Contents

1 Introduction

The IceCube Neutrino Observatory is the largest neutrino experiment ever constructed. It consists of over 5000 light-detecting digital optical modules (DOMs), which are burried in a volume of over $1\,\mathrm{km}^3$ in the ice of Antarctica. Such an experiment with as many individual detectors naturally yields tons of raw data, hence manual interpretation/analysis is not appropriate. One way to analyse the data is the use of Machine Learning (ML) models, more specifically in our case the use of Graph Neural Networks (GNN). The network relavant to this work models the triggered detectors, as point clouds which are then connected into a graph representing the geometry and data gained from the measurement. The data fed to the network for interpretation/evaluation includes the measurements and positions of the detectors, as well as the time of measurement. In this thesis an effort is made to feed additional data to the network, which describes the local properties of the ice, in order to help the network understand the data context better.

2 Particle Physics

Search for the building blocks of matter has long been an interesting field for many scientists and philosophers. For a long time humans believed elements to be fundamentally different from another. Actually not too long ago, in the late 1800s-1900s it was discovered that matter is made from atoms, which in turn consist of protons, neutrons and electons. The idea that these, long thought of as elementary particles, are dividable even further into quarks has not even been around for more than 60 years at the time this work is written.

2.1 Standard Model

The Standard Model of particle physics (shown in figure 1) is a theory to describe all currently known elementary particles. It also describes three of the four fundamental forces we have yet discovered, them being electromagnetic, weak and strong interactions, with gravity being the force that is not (yet, hopefully) able to be integrated consistently into the theory.

Although the Standard Model explains most phonomena fully and logically, gravity, dark matter, as well as a few other phenomena, are not able to be properly explained/modelled by the Standard Model. The model is a great achievement of science, and new particles continue to be discovered by great scientists to this day. But the weaknesses of it mark the theory as incomplete.

2.2 Neutrinos

Neutrinos, shown at the bottom of the Standard Model above, are uncharged leptons first hypothesized by Wolfgang Pauli in 1930 in order to fix the observed possible variation of electron energy during beta decay. He reasoned, that since energy must be conserved, the 'missing' energy is carried by a particle previously not detected/thought of by anyone.

$$n^0 \longrightarrow p^+ + e^- \tag{1}$$

Therefore he extended the known equation of the beta decay (above) with his new electron neutrino particle, which solved the until then continous beta decay spectrum.

$$n^0 \longrightarrow p^+ + e^- + \bar{\nu}_e^0 \tag{2}$$

Since neutrinos exist everywhere in huge numbers, this raises the question why have not been detected before. This is due to them interacting only via the weak force over short distances (as well as gravity), which makes them very hard to detect. With an estimated cross section of $\sigma \sim 10^{-38}\,\mathrm{cm}^2 = 10^{-42}\,\mathrm{m}^2$ their probability of interaction is very small. For comparison, the standard unit for crosssections in particle physics, the `barn`, equals $1\,\mathrm{b} = 100\,\mathrm{fm}^2 = 10^{-28}\,\mathrm{m}^2$.

Because of their low interaction probability and resulting high penetration neutrinos carry information of events throughout the universe. Observing them may lead to insight and understanding what happened a long time ago in a galaxy far, far away....

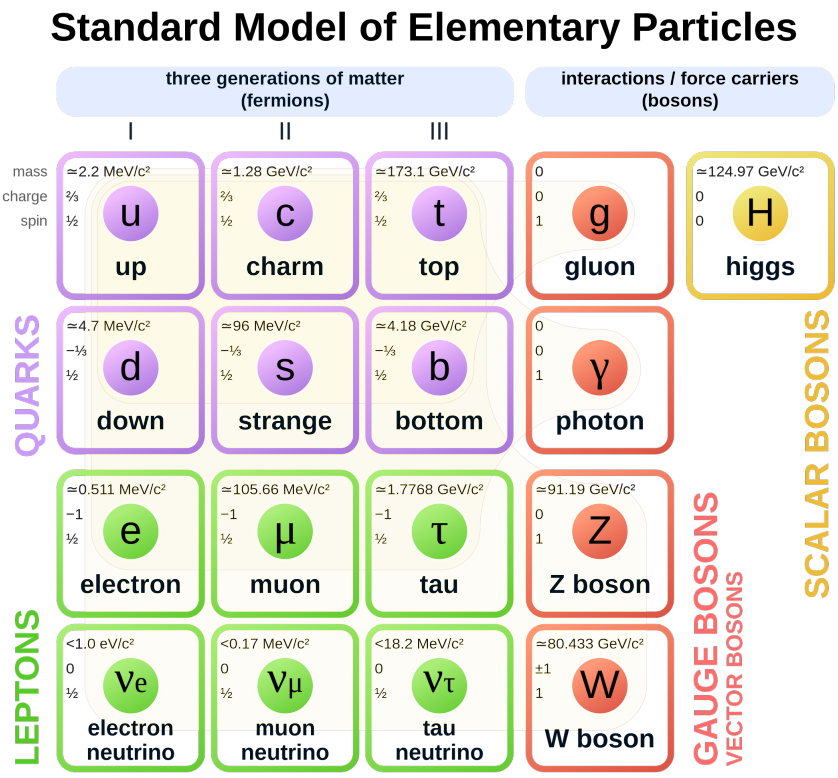

Figure 1: The Standard Model of particle physics.

2.3 Neutrino Oscillation

In the 1960s the solar neutrino flux was measured in the Homestake experiment performed by Ray Davis and John N. Bahcall. They measured the amount of neutrinos interacting inside a $380\,\mathrm{m}^3$ tank of perchloroethylene. The solar neutrinos originate from the nuclear fusion process, mostly from the pp-chain (proton-proton).

$$p^+ + p^+ \longrightarrow {}^2H + e^+ + \nu_e \tag{3}$$

The observed neutrino flux was only about $\frac{1}{3}$ of the theoretically expected value. This problem puzzled scientists for decades until it was resolved around 2002 through the explanation of neutrino oscillation. Neutrino oscillation is a flavour transition of the neutrino, where any neutrino given will oscillate/change its flavour (electron, muon, tau) when traveling through space. Therefore statistically only about every third neutrino will be a electron neutrino, and only those are being measured by the Homestake experiment - explaining the measured flux deficit.

The three neutrino flavours carry different masses, therefore they travel through space with different velocities. The actual weight of any neutrino is a superposition (linear combination) of the three individual mass eigenstates.

Because the different neutrino mass states travel through space at different rates, a phase shift is introduces as the neutrino travels. This results in a pure electrion neutrino gaining some mixture of mu and tau neutrino as well. As each mass state is periodic, after some distance the neutrino will periodically return to its pure electron form, as long as the states maintain coherence. If coherence is reduces, the neutrino will consist increasingly of an equal mixture of all states, therefore the Solar Neutrino Problem is solved.

The superposition state of a neutrino can be described by the mixing angles θ_{ij} between any two states i and j.

