Summary of Work on IceTop Muons

John Franklin Crenshaw

August 2018

This document summarizes all of the scripts and iPython notebooks that were written for IceTop by John Franklin Crenshaw in the summer of 2018. The goal is to develop a neural network that can predict the number of muons intersecting IceTop for individual air showers. First is an outline that shows how the data scripts are used to convert the data from one form to another, and where in this process the plotting scripts and neural network notebooks come into play. Then in the following sections, the data, plotting, and neural network files are listed and explained. All of the files[1] described in this document (including this document) can be found in the GitHub repository at `https://github.com/jfcrenshaw/icetop_muons.git`.

Contents

[1]The i3 data files, the IceTop geometry file, and the numpy arrays of shower objects are not in the GitHub repository due to their large size. They can be found in found in /cr/users/crenshaw/icetop in the KIT computer system

1 Outline

There are basically 3 different types of files in this set. The first is the data scripts that apply quality cuts and repackage the data to be saved in various convenient forms. The second are the plotting scripts, which are used to make plots of the air shower data for analysis. The third are the neural network notebooks. These are iPython notebooks that use the neural network data to create neural networks to predict muon charge deposition. The use of these files can be seen in Figure 1.

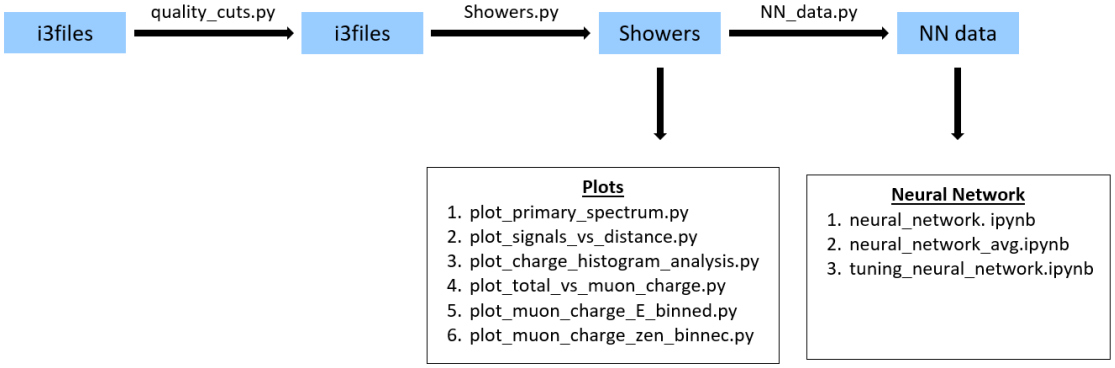

Figure 1: An outline of how the data scripts are used and where in the process the plotting scripts and neural network notebooks come into play.

The original IceTop simulation data is stored in the i3 files. These are stored in /cr/data01/hagne/John_project/Donghwa/. The first data script is quality_cuts.py which goes through the files in this directory, and saves all of the showers that pass the quality cuts in the data directory.

This reduces the number of i3 files, but it still takes a long time to read data from these files. To make analysis quicker, the script Showers.py takes all of the relevant data from the i3 files and stores them in shower objects. These shower objects are stored in numpy arrays and saved in the data folder as 'proton_showers.npy' and 'iron_showers.npy'. See the section "Shower Data Class" for more details.

Once the shower objects have been created by the Showers.py script, the plotting scripts can be used. These scripts go through all the shower objects and plot specific data in various configurations. For more information, look at the section "Plotting Scripts".

For the neural networks, it is convenient to make a single array for each shower that holds the handful of data points we want to use as input for the neural network. This is accomplished by the script NN_data.py. These arrays are saved in the data folder as 'NN_data.npy'. This file is then used for making the neural networks.

There are other data scripts, for example intersect.py calculates muon intersections from CORSIKA files and saves them in the shower objects. For more details on the data scripts, the plotting scripts, and the neural network notebooks, see the respective sections below.

2 Data Scripts

Following is a list of the scripts used to process and store IceTop shower data.

