loss at each data point. Instead of looping through the entire training set, however, we can instead divide the data into smaller "batches". Rather than computing the loss at every point in the training set, the loss is computed at every point in the batch. The batch size is the number of samples in each batch.

(b) **Number of Epochs**: When the algorithm is training, it loops through the training set a certain number of times. This number is called the number of epochs. If the training set has been divided into batches, then within a single epoch, the algorithm will iterate through each batch and update the model parameters.

For example, say we have a training set that has 100 total samples (i.e. 100 sets of inputs and their corresponding outputs). If we choose a batch size of 1, then we will have 100 total batches, each containing a single sample. During *each* epoch, the network will compute the loss at each of the 100 batches. If we choose a larger batch size — for instance, a batch size of 20 — then we will have 5 total batches, each containing 20 samples. Rather than computing the loss 100 different times during each epoch, the network will compute the loss 5 times per epoch [11].

The number of epochs and the batch size are completely dependent on the individual network. Changing these parameters impacts both the accuracy of the model that is created and the time taken to train the model. The values of these parameters are generally chosen through trial and error [11].

3.2.2 LSTM Networks

Long short term memory (LSTM) networks are a type of recurrent neural network (RNN). RNNs consist of chains of repeating units which allow them to "remember" previous information and use that information to inform later decisions. However, standard RNNs have trouble dealing with data that has long term dependencies. LSTMs were developed to deal with this issue. The difference between a standard RNN and an LSTM lies in the structure of their repeating units. Where the repeating unit of a basic RNN will consists only of a single activation function layer (tanh, for example), the repeating unit of an LSTM consists of five different layers. Figure 3.7 shows the structure of an LSTM unit [12]:

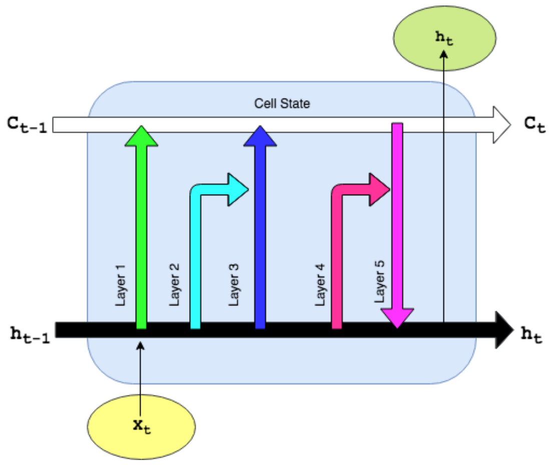


Figure 3.7: Repeating unit in a LSTM network - The structure of a LSTM network allows the algorithm to "remember" information over time. Information from preceding units is passed into the current unit and used to update the so-called "cell state", which is passed on to the next unit. (Adapted from [12]).

The horizontal white arrow across the top of the diagram represents the cell state of the unit. The previous unit's cell state, C_{t-1}, enters the current cell state. As it passes through the unit, the five layers (indicated by the colored arrows) update the cell state with information from the previous unit's output (h_{t-1}) and the current unit's input (x_t). The updated cell state, C_t, determines the current unit's output,h_t, and both of these values are passed on to the next unit, where the process repeats [12].

Each layer in the LSTM performs a different task [12]:

- Layer 1 is called the "forget gate". A sigmoid function $\left(\sigma(y) = \frac{1}{1+e^{-y}}\right)$ takes $1 + e^{-y}$ values from the input x_t and the previous unit's output h_{t-1} and returns a vector f_t of values between 0 and 1. These values indicate how much of the information to "forget", with 0 meaning "forget everything" and 1 meaning "keep everything". These values are multiplied by the values in C_{t-1}.

- Layer 2 is the "input gate". Again, a sigmoid function takes x_t and h_{t-1} and outputs a vector i_t of values between 0 and 1. This vector is then multiplied by the output of layer 3, which is a hyperbolic tangent layer. It takes the same inputs as layer 2, and returns a vector p_t of values between -1 and 1. This vector represents possible updates to the cell state; in a similar manner to the forget gate, the network decides which potential updates to keep by multiplying p_t by i_t. The update is added to the product of the previous cell state and the forget gate to create the new cell state, C_t.

- Layers 4 and 5 determine the output of the unit, h_t. A final sigmoid function in layer 4 outputs a vector o_t of values between 0 and 1. In layer 5, the new cell state C_t passes through a tanh function. The unit output h_t is the product of o_t and $\tanh(C_t)$. h_t is sent to the next unit as well as sent out of the LSTM chain.

Chapter 4

Overview of Simulated Data

In order to develop our neural network, we required a dataset. Since the MOLLER experiment has not yet been conducted, we used data generated by the GEANT 4 Monte Carlo Simulation for the MOLLER experiment. The dataset contains simulated measurements from a series of five virtual detector planes which play the same role as the GEMs and the main quartz detector will in the actual experiment. The first four virtual detector planes are tracking planes, corresponding to the GEM detectors; these are located upstream of the fifth virtual detector plane, called the main detector plane, which corresponds to the quartz detector which will measure the A_{PV}. This arrangement is shown in Figure 4.1:

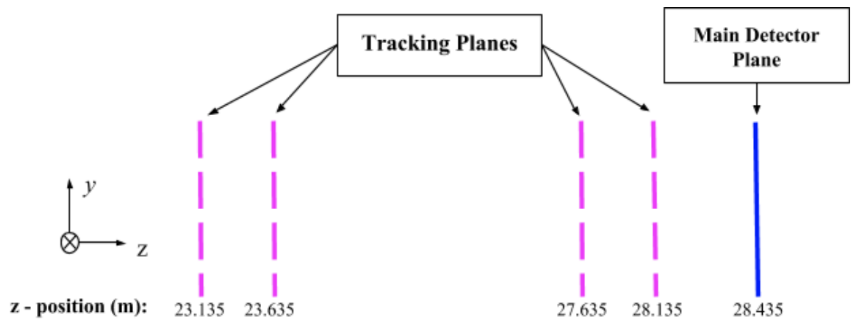


Figure 4.1: Virtual tracking and main detector planes - This is the arrangement of the virtual tracking and main detector planes in the GEANT 4 simulation for the MOLLER experiment. The simulated electron beam travels from the left to the right along the z axis. The locations of the virtual tracking planes and main detector planes correspond to the planned locations of the GEM and quartz detectors for the experiment (see Figure 2.3).