TODO: Right formula

$$\begin{bmatrix} \nu_e \\ \nu_\mu \\ \nu_\tau \end{bmatrix} = U \begin{bmatrix} \nu_1 \\ \nu_2 \\ \nu_3 \end{bmatrix} = \begin{bmatrix} U_{e1} & U_{e2} & U_{e3} \\ U_{\mu 1} & U_{\mu 2} & U_{\mu 3} \\ U_{\tau 1} & U_{\tau 2} & U_{\tau 3} \end{bmatrix} \begin{bmatrix} \nu_1 \\ \nu_2 \\ \nu_3 \end{bmatrix} \tag{4}$$

2.4 Charged/Neutral Current Decays

As touch on earlier, neutrinos interact in two distinct ways, gravity and the weak force. The neutrino takes part in two weak interaction types, charged current and neutral current interactions. The charged current interaction is mediated by the W-boson and allows for interaction with a charged particle. The Z-boson on the other hand allows for interaction with neutral particles and only affects spin and momentum of the particles involved.

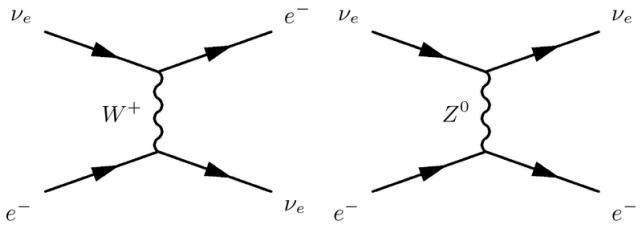

Figure 2: Feynman diagrams of charged and neutral current interactions.

2.5 Cherenkov Radiation

The detectors (DOMs) within IceCube can not measure incoming neutrinos directly, instead they detect Cherenkov Radiation. This electromagnetic radiation is emitted when a charged particle is moving through a dielectric medium at a speed greater than its phase velocity (speed of light within the medium). This phenomena is most familiar for its occurence in nuclear reactors, which glow blue because of emitted beta particles that travel faster then speed of light in water.

$$c_{Water} = \frac{c_{Vacuum}}{n_{Water}} = \frac{3 \cdot 10^8 \, \frac{\mathrm{m}}{\mathrm{s}}}{1.3} = 2{,}25 \cdot 10^8 \, \frac{\mathrm{m}}{\mathrm{s}} \tag{5}$$

When a charged particle travels through a dielectric medium it polarizes the surrounding molecules. After the charged particle has passed the molecules rearrange themselves into their original orientation. This change in charge distribution causes electromagnetic radiation known as Cherenkov Radiation. In case the incoming particle travels faster then the speed of light within the medium, a wavefront is created trailing the particle, similar to a sonic boom (see figure 3).

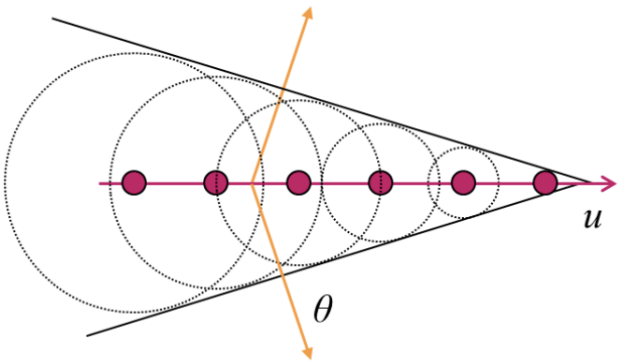

Figure 3: An illustration representing a Cherenkov Radiation wavefront.

3 IceCube

3.1 Location and Structure

The IceCube Neutrino Observatory is located at the geographic South Pole, where it makes use of the abundance of natural ice as material for the incoming particles to interact with. Over a volume of over $1\,\mathrm{km}^3$ 5160 Digital Optical Modules (DOMs) have been placed over a span of about 10 years. The DOMs are positioned on 86 individual strings in a hexagonal grid pattern at depths between 1450 m and 2450 m (shown in figure 4). The strings are placed about 125 m apart horizontally and all DOMs are spaced about 17 m vertically. Each DOM contains a 10" Photomultiplier Tube (PMT) that faces downwards, towards the bedrock below the ice. In the center of the detector, where the ice is clearest, additional strings have been placed, which increases the detector density in that volume (DeepCore).

The detector was built in order to study neutrino and muon interactions with energies between singular GeV up to a few PeV. It was build so large, not only to increase the probability of a neutrino interaction within its borders, which is minimal (see section 2.2). But also to provide enough space in order to hopefully capture full traces of interactions, which can extend up to a few hundred meters in length, especially interactions with high energies can involve over 100 DOMs.

3.2 Working Principle

The neutrino interaction with the ice produces charged particles, which in turn emit Cherenkov radiation when interacting with the ice. This near-UV radiation is then recorded by the PMTs and registered as a DOM trigger.

However, not all DOM triggers are caused by a neutrino interaction. Radioactive decay, as well as noise of the electrinic components are also recorded ($> 1000\,\mathrm{Hz}$). These events are of course not desired, and many ways of filtering have been implemented to combat such noise in the recorded data.

At trigger level a minimum number of DOMs have to be triggered for them to be registered as a single event. These triggers have to occur within a specified time interval (8 DOMs within $5\,\mu\mathrm{s}$), for them to be counted towards one event. This tries to ensure that an event is does not solely consist of multiple noise hits.

Additionally a local filtering is applied to all events, it filteres for triggers that happen closeby (150 m radius with a given time frame of 1000 ns). This assures that all DOMs registered as a single event are locally correlated and therefore most likely to be caused by a single neutrino interaction. For example a single noise-influenced DOM at the outside of the detector is not included in an actual event at the center of the detector.

3.3 Ice Variance

As the ice of Antarctica was not created all at once irregularities within the ice are present. These irregularities influence the measurements of IceCube in various ways. For example a high concentration of dust at a depth of around 2000 m (dust layer) results in a higher absorption rate than elsewhere in the detector. Besides absorption rate, the amount of scattering and birefringence (as well as other properties) vary over the volume, since they were formed at different times, with inconsistent densities and temperatures.

This work will incorporate measurements of ice properties measured by the SPICE project (see section 5) into the learning process of a neural network (see section 4 and 6.1) in order to try to help it understand the influence of the ice on neutrino events and improve its performance.

3.4 Retro

The most commonly used algorithm for reconstructing events used in IceCube is Retro (Retro Reco). Retro reconstructs events by simulating photons traveling trough the ice between DOMs and builds lookup tables mapping simulated physics to observed DOM detections. By comparing a recorded event to these tables and looking for the entries that are most likely to have occured (the entries that have the highest aggreement between them), it predicts the values of the event.

Since Retro provides great accuracy and is widely used within IceCube, it will serve as a baseline for all comparisons in this work.