- **env-shell.sh -** script from Agnieszka that starts an icecube environment. It must be run before 'quality_cuts.py' and 'Showers.py'.
- **quality_cuts.py -** This script opens i3 files from 'data_location', checks which frames pass the given quality cuts, and stores the passing frames in new files located at 'cut_location' (set these variables at the top of the script). The quality cuts are:
  1. 5 stations are triggered
  2. $S_{125} \geq 1$
  3. $cos(\theta) \geq 0.8$
  4. core distance $\leq 400m$ (this is approximately equivalent to rejecting showers whose cores are outside the containment area)
  5. The tank with the largest signal is less than 300m from the center of IceTop (this is approximately equivalent to rejecting showers whose largest signals are on the edge of the array)
  6. the maximum signal is greater than 6 VEM
- **Showers.py -** This script goes through every file and frame of the i3files in the folder './data/i3files', and saves important data in the Shower data class. This data class (and some relevant functions) are defined in the file 'ShowerClass.py'. The class is explained in the following section. This script saves 'proton_showers.npy' and 'iron_showers.npy' in the 'data' folder. The data is saved as an array of shower objects, that can be loaded into another script using numpy.load().
- **intersect.py -** This script adds the number of muon intersections to the shower objects. It does this by taking a shower object, opening the corresponding CORSIKA muon file (located at '/cr/data01/hagne/John_project/CORSIKA/muonsPROPER/'), calculating how many muons hit which tank, and then storing these numbers in the shower object. The new shower objects (now containing muon numbers) are then saved in './data/proton_showers_new_save.npy'. This is because not all of the CORSIKA files are currently available. Hopefully, they will eventually all be available, and the script can be run to add the muon numbers to every shower. However, the script is currently set up to work with a file 'corsika_list.npy'. This file should contain a numpy array that is a list of the run numbers for showers that don't yet have muon numbers and for which there is now a CORSIKA file. I have been supplying the script a list of new files, running the script, then merging these new saves into the file './data/proton_showers_nmuons.npy' (there are currently no iron files). Thus I am slowly accumulating a list of showers that have muon numbers, that exists parallel to the full list of showers that contain no muon numbers. Eventually, './data/proton_showers_nmuons.npy' will contain every shower, and the other list ('./data/proton_showers.npy') can be deleted.
- **short.py -** This script collects all the showers that have $log_{10}(E_{primary}) \in (16.5, 17.0)$ and saves them in a new file. This creates a smaller subset of the showers, that can be used to more quickly prototype new scripts. The shorter lists are saved in the data folder with '_short' appended to the file name.
- **NN_data.py -** this script collects the data from all the showers to be used in the neural networks. It saves the relevant data in arrays in the file './data/NN_data.npy'. It also creates another set that averages each shower feature over the other showers in its run, and saves this averaged shower data in the file './data/NN_data_avg.npy'.

3 'Shower' Data Class

The shower data class is a class of objects I created to store relevant information about the showers. This allowed me to quickly access data to create new analyses and plots, without having to go back to the i3 files, which takes a long time. The class is defined in ShowerClass.py. The script Showers.py takes the data from the i3 files and puts it in Shower objects, which are then saved in numpy arrays in the data folder. More information can be found in the previous section.