The set of the four virtual tracking planes is labeled as a single "Detector 30" in the simulation. The virtual main detector plane is labeled "Detector 28". Geometrically, these detectors are identical: when viewed head on, each of them is a circle with a radius of 2 meters (see Figure 4.2). They are arranged along the z axis in the x, y plane. The simulated electrons travel in the $+z$ direction, starting at the far left ($z = 23.15$ m) and ending at the main detector plane ($z = 28.435$ m). There is no magnetic field in the simulated detector system, but there *is* air, which means the simulated electrons can lose energy as they move.

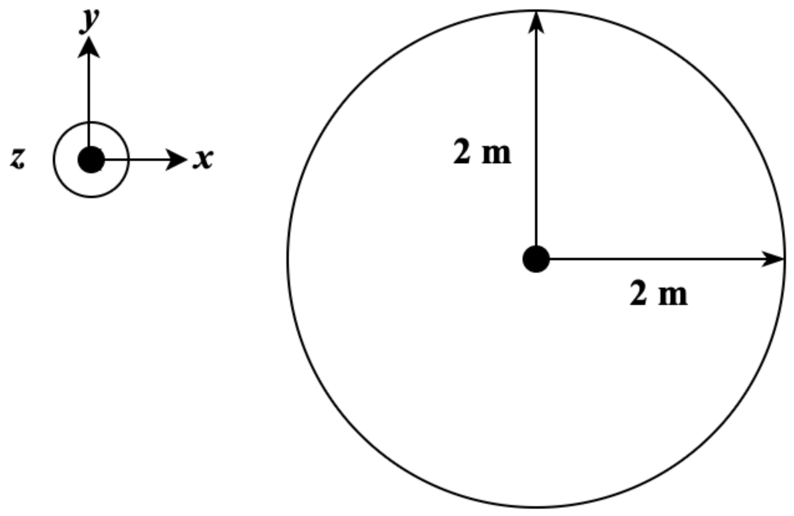
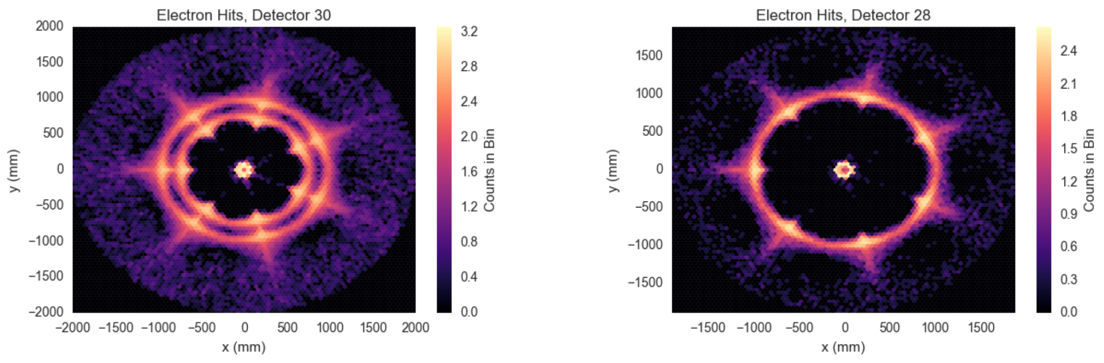


Figure 4.2: Dimensions of the virtual detectors - Each of the virtual detectors is a circle with a radius of two meters.

The data generated by the simulation were from Møller scattered electrons from an 11 GeV beam on a liquid hydrogen target. The dataset contains measurements for 100,000 separate events. A new event was defined each time a new electron is sent into the virtual detector system. This initial electron was labeled a *primary* electron. Each individual event contained all of the information about all of the processes that occurred as a result of each primary electron. This information included measurements of the positions where each electron passed through each detector plane.

Each time a simulated electron traveled through one of the virtual detector planes, a position measurement was recorded in the form of x, y, and z coordinates, in units of millimeters. We call such an occurrence a *hit*. The distribution of electron hits in the virtual tracking planes and the main detector plane is displayed in Figure 4.3:



(a) Virtual Tracking Planes **(b)** Main Detector Plane

Figure 4.3: Distribution of electron hits in virtual detectors - These histograms show the distribution of electron hits in (a) the four virtual tracking planes and (b) the main detector plane over all 100,000 events. The majority of the hits are located in ring(s) at a radius of approximately one meter. These are (primarily) the Møller electrons. The two-ring structure in figure (a) occurs because the 4 meter separation between the first two tracking planes and the last two (see figures 4.1 and 4.4) has been flattened into one dimension.

The locations on the virtual detector planes with the highest concentration of electron hits are the center (where the unscattered beam passes through) and the ring(s) at a radius of approximately one meter. Recall the setup of the MOLLER experiment (Figure 2.2): after the Møller scattering occurs in the liquid hydrogen target, the electrons pass through a toroidal spectrometer. The toroidal spectrometer focuses the Møller electrons onto ring with a radius of about a meter [1]. The ring(s) of hits on the virtual detector planes are the ring(s) of (predominantly) Møller electrons. The two separate rings shown in Figure 4.3(a) are a result of "flattening" the hits from the four separate virtual tracking planes onto a single image.

Figure 4.4 shows these same hit distributions as a three-dimensional scatter plot. Here, the hit distributions in each of the five detector planes can be seen.

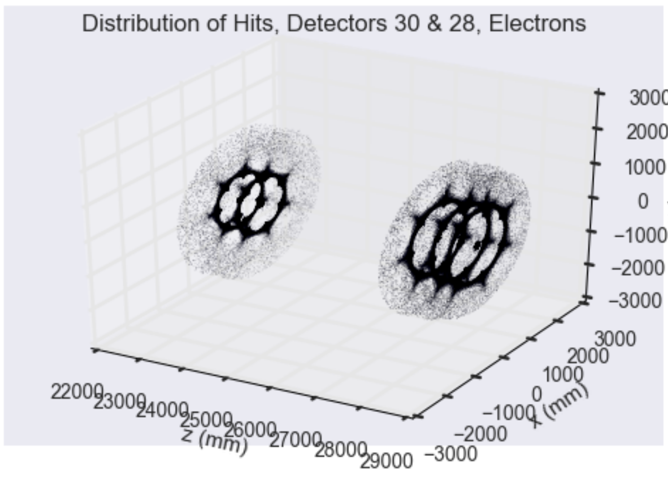


Figure 4.4: Distribution of electron hits in virtual detectors (3-dimensions) - This scatter plot shows the distribution of electron hits in all five virtual detector planes over all 100,000 events. The two leftmost planes are the first two tracking detectors; the three on the right are the final two tracking detectors followed by the main detector. Again, the highest concentration of hits in each detector occurs in a ring at a ~ 1 m radius.