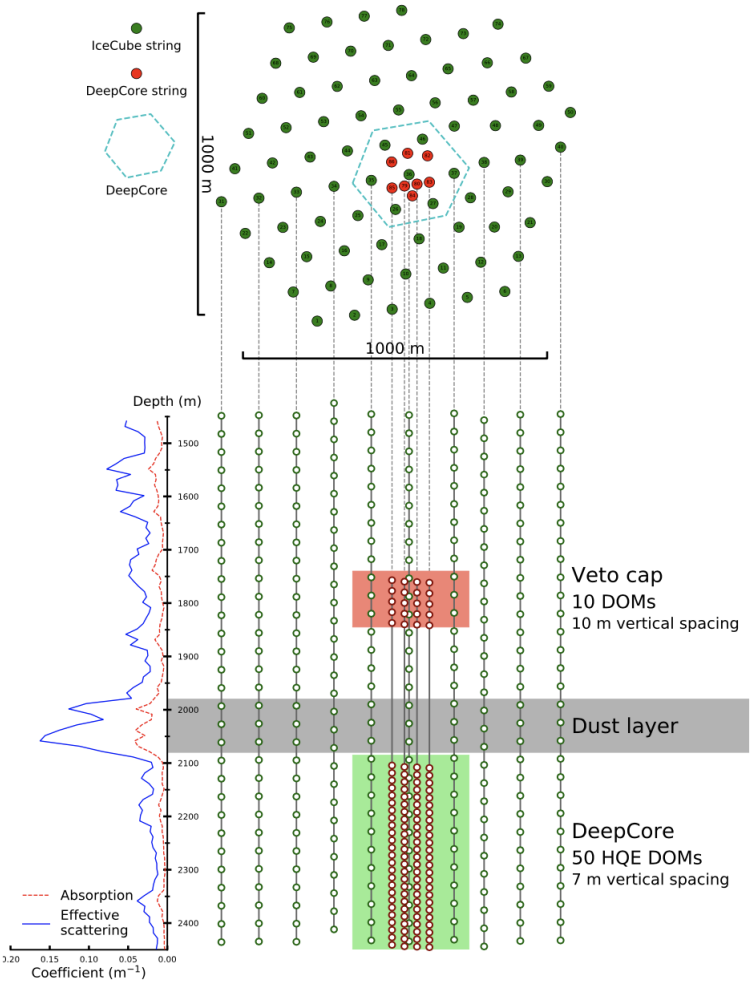

Figure 4: The structure of IceCube.

3.5 Simulation Data

As will be aparant in section 4.3, it will be necessary for the process of this work to have labeled data of the behaviour of IceCube. That is, having a dataset with not only the event recorded by IceCube, but also the true information of the particle it came from. As this information is of course not known for recorded events (this is what we want to know in the first place), simulated data is used.

The used datasets are generated by `Genie`, a state-of-the-art neutrino simulation engine. It provides the most physically accurate data accessible, because it simulates neutrino interactions with all ice molecules.

3.6 Event Variables

For each recorded event IceCube produces a set of data consisting of the measurements each DOMs participating in that event has made. This is the data that is available for real recorded events, as well as for the simulated data. The following table will outline the variables generated by each DOM and will serve as input for the Neural Network introduced in section 6.1.

Variable Name	Description
dom_x	X coordinate of the DOM
dom_y	Y coordinate of the DOM
dom_z	Z coordinate of the DOM
dom_time	Time the DOM was triggered
dom_charge	Charge the DOM measured
rqe	Relative quantum efficiency
pulse_width	Width of the pulse

For the simulated dataset there exists an additional set of data, the truth table. It holds the true information about the incoming particle that produces the data above. It only exists for the simulated data (actually the simulated data is made from it), and serves as the expected output for our Network to learn (see section 4.3).

Variable Name	Description
energy_log10	The total energy of the incoming particle [log10 GeV]
position_x	X coordinate of the incoming particle vector
position_y	Y coordinate of the incoming particle vector
position_z	Z coordinate of the incoming particle vector
azimuth	Azimuth angle of the incoming particle vector in a spherical coordinate system
zenith	Zenith angle of the incoming particle vector in a spherical coordinate system
pid	Particle ID

4 Machine Learning

The field of Machine Learning (ML) focuses on building and studying methods/models that are able to 'learn' from (sample) data. ML models learn to make decisions/predictions based on the sample data (training data) it is provided, with no logic being directly programmed into them. In order to learn and generalize to a given problem a model relies on a large set of training data in order to understand varying scenarios and contexts. When a large exhaustive dataset is used for learning, the resulting model can adapt to many different situations. This is why they are used a multitude of different areas, from voice and image recognition, self-driving cars, to deciding on whether or not a given client should receive a loan.

4.1 Fully-Connected Feed-Forward Multi-Layered Perceptron Networks

This section will explain how a simple neural network works and operates, by taking a fully-connected feed-forward multi-layered perceptron network (MLPs) as an example.

Such a neural network consists of a fixed number of layers each with a fixed number of nodes (can vary by layer). The layers between the input and output layers are called hidden layers. The more of them exist, the 'deeper' the network gets. As represented in figure 20, the layers are interconnected by edges, which connect all nodes from the previous layer to all nodes of the next layer (fully-connected). This structure and dataflow is supposed to simulate what happens in the brain. As long as edges only 'move' data into one direction, that is to speak the connections do not form circles/loops, the network is called feed-forward.

To more mathematical describe how the network operates: Data in nodes x_i of the previous layer is transformed/scaled by the edges weight w_{ji} and offset by the bias b_i of the next node. Therefore the data h_i of the next layer is given by:

$$h_i = \sum_j w_{ji} x_j + b_j \tag{6}$$

In matrix notation the transformation for all nodes in a layer can be expressed in one formula, where for any given layer index l: h_l is the data values, W_l is a matrix of edge weights and b_l is the bias (offset).

$$h_{l+1} = W_l h_l + b_l \tag{7}$$

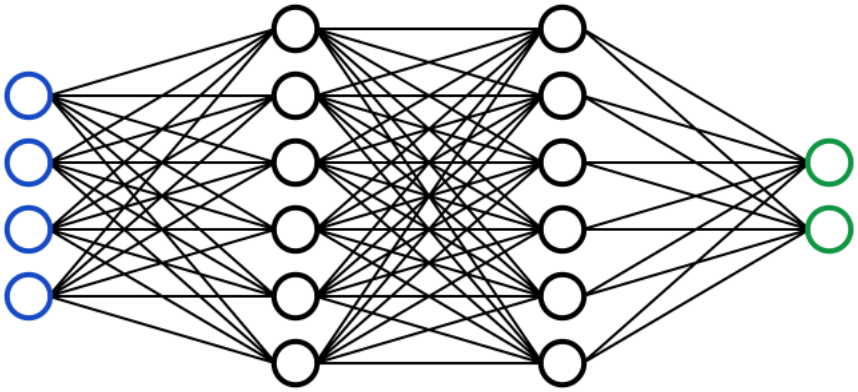

Figure 5: A graphical representation of a fully-connected feed-forward multi-layered perceptron network.

