Photon Angle Reconstruction for the K0T0 Experiment at JPARC

Ben Brubaker

Advisor: Yau Wah

May 11, 2011

Abstract

I set out to design and train a regression neural network to optimally reconstruct the angle of a photon incident on a CsI calorimeter. I ultimately used a 2-layer MLP network with 10 inputs, 30 hidden units, and a single output. This network was able to reconstruct angle with 6.6° accuracy using real data.

1 Introduction

1.1 Overview of Experiment

The K0T0 experiment at JPARC, also called E14, aims to measure the branching ratio of the so-called 'golden mode' decay of the K-Long particle, which is very precisely predicted by the standard model [1]:

$$Br(K_L^0 \longrightarrow \pi^0 \nu\bar{\nu}) = (2.49 \pm 0.39) \times 10^{-11} \tag{1}$$

The golden mode is of particular interest because it is a CP-violating decay, and its small theoretical uncertainty makes it ideal for probing unanswered questions of CP Violation in the Weak Interaction.

The general strategy of E14 is as follows: we produce K_L^0 in the collision of high-energy protons with a stationary target, and then count decays by the golden mode and three other *normalization modes* with experimentally verified branching ratios [2]:

$$\begin{aligned} Br(K_L^0 \longrightarrow 3\pi^0) &= (1.952 \pm 0.012) \times 10^{-1} \\ Br(K_L^0 \longrightarrow \pi^0\pi^0) &= (8.65 \pm 0.06) \times 10^{-4} \\ Br(K_L^0 \longrightarrow \gamma\gamma) &= (5.47 \pm 0.04) \times 10^{-4} \end{aligned} \tag{2}$$

We can use the normalization modes to calculate the total number of K_L^0 decays, and thus the branching ratio of the golden mode.

In identifying golden mode decays, we ignore the neutrinos, which are very difficult to detect in practice, and focus on the π^0, which itself decays according to $\pi^0 \longrightarrow \gamma\gamma$ with a branching ratio of $\sim 99\%$. The apparatus used in E14 consists of a large Cesium Iodide (CsI) calorimeter used to detect photons, as well as many veto detectors specialized for different kinds of background reduction. Golden mode decays will appear as events with exactly two photons simultaneously incident on the CsI and nothing on the veto detectors. Many cuts can then be applied to thin out the resulting pool of potential golden mode decays, the most important being the requirement of nonzero net momentum transverse to the K_L^0 beamline, due to the momentum carried away by the undetectable neutrinos.

The similarity of the end products of the normalization modes to those of the golden mode allows us to detect them with the same apparatus, but it can also cause problems. $K_L^0 \longrightarrow \pi^0\pi^0$ in particular is a major background for the golden mode: if we fail to detect two of the four photons, the two that

remain will have nonzero transverse momentum, and thus may appear very much like photons from a golden mode decay.

1.2 Motivation for Angle Measurement

We could reduce such misdiagnoses greatly by introducing a cut based on how likely it is that a given pair of photons originated in the same π^0 decay, and an accurate measurement of the angle at which each photon is incident on the CsI would enable us to implement such a cut. To see this, consider a single π^0 decaying into two photons which enter the CsI array, as illustrated in Figure 1:

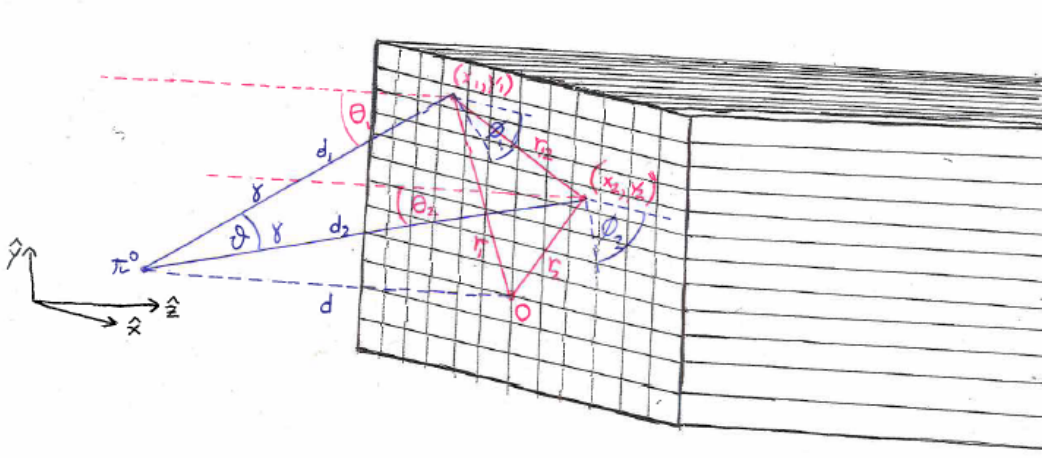

Figure 1: Geometry of π^0 decay

From geometry alone, we get

$$r_{12}^2 = d^2 \left(\sec^2 \theta_1 + \sec^2 \theta_2 - 2 \sec \theta_1 \sec \theta_2 \cos \vartheta \right) \tag{3}$$

The K_L^0 beam is collimated upstream of the region in which decays are likely to occur, and the very short lifetime of the π^0 (On the order of 10^{-17} s [2]) ensures that the decay vertex of any π^0 produced in a K_L^0 decay will lie on the K_L^0 beamline to a good approximation. Thus, we can set our origin on the beamline in the plane of the array surface. If we can reconstruct the position (x, y) at which each photon hits the detector with respect to this origin, we can calculate r_{12}. It is also evident that if we can reconstruct the angle of incidence θ and the azimuthal angle ϕ of each photon in addition to the hit position, we will have uniquely determined d, the z-coordinate of the decay

vertex.[1] Thus, we can use equation (3) to find ϑ, the angle between the photon trajectories, as a function of the reconstructed quantities x_1, y_1, θ_1, ϕ_1, x_2, y_2, θ_2, and ϕ_2.

Everything I have discussed thus far has been purely geometrical. We can bring physics into the picture by considering conservation of 4-momentum [1], which, in units with $c = 1$, requires that

$$\begin{aligned} m_\pi^2 &= -(p_1 + p_2)^2 \\ &= -\left(|p_1|^2 + |p_2|^2 + 2p_1p_2\right) \\ &= 2(\varepsilon_1\varepsilon_2 - \mathbf{p}_1 \cdot \mathbf{p}_2) \\ &= 2\varepsilon_1\varepsilon_2(1 - \cos\vartheta) \end{aligned} \tag{4}$$

If we can reconstruct the initial energy with which each photon struck the detector, we can determine what value of ϑ we should expect if the two photons did in fact decay from a particle with mass m_π.

Provided we can we reconstruct the position, angle, and energy of photons entering the CsI, we can define a cut as follows: we construct a χ^2 statistic for ϑ with the result we get from conservation of momentum as the expected value and the result we get from geometry as the observed value, and discard all events for which χ^2 is above a certain threshold value determined by the errors on our reconstructed quantities.

The experiment that preceeded E14, called E391a, was able to circumvent angle reconstruction by *assuming* that pairs of coincident photons originated in the same π^0 decay, and then solving for ϑ in terms of ε_1 and ε_2 to find d. But this method does not allow us to implement the cut defined above; thus, developing a method to accurately reconstruct photon angle is a worthwhile endeavor.

1.3 My Goals

My goal in this project was to design and train a regression neural network to optimally reconstruct the angle of a photon incident on a CsI calorimeter. I experimented with many different network architectures, but ultimately used a 2-layer MLP network with 10 inputs (nine energy readings from the CsI calorimeter and the total energy deposited in the calorimeter), 30 hidden units, and one output (the angle θ_x). I trained and evaluated this network using both simulated data and real data.

When using this network, I was constrained by the available data to assume that the angle of

[1] d can then be used to identify normalization mode decays, but the details of this process are outside the scope of my project.

incidence lay in the xz-plane, and that the calorimeter could give us no time information. I was able to experiment with angular reconstruction not subject to these restrictions by constructing a variant of the above network with nine additional inputs (corresponding to the times at which the nine energy deposits occurred) and another variant with two outputs (θ_x and θ_y). I was only able to test these network variants using simulated data, but the results of these tests still provide some indication of how we might expect them to perform on real data.

Throughout this paper, I will assume a familiarity with neural networks in general, and the characterization, training, and testing of multilayer perceptron (MLP) networks in particular. I encourage readers unfamiliar with the neural network literature to consult Appendix A, or the excellent introduction by Christopher Bishop [3].

2 Real Data

The real data which I used as the ultimate test of my network came from an April 2010 test of the detector and DAQ systems in Sendai, Japan. I will begin with an account of the detector used in Sendai, upon which my simulated detector was modeled, and a discussion how the energy and time readings my neural network takes as inputs can be computed from the raw data. Then I will turn to a description of the particular measurements that were made in Sendai, the form of the resulting datasets, and the code I wrote to process them for use in my network.

2.1 Detector Geometry

The detector used in the Sendai test is a 12×12 array of CsI crystals, each 50 cm long and with 2.5 cm sides. A photomultiplier tube is attached to the back of each crystal. The calorimeter to be used in E14 itself is much larger and less homogeneous, but works the same way. The array, with the kinematics of a hypothetical sample event, is shown in Figure 2.

As Figure 2 indicates, the surface of the CsI array is parallel to the xy-plane. The individual CsI blocks are indexed by $k = i + 12(j - 1)$, where i and j index columns and rows, respectively. Thus, the block index counts from left to right, and from bottom to top, and the geometrical center of the array (marked by a small red dot) lies as the intersection of blocks 66, 67, 78, and 79.