Each simulated electron is associated with a series of hit position measurements. If we connect the hits for any given electron, we form a *track*. An example of a track is shown in Figure 4.6:

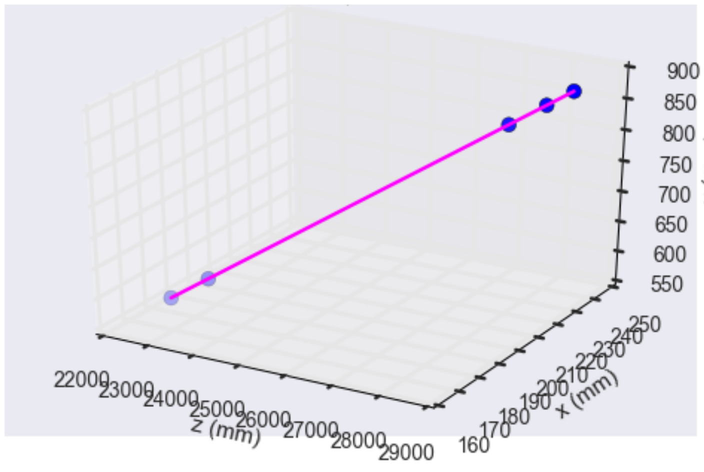


Figure 4.5: Example of an electron track - The electron travelled from left to right in the positive z direction. Each circle marks the location (the (x, y, z) position, a.k.a. the *hit*) where the electron passed through a virtual detector plane. For simplicity, we only consider electrons which pass through all five virtual planes. The *track* is formed by connecting the five hits to form a path.

The sample track in Figure 4.6 is an ideal case in the sense that the electron hits each of the five virtual detectors. This was not true of all of the electrons in the dataset — for instance, some of the electrons only had hits recorded on two of the five virtual detector planes. Also, it is important to note that not every track was formed from a primary electron. The primary electrons are the *first* electrons in a given event to enter the simulated detector system, but this does not mean they are the *only* electrons in said event. Various reactions can occur within the detector system, and this is accounted for in the simulated data. For consistency and simplicity, however, we considered only the tracks which (1) were formed directly by primary electrons and (2) intercepted each of the five detectors once (and only once).

Chapter 5

Track Reconstruction Network

We created a LSTM network which takes in a given electron's hits from the four virtual tracking planes, fits those hits to a track, then "projects" that track onto the virtual main detector plane. This "projection" is a prediction of the hit position on the virtual main detector plane for that electron, assuming said electron continues to follow the same path. This network allows the data from the virtual tracking planes to be connected with the data from the virtual main detector. If we match the network's prediction to actual measured hits on the main detector plane, we can determine the track that any given electron "belongs" to. This, in turn, gives us information about the electron's behavior from the time it entered the virtual detector system until the time it hits the main detector plane.

5.1 The Baseline Network

Creating the best version of a neural network requires quite a bit of fiddling with the various network parameters; for clarity, we will describe what we will call our "baseline" network. Our network was created using Python and Keras, an Application Programming Interface (API) built on top of TensorFlow (Google's machine learning library). See [13] and [14] for more information.

The dataset used to develop our network contained track data for primary electrons which hit each of the five detector planes one time (see Chapter 4 for more details). We used 80% of the dataset for training and 20% for testing. The input values were the first four hit measurements in each track, corresponding to the hits in the four virtual tracking planes. The output values were the final hit measurements in each track, corresponding to the main detector hits. Since the hit data was given component-wise, we made separate (but identical) networks and datasets for each component of the measurements.

In order to be fed into our network, the data required some reformatting. The first step involved splitting the dataset into the desired training and testing sets, and then further into the respective input and output sets. Now, recall from the previous chapter that all of the hit data was given in units of millimeters. The virtual detector planes have a radius of 2 m, or 2000 mm; as a result, the data consisted of numbers ranging anywhere from 0 to 2000. The network does not know the units that the data is in, so it only sees this wide range of values. This can make optimization both slow and difficult. To address this issue, we rescaled all of the data so that it normally distributed with a mean of 0 and a standard deviation of 1.

Keras also requires the input data arrays to have specific dimensions. The original input arrays had 4 columns (for each of the four virtual tracking plane hits) and n rows, where n was the number of tracks in the training / testing set. We reshaped the input arrays so they had the following dimensions: (# of samples, # of timesteps, # of features). The number of samples is the number n of tracks in the dataset. The number of timesteps is the number of columns in the original array (so 4, in this case). Each "timestep" contained a single measurement (the hit position), so the number of features in this situation was 1. We created a function in Python which performs all of this reformatting, including the splitting into training and testing sets; this can be seen in Appendix A.

Below we present the code which builds and trains our baseline network:

```
model = Sequential()
model.add(LSTM(32, input_shape=(4,1)))
model.add(Dense(1, activation='linear'))
model.compile(loss='mse', optimizer='adam',␣metrics=['mse'])
history = model.fit(training_input_model, training_output_model,
        batch_size=100, epochs=100, validation_split=0.1,
        verbose=1)
```

The first line, `model = Sequential()`, initiates the construction of our network. It indicates that we want our network to consist of a linear stack of layers. The `model.add()` function adds layers to our network. We have two layers: the input layer, which is also the LSTM layer, and the output layer. The LSTM layer takes in the following arguments:

- The number of LSTM nodes, or cells, also called the dimensionality; this number is up to us to choose
- The shape of the input array, which is a vector of the form (# of timesteps, # of features)

The output layer is "dense", which indicates that it is a so-called "fully-connected" layer. This means that every node in this layer is connected to every node in the previous layer. This arguments that this layer takes in are

- The number of output nodes; this number must be 1, since we want the network to predict a single number
- The "activation" function, which indicates how we want the network to combine the inputs in order to predict the outputs. We chose a linear activation function, since the tracks are straight lines.