Therefore a simple NN can be thought of as a repeated process of matrix-vector multiplications and vector additions. Such an algorithm produces behaviour that can be adjusted to many given problems, as long as the size of the network is adequate to the problem at hand. The size of the network is usually determined by the number of learnable parameters, which are adjusted by the number of layers and their width. Because MLPs only contain linear operations, their behaviour is limited to modeling linear relationships. This is a limitation that might still work well for very simple applications, more difficult problems may require more dynamic behaviour. To produce such Activation Layers are used and built into the network.

4.2 Activation Layers

In order to introduce non linear behaviour to the network, Activation Layers are applied to the data at selected layers as it is transformed and fed through the network. Activation Layers are differentiable but typically non-linear functions, we will see in section 4.5 as to why they have to be diffentiable.

One widely used activation functions and also most relevant to this work are ReLU (Rectified Linear Unit) and LeakyReLU (Leaky Rectified Linear Unit). They both consist of two parts, in the positive domain both are $f(x) = x$, but in the negative domain ReLU is $f(x) = 0$ and LeakyReLU is $f(x) = c * x$ where c is a parameter.

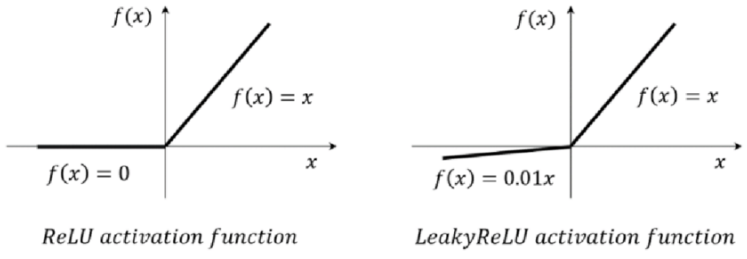

Figure 6: Plots of ReLU and LeakyReLU activation functions.

4.3 Supervised Learning

The type of NNs we discuss relevant to this work all learn by the principle of Supervised Learning. Supervised Learning is the process of learning based on sample input-output data (training data), which is fed into the network. Since training data has to be created before a model can be trained, this limits the applicability to problems, where a solution (truth) is known. For example the best next chess move is not easily decided on, therefore this approach would not be feasable for a chess problem.

By comparing the network's results with the truth during training a Loss is calculated by a Loss Function (see section 4.4). This loss represent the 'wrongness' of the network. It is then used by the Optimizer (see section 4.5), an algorithm that adjusts the weights and biases (parameters) of the network in such a way, that the loss decreases and therefore the results are getting closer to the truth each iteration. This adjustment of parameters based of the loss is only possible because all the operations the network performs are differentiable and the gradient of the 'loss landscape' (n-dimensional space) can be calculated.

4.4 Loss Functions

There are many Loss Functions a network architect can choose from. Depending on the characteristics of the function, the networks learning behaviour changes, as we understand in the following sections.

In general a loss function produces a minimal value (0) if the networks output y_i matches the expected truth $\hat{y}_i$.

Mean Absolute Error / L1 Loss

L1 Loss or Mean Absolute Error Loss returns the absolute difference beween the network output and the truth, which is averaged over the individual training batch.

$$MAE = \frac{\sum_i^n |y_i - \hat{y}_i|}{n} \tag{8}$$

Mean Squared Error / L2 Loss

L2 Loss or Mean Squared Error Loss behaves different to L1 Loss in the way that it does not calculate the absolute difference, but instead squares the difference of each value before averaging.

$$MSE = \frac{\sum_i^n (y_i - \hat{y}_i)^2}{n} \tag{9}$$

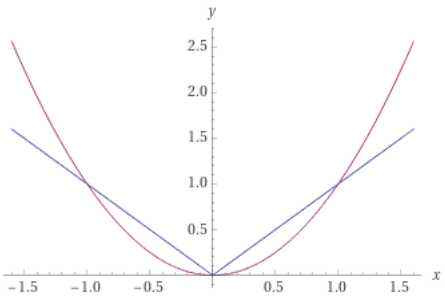

Figure 7: A comparison of L1 and L2 Loss.

As seen in the figure above, the behaviour of L1 is different when compared to L2. This has consequences for the learning behaviour of the network, in the case of $|y_i - \hat{y}_i| > 1$ L2 Loss has a greater values, which will cause the optimizer to change the networks parameters more drastically than in the case of L1. This can be an advantage or disadvantage depending on what you are interested in achieving and the scale of output data of your network. Another advantage of MSE loss is the tangent and differentiable behaviour around 0. By having the gradient approach 0 near the origin this loss functions ensures a 'homing in onto the minima' effect, which makes it harder for the optimizer to overshoot the minima.

Log-Cosh Loss

Another popular loss function is the Log-Cosh Loss Function. It combines the low gradient near 0 advantage of L2 Loss with linear loss behaviour of L1 Loss.

$$LC = \frac{\sum_i^n log(cosh(y_i - \hat{y}_i))}{n} \tag{10}$$

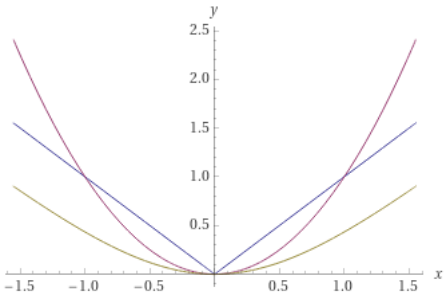

Figure 8: A comparison of L1, L2 and Log-Cosh Loss.

4.5 Optimizers

The task of an optimizer is, as its name states, to optimize a given function. In our case it optimizes/reduces the loss of the network by iteratively tweaking its parameters, which are initialized randomly. The optimizers knows what parameter to adjust in which way by performing a stochastic gradient descent, it calculates the gradient based off a subset (batch) of the training data.

It decides on how to tune these parameters based on its underlying algorithm and optional hyperparameters. Probably the most important hyperparameter is the learning rate, it is basically the step-size, the amount the optimizer changes the network's parameters each iteration. When the learning rate is too high, the optimizer traverses the loss landscape (see figure 9) too quickly, which may result in minimas being skipped over. A too low learning rate will result in the training loop taking much more time than necessary in order to find a minima. As the choice of selecting a learning rate is non-trivial, one can apply a learning rate finder. It runs the training loop with many different learning rates in order to find a value that converges quickly, while keeping its divergence during training minimal.

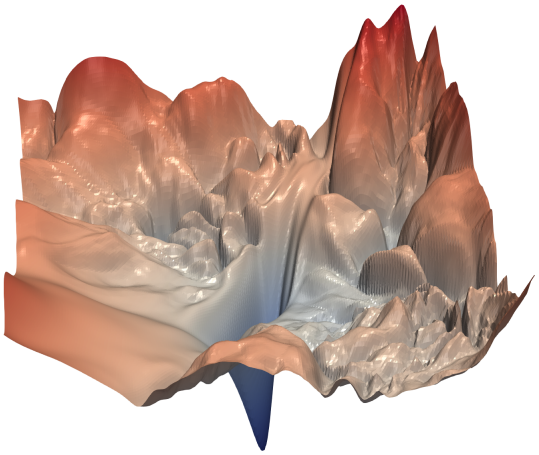

Figure 9: A simplified 3D representation of the loss landscape of a complicated function.