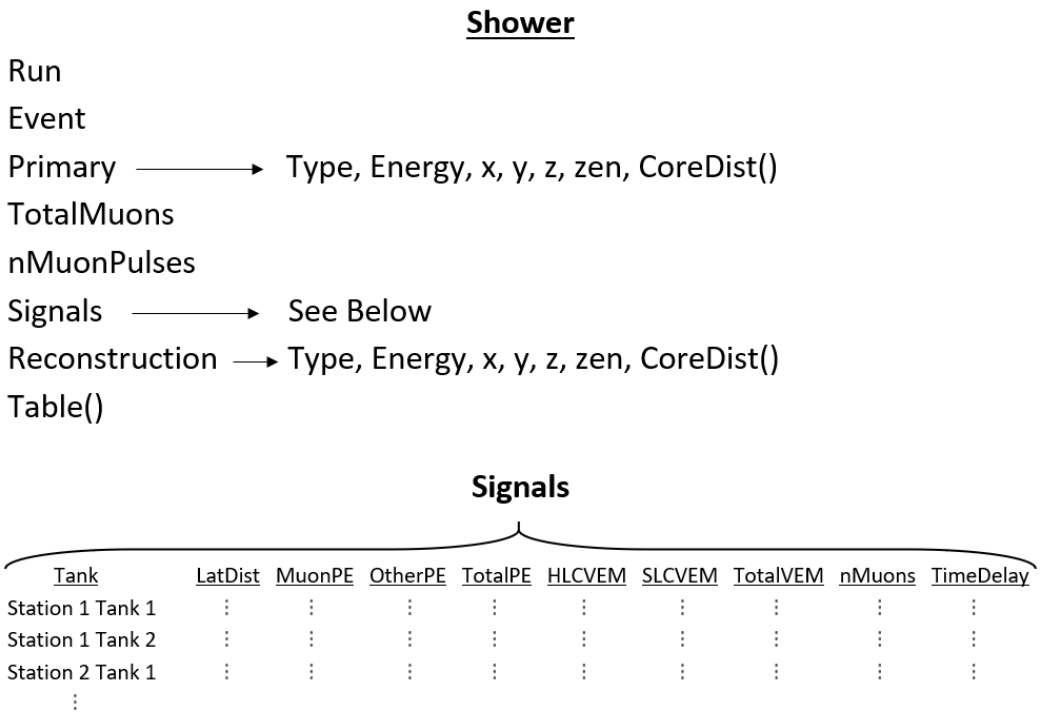

Figure 2: A visual summary of the shower class

The structure of the Shower class is as follows:

Each shower has 7 attributes:

1. shower.Run - the run number
2. shower.Event - the event number
3. shower.Primary - information about the primary particle. See below for more details
4. shower.TotalMuons - the total number of muons that intersected IceTop tanks in the shower. Currently this number is useless. The muon numbers need to be retrieved from the Corsika files using the intersect.py script
5. shower.nMuonPulses - the number of muon pulses in the shower
6. shower.Signals - information about all of the HLC and SLC signals in the shower. See below for more details
7. shower.Reconstruction - information about the shower reconstruction. See below for more details

The shower object also has a function Table(), that prints a summary of the data in the shower object. It prints some information about the shower as a whole, then prints a tank-by-tank table of the signals. It can take a distance argument so that it only prints tanks past a certain distance. For example, shower.Table(300) only prints the tanks that are $\geq 300m$ from the shower core.

Information about the primary particle: shower.Primary stores information about the primary particle. It has 7 attributes:

1. shower.Primary.Type - type of the primary particle
2. shower.Primary.Energy - energy of the primary particle
3. shower.Primary.x - x position of the primary particle
4. shower.Primary.y - y position
5. shower.Primary.z - z position
6. shower.Primary.zen - zenith angle of primary particle
7. shower.Primary.CoreDist() - the distance of the primary particle from the center of IceTop

Information about the shower signals: shower.Signals stores information about HLC and SLC sigals in the shower. This information is stored in 10 arrays, each of them ordered by station and tank number:

1. shower.Signals.Tank - array of station and tank numbers. For example, shower.Signals.Tank[0] = "Station1_Tank1". Note that Tank 1 is the tank with DOMs 61,62 and Tank 2 is the tank with DOMs 63,64.
2. shower.Signals.LatDist - array of lateral distances for each tank. For example, shower.Signals.LatDist[0] is the lateral distance from the shower core to Station 1 Tank 1.
3. shower.Signals.MuonPE - array of photoelectrons deposited by muons in each tank. For example, shower.Signals.MuonPE[0] is the number of PEs deposited by muons in Station 1 Tank 1.
4. shower.Signals.OtherPE - array of photoelectrons deposited by other particles in each tank.
5. shower.Signals.TotalPE - array of total photoelectrons deposited in each tank.
6. shower.Signals.HLCVEM - array of HLC signals in each tank (in units of VEM).
7. shower.Signals.SLCVEM - array of SLC signals in each tank (in units of VEM).
8. shower.Signals.TotalVEM - array of total signals in each tank (in units of VEM).
9. shower.Signals.nMuons - array of number of muons intersecting each tank. Currently this number is useless. The muon numbers need to be retrieved from the Corsika files using the intersect.py script
10. shower.Signals.TimeDelay - array of the time delay for the signal in each tank.