Next, the `model.compile()` function defines the model's learning process. We choose a loss function (`loss='mse'` means that we chose mean squared error) and an optimizer

(`optimizer='adam'` refers to the Adam optimizer, which is a popular variant of gradient descent — see Appendix C). `metrics=['mse']` indicates that we want the network to save the mean squared error values from each epoch. This allows us to, for instance, plot the mean squared error per epoch in order to see how quickly the loss decreases.

Finally, the `model.fit()` function trains our model. We give it an input set (`training_input_model`) and an output set (`training_output_model`). Then, we specify the batch size (100), the number of epochs (100), and the percentage of the training data that we would like to use as a validation set (10%). `verbose=1` will print the loss after each epoch when the code is run.

The following sections detail our network's performance.

5.2 Results

We wanted our network to be both efficient and accurate. As such, we required methods of quantifying both efficiency and accuracy. For efficiency, we chose to study the (real) time taken by the network to train. We did this using Python's `timeit` function. For accuracy, we evaluated the model's performance on the testing set. In order to do this, we defined

$$\delta = \frac{1}{N} \sum_{(x,y)\in D} |y - p(x)| \tag{5.1}$$

Here, D is the testing set, (x, y) represents a single sample in D, N is the total number of samples in D, and $p(x)$ is the model's prediction at input value x. δ, therefore, is the average absolute difference between the actual output values in the testing set and the model's prediction of these output values. For our purposes, we consider $\delta \leq 1$ mm to mean the model's predictions are accurate.

Our baseline network, given in the previous section, took 47.48 s to train and had a δ value of 1.3 mm. This means that the model's predictions were fairly close to the actual output values of the testing set. In order to better visualize the model's accuracy, we made a histogram of the absolute difference between the actual output values and the model's predictions at each sample in the data set (Figure 6.1):

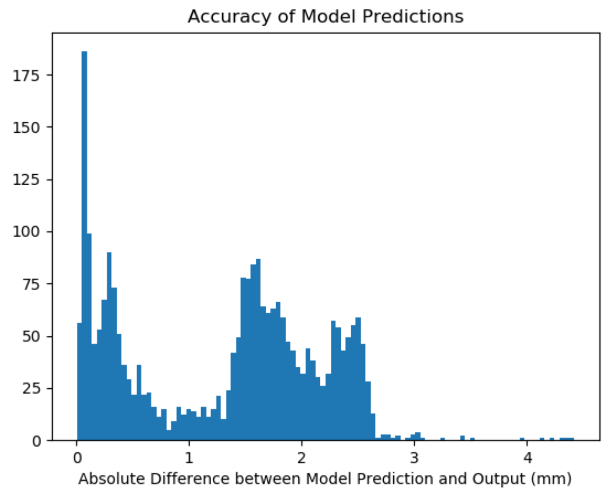


Figure 5.1: Accuracy of baseline network - Most of the model's predictions differed from the actual output values by about 0.2 mm.

The following two sections describe how various changes to the network and the data impacted efficiency and accuracy. For consistency, the same set of tracks was used in each case. This set contained data for 13,000 total tracks. We took 80% of this 13,000 (10,400 tracks) to be the training set, and the remaining 20% to be the testing set.

5.2.1 Impact of Network Parameters

In our network, we are able to control three major parameters: the batch size, the number of epochs, and the number of nodes (also called the dimensionality) in the LSTM layer. We wanted to determine how each of these parameters impacted the efficiency and accuracy of the model when the other two parameters were held constant.

Impact on Efficiency

1. **Batch Size**: As the batch size increases, the number of batches with which the network computes the loss during each epoch decreases. This means that the number

of times that the network updates the model parameters per epoch also decreases. As such, we expect the network to be more efficient as the batch size grows. Figure 6.1 shows that this is indeed the case:

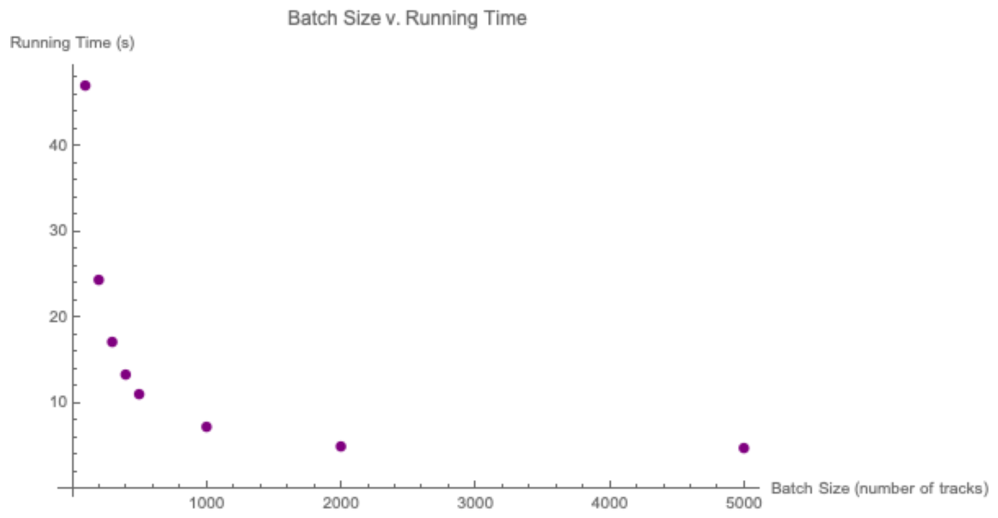


Figure 5.2: Impact of batch size on efficiency - As the batch size increases, the time taken to train the model decreases rapidly.

At the smallest batch size (100), the running time was $\sim$ 47 seconds. At the largest batch size (5000), the running time was less than 5 seconds. Clearly, the batch size has a strong impact on the model efficiency.