Adam

Adam (Adaptive Moment Estimation) is one of the most popular optimizers, mainly due to its adaptive step-size, which allows it to change the network's parameters more drastically when it is confident the loss landscape is smooth. This makes it a very versatile optimizer in many scenarios, since it automatically

adjusts its learning rate in order to speed up training. It starts of with the user-provided base learning rate, but then assigns each parameter its own learning rate which it adjusts dynamically.

4.6 Overfitting and Early Stopping

Overfitting

When a large enough neural network trains on a dataset for long enough, it will eventually reach a perfect loss on the training data since it will adjust its parameter to match/model the dataset at hand. Its predictions get increasingly closer to the expected values as the loss is decreased over time. However, once the trained model is evaluated another dataset, it is apparent, that the network has adjusted itself to the specifics of the dataset it has been trained on. Thus it fails to generalize to the underlying truth, because picked up on the noise in the training data. This is a problem that has to be solved, because the network is expected to perform its best even on new data it has never seen before.

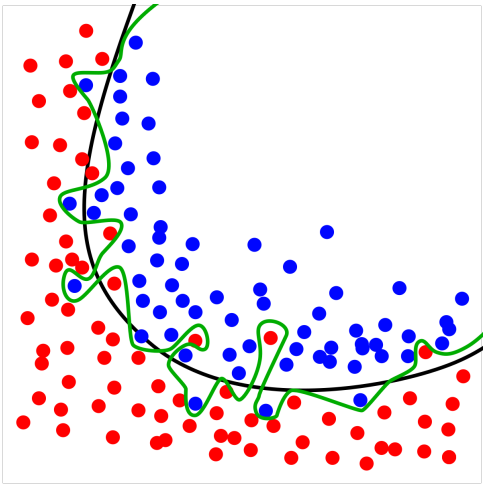

Figure 10: A graphical representation of Overfitting. In this case the network adjusts to the details of the noisy training data (red and blue points) instead of picking up on the general black division of the colors.

Early Stopping

The aim of Early Stopping is to stop the training process at the point where the network performs its best on not yet seen before data. The point at which it has not yet picked up on the specifics (noise) of the training data. As vaguely introduces in the previous section, the whole dataset that is available for training is usually split into three parts: Training, Validation and Test Datasets.

The Training Dataset is fed to the network in order to calculate gradients (backpropagate) in order to descent in the loss landscape. The Validation Dataset on the other hand is only fed to the network in order to evaluate its performance (calculate its loss), but never to backpropagate. Finally, after training, the Test Dataset is used to evaluate the networks final performance, it is used seperately from the Validation Dataset in order to suppress any influence of it being used as the indicator of when to stop training.

More accurately, after each Epoch (the network has seen all of the Training Data exactly once) it calculates and logs the networks loss on the Validation Dataset. First the loss on the training data, as well as the loss on the validation data, both decrease together (see figure 11). After a certain number of Epochs, the validation loss will incease again due to the network overfitting to the training data. When this happens Early Stopping will continue training for a set number of epochs (patience) before stopping the training and rolling the networks parameters back to the state it was when the Validation Loss was minimal. This results in a optimally performing network, that generalizes as good as possible to the underlying problem.

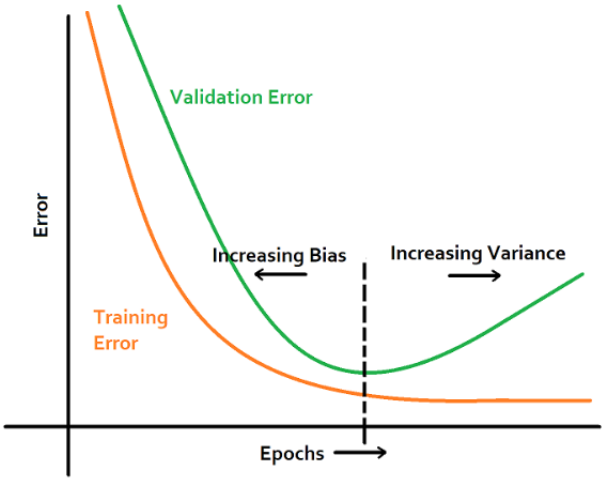

Figure 11: A simple plot of the training and validation loss during training.

4.7 Learning Rate Schedulers

In order to speed up training, but also in order to improve the accuracy of the optimizer hitting the/a minima in the loss landscape learning rate schedulars are used. They change the learning rate of the optimizer during the training process. Initially usually the learning rate is set high, in order to allow the initially completely unaware network to start learning quickly. This has the disadvantage that the network most likely skips over (many) minima (as previously in section 4.5). After a few epochs the learning rate scheduler usually decreased the learning rate gradually, as to allow the optimizer to reach the minima without overshooting it or oscillating around it.

The use of a learning rate scheduler generally results in quicker training with better results. Its use is most advantageous when using an optimizer that does not adjust its learning rate itself (unlike Adam).

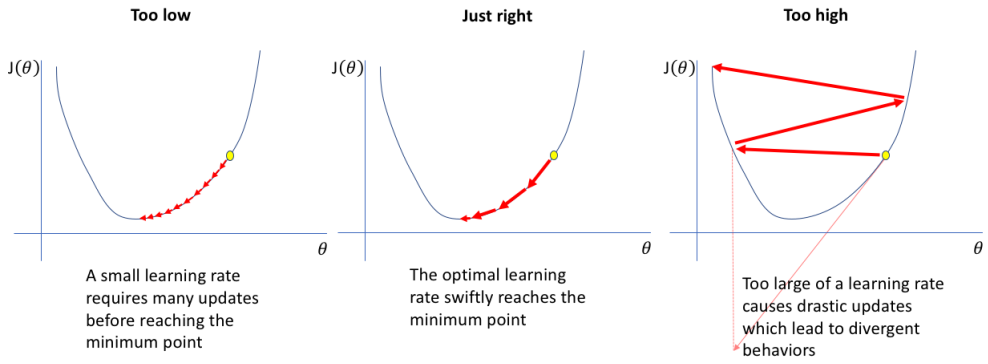

Figure 12: An illustration of good and bad learning rates.

4.8 Weighted Loss

Usually the loss for all data records is handled equally. But in some cases it makes sense to apply a weight to all losses individually. A common case for weighing losses in classification (binary network output) is when the given training dataset is imbalanced. A imbalanced dataset contains records of which a majority classifies as one of the two possible binary values. This would/will lead the network to prefer this value over the other, this behaviour is known as bias. Bias is generally undesirable since it discriminates agains the minority and therefore yields poor performance in identifying records of the minority class.