Figure 2 shows a photon hitting the center of block 66 at an angle θ with respect to the z-axis and an angle ϕ in the xy-plane. Alternatively, we can specify the photon's orientation with the angles

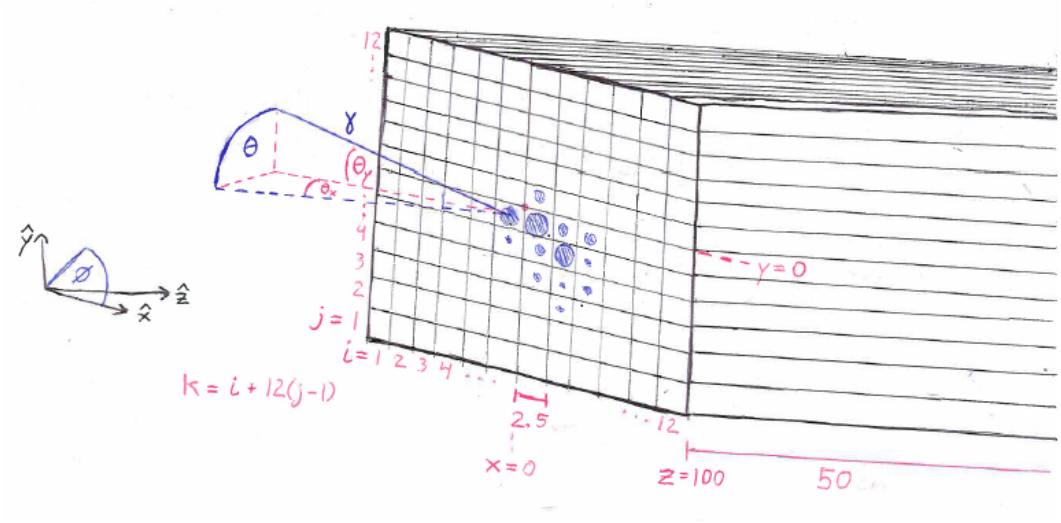

Figure 2: Geometry of CsI array, showing indexing conventions, coordinate system, angles, and energy deposition in a typical event. All distances are in cm.

$\theta_x = \tan^{-1}\left(\frac{x}{z}\right)$ and $\theta_y = \tan^{-1}\left(\frac{y}{z}\right)$. We can transform easily between this parameterization and the more familiar angles of spherical polar coordinates with the relations

$$\begin{aligned} \tan\theta &= \sqrt{\tan^2\theta_x + \tan^2\theta_y} \\ \tan\phi &= \frac{\tan\theta_y}{\tan\theta_x} \end{aligned} \tag{5}$$

2.2 Energy and Time Distributions

The photons incident on the detector in the E14 experiment will typically have energies between a few hundred MeV and 1 GeV, and when a photon with energy in this range is incident on CsI, it sets off an electromagnetic shower, distributing the initial energy of the incident photon throughout the array. Since CsI is a scintillator, the end results of this process are lots and lots of visible photons, which are prone to absorbing on the interface between two crystals. To prevent such absorption, the latter half of each individual cystal in the CsI array is wrapped in a layer of highly reflective aluminized mylar, which effectively causes each visible photon to get "stuck" in whichever crystal it happens to have been produced in, and ultimately make its way to the PMT attached to the back of that crystal.

Each PMT in turn will output a current proportional to the energy of the entering photon. Since hundreds of millions of visible photons are produced in each electromagnetic shower, many will end

up arriving at the PMT in each block to which the shower spreads. But reflections from the mylar cause different photons to travel along paths of different length, so the visible photons do not reach the PMT at the same time. Thus, each PMT output ends up being a current pulse with some finite, reasonably well-defined temporal extent, which is then digitized at 125 MHz by our DAQ System and recorded. An example of such a pulse is shown in Figure 3.

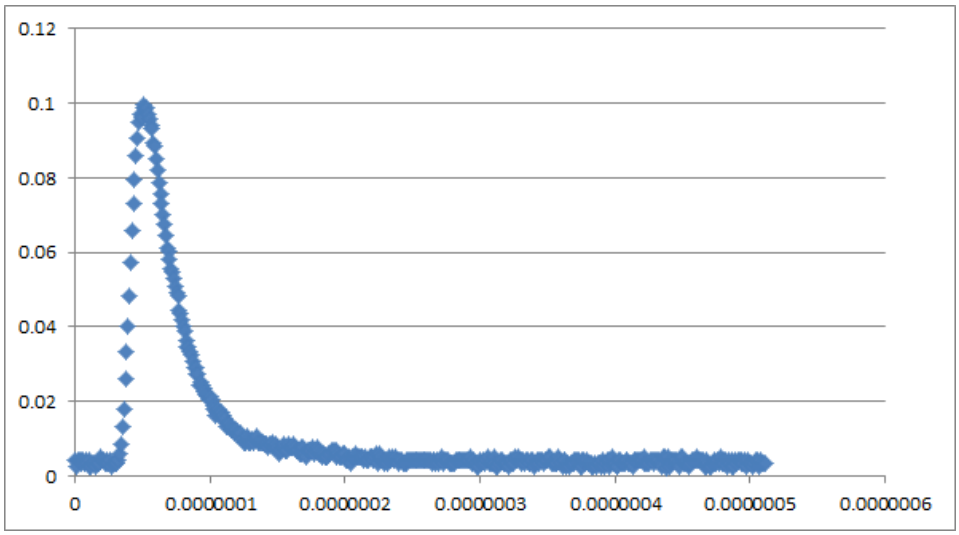

Figure 3: A Digitized PMT pulse from a test of the detector in Kyoto. On the x-axis is time in seconds; on the y-axis is voltage in volts.

For a given event, we can integrate over the pulse received by a given DAQ channel to get the total energy "deposited" in the corresponding CsI block. We can extract timing information from the digitized PMT pulses in one of two ways: we can record the time at which the amount of energy in the PMT pulse surpasses a 1 MeV threshold,[2] or we can fit a curve to each pulse, and assign it a timestamp corresponding to the mean value of the fit. The relevant information will be encoded in the relative timestamps of adjacent crystals, rather than the magnitude of individual timestamps, so the difference between the timestamps that these two methods yield is subtler than it may initially seem. E391a used the "threshold timestamp" method, and recent advances in time measurement resolution will enable E14 to use the "mean timestamp" method.

The detector is thus capable of giving us two pieces of information for each of the 144 blocks in the array: the energy deposited in the block, and some indicator of when that deposit occured. The value of this information in reconstructing the photon's angle is intuitively obvious: we expect

[2]For a sense of perspective, individual blocks can end up with deposits as large as a few hundred MeV, depending of course on the initial energy of the incident photon.

the skewedness of the energy and time distributions to correlate with larger incident angles. The blue circles on some of the blocks in Figure 2 are supposed to represent the energy deposits in those blocks for a hypothetical event: for a photon entering the array from the top left, energy will spread primarily (but not exclusively) down and to the right.

Another thing we can see from the example in Figure 2 is that the electromagnetic shower ends up being a fairly local phenomenon: at the energy scales we are using, most blocks pick up negligible energy or no energy at all; for these blocks, there is clearly no well-defined time. Thus, the number of relevant variables we can glean from the CsI for each event will actually be much smaller than 288. I will explain my procedure for picking out the relevant variables for each event in greater detail in Section 4.1.

2.3 The Sendai Beam Test

In the Sendai beam test, the detector was placed in the path of a beam of electrons with known energy, on top of a platform that could be rotated in the xz-plane, to achieve the effect of particles entering the array at different angles. The platform could also be translated along the x-axis, to ensure that the electromagnetic shower was completely contained within the array at each incident angle. About 60,000 events were recorded at each combination of beam energy and angle; datasets like this were generated at 200, 300, 460, 600, and 800 MeV, and 10, 15, 20, and 30°. The fact that the particles incident on the detector were electrons rather than photons is not a significant difference between Sendai data and the data we hope to get from E14: the electromagnetic shower begins with bremmstrahlung rather than pair production, but then proceeds exactly as it would had a high-energy photon initiated it.

More significant is the fact that it proved impossible to compute meaningful timestamps from the Sendai data, due to pulse deformations caused by the saturation of preamps within the PMTs. The purpose of the Sendai test was precisely to catch and diagnose such hardware problems, and unfortunately, no other real data from E14 exists at this time. Thus, I was unable to use real data to test the degree to which time information helps with angle reconstruction.

It is also significant that data only exists for certain discrete values of θ_x. In the real experiment, we expect θ_x to vary continuously from about 5° to about 40°, so I will treat it as a continuous variable whose values just happen to be clustered in a few places rather than as a categorical variable. The outputs of a network trained using this data will be skewed towards the peaks in the target

distribution, but lacking a more complete dataset, we won't be able to do any better with real data.

The datasets that I received from Eito Iwai, a graduate student studying the Sendai data, were in ROOT format, so I wrote the C++ code to process them to be interpreted and executed within ROOT. Each dataset contained the energy deposited in each block that registered nonzero energy for approximately 60,000 events at a single energy-angle combination. My code mixed these input datasets together to get training and test datasets for my neural network, each containing 20,000 events with fixed energy and a uniform angle distribution. Within each input dataset, events were drawn sequentially, starting with a randomly chosen "event offset." To generate an independent dataset, I could simply change the offset on each input dataset by at least 5,000, and thus guarantee that the new dataset drew from a non-overlapping subset of the input data.

3 Simulated Data

All of my simulations were performed in GEANT 3.21, a FORTRAN 77-based particle physics simulation program developed at CERN. The vast majority of the routines in GEANT don't need to be explicitly addressed by the user, but there are a number of so-called "user routines," which collectively perform four main functions: simulating particle kinematics, simulating physics processes, defining the detector geometry, and deciding what information to output when the desired number of events have been simulated. The latter two functions are self-explanatory, but I will elaborate here briefly on the first two so that the reader can understand to what extent my simulated data and the real data from Sendai may be compared.

3.1 Physics Simulation

I will begin by describing what physical processes my simulation takes into account and what it omits, as this information is necessary to make sense of other aspects of the simulation. GEANT simulates the interactions of every particle produced in the electromagnetic shower except visible photons, which are likely to be absorbed and then re-emitted by impurities in the CsI. By not attempting to track visible photons, I was able to model the detector as an array of homogeneous CsI crystals, omitting the mylar, the impurities in the CsI, and the PMTs, the properties of which only become important at the end of each electromagnetic shower.

My simulation calculates the energy and time distributions of each event as follows:[3] at each discrete step in the tracking of each particle, if a process occured that would produce visible photons, GEANT records the total energy transfered to these photons in an array indexed by the block in which the process occurred and the tracking step. The time and depth (in the z-direction) at which each such process occurred are also recorded. At the end of each event, the energy entries for each block are summed over all tracking steps, yielding a 144-element array whose entries represent the total energy deposited in each block. The code also loops over the time of each deposit, and checks whether the total energy deposited in the block up to that point has surpassed a 1 MeV threshold. It then records the earliest time for which this is true, as well as the depth at which the threshold-crossing process occured.