2. **Number of Epochs**: As the number of epochs increases, the number of times that the model loops through the entire training set increases. Intuitively, then, the number of epochs should be directly proportional to the running time. Figure 6.2 confirms this hypothesis:

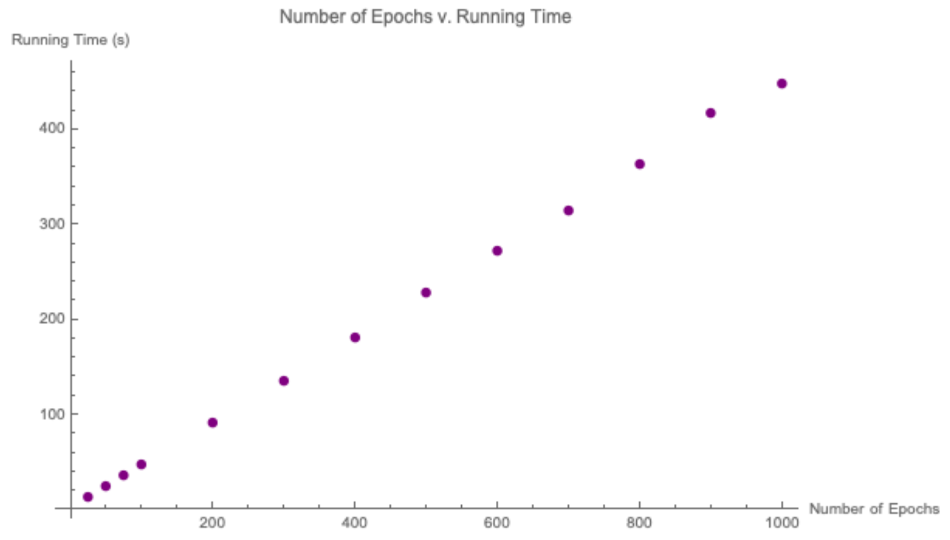


Figure 5.3: Impact of number of epochs on efficiency - As the number of epochs increases, the time taken to train the model increases linearly.

3. **LSTM Dimensionality**: The dimensionality of the LSTM layer is the number of LSTM cells, or nodes. As we increase the dimensionality, we see that the running time increases, as well (Figure 6.3):

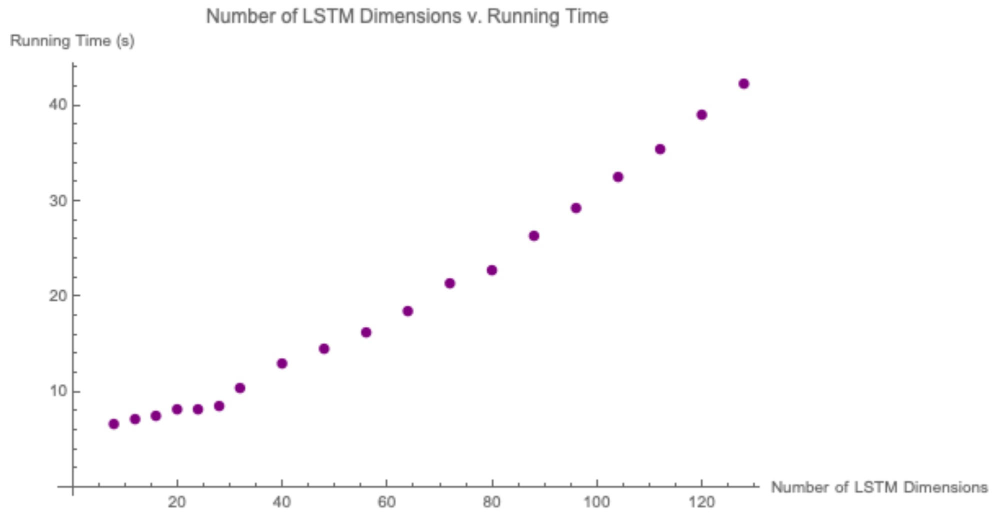


Figure 5.4: Impact of LSTM dimensionality on efficiency - As the dimensionality of the LSTM layer increases, the time taken to train the model increases.

While this relationship appears to be almost linear, there are several small fluctuations, suggesting that the relationship between the LSTM dimensionality and the model efficiency is not as direct as the batch size / number of epochs relationships.

Impact on Accuracy

1. **Batch Size**: As we increase the batch size, the value of δ increases, then stays relatively constant, and then increases once more (Figure 6.4):

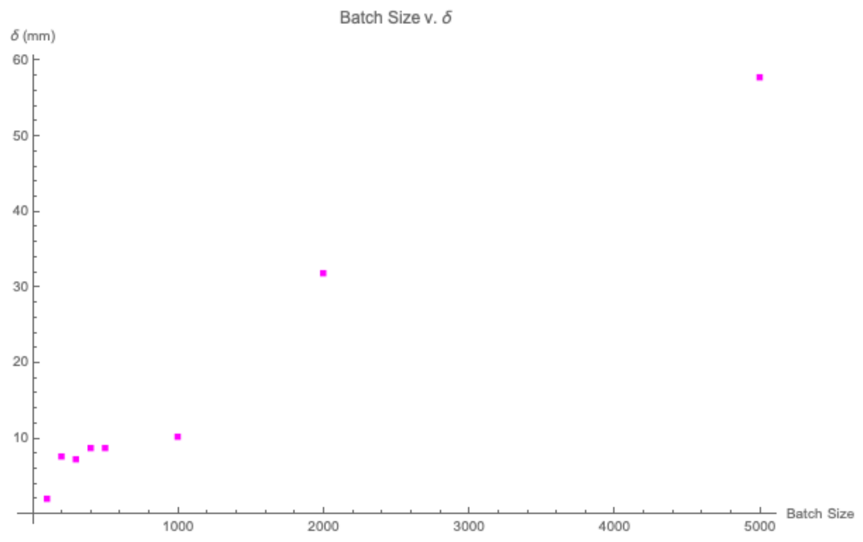


Figure 5.5: Impact of batch size on accuracy - As batch size increases, δ increases overall. This increase does not, however, appear to be consistent.

2. **Number of Epochs**: As we increase the number of epochs, the number of times that the training data is fed through the model increases. The more the model "sees" the dataset, the more likely it is to pick out the correct relationship between the inputs and the outputs. Thus, we expected δ to decrease with increasing number of epochs. This is exactly what we found (Figure 6.5):