To solve this problem one can cut a percentage of the majority from the training dataset in order to produce a balanced dataset in which both classes appear equally often. This can be problematic when

majority outnumbers the minority by a very large factor, since then many individual records are lost when balancing out the dataset. With way fewer cases to learn from the network may struggle to fully understand all cases. When facing such an issue one can apply weighted loss to the training data. Instead of cutting a percentage from the majority class, a lower weight is applied to all records of the majority class. The weights are picked in such a way that the sum of all losses of the majority class equals the sum of all losses of the minority class. This way the network can learn from all available records without getting biased towards the majority.

4.9 Graph Neural Networks

Since the DOMs of IceCube are located in a irregular hexagonal grid patter, the use of MLPs is unsuitable. MLPs view all edges as equals, therefore assuming a equal distribution of nodes and equal distance between them, like a cubic grid pattern. Because this not not the case for IceCube, a more flexible model is required.

Graphs are a set of nodes and edges, where each node can be connected to all other nodes (even themselves) via edges. Edges can be either one- or two-directional, allowing two nodes to influence another, in contrast to the feed-forward rule of MLPs. Since not all nodes have to be connected, this allows for much more freedom when building a graph. The decision of which nodes are connected via edges is an important one, it heavily influences how the network behaves and models relationships like social connections or molecular connections. In contrast to MLPs, each node can hold multiple values (its features), instead of solely containing a singular value.

In the case of IceCube, each DOM taking part of a event is modelled as a node. These nodes are then connected via edges, decided on by the k-nearest-neighbours algorithm (kNN). It connects each node with the (in this case) 8 nearest neighbours in space.

This generated graph is then interpreted by a Graph Neural Network, which takes the graph as its input. In contrast to the "one shape fits all" aproach of MLPs (or similar), this allows GNNs to adjust much better to different shapes of input data. GNNs have proven great performance in various field, such as social modeling at e.g. Facebook, interpretation and classificatons of point clouds objects, as well as protein folding predictions.

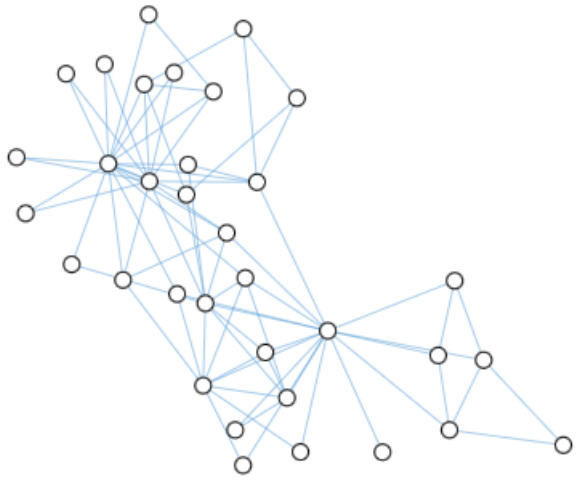

Figure 13: A graphical representation of a graph.

Lets now look into how GNNs operate on the input graph in more detail.

A GNN operates on a given graph by performing several (arbitrary amount) iterations of Message Passing. During Message Passing each node sends "messages" to its neighbours iteratively updating their data based on the messages received. Messages are typically based on the data the sending node holds, but can also contain other information like number of edges. Therefore messages are usually of the shape of the data contained in the node, but like layers can change width in MLPs, the shape of messages is not fixed. Each node combines the messages it receives, this aggregation logic could be taking

the maximum, minimum, average, median, or any other sort of combining, as long as it is permutation invariant. Permutation invariance is required, since a graph does not hold information about ordering of edges and therefore received messages. Lastly, a update function generates the new data for each node, based on its old data and the aggregated messages. The shape of data can change between iterations, based on the choice of message passing, aggregation and update functions.

As the data contained in the graph has the shape `Number of Nodes` × `Number of Features`, in order to produce a prediction vector (or singular value) this shape has to be reduced. To achieve this the data of all nodes is aggregated by using similar function as for the aggregation of messages. Such functions can also be used in combination, as `dynedge`, the GNN used in this work, implements its node aggregation layer. Finally the shape `1` × `Number of Features` is reduced again to the shape `1` × `Number of Output Variables`, usually by the use of a MLP.

EdgeConv

The GNN used in this work makes use of the `DynEdge` convolutional layer. `EdgeConv` was designed in order to make neural networks able to learn on graph point clouds, classifying or identifying objects based on their point cloud. `EdgeConv` incorporates local neighbourhood information into its message passing, allowing for the network to learn about the geometrical structure by analysing each point in relationship with its neighbourhood.

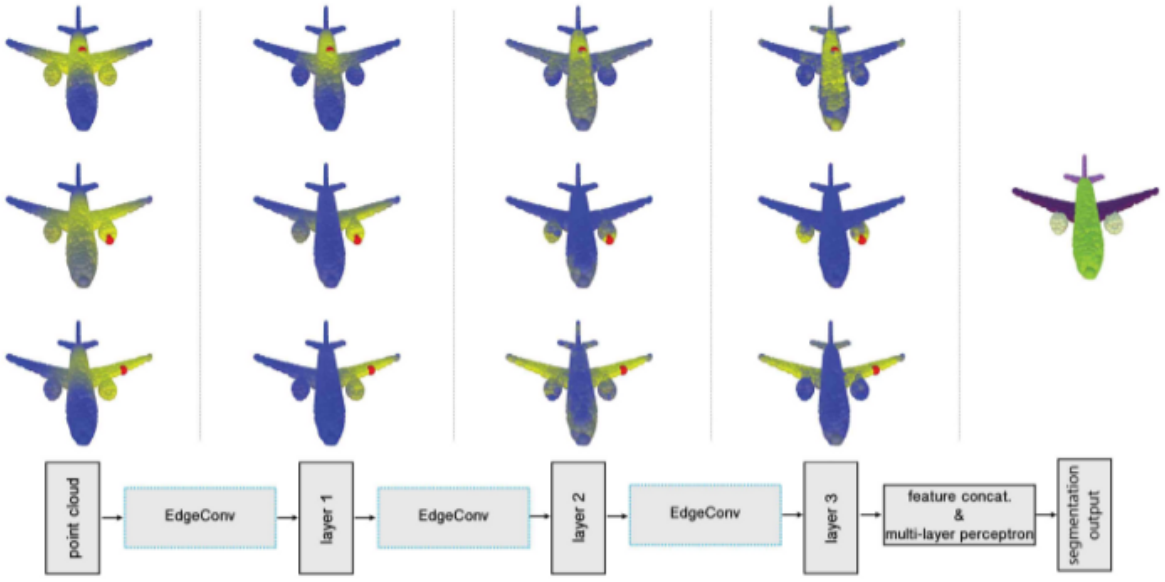

Figure 14: An example application of EdgeConv.

5 Icedata

This section describes the IceCube ice model data and the process of preparing (reading, converting, interpolating) it for supervised learning. In figure 4 the absorption and scattering is visualized, giving an overview of how the ice changes vertically.