Thus, we can compute for each block and each event a variable analogous to the threshold timestamp defined in Section 2.2. But there is a crucial difference between these two quantities. The (measurable) threshold timestamp of a given block is the time at which the DAQ system sees that a total of 1 MeV has been deposited in the form of visible photons in that block, whereas the simulated timestamp defined in this section is the (non-measurable) time at which the process which produced the final visible photons that pushed the energy over the 1 MeV threshold actually occured.

The former time is offset from the latter by two independent terms: the easily calculable (and thus uninteresting) time for the PMT output signal to reach the module which registers that the threshold has been surpassed, and the time it takes a visible photon to travel from its point of origin within the CsI to the PMT.

We can model this latter correction to the simulated timestamp using the depth at which the visible photons were produced and the average effective speed of visible photons in the E14 CsI array, taking into account reflections from the mylar. We can divide the particle's distance from the back of the array by this average speed (found experimentally to be 8.2 cm/ns [4]) to get the average time it takes a visible photon to travel to the PMT, and then superimpose some Gaussian noise (drawn from a distribution with mean 0 and $\sigma = 200$ ps) to reflect the limited precision of real time measurements. In this way, we can compute energy and time distributions from simulated data that reflect measurable information.

[3]This code is largely due to my colleague Jiasen Ma, a postdoc in Dr. Wah's research group, who was working on simulating aspects of the electromagnetic shower in GEANT well before I began this project.

3.2 Kinematic Simulation

Until it interacts stochastically inside the detector, an incoming photon can be completely specified by its energy ε, its angle (parameterized by (θ_x, θ_y) or (θ, ϕ)), and its hit position (x, y). Before simulating the generation of a large number of photons in GEANT, we need to consider within what range we expect each of these variables to fall.

Because the electromagnetic shower is a local phenomenon, the macroscopic distribution of hit position (i.e., over the whole array) is irrelevant, provided the shower is entirely contained within the CsI. On the other hand, the microscopic distribution (i.e., within a single block) must be uniform, because there is no conceivable reason incoming photons should strike one side of a given 2.5 cm square in preference to the other. Thus, without loss of generality, we can generate photons whose hit positions are uniformly distributed within a single block. It is similarly easy to see that ϕ must have a uniform distribution by azimuthal symmetry, and can be restricted to a single quadrant without loss of generality.

To get a sense of the expected ε and θ distributions, I looked to the data amassed by E391a, in particular, to Monte Carlo simulations of photons resulting from $K_L^0 \longrightarrow \pi^0 \nu\bar{\nu}$ and $K_L^0 \longrightarrow \pi^0\pi^0$ decays. The details of these distributions are not particularly important for my purposes, since I will want to feed my neural network training sets with uniformly distributed target values so as not to bias it towards more common angles, but we will want to know the upper and lower bounds between which the bulk of each distribution lies. From the histograms of these distributions, reproduced in Appendix B, we see that the ε distributions are mostly contained between 150 MeV and 1 GeV, and the θ distributions are mostly contained between 5° and 40°.

Angles as large as 40° are unproblematic in E14, for which the array is much bigger, but photons entering a 12×12 array at extreme angles may escape the array after leaving behind only a fraction of their initial energy. If I included such events in my simulation, I would be creating a complication that doesn't exist in the data I'm trying to model. Thus, I had to choose a maximum angle θ for which the probability of a photon passing through the array without interacting would be less than some small value ϵ. I will omit the details of this calculation, and simply quote my results: a maximum angle of $\theta = 32°$ guarantees that each photon will pass through at least 15 *Radiation Lengths* [5] of CsI, and thus that $\epsilon < 10^{-5}$, provided we choose photons incident in block 66, and restrict ϕ to the third quadrant (or set $\phi = 180°$, for datasets with only one angular degree of freedom), to ensure that the shower spreads primarily up and to the right.

I produced simulated datasets at the same fixed energies at which real data was available, and the other kinematic variables – x, y, θ, and ϕ – were drawn from predefined sequences of uniform random numbers in GEANT. To generate an independent dataset, I could change the indices specifying which of these independent sequences to draw from.

4 Neural Network Procedure

I implemented my angular reconstruction network using the MLP framework of the CERN data analysis package ROOT, which is written in C++. In this section, I will detail precisely how I went about doing so. I will describe the variants of the pre-processing scheme I used[4] and how I optimized my network for use in angle reconstruction.

4.1 Pre-processing

Computing energy and time distributions from raw data, as described in Section 2, can be regarded as a form of pre-processing, as it reduces the number of neural network inputs D from some number on the order of 10^4 to 288. But as I noted at the end of Section 2, most of the remaining 288 variables do not contain useful information. We can do better if we can come up with an algorithm which allows us to discard the unhelpful majority of the 288 energy and time inputs, and not necessarily the same ones for each event.

In order to be applicable to data from real pion decays, the pre-processing scheme we choose must produce a set of inputs that is translation-invariant, in the sense that the inputs chosen should be equally useful regardless of which block actually contains the photon hit position, and azimuthally symmetric. My pre-processing algorithm (a piece of standalone C++ code) scans over the array and picks out a block with a significant energy deposit, and then records the energies (and timestamps if available) of all the blocks in a smaller output array centered at the chosen block. This algorithm will produce a set of inputs that is both translation-invariant and azimuthally symmetric, provided we specify how it chooses the center of the output array (μ) and the size of that array (ν).

It seems reasonable to choose μ to be either the block with the single largest energy deposit or the block containing the array's center of energy. A quick glance at a sample event in the simulated data suggests that ν could be 3×3 or 5×5, but not bigger, as very little useful information will be

[4] See section A.3 in the appendix for a more thorough introduction to what pre-processing is and why it's necessary.

encoded in the crystals further than 5 cm from the center of an energetic cluster. But there is no way we can decide a priori which of these choices for μ and ν are to be preferred. Rather, we must use an independent validation dataset to optimize the network with respect to pre-processing. I will discuss this process in greater detail in the coming section.

4.2 Network Optimization

To train, construct, and evaluate an MLP in ROOT, we must specify a training dataset and either a validation dataset or test dataset, each containing N events. We must also specify the number of inputs D, the number of outputs C, the number of hidden units J, the learning method the network will use during training, values for any adjustable parameters specific to the the learning method, and the number of training epochs M. All of these terms are defined and discussed in greater detail in Appendix A.

My first goal was to optimize my angle reconstruction MLP with respect to all of these parameters, which I did by training and evaluating variants of my network characterized by different parameter values on the same datasets, and comparing their respective validation errors, defined by

$$E' = \frac{\sum_{n=1}^{N} \|\mathbf{y}(\mathbf{x}^n, \mathbf{w}) - \mathbf{t}^n\|^2}{\sum_{n=1}^{N} \left\|\bar{\mathbf{t}} - \mathbf{t}^n\right\|^2} \tag{6}$$

where $\mathbf{x}$, $\mathbf{y}$, $\mathbf{w}$, and $\mathbf{t}$ are the input, output, weight, and target vectors, respectively, and $\bar{\mathbf{t}}$ is the mean value of the target variable over the validation dataset. Thus, the denominator of this expression is simply the variance of the target data: a result of $E' = 1$ would mean that the network did not use its inputs at all, but rather simply set $\mathbf{y}(\mathbf{x}^n, \mathbf{w}) = \bar{\mathbf{t}}$. The lower the value of E', the greater the extent to which the network was able to use its inputs to make informed rather than random guesses as to the values of the outputs. Thus, the validation error allows us to determine at a glance the effectiveness of any network model we might consider.

In general, we can expect energy distributions to be more distinctive, and angular reconstruction to be more accurate, at higher incident energies, given that there is simply more information to go around. Thus, I chose to optimize my MLP using a pair of simulated 1 GeV datasets that used timestamp inputs as well as energies, with the hope that they might reveal differences in the performance of different network variants which would be obscured by worse performance across the board at lower energies or without time information.

The number of inputs D in my neural network depended on the particular pre-processing scheme I used: $D = 18$ for a 3×3 array, and $D = 50$ for a 5×5 array. I also experimented with adding an additional input ε_{tot} representing the total energy collected by the whole 12×12 array, which I imagined might differ significantly from the energy picked up by the output array in the case of a particularly skewed energy distribution, and thus encode useful information about the angle of incidence. I tried out datasets with several values of N, in each case choosing values of J consistent with the rule of thumb $N > 30W$, where $W = (D + C + 1) \times (J + 1)$, to avoid overfitting.

I had some theoretical help when attempting to choose the optimal option among the six learning methods implemented in ROOT. The *BFGS* algorithm, discussed at greater length in section A.2, involves the computation of a $W \times W$ matrix at each epoch, and is therefore thought to be more effective than its primary competitor, *Conjugate Gradients*, for a relatively small number of weights [6], but less effective for many weights, where sheer computational effort drags it down.

Bishop gives $W = 1000$ as the upper bound below which BFGS should be preferred [3]. For $N = 20,000$, the largest value of N I will consider, the maximum value of W I could choose and still be consistent with $N > 30W$ is less than 1000, which suggests I should train my MLP with BFGS. I compared validation errors on networks trained with different learning methods, to lend experimental backing to this theoretical prediction. Lastly, I optimized my network with respect to the number of epochs M, and, since BFGS did indeed end up outperforming its competitors, I optimized it with respect to the BFGS parameters R and τ, both defined in section A.2.

The performance of network variants that differ in the respects outlined above is summarized in Table 1, with the optimal choice for each parameter in bold. In the first column, the two entries marked GD refer to training by the Gradient Descent method, with fixed step size and step size determined by line search, respectively, and the entries marked CG refer to training by the Conjugate Gradients method, using the Ribiere-Polak and Fletcher-Reeves formulas, respectively.