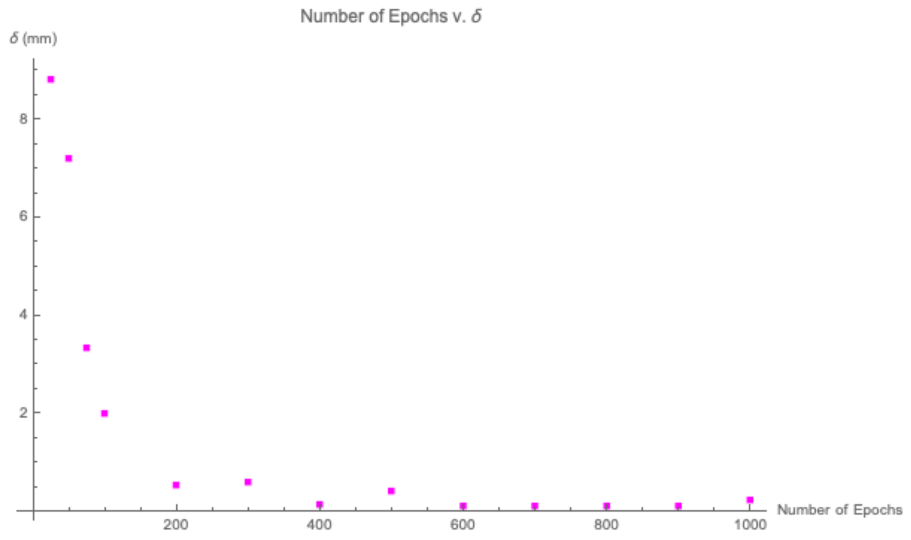


Figure 5.6: Impact of number of epochs on accuracy - As the number of epochs increases, the value of δ decreases rapidly, which means the accuracy increases.

3. **LSTM Dimensionality**: We did not find a clear correlation between the dimensions in the LSTM layer and the value of δ. As shown in Figure 6.6, δ just appears to fluctuate:

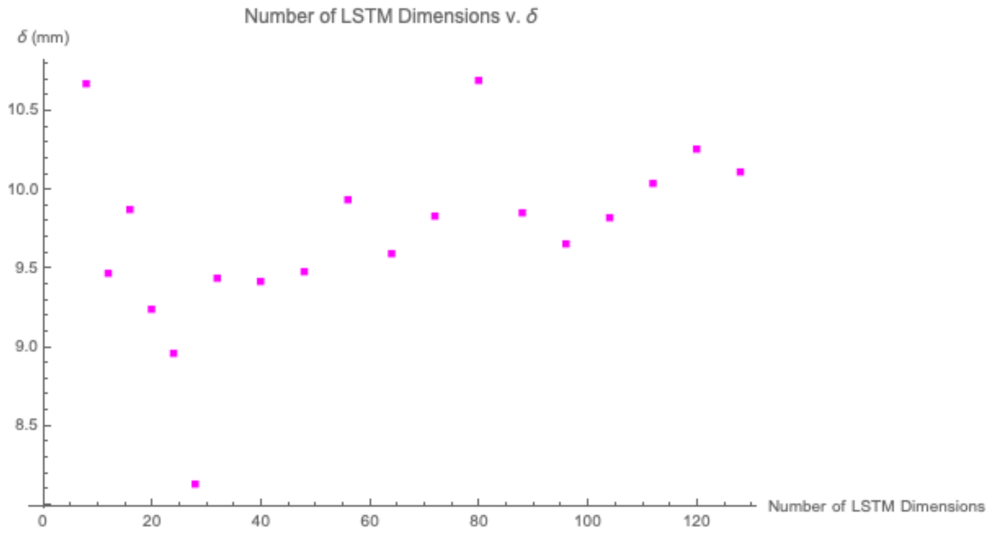


Figure 5.7: Impact of LSTM dimensionality on accuracy - We were unable to determine a clear relationship between the number of LSTM dimensions and the value of δ

As described above, while we found a clear relationship between the number of epochs and the accuracy of the model, we were unable to determine any meaningful correlation between the batch size / the LSTM dimensionality and the accuracy. In order to better understand these results, we needed to understand how much δ fluctuates when every network parameter is held constant. We trained the network ten times with the same batch size, number of epochs, and LSTM dimensionality. The results from this test are shown in Figure 6.7:

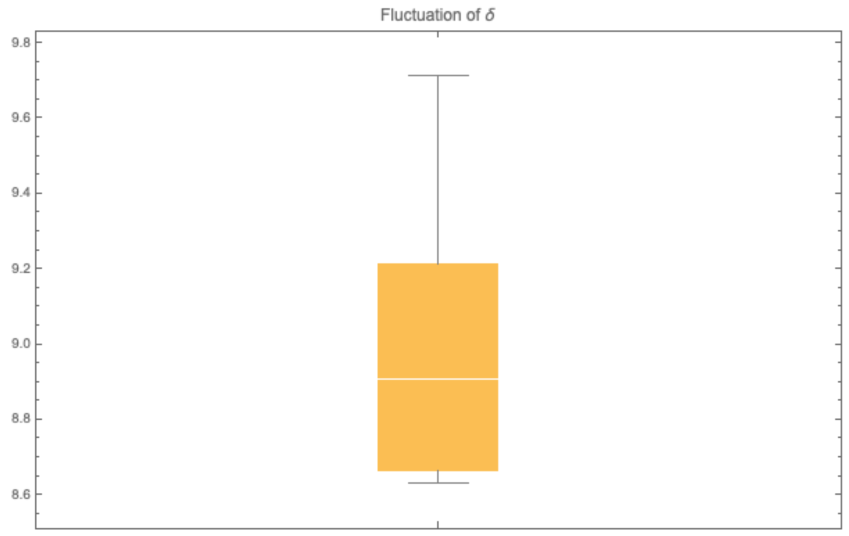


Figure 5.8: Fluctuation of δ - Over ten repeated trials with every network parameter held constant, there was a maximum fluctuation of approximately 1 mm in the value of δ

This plot shows the range of δ values over the ten repeated trials, indicated by the thin vertical bar. The orange boxes represent the middle 50% of the data. We found that the value of *delta* fluctuated by 1.08 mm. This is likely because different batches of training data were selected randomly by TensorFlow in each training cycle, causing the accuracy to vary somewhat. Based on this, the fluctuations seen in Figure 6.6 are most likely due to the natural fluctuations of δ.

Adding Detector Resolution

The network performed well on the simulated MOLLER data. However, the simulated data represents an idealized situation. The data is "perfect" in the sense that it does not include any of the uncertainty or noise that will be present in the actual experimental data. As a result, the fact that the network performs well on the simulated data does not necessarily imply that it would perform well on real data.

One such source of uncertainty is the resolution of the detectors. Each of the detectors in the MOLLER experiment will have a certain precision. The hit position measurements will not be exact points, but will instead be within a certain range of values. This means that the hits will not necessarily form perfectly straight tracks. We were able to add detector resolution to the simulated data by shifting the hit positions by a chosen amount. We first chose a particular detector resolution σ. Then we mapped each hit in the dataset to a shifted hit position based on σ. This shifted hit position was chosen randomly from a Gaussian distribution with the hit position as its mean and σ as its standard deviation.