Information about the shower reconstruction: Information from the shower reconstruction is stored in shower.Reconstruction. It has the exact same structure as the Primary Particle data, except all of the values are reconstructed values instead of truth values. For details about the Primary Particle data, see above.

4 Plotting Scripts

Following is a list of the scripts used to create various plots. Every script's name starts with 'plot_' and creates a figure that lives in './figures/'. This list does not include the plots associated with the neural networks. Those can be found in the next section.

- **plot_primary_spectrum.py -** This script creates a plot that shows the primary particle energy spectrum for the i3 files in './data/i3files'. The plot is named 'primary_energy_spectrum.png'.
- **plot_signals_vs_distance.py -** This script creates 2 different plots that analyze the HLC, SLC, and total signals at various lateral distances. The first plot ('charge_vs_lateral_distance_{primary}.png') is the 2D histogram of signal charge vs lateral distance. The second plot ('signals_at_various_distances_{primary}.png') is the 4x3 multi-plot, that shows histograms of the various signals at a range of specific lateral distances. This script has 1 setting that can be adjusted: at the top of the file, set the variable 'element' to 1 for proton primaries, and to 2 for iron primaries.
- **plot_charge_histogram_analysis.py -** This script creates histograms of the charge deposited, breaking it down into muon/non-muon components, and HLC/SLC components. The plot is saved as 'analysis_charge_past_{distance}_{primary}.png'. The script has 4 settings that can be adjusted at the top of the file: primary type, cut distance, and lower and upper limits on primary energy.
- **plot_total_vs_muon_charge.py -** This script plots total VEMs vs muon VEMs. A line is fit by binning the points and averaging, but the original points are also plotted for comparison. The plot is saved as 'total_charge_vs_muon_charge_past_{distance}_{primary}.png'. The script has 6 settings that can be adjusted at the top of the file: primary type, cut distance, and lower and upper limits on charge deposited and primary energy.
- **plot_muon_charge_E_binned.py -** This script makes a plot of signals vs VEMs from muons. It bins showers by energy ranges (found in settings) and plots each range as a different color. The points on the plot are also binned and averaged, and a line is fit to the averages. The plot is saved as 'muon_charge_E_binned_past_{distance}_{primary}.png'. The script has 5 settings that can be adjusted at the top of the file: primary type, cut distance, lower and upper limits on charge deposited, and an array of energy ranges.
- **plot_muon_charge_zen_binned.py -** This script does the same thing as the previous script, except it bins by zenith angle rather than energy. The plot is saved as 'muon_charge_zen_binned_past_{distance}_{primary}.png'.

5 Neural Network Notebooks

Following is a list of the Jupyter Notebooks created to make neural networks. Every notebook's name starts with 'neural_network_'.

- **neural_network.ipynb -** The initial neural network notebook. It creates a simple neural network and some plots to analyze its effectiveness.
- **neural_network_averaged.ipynb -** This was a brief look at training the neural network using averaged shower data. It uses 'NN_data_avg.npy' instead of 'NN_data.npy'. This method was abandoned.
- **neural_network_manual_tuning.ipynb -** This notebook is manual tuning of the neural network. I made many different networks by hand, and printed some statistics and plots to compare their effectiveness. I used this to determined the best configuration I could find relatively quickly. However, the tuning process should be automated to be much more exhaustive and to tune all of the parameters simulateously, rather than individually. An example of how that might work can be found in "neural_network_automated_tuning.ipynb"
- **neural_network_manual_tuning.ipynb -** This notebook demonstrates how tuning the neural network should be perform. It is much more thorough and can systematically look for the best combination of hyperparameters. However, this process takes a long time, which is why I started with manual tuning as seen in the previous notebook.
- **neural_network_checking_inputs.ipynb -** This notebook investigates which inputs the neural network really uses. I discovered that it was really only using the charge past 400m, and ignoring all of the other information. Thus it seems like the neural network probably doesn't perform any better than a simple linear fit would.