As Table 1 indicates, I first confirmed that BFGS does in fact outperform its competitors, at least for a small number of epochs. Then I fixed the learning method and observed that, while validation error continued to fall as I increased M, it did so at a progressively slower and slower rate; with about 2 to 6 seconds of processing time for each epoch, simply cranking up M would clearly not be the most efficient way to reduce error. I decided to perform the rest of my optimization at $M = 200$, and return to using $M = 1000$ only after settling on final values for all the other parameters. I saw that increasing R decreased the error dramatically without any increase in processing time up until about

Learning Method	**M**	τ	**R**	**N**	**J**	$\varepsilon_{\mathbf{tot}}$	μ	ν	**E′**
BFGS	10	3	10	20000	10	no	$\varepsilon_{\max}$	3×3	0.270
GD (fixed step size)	10	N/A	N/A	20000	10	no	$\varepsilon_{\max}$	3×3	0.418
GD (line search)	10	3	N/A	20000	10	no	$\varepsilon_{\max}$	3×3	0.337
CG (R-P)	10	3	10	20000	10	no	$\varepsilon_{\max}$	3×3	0.300
CG (F-R)	10	3	10	20000	10	no	$\varepsilon_{\max}$	3×3	0.297
BFGS	100	3	10	20000	10	no	$\varepsilon_{\max}$	3×3	0.210
BFGS	200	3	10	20000	10	no	$\varepsilon_{\max}$	3×3	0.204
BFGS	500	3	10	20000	10	no	$\varepsilon_{\max}$	3×3	0.199
BFGS	**1000**	3	10	20000	10	no	$\varepsilon_{\max}$	3×3	0.192
BFGS	200	3	20	20000	10	no	$\varepsilon_{\max}$	3×3	0.200
BFGS	200	3	**50**	20000	10	no	$\varepsilon_{\max}$	3×3	0.184
BFGS	200	**3**	100	20000	10	no	$\varepsilon_{\max}$	3×3	0.184
BFGS	200	2	50	20000	10	no	$\varepsilon_{\max}$	3×3	0.184
BFGS	200	3	50	20000	5	no	$\varepsilon_{\max}$	3×3	0.199
BFGS	200	3	50	20000	20	no	$\varepsilon_{\max}$	3×3	0.183
BFGS	200	3	50	20000	**30**	no	$\varepsilon_{\max}$	3×3	0.175
BFGS	200	3	50	5000	8	yes	$\varepsilon_{\max}$	3×3	0.189
BFGS	200	3	50	10000	15	yes	$\varepsilon_{\max}$	3×3	0.184
BFGS	200	3	50	**20000**	30	**yes**	$\varepsilon_{\max}$	3×3	0.1720 ± 0.0005
BFGS	200	3	50	20000	30	yes	CoE	3×3	0.1723 ± 0.0006
BFGS	200	3	50	20000	12	yes	CoE	5×5	0.2034 ± 0.0002
BFGS	200	3	50	20000	12	yes	$\varepsilon_{\max}$	5×5	0.1759 ± 0.0004
BFGS	**1000**	**3**	**50**	**20000**	**30**	**yes**	$\varepsilon_{\mathbf{max}}$	$\mathbf{3 \times 3}$	0.160

Table 1: Optimization of MLP using validation set at 1 GeV

$R = 50$, and that decreasing τ from the value $\tau = 3$ recommended by the ROOT User's Guide [7] slowed down the line search without appreciably increasing its accuracy. I next tried varying J, and got best results for $J = 30$, approximately at the top of the range permitted by $N > 30W$ for $D = 18$ and $C = 1$. Decreasing N from 20,000, thus reducing the maximum permissible value of J, predictably led to an increase in E', and the extra input ε_{tot} did indeed prove informative.

To optimize my network with respect to the pre-processing options μ and ν, I wrote a program to train and evaluate my network 100 times in succession, recording the validation error and then re-randomizing the initial weights each time. This way I could compute not only the average validation error but also its standard deviation, a high value for which would indicate that attempts to train the network are falling into different local minima, and cast doubt on the validity of any attempt to decide between two network variants based on a single reading of the validation error of each.

Fortunately, the standard deviation was low for each of my choices of pre-processing algorithm; since it would be impractical to train the optimal network 100 times in sequence every time for each measured value of E', I simply assigned all my other values of E' the slightly conservative

error of $\delta\ E' = 0.001$.

As for the mean values of E' I found this way, they indicated that $\nu = 3 \times 3$ should be preferred over $\nu = 5 \times 5$. For $\nu = 3 \times 3$, there was little difference between the "maximum energy" and "center of energy" variants. I can think of no good reason why CoE, 5×5 pre-processing fared so much worse than all the others, but I decided to go with the $\mu = \varepsilon_{\max}$ variant just in case this unusually bad result was indicative of some deeper systematic problem with the CoE method.

5 Results

My next order of business was to check the standard deviation of the error on real and simulated test datasets, and make sure it was small enough that the test set error (also called the *generalization error*) on two different datasets could be meaningfully compared. I performed 100 successive trainings using the optimal network, except with $M = 200$ instead of 1000 to cut down on processing time, on three datasets, all at 200 MeV. The results are summarized in Table 2:

Type	$\mathbf{E'_{test}}$
S(ε,t)	0.5318 ± 0.0005
S(ε)	0.6408 ± 0.0004
R(ε)	0.6827 ± 0.0003

Table 2: Standard Deviation of Generalization Errors

S(ε,t) denotes simulated data that makes use of both energy and time inputs, S(ε) denotes simulated data that only uses energy inputs, and R(ε) denotes real data. We see that the standard deviation on the test set error is indeed suitably small. Though it doesn't seem likely that increasing M or ε would increase the variability in E', I chose to assign each of my other generalization error readings a a conservative error of $\delta E' = 0.001$.

Having done this, I proceeded to train and test the optimal network (characterized by the values in the last line of Table 1) at energies ranging from 200 MeV to 1 GeV, on real data, simulated data using only energy inputs, and simulated data using both energy and time. My results are summarized in Table 3.

θ (°)	ϕ (°)	ε **(GeV)**	**Type**	σ (°)	$\mathbf{E'}_{\text{test}}$	$\mathbf{E'}_{\text{train}}$
U	180	1	S(ε,t)	3.80	0.158	0.149
U	180	0.8	S(ε,t)	4.15	0.187	0.175
U	180	0.6	S(ε,t)	4.56	0.230	0.219
U	180	0.46	S(ε,t)	5.27	0.303	0.284
U	180	0.3	S(ε,t)	5.96	0.391	0.359
U	180	0.2	S(ε,t)	6.68	0.496	0.447
U	U	U	S(ε,t)	5.19, 5.22	0.382	0.369
U	180	1	S(ε)	4.44	0.221	0.212
U	180	0.8	S(ε)	4.90	0.265	0.252
U	180	0.6	S(ε)	5.31	0.319	0.310
U	180	0.46	S(ε)	5.76	0.378	0.367
U	180	0.3	S(ε)	6.56	0.497	0.472
U	180	0.2	S(ε)	7.30	0.618	0.585
U	0	0.8	R(ε)	4.55	0.362	0.351
U	0	0.6	R(ε)	4.99	0.430	0.409
U	0	0.46	R(ε)	5.29	0.488	0.467
U	0	0.3	R(ε)	5.85	0.599	0.575
U	0	0.2	R(ε)	6.15	0.657	0.626

Table 3: Neural Network Results on Test Data

Here, "U" indicates a uniform distribution over the range specified in Section 3.2, and σ is the standard deviation of the distribution of network targets minus network outputs evaluated on the test dataset. Since the mean value of $\mathbf{t} - \mathbf{y}$ is zero, we can write

$$\sigma = \sqrt{\frac{\sum_{n=1}^{N} \|\mathbf{y}(\mathbf{x}^n, \mathbf{w}) - \mathbf{t}^n\|^2}{N}} \tag{7}$$

Whereas E'_{test} is a dimensionless quantity, σ has units of degrees, and it is tempting to compare the network's performance on real and simulated data by simply comparing the corresponding values of σ. But this is misleading. Comparing equations (6) and (7), we see that $\sigma \propto \sqrt{E'_{\text{test}}}$, but the "constant" of proportionality (which I shall call β) is really a function of the variance of the target data. For both the S(ε,t) and S(ε) datasets, the target distribution is uniform from 0° to 32°, so we should expect the same $\beta = \beta_{\text{sim}}$ for all simulated datasets.[5] For the R(ε) datasets, on the other hand, the target distribution is a series of spikes at 10°, 15°, 20°, and 30°, with an accordingly smaller standard deviation. Thus, a smaller value of σ for real data than for simulated data does not necessarily indicate that the network is reconstructing real data more accurately than simulation,

[5]Excepting, of course, the dataset with two target angles, for which there is no such proportionality at all. It is worth noting that β is not exactly constant even among simulated datasets, because, for any finite dataset, the mean value of $\mathbf{t} - \mathbf{y}$ will not be exactly 0. Therefore, equation (7) is only an approximation, albeit a very good one, to the standard deviation of the distribution.

since the network's job is easier for real data. We avoid this problem by comparing datasets using E'_{test}, which is normalized to the variance of the target data.

However, we would still like an index of the accuracy of the MLP which is measured in degrees, rather than dimensionless. We can construct such an index by noting that, if we had a real dataset with a uniform target distribution from 0° to 32°, it would have $\beta = \beta_{\text{sim}}$, and its value of E'_{test}, representing the degree to which the network's inputs allow it to make informed guesses about the correct targets, would not change much. So we can define $\sigma' = \beta_{\text{sim}} \times \sqrt{E'_{\text{test}}}$: for simulated datasets, we simply have $\sigma' = \sigma$, and for real datasets, σ' will be an indicator of how accurately the MLP would be able to reconstruct angle on uniformly distributed data. Figure 4 shows a plot of σ' vs. ε, which we can use to compare real data to simulated data.

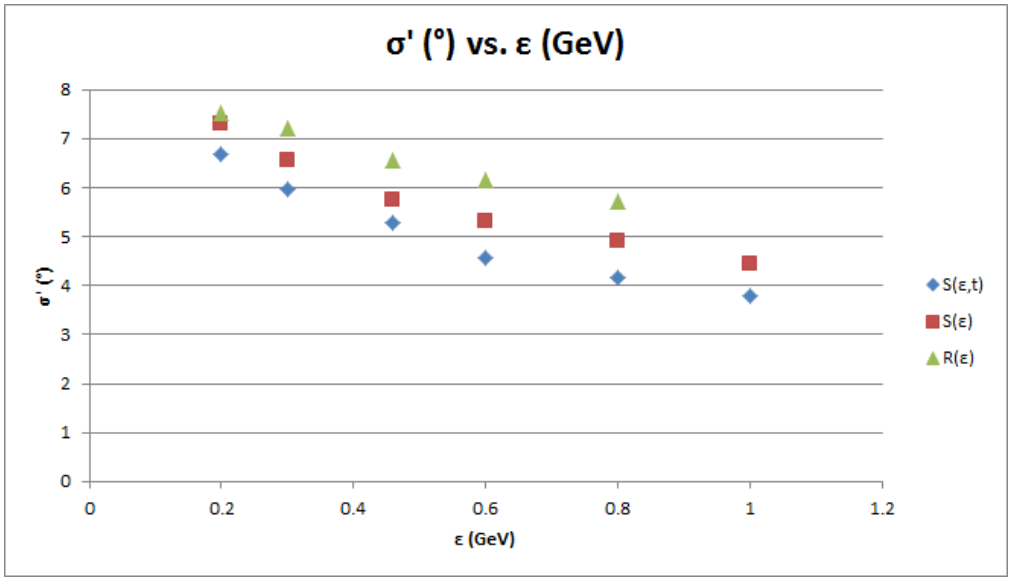

Figure 4: σ' vs. ε for real data and simulated data with and without time information.