The project SPICE measured the ice properties (icedata) and organized it into layers (ice of equal properties) indexed by depth. Measurements were taken along 6 strings every 10 m vertically. The recorded values are not constant horizontally, therefore the layers are (slightly) tilted. This tilt also varies across depth, which will complicate the interpolation process (section 5.3).

Interpolation of the data is required, because the position of each DOM must be associated with the ice properties of the layer it resides in. Also, the 10 m granularity of the data makes a interpolation between them sensible.

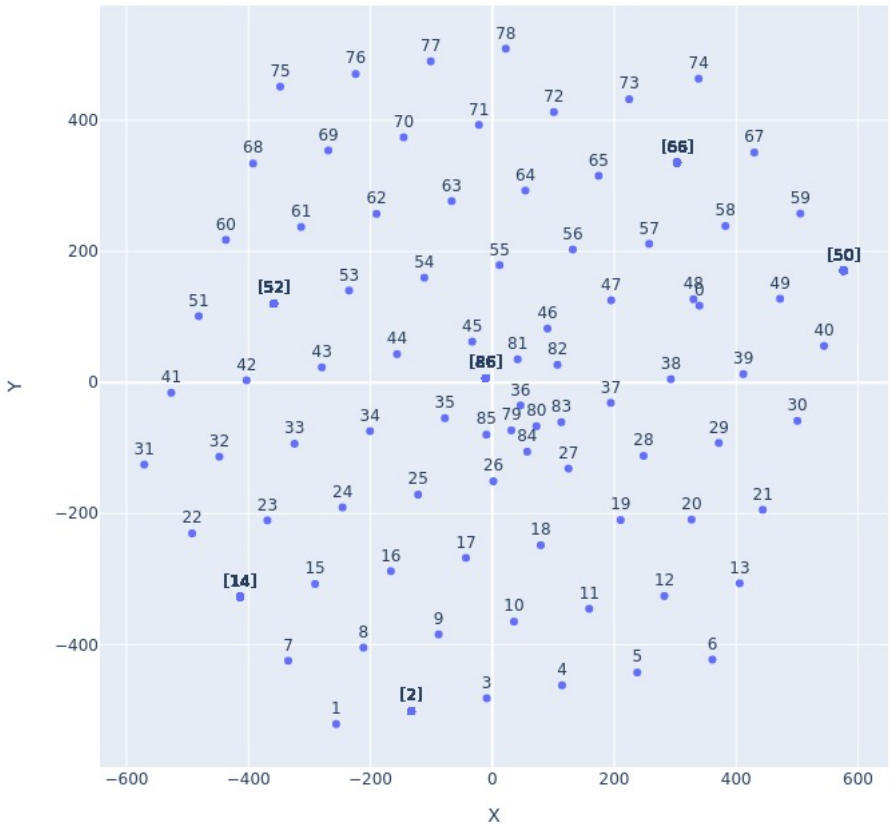

Figure 15: An overview of where SPICE measurements have been taken. Strings where measurements have been made are marked with square brackets.

5.1 Coordinate Systems

Within IceCube there exist multiple types of coordinate system which are commonly used. This section will outline the differences between the ones relevant to this work and provide conversions used when working with the icedata.

IceCube

Most of IceCube uses the IceCube coordinate system which has its origin 1948,07 m below the surface, its z-coordinate will be called $z_{IceCube}$ and grows positive upwards.

Spice

The Spice dataset is organized and labeled by depth, which is 0 at the surface of the ice and grows positive downwards. We will call this z-coordinate z_{Depth} or simply Depth.

Global

The z-coordinate we will be working with (and converting the other coordinates into) is z_{Global}, a coordinate system that has its origin at the surface and grows upwards. This definition results in the following conversions:

$$z_{Global} = -z_{Depth} \tag{11}$$

$$z_{Global} = z_{IceCube} - 1948{,}07\,\mathrm{m} \tag{12}$$

5.2 Data files from SPICE

The data from SPICE comes in space/tab-seperated-values format and are read by using the Python `pandas` library. For example, the file `tilt.par` is read like so:

```
import pandas as pd

df_tilt_par = pd.read_csv('tilt.par', sep=' ', names=['string_no', 'dist_sw_225'])
```

icemodel.dat

This file contains the ice properties listed below. As loosely described above, the data is indexed by depth, which identifies the layer the row of data is related to.

The columns of data contained in this file represents the following (physical) properties:

- `depth`: depth below surface (used as layer identifier)
- `be(400)`:
- `adust(400)`: absorption at wavelength of 400nm
- `k1`:
- `k2`:
- `BFRA`: birefringence (axis 1)
- `BFRB`: birefringence (axis 2)

TODO: Describe properties more thurrowly

Depth	be(400)	adust(400)	k1	k2	BFRA	BFRB

tilt.par

This file describes where the ice layer tilt offsets are measures at. This means that the file contains a horizontal (xy) description of where the offsets are located. `SW_255` is the horizontal distance in the SouthWest (225deg) direction with its origin located at string 86.

Due to the natural relationship between string number and its location the data is redundant and only `SW_255` will be used from now on.

- `string_no`: string number
- `sw_225`: distance [m] into the south-west direction (225 °) direction

String	SW_225
50	-531.419
66	-454.882
86	0
52	165.202
2	445.477
14	520.77

tilt.dat

This file describes how much each layer of ice is tilted at the locations specified in `tilt.par` relative to the flat assumption. It holds the 6 offset values at all depths. Together with the locations from above, it will be used to build a model of tilted layers.

Depth	Offset 50	Offset 66	Offset 86	Offset 52	Offset 2	Offset 14

5.3 Interpolation

As discussed in section 5, the layer tilt is not constant across depth, therefore the process of assigning a layer to an arbitrary point is non-trivial. This is especially evident for the bottom-most layers, whose proximity to the bedrock gives them a curved shape, as seen in figure 16. Not only do the layers have to be interpolated over the horizontal axis, but also the icedata has to be interpolated vertically between the layers. This process and all decisions/assumptions made are described in this section.

Since the layers are described at the locations defined in `tilt.par`, we use its z-southwest coordinate system. More specifically we use z_{Global} as the y-axis and distance in the south-west direction as the x-axis with string 86 as the origin. Onward we will use the function `sw_225` to calculate the sw-coordinate of any given point.

```
import numpy as np

SW_225 = np.array([-1, -1]) / np.sqrt(2)  # normalized direction vector
ORIGIN = XY_STRING[86]                    # string 86 is zero of ‘tilt.dat‘

def sw_225(xy):
    return np.dot(xy - ORIGIN, SW_225)
```

5.3.1 Layers

First a function/algorithm is required to decide on what layer a given point is located in, or more accurately, what layers a given point is located between. To build such a function, we interpolate the measured layer tilt offsets across the sw-coordinate. By adding/subtracting the layer depth to the interpolated offset we get the actual layer heights at any point. Doing this for all positions, we get the resulting layers shown in figure 16.