First, we shifted the training dataset, and left the testing set untouched. We did this for a range of σ values. Then we did the opposite, shifting the testing set and leaving the training set untouched. In both cases, we computed δ at each σ in order to gauge the accuracy of the model. The results are displayed in Figure 6.8:

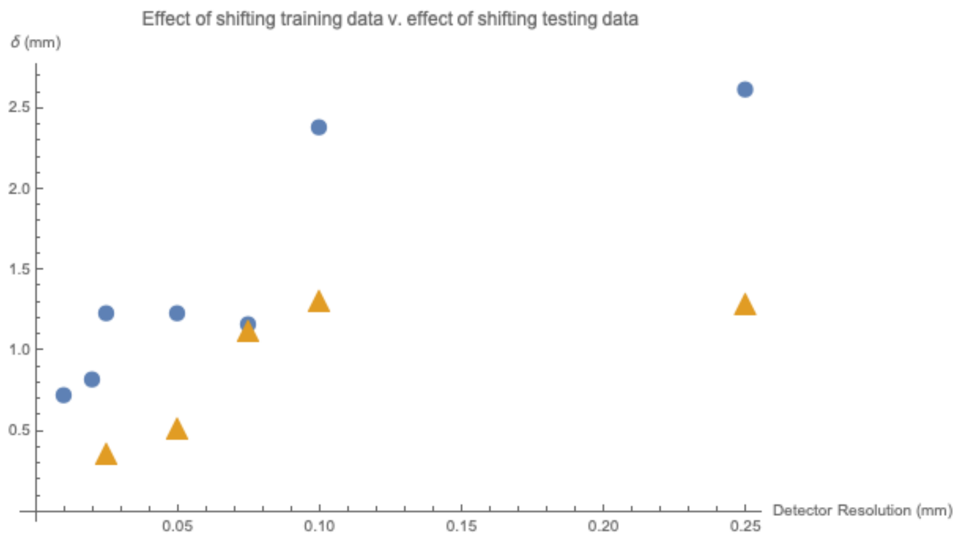


Figure 5.9: Impact of detector resolution on model accuracy - The circles are the data from shifting the training set, and the triangles are the data from shifting the testing set. At each point, δ is higher when the training set is shifted than when the testing set is shifted. In both cases, δ increases with increasing detector resolution.

Shifting the training data resulted in higher δ values at each σ than shifting the testing data did. Focusing on the data from shifting the testing set, we see that δ is less than 1.5 mm at each point. This means that when the model is trained on ideal data and then tested on data that contains some uncertainty, it is still capable of making accurate predictions.

Chapter 6

Conclusion and Outlook

The goal of this project was to determine whether or not a neural network could be used as a tool for track reconstruction in the upcoming MOLLER experiment. We were able to construct a LSTM recurrent neural network which predicts the final hit position in an electron's track with relatively high accuracy. Our current results indicate that neural networks have the potential to perform very well as methods for track reconstruction.

However, there are still many ways we could continue to test and improve the performance of our network. While we addressed the issue of detector resolution, there are still other, unaddressed sources of error and uncertainty that will be present in the MOLLER data which could impact the network's accuracy. These include background noise from other processes that occur in the detector system as well as detector inefficiency. If we are to use a neural network in an actual experimental setting, it needs to be able to make accurate and efficient predictions about realistic data.

In addition to this, we would also like to continue to study how the network parameters impact performance. Ideally, we want to quantitatively determine which combination of parameters allows for the fastest and most accurate predictions to be made.

Overall, we believe this project was a success and we hope to continue to study neural networks as potential tools for track reconstruction.

Appendix A

Data Processing Function

This function organizes and splits a set of tracks into training and testing sets. It scales and reshapes these datasets for use in training and testing the model. This function can also add detector uncertainty to both the training and the testing sets.

```
# Description: Processes data for training / testing the model
# Last Updated: 03/16/2019
# Author: Mary Robinson

def process_data(file_name, training_percentage, detector_resolution_training,...
 detector_resolution_testing):

import numpy as np
import sklearn
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()

# first - loads the full data file

data = np.loadtxt(file_name)

# splits the list of hits into sets of tracks
# (NOTE: assumes each track consists of 5total hits;
# does not work for tracks of variable length)

tracks = np.split(data, len(data)/5)
tracks = np.stack(tracks)

# splits the list of tracks into training and testing sets
#for the reconstruction algorithm based on the percentage of
#data the user wishes to train the algorithm on (training_percentage)

number_training_samples = int(round(len(tracks)*training_percentage))

training_data = tracks[0:number_training_samples]
testing_data = tracks[number_training_samples:len(tracks)]

# Adding detector resolution
```

```
Shifted_training_data = []

for hit in training_data:
mu = hit
shifted_hit = np.random.normal(mu,detector_resolution_training)
Shifted_training_data.append(shifted_hit)

Shifted_training_data = np.array(Shifted_training_data)

Shifted_testing_data = []

for hit in testing_data:
mu = hit
shifted_hit = np.random.normal(mu,detector_resolution_testing)
Shifted_testing_data.append(shifted_hit)

Shifted_testing_data = np.array(Shifted_testing_data)


# then splits the training and testing sets into input sets /
# output sets (i.e. first four hits / last (predicted) hit)

training_input = Shifted_training_data[:, 0:4]
training_output = Shifted_training_data[:,4:5]

testing_input = Shifted_testing_data[:,0:4]
testing_output = Shifted_testing_data[:,4:5]

# Scalers

training_input_scaler = scaler.fit(training_input)
training_output_scaler = scaler.fit(training_output)

testing_input_scaler = scaler.fit(testing_input)
testing_output_scaler = scaler.fit(testing_output)

# reshapes and scales input sets

training_input_scaled = training_input_scaler.transform(training_input)
testing_input_scaled = testing_input_scaler.transform(testing_input)

training_input_model = training_input_scaled.reshape(len(training_input_scaled), 4, 1)
testing_input_model  = testing_input_scaled.reshape(len(testing_input_scaled), 4, 1)

# reshapes output sets for use in algorithm

training_output_scaled = training_output_scaler.transform(training_output)
testing_output_scaled = testing_output_scaler.transform(testing_output)

training_output_model = training_output_scaled.reshape(len(training_output_scaled),1)
testing_output_model  = testing_output_scaled.reshape(len(testing_output_scaled), 1)

return data, tracks, training_data, testing_data, training_input, training_output,...
 testing_input, testing_output, training_input_model, training_output_model,...
 testing_input_model, testing_output_model, testing_output_scaler
```