All three σ' vs. ε curves are very similar; in particular, we see that we can reconstruct the incident angle using energy distributions alone with accuracy better than 10°, and that we can reconstruct this angle with real data almost as well as with simulated data. The slight discrepancy that does exist is to be expected, and is due to the fact that the real data may contain many processes that I could not possibly account for in my simulation, such as beam electrons interacting with stray positrons before they reach the CsI. To get a single number that may be used as an index of the network's accuracy on real data, we can simply average the values of σ' at different energies:

Type	**Average σ (°)**
S(ε,t)	5.07
S(ε)	5.71
R(ε)	6.63

Table 4: Average σ' for real data, and simulated data with and without time inputs

We see that my MLP can reconstruct a photon's angle of incidence with an accuracy of about 6.6° using only energy inputs. My simulation also suggests that, if real data that included time information were available, my MLP could use it to improve the accuracy of its reconstruction by about 0.6°. Lastly, comparing the first line of Table 4 with the values of σ_x and σ_y in the 2-angle simulated data in Table 3, we see that, according my simulation, my MLP should be just as accurate when reconstructing angles not confined to the xz-plane.

6 Conclusion

I was able to construct and train a neural network to reconstruct angle fairly accurately using real data, and thus I accomplished exactly what I set out to do when I began this project. I discovered that a surprising ammount of angular information was encoded in the calorimeter's energy distributions alone, and that the time distributions didn't contain much additional helpful information. Future studies with more complex timing simulations, or access to real data containing meaningful timestamps, will help to shed light on whether or not time information might be used to further improve the accuracy of my angular reconstruction.

A Neural Network Theory

This appendix is intended as a short guide to the aspects of neural network theory that are relevant to my project. For a more thorough treatment, I highly recommend *Neural Networks for Pattern Recognition* by Christopher Bishop.

Bishop defines a neural network as "a very powerful and very general framework for representing non-linear mappings from several input variables to several output variables... governed by a number of adjustable parameters" [3]. Vague as it is, this definition has the virtue of generality; in particular, it highlights the flexibility of neural networks, and their applicability to tasks other than the binary classification problems like signal-noise discrimination for which they are most commonly known.

Since the electromagnetic shower relating the photon angle to energy and time distributions is a fundamentally stochastic process, the angle cannot be expressed as a deterministic function of the network inputs: events at different angles can give rise to very similar energy and time distributions. My goal in reconstructing the network outputs is to find the systematic relationship between inputs and outputs underlying these stochastic fluctuations, or in other words, to find the conditional average, or regression, of the outputs on the inputs.

A.1 The Multilayer Perceptron

The particular type of neural network I will be using is called a *Multilayer Perceptron* (MLP), and it is an example of a more general class of networks that are *layered* and *feed-forward*.

A layered network is one in which a fixed number of transformations act on the input vector in series; each layer corresponds to a single transformation (governed by adjustable parameters called *weights*) between one set of variables and another, where the first set is the input vector, and the last set is the output vector.

A feed-forward network, as one might expect, is one in which there are no feedback loops: the outputs of each transformation may only be used as inputs by the transformation that immediately follows it. This may seem like a natural way to structure a network, but it is by no means the most general way to do so.

More precisely, a multilayer perceptron is a feed-forward layered network in which each transformation except the last is a sigmoidal *activation function* $g(\cdot)$ acting on a linear combination of

its inputs, and the final transformation is simply a linear combination of its inputs (it is thus said to have a linear activation function). In both cases, the coefficients in the linear combination are the weights for the layer in question. The meaning of this is more clearly conveyed by a network diagram, shown in Figure 5.

In this two-layer network,[6] $\mathbf{x}$ is a $(D+1)$-dimensional vector whose elements x_i are simply the network inputs for $i = 1, \ldots, D$. $\mathbf{z}$ is a $(J+1)$-dimensional vector of *hidden units* whose elements z_j are the outputs of the first transformation and inputs to the second, for for $j = 1, \ldots, J$. $\mathbf{y}$ is a C-dimensional vector whose elements y_k are the network outputs. For the particular MLP illustrated in Figure 5, $D = 18$, $C = 2$, and $J = 30$. $w_{ij}^{(1)}$ and $w_{jk}^{(2)}$ are the weights that parameterize the first and second transformations, respectively. The units x_0 and z_0 are both given a fixed value of 1, and the corresponding weights $w_{0j}^{(1)}$ and $w_{0k}^{(2)}$ are called *biases* for the first and second layers, respectively. The role of the biases is to allow for an adjustable offset to the linear combination passed on from each layer to the next.

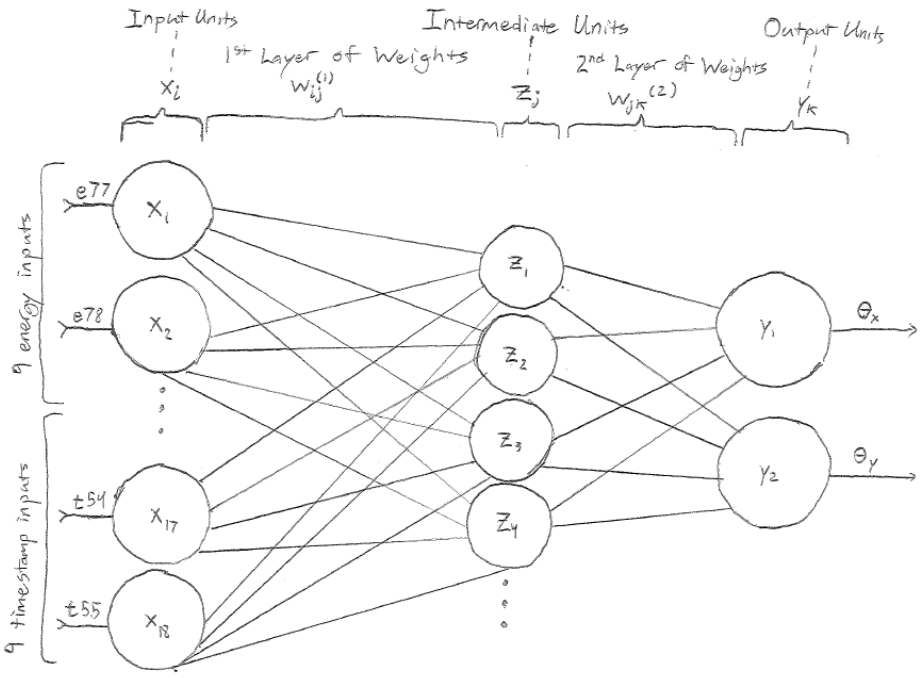

Figure 5: Schematic for my MLP

[6]There is some potential for confusion in this terminology, as layers are sometimes taken to refer to layers of nodes rather than layers of weights. By this convention, of course, the MLP in Figure 5 would be a 3-layer network. I prefer the "layers of weights" convention because it allows us to identify each layer with a mapping. I shall use this convention exclusively throughout this paper.

The relation between the inputs and the outputs are specified by the equations

$$y_k = \sum_{j=0}^{J} w_{jk}^{(2)} z_j \tag{8}$$

and

$$z_j = g\left(\sum_{i=0}^{D} w_{ij}^{(1)} x_i\right) \tag{9}$$

where

$$g(a) = \frac{1}{1 + e^{-a}} \tag{10}$$

is the logistic function, which is also called a sigmoid on account its s-shaped graph.

The most important point to take away from the above formalism is that the outputs of the MLP can be expressed as differentiable, if convoluted, functions of its inputs and the weights in both layers: $\mathbf{y} = \mathbf{y}(\mathbf{x}, \mathbf{w})$, where the dimensionality of $\mathbf{w}$ is $W = (D + C + 1) \times (J + 1)$; $W = 651$ for the network in Figure 5. In order for our neural network to have any predictive power, we must *train* it by giving it a large dataset in which each set of inputs is matched to a C-dimensional target vector $\mathbf{t}$ representing the values that we would like the neural network to output for that event. We can define an *error function* of the form

$$E = \frac{1}{2} \sum_{n=1}^{N} \|\mathbf{y}(\mathbf{x}^n, \mathbf{w}) - \mathbf{t}^n\|^2 \tag{11}$$

where n indexes the events in the training dataset, and N is the size of the dataset. This particular error function is called the sum-of-squares error function. It is common in many optimization problems, and particularly apt for training regression neural networks, since it can be proved that for a sufficiently large training dataset, the outputs of a trained neural network are precisely the conditional averages of the corresponding targets with respect to the input vector [3].

We see that E is a differentiable function of the weights. In particular, the partial derivatives of E with respect to the weights in the second layer can be calculated in terms of the inputs, the targets, and the current values of the weights for each event in the training dataset. The partial derivatives of E with respect to the weights in the first layer can then be calculated in terms of the partial derivatives of E with respect to the weights in the second layer. This method of computing the gradient of the error function in weight space ∇E by beginning with the network outputs and working backwards is called *back-propagation*.

The MLP is a particularly useful variety of neural network precisely because of the guarantee

that we can always calculate ∇E this way, and as we shall see, the gradient is crucial to any effective approach to training neural networks. The motivation for using a two-layer MLP in particular is a theorem that says any continuous function can be approximated to arbitrary accuracy by a linear combination of J sigmoids, for sufficiently large J [3].

A.2 Training MLPs

Before we begin the process of training an MLP network, we must initialize the network weights to random values. But not just any values will suffice; we don't want to initialize the weights in such a way as to saturate the sigmoidal activation functions feeding out of the hidden units. For very low a and very high a, the derivative of $g(a)$ defined in equation (10) is very small. Thus, the weights will be pushed towards extreme values during training in order for the output to better approximate its target, and we may end up with a network in which some of the weights are arbitrarily treated as very highly correlated with the targets while others are more or less ignored.