Extrapolation

As is visible in figure 15 the strings 7 and 1 extend further into the south-west direction than string 14, the last string a measurement has been made for. This is also the case with strings 74, 67 and 59, which extend past the last measured string in the negative southwest direction, string 50. This makes the choice to extrapolate the layers in both directions necessary.

Quadratic Interpolation

As the layer offset is measured at 6 points in the southwest direction, we have to interpolate these values. A quadratic interpolation was chosen instead of a linear interpolation. This decision is clearly biased, but hopefully the better choice. The resoning being, that the layers of ice are most likely not spikey/angled at the points of measurement, but rather smoothly changing their angle. Therefore a quadratic interpolation will provide for more even and flowy layers, instead of spikey, undifferentiable layers.

TODO: Show layer interpolation 2d plot with linear vs quadratic interpolation

Continued Layers

Strangely the last layer contained in `tilt.dat` is at a depth of 2448,47 m, which is a few dozen meters above the lowest DOMs. Though the data contained in `icemodel.dat` extends all the way to a depth of 2798,47 m. To incorporate this existing data, the decision was made to simply copy the shape of the lowest layer 7 times. This allows for a interpolation that respects the given icedata while hopefully being as close to the actual shape of layers as possible. These additional layers are visible in figure 16.

Code

The interpolation was done using the 1D interpolation function provided by the python library `scipy`. The data used is stored in `df_tilt_par` and `df_tilt_dat`, which comes from the corresponding files described in 5.2.

Since `interpolation_dzs` only interpolates the offsets, we create another funtion `interpolation_zs`, which adds the z-coordinate of every layer to the offsets and converts between the Spice coordinates and the global coordinates (see 5.1). It will return an array of the heights of all layers at the location it is given `sw`.

```
from scipy.interpolate import interp1d

interpolation_dzs = interp1d(
    df_tilt_par.dist_sw_225.to_numpy(),  # positions where offsets were measured (6,)
    df_tilt_dat.iloc[:, 1:].to_numpy(),  # offsets = dzs (125, 6)
    kind='quadratic',
    axis=1,
    bounds_error=False,
    fill_value='extrapolate',
)

def interpolation_zs(sw): return -(df_tilt_dat.z - interpolation_dzs(sw))
```

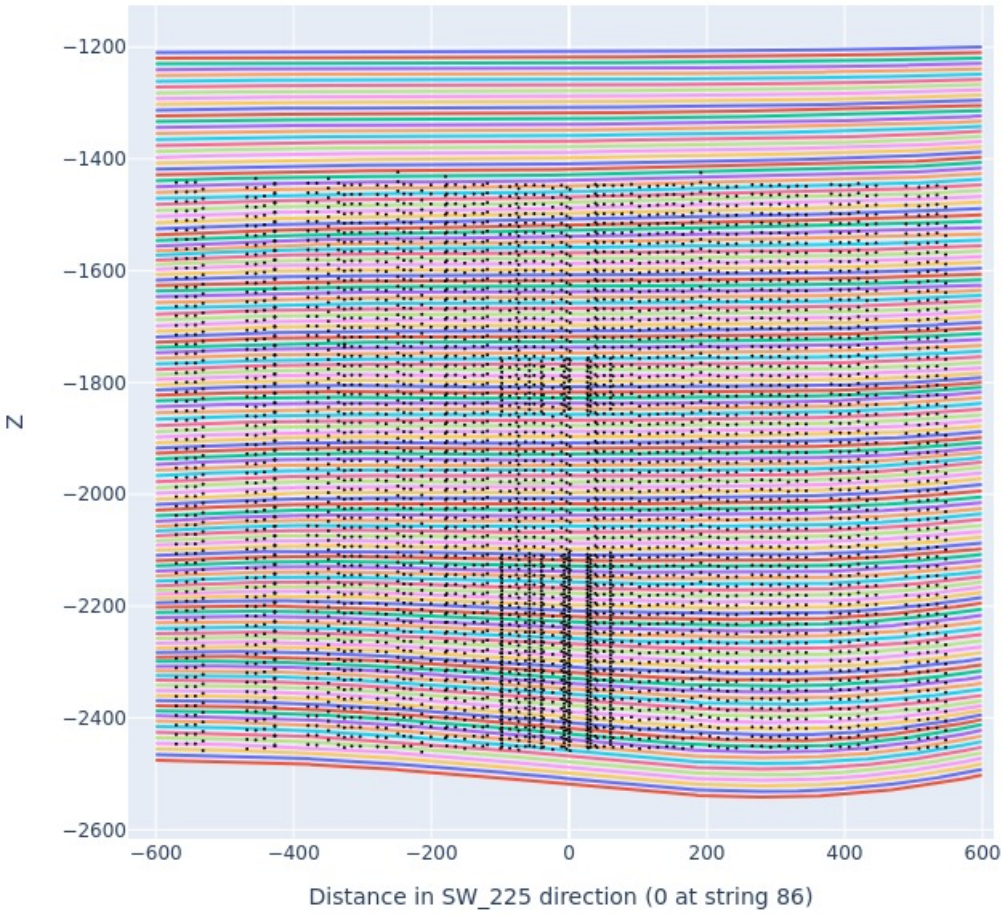

Figure 16: The interpolated ice layers (color) and all DOMs (black).

5.3.2 Ice Properties

To now compute the icedata at any point we first use `interpolation_zs` designed above to get the heights of all layers at the location point of interest. Then we do another interpolation, this time interpolating the icedata over the just calculated layers heights. This gives us a function that will interpolate the icedata between the layer above and below the point of interest at a given height.

Quadratic Interpolation

A quadratic interpolation was chosen for similar reasons as before for the layers. It is intended to smooth over outliers in the data and make it less spikey. Having a differentiable function to describe the properties is probably also a more correct approach.

Extrapolation

If we had not previously continued the layers in `tilt.dat`, we would have to use extrapolation in order to calculate the icedata of the lowest DOMs. But because we have extended the layers in order to contain all DOMs between them, extrapolation is not required here.

Code

The functionality discussed above relies on the layer heights at the point of interest. Therefore we run the function `interpolation_zs` created before in order to get the layer heights. We use use this data together with the icedata from `icemodel.dat` to build the second interpolation, again using the 1D interpolation function from `scipy`.

The function `interpolation_features` will return a 2D array of icedata specific to the point of interest identified by its sw- and z-coordinate.

```
layer_zs_global = interpolation_zs(sw)

interpolation_features = interp1d(
    layer_zs_global,  # layer heights from before (125,)
    layer_features,   # icedata from icemodel.dat aligned with layers from tilt.dat (125, 6)
    kind='quadratic',
    axis=0,
    bounds_error=False,
    fill_value='extrapolate',
)
```

5.4 Lookup Table

A lookup table containing pre-interpolated ice properties for all DOMs is created in order to minimize computational overhead during training. So instead of interpolating the features on every occurence during training, we precalculate the values for all DOMs beforehand. In order to achieve this, we loop over all DOMs and their positions, calculate their sw-coordinate and then calculate their icedata. The lookup table itself is implemented as a simple key-value store (Python dictionary), which uses