Appendix B

Track Reconstruction Network

This is the full track reconstruction network. It is written in Python using Keras. This includes the network itself, as well as the necessary code to train and test the model.

```
import keras
from keras.models import Sequential
from keras.models import load_model
from keras.layers import Dense
from keras.layers import LSTM
import track_recon_data_processing #see Appendix A
import numpy as np
import timeit

# Use the data processing function to load the dataset

data, tracks, training_data, testing_data, training_input, training_output,...
 testing_input, testing_output, training_input_model, training_output_model,...
 testing_input_model, testing_output_model, testing_output_scaler =...
track_recon_data_processing.process_data("13000_tracks.csv", 0.80, 0.0, 0.0)

# Builds the network

model = Sequential()
model.add(LSTM(32, input_shape=(4,1)))
model.add(Dense(1, activation='linear'))
model.compile(loss='mse', optimizer='adam', metrics=['mse'])

# Train the network

tic = timeit.default_timer() # This will time how long the training takes

history = model.fit(training_input_model, training_output_model, ...
batch_size=100, epochs=100, validation_split=0.1, verbose=1)

toc = timeit.default_timer()

# Plotting MSE / Epoch (optional)

fig1 = plt.figure()
plt.plot(history.history['mean_squared_error'])
plt.title('Mean Squared Error (baseline)')
plt.xlabel('Epoch')
plt.ylabel('Mean Squared Error')
fig1.savefig('model_mse.png')


# Testing the model

prediction = model.predict(testing_input_model, verbose=1)
prediction = testing_output_scaler.inverse_transform(prediction)

# Evaluating the model's performance on testing set

prediction_list = list(prediction)
testing_output_list = list(testing_output)

difference = [0]*len(testing_output_list) # Computes absolute difference

for i in range(0, len(testing_output_list)):
difference[i] = abs(prediction_list[i] - testing_output_list[i])

difference = np.array(difference)

# Plots absolute difference

fig3=plt.figure()
plt.hist(difference, bins=100)
plt.title('Accuracy of Model Predictions')
plt.xlabel('Absolute Difference between Model Prediction and Output (mm)')
fig3.savefig('model_acc.png')

# Calculates and prints the average absolute difference (called delta in the report)
# and the time taken to train

print("mean difference: ", np.mean(np.array(difference)))
print("time: ", toc - tic)
```

Appendix C

The Adam Optimizer

The amount by which a gradient descent algorithm adjusts the model parameters as it moves along the loss function is called the *learning rate*. Choosing an appropriate learning rate presents a challenge: if the learning rate is too large, the algorithm could overstep the minimum point, and if the learning rate is too small, the algorithm will approach the minimum extremely slowly.

The Adam (short for adaptive moment) optimizer [9] is a variant on gradient descent optimization which updates the learning rate as optimization progresses. It works by keeping track of the mean (also called the "first moment") and the uncentered variance (the "second moment") of the gradient of the loss function.

Consider the following:

- $E(w_t)$, the loss as a function of the model parameters w_t at iteration t of training;
- g_t, the gradient of $E(w)$ at iteration t
- m_t, the estimate of the first moment of the gradient at iteration t;
- s_t, the estimate of the second moment of the gradient at iteration t;
- η_t, the learning rate at iteration t;
- β_1, β_2, the "forgetting factors", typically equal to 0.9 and 0.99, respectively;
- ϵ, typically equal to 10^{-8}, a factor intended to prevent division by 0

The estimates of the first and second moments of the gradient are given by

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

and

$$s_t = \beta_2 s_{t-1} + (1 - \beta_2) g_t^2$$

.

The algorithm then computes

$$\hat{m_t} = \frac{m_t}{1 - \beta_1^t}$$

and

$$\hat{s_t} = \frac{s_t}{1 - \beta_2^t}.$$

The model parameters are then updated by

$$w_{t+1} = w_t - \eta_t \frac{\hat{m_t}}{\sqrt{\hat{s_t}} + \epsilon}.$$

Bibliography

[1] MOLLER Collaboration (J. Benesch (Jefferson Lab) et al). Nov 14, 2014. 14 pp. JLAB-PHY-14-1986. e-Print: arXiv:1411.4088[nucl-ex]

[2] Peskin, Michael E. and Schroeder, Daniel V. An Introduction to Quantum Field Theory. Reading: Addison-Wesley, 1995.

[3] Precision Measurement of the Weak Mixing Angle in Møller Scattering (P.L. Anthony (SLAC E158 Collaboration) et al). Aug 17, 2005. Phys. Rev. Lett.95, 081601.

[4] The brilliant mind of Alexander S. Michel, who explained parity violation to me about 10 times over the course of the past year.

[5] The gas electron multiplier (GEM): Operating principles and applications (Fabio Sauli (CERN)). Jan 1, 2016. https://doi.org/10.1016/j.nima.2015.07.060.

[6] A high-bias, low-variance introduction to Machine Learning for physicists (Mehta, Pankaj et al). March 2018. 119 pp e-Print: arXiv:1803.08823[physics.comp-ph]

[7] Google's Machine Learning Crash Course. https://developers.google.com/machine-learning/crash-course/.

[8] What is the difference between test and validation sets? (Brownlee, Jason). Jul 14, 2017. https://machinelearningmastery.com/difference-test-validation-datasets/.

[9] An overview of gradient descent optimization algorithms (Ruder, Sebastian). Jan 19, 2016. e-Print: arXiv:1609.04747[cs.LG]

[10] Neural Network Structure. https://researchgate.net/figure/MLP-neural-network-structure_fig2_276880129.

[11] What is the difference between a Batch and an Epoch in a Neural Network? (Brownlee, Jason). Jul 20, 2018. https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/.

[12] Understanding LSTM Networks (Olah, Christopher). Aug 27, 2015. http://colah.github.io/posts/2015-08-Understanding-LSTMs/

[13] TensorFlow Documentation: https://tensorflow.org

[14] Keras Documentation: https://keras.io/