The solution to this apparent problem comes in rescaling the inputs by whatever linear transformations are necessary to ensure that the each input has mean 0 and standard deviation 1 when evaluated over the whole training dataset. We can then save the parameters of each rescaling transformation in order to apply the same transformations to any other datasets we may later use to test the network. Then the weights can be randomized by drawing from a higher-order Gaussian with a mean of 0, and a standard deviation computed as a function of the number of inputs D so as to ensure that the linear combination $\sum_{i=0}^{D} w_{ij}^{(1)} x_i$ is of order unity, and thus we get a nice, symmetric initialization.

The training process itself consists of two phases. First, we evaluate the gradient using back-propagation. Next, the gradient is used to update the weight vector according to a formula specified by a particular *learning method*. The gradient is re-evaluated using the new weights, and the whole process is repeated for some fixed number M of iterations over the training dataset, which are also called *epochs*. The iteration formula will thus be of the general form

$$\mathbf{w}^{(m+1)} = \mathbf{w}^{(m)} + \Delta\mathbf{w}^{(m)} \tag{12}$$

where m indexes the epoch.

As a simple example of a learning method, we can consider *gradient descent*, in which the

weight vector is updated according to

$$\Delta \mathbf{w}^{(m)} = -\eta \nabla E(\mathbf{w}^{(m)}) \tag{13}$$

where η is a positive parameter called the learning rate. We thus expect the gradient descent algorithm to seek out a stationary point on the *error surface* in weight space, where $\nabla E(\mathbf{w}^*) = 0$, and remain there indefinitely. Since $-\eta \nabla E$ always defines a direction of descent on the error surface, we can hope that the stationary point we arrive at this way will be a minimum, though of course it could also be a higher-dimensional saddle point of sorts.

The most serious difficulty with gradient descent lies in the fact that η is a fixed parameter. If η is too large, the algorithm has a good chance of overshooting the minimum it is trying to find, and possibly ending up in divergent oscillations about the minimum. If η is too small, convergence towards a minimum may be prohibitively slow. Thus, we are led to consider more complex learning methods.

Line search is a useful procedure for optimizing the magnitude of a step through weight space, provided we have already decided upon a direction. A line search algorithm proceeds by moving from the initial point $\mathbf{w}$ along the specified search direction $\mathbf{d}$ (which we will take to have unit norm) in steps of progressively greater length, and evaluating the error function at each step. Specifically, the algorithm iterates over l, setting

$$\begin{aligned} \alpha_1^{(l)} &= \alpha_2^{(l-1)} \\ \alpha_2^{(l)} &= \alpha_3^{(l-1)} \\ \alpha_3^{(l)} &= \tau \alpha_3^{(l-1)} \end{aligned} \tag{14}$$

where $\alpha_1^{(0)} = 0$, $\alpha_2^{(0)} = 0.01$, $\alpha_3^{(0)} = \tau \alpha_2^{(0)}$, and τ is a parameter set by the user, until the condition

$$E\left(\mathbf{w} + \alpha_3^{(l)} \mathbf{d}\right) > E\left(\mathbf{w} + \alpha_2^{(l)} \mathbf{d}\right)$$

is satisfied. Since the line search did not stop at the $(l\text{-}1)^{\text{th}}$ step,

$$E\left(\mathbf{w} + \alpha_1^{(l)} \mathbf{d}\right) > E\left(\mathbf{w} + \alpha_2^{(l)} \mathbf{d}\right)$$

We see that the line search has bracketed a minimum between α_3 and α_1 along the specified search direction. It then fits a parabola to the three points $E(\mathbf{w} + \alpha_1\mathbf{d})$, $E(\mathbf{w} + \alpha_2\mathbf{d})$, and $E(\mathbf{w} + \alpha_3\mathbf{d})$. The minimum of this parabola will necessarily be located at $\mathbf{w} + \alpha\mathbf{d}$, where $\alpha_1 < \alpha < \alpha_3$, and α is then taken as the magnitude of the step in the specified direction through weight space.

Obviously, the line search will be more precise for smaller values of τ, but it will also be slower. Performing a line search will clearly be slower than simply choosing a fixed step size η as in equation (13), since we must bracket the minimum and perform a parabolic fit for each epoch m. Still, we will likely gain more than we lose in computational efficiency given that the line search allows us to move to the vicinity of a minimum along any given search direction in just one epoch, and it will prevent divergent oscillations caused by too large values of η in regions of weight space where the error surface has high curvature.

Introducing line search has enabled us to choose a more optimal value for the magnitude of $\Delta\mathbf{w}^{(m)}$ at each epoch m. We might ask whether we can likewise choose a more optimal value for the direction. This question motivates the introduction of a powerful but computationally demanding class of learning methods called *Quasi-Newton Methods*, which include the *Broyden-Fletcher-Goldfarb-Shanno Algorithm* (BFGS).

We can understand the basic idea behind Quasi-Newton Methods by Taylor expanding the error function in the vicinity of a minimum $\mathbf{w}^*$ to second-order, since it will be approximately quadratic in the vicinity of a minimum. The first-order term vanishes because $\nabla E(\mathbf{w}^*) = 0$, so

$$E(\mathbf{w}) \approx E(\mathbf{w}^*) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^*)^{\mathbf{T}}\,\mathbf{H}\,(\mathbf{w} - \mathbf{w}^*) \tag{15}$$

where $\mathbf{H}$ is the Hessian (the matrix of second partial derivatives) evaluated at $\mathbf{w}^*$.[7] Taking the gradient of (15), we get

$$\nabla E(\mathbf{w}) \approx \mathbf{H}\,(\mathbf{w} - \mathbf{w}^*)$$

which we can solve for $\mathbf{w}^*$ to obtain

$$\mathbf{w}^* \approx \mathbf{w} - \mathbf{H}^{-1}\nabla E(\mathbf{w}) \tag{16}$$

The vector $-\mathbf{H}^{-1}\nabla E$ is called the *Newton Step*. Evaluated at any weight vector $\mathbf{w}$ in the vicinity of

[7] I have not made this dependence explicit for the sake of reasonably clear notation. Note that the Hessian *acts on* the vector $(\mathbf{w} - \mathbf{w}^*)$; it is not evaluated there.

a minimum, it points approximately towards the minimum, unlike the negative gradient, which may point in a much less useful direction.[8]

Quasi-Newton algorithms are defined by iteration formulae of the form

$$\Delta \mathbf{w}^{(m)} = -\alpha^{(m)} \mathbf{G}^{(m)} \nabla E(\mathbf{w}^{(m)}) \tag{17}$$

where $\alpha^{(m)}$ is found by a line search at each epoch, and the sequence of matrices $\mathbf{G}^{(m)}$ are increasingly accurate approximations to the inverse Hessian constructed in such a way so as to always be positive-definite. This ensures that the direction specified in (17) is guaranteed to be one of descent rather than ascent through weight space. Of course, this comes at a cost to the accuracy of the approximations $\mathbf{G}^{(m)}$, since the real Hessian is almost certainly not positive definite everywhere, but it guarantees that the algorithm will not diverge.

The BFGS algorithm differs from other Quasi-Newton methods in the particular iteration scheme it uses for $\mathbf{G}^{(m)}$, which is a hideously complex matrix equation constructed using various combinations of outer products involving $\mathbf{w}^{(m)}$ and $\nabla E(\mathbf{w}^{(m)})$. Presenting the iteration scheme here would not be enlightening, and BFGS is widely acknowledged to be superior to other Quasi-Newton algorithms anyway [3], so there is little point to fleshing out all the details.

However, I do want to draw attention to two specific points. First, the iteration scheme is initialized with $\mathbf{G}^{(0)} = \mathbf{I}$, the identity matrix. Thus, the first epoch of training with BFGS is simply a line search in the direction of the negative gradient, which should bring the weight vector close enough to a minimum that the quadratic approximation with which BFGS was motivated is applicable.

Second, roughly speaking, the BFGS algorithm will typically get us "close enough" to a minimum fairly quickly, and if we let it run much longer, it will continue to tweak the weights slightly in an effort to move even closer. But nothing guarantees that the minimum we have found is actually a minimum with respect to all of the many orthogonal directions of weight space, so this extra precision may be wasted if we actually need the algorithm to take a big step through weight space to the vicinity of a different minimum. Thus, practical implementations of BFGS should include a provision for a number of epochs $R < M$ after which $\mathbf{G}$ is reset to $\mathbf{I}$.

We can only meaningfully claim that a neural network has been trained if it has the capacity to *generalize* from the training data. We can evaluate a trained network by feeding it a *test dataset*

[8] Consider the simple example of a narrow valley in 2-dimensional weight space: the negative gradient always points directly across the valley, rather than towards the center.

independent of the one on which it was trained and computing the error on this dataset (called the *generalization error*). We may also want to compare the performance of two or more alternate variants of the same network. In this case, we feed both networks the same independent *validation set* and choose the network with the smaller *validation error*. This terminological distinction helps make the point that if one dataset is used to pick out the optimal network from its competitors, an independent dataset should be used to evaluate the generalization error. Otherwise, the network will be biased towards the validation set.

We could in principle compute the generalization and validation errors using equation (11). But it is often more convenient to use a normalized variant of the sum-of-squares error:

$$E' = \frac{\sum_{n=1}^{N} \|\mathbf{y}(\mathbf{x}^n, \mathbf{w}) - \mathbf{t}^n\|^2}{\sum_{n=1}^{N} \left\|\bar{\mathbf{t}} - \mathbf{t}^n\right\|^2} \tag{18}$$

where $\bar{\mathbf{t}}$ is the (unconditional) mean value of the target variable over the dataset in question. I discuss the properties of this error function at greater length in the body of my thesis.

A.3 Potential Problems with Neural Networks

Neural networks have the potential to be incredibly powerful when the relation between inputs and targets is unknown, very complicated, and non-deterministic. But they too have their limitations, and there are a number of perennial problems in optimization that can cripple a neural network if care is not taken to avoid them. In this section, I consider some of these problems.

First of all, we might naively expect that the greater the number of relevant inputs D we give our neural network, the more accurately it will be able to find the correct outputs. But this is only true up to a point. Like many other methods of nonlinear function approximation, neural networks suffer from something called the *curse of dimensionality*, which is the colorful name that has been given to the phenomenon of reduced performance for increasing D. Roughly speaking, this is because as the input space of the approximater increases, so does the size of the sample dataset needed to cover enough of the input space for the approximater to correctly extrapolate to the rest of it, and the rate at which it does so is often quite rapid.

The curse of dimensionality is the reason we cannot train a character recognition neural network by simply giving it one input for each pixel in the image to be classified, and, more to the point, the reason we cannot reconstruct angle by giving the network every discrete voltage reading and

the corresponding time from each digitized PMT pulse seen by the E14 DAQ system. It is worth stressing that there is no upper limit on D for which the curse of dimensionality is theoretically unbeatable, but for networks with large numbers of inputs, the required training set size N may be prohibitively large.

The observation that a neural network that took literally every piece of directly measurable information from the E14 DAQ system as an input would be outperformed in every sense by a network taking only a few energy and time readings, all of which are derived solely from the information in the DAQ system, is an example of the importance of *pre-processing*. Any process by which the dimensionality of the input data is reduced in such a way as to make each input correlate more strongly with the outputs can be considered a form of pre-processing.

For any given regression problem, there should be no ambiguity about the number of outputs C our neural network should have, and we have seen that the number of inputs D must be chosen according to some pre-processing scheme with regard to the curse of dimensionality. Assuming we're working with a 2-layer MLP, we might now think to ask about the optimal number of hidden units J. There is no simple answer to this question in the neural network literature. But a good start is to note that, for given C and D, $J + 1$ will be proportional to the number of weights W, which should in turn be significantly smaller than the number N of data points in the training set.

To see why this is, recall that the weights in a neural network are analogous to the coefficients in a W^{th}-order polynomial regression. If N is comparable in magnitude to W, the network may suffer from *overfitting*, which is to say the outputs model not only the underlying systematic relationship between inputs and targets but much of the stochastic variation as well. Since this variation will by definition differ between datasets with identical input distributions, overfitted neural networks will tend to generalize very poorly to independent datasets.

Overfitting can be avoided by taking N to be much greater than W, by a factor of at least 30 according to one common rule of thumb [6]. For given N, D, and C, this rule will give us a range of acceptable values for W and thus for J. Like all rules of thumb, the "factor of 30" rule is only approximate; while it's very unlikely that a network with $N > 30W$ will be overfitting, it may very well be *underfitting*, depending on W (and thus, for a given J, on D and C), as well as on the complexity of the underlying function to be learned, and on the amount of random variation about that function. Ultimately, there is no hard-and-fast rule for determining an optimal value for J. The best we can hope to do is compare the performance of networks trained with different values of J

and hope that an answer presents itself.

Another major problem plaguing neural nets is that of *local error minima*. None of the learning methods I discussed, nor any of their major competitors, are capable of discriminating between local and global minima, so the existence of many local minima can cause neural networks to finish training when they have not in fact minimized the error very far. We can combat the problem of local error minima by training a network many times in succession, randomizing the initial weights every time, and seeing whether any random initialization yields significantly lower generalization error than others. More specifically, we can compute the mean and standard deviation of the generalization error after training the network many times. A low standard deviation is a good indicator that the error has in fact reached a global minimum.

Yet another possible confounding factor for networks trained with sum-of-squares errors has to do with the whether or not the data is *single-valued*. I do not mean single-valued in the strict sense of the term, for we already know that the targets are not deterministic functions of the inputs. But the performance of a network traind with a sum-of-squares error may be seriously compromised if the underlying systematic relationship between the inputs and the targets is not single-valued.

When I introduced the sum-of-squares error function above, I cited the result that for a sufficiently large training dataset, the outputs of a trained neural network converge to the conditional averages of the corresponding targets with respect to the input vector. For a single-valued output function, these conditional averages will be good approximations to the systematic relationship we are trying to model. But if two distinct targets are associated with inputs occupying the same cluster in input space, the conditional average of the target with respect to the input vector might end up at a value between the two targets which is not associated with any input in that cluster. Therefore, in the case of a non-single-valued relationship between inputs and targets, the network outputs may be very poor approximations to the underlying systematic behavior.

Unfortunately, while it's very easy to glance at a graph to determine whether a function of a single variable is single-valued, this doesn't work for a neural network with, say, 19 inputs. The failure of neural networks when confronted with non-single-valued data can be addressed by adopting more complicated error functions, but only the sum-of-squares error function is implemented in ROOT, so I consider such measures outside the scope of my project.

B E391a Histograms

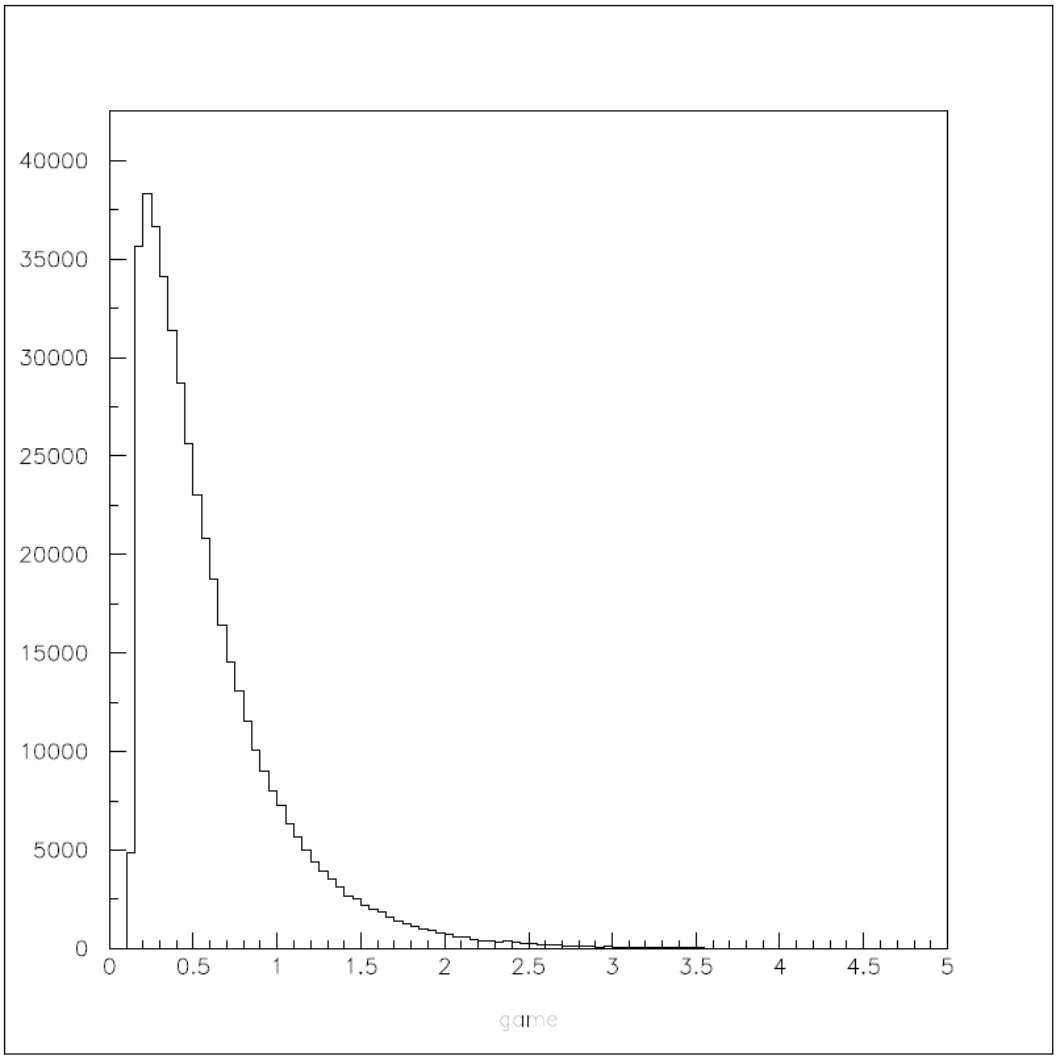

Figure 6: Distribution of ε (GeV) for photons produced by $K_L^0 \longrightarrow \pi^0 \nu\bar{\nu}$

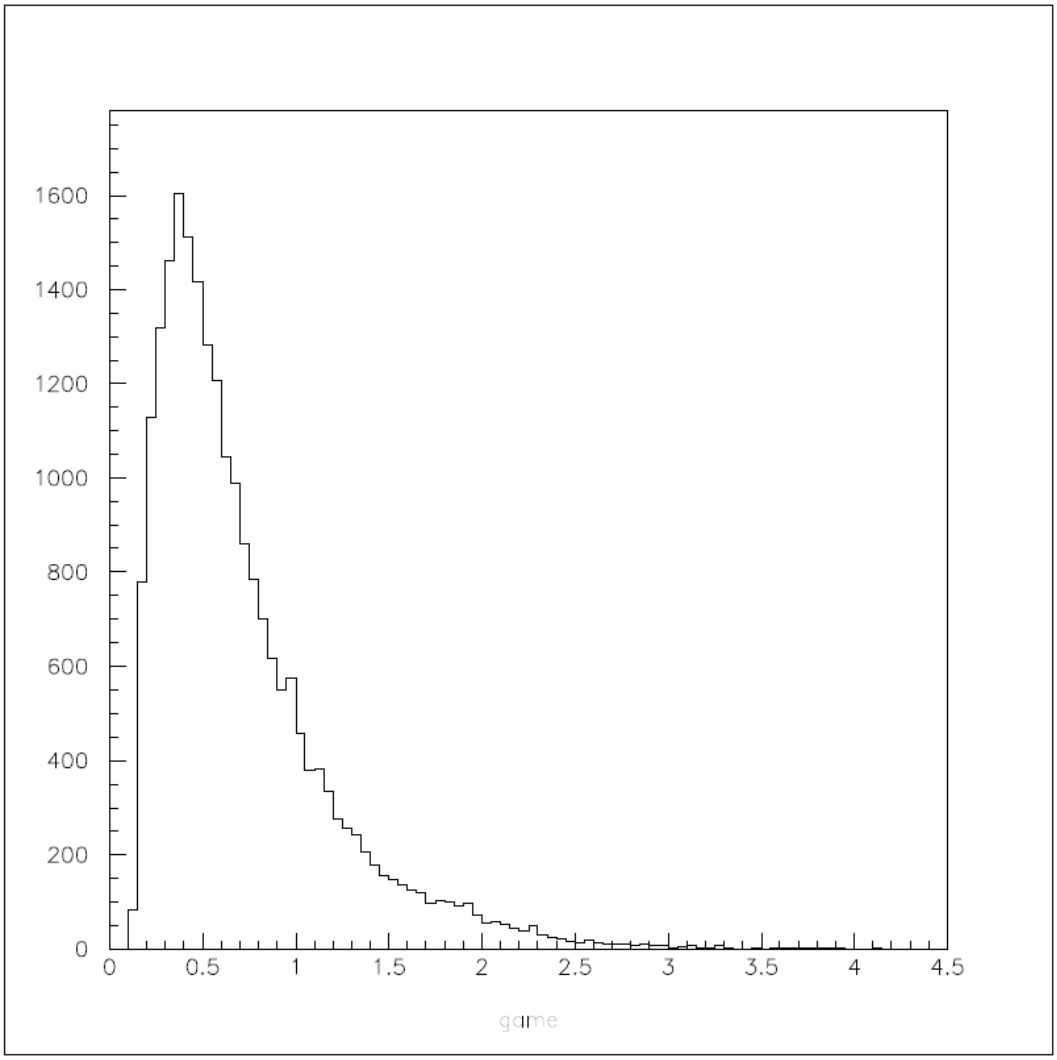

Figure 7: Distribution of ε (GeV) for photons produced by $K_L^0 \longrightarrow \pi^0\pi^0$

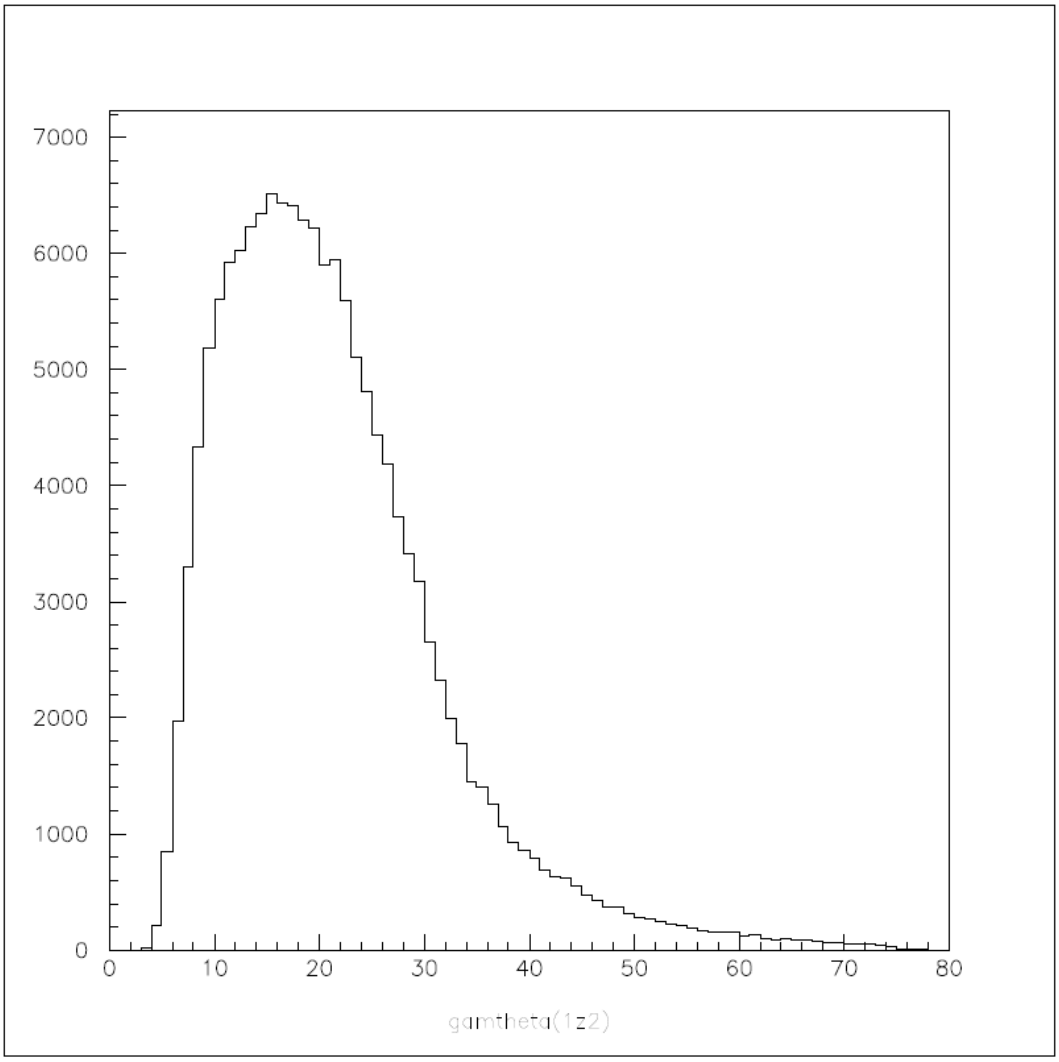

Figure 8: Distribution of θ (°) for photons produced by $K_L^0 \longrightarrow \pi^0 \nu\bar{\nu}$

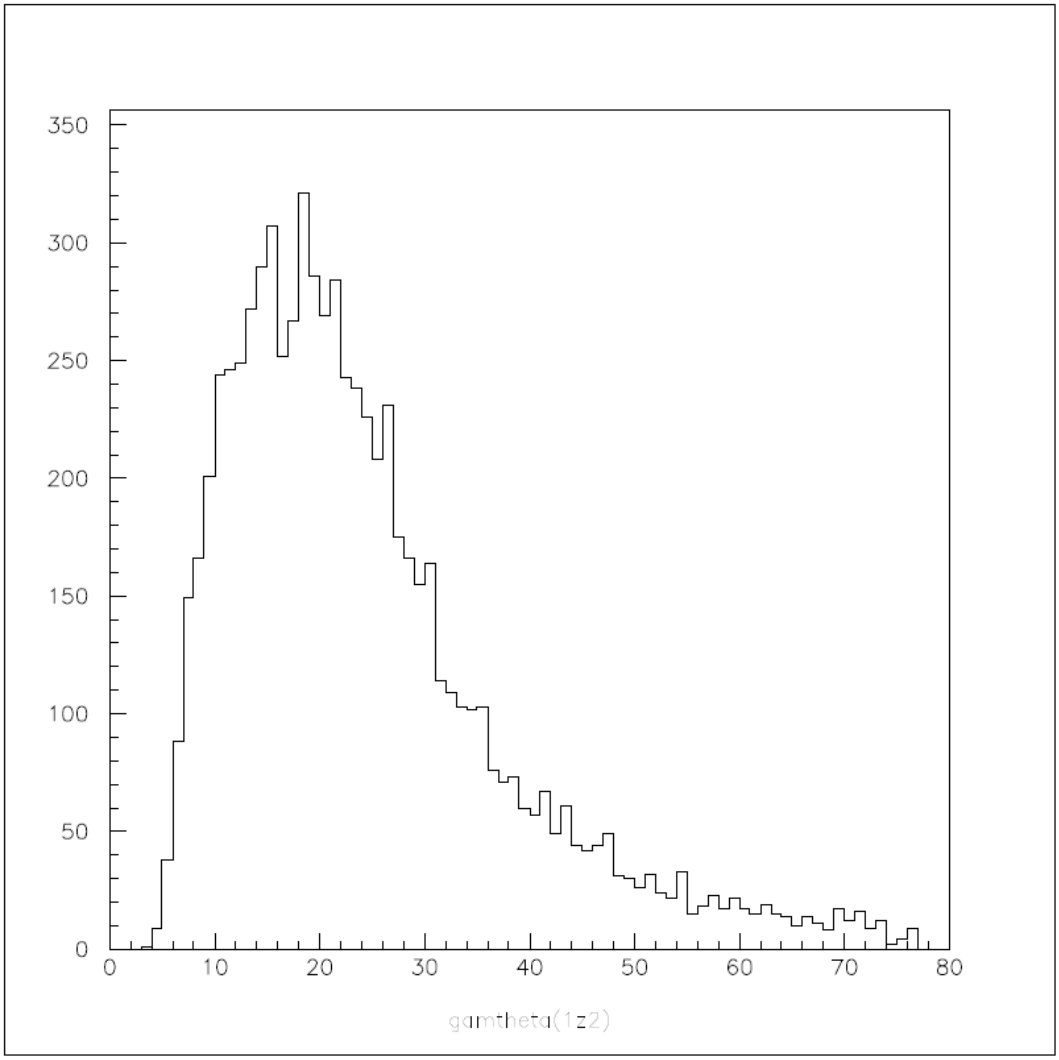

Figure 9: Distribution of θ (°) for photons produced by $K_L^0 \longrightarrow \pi^0\pi^0$

C Acknowledgements

First and foremost, I would like to thank my advisor, Professor Yau Wah, for providing me with a very enriching opportunity to work in a lab and conduct research as an undergraduate, and for entrusting this project to me.

I would also like to thank Jiasen Ma for his willingness to help me work through hours of inexplicable problems in GEANT, and for patiently answering all my questions without fail.

I would like to thank Eito Iwai, of the University of Osaka, for compiling and providing me with the Sendai beam test datasets I used to test my neural network.

I would like to thank Eugene Lee for the assembler.sh shell script, which has by now saved me hours that would have otherwise been wasted compiling user routines in GEANT.

I would like to thank Evan Hall for letting me partake of his apparently bottomless knowledge of LaTeX.

Lastly, I would like to thank Professors Mark Oreglia and Tom Witten for running the undergraduate thesis seminar this year, and providing me with invaluable feedback on this project.

References

[1] J. Ma. "Search for the Rare Decay $K_L^0 \longrightarrow \pi^0 \nu\bar{\nu}$." PhD Thesis, University of Chicago, 2009.

[2] K. Nakamura. *et al.* (Particle Data Group). "Particle Listings." Last modified January 15, 2010. http://pdg.lbl.gov/2010/listings/contents_listings.html

[3] C. M. Bishop. *Neural Networks for Pattern Recognition.* Oxford: Clarendon, 1995.

[4] M. Doroshenko. "Calibration of E391 Detector." Internal Technical Note, 2007.

[5] K. Nakamura. *et al.* (Particle Data Group). "Atomic and Nuclear Properties of Materials." Last modified April 2010. http://pdg.lbl.gov/2010/AtomicNuclearProperties/index.html

[6] W. S. Sarle. "Neural Network FAQ." Last modified May 17, 2005. ftp://ftp.sas.com/pub/neural/FAQ.html.

[7] ROOT Development Team. "ROOT User's Guide: Fitting Histograms." Last modified February 18, 2011. http://root.cern.ch/download/doc/5FittingHistograms.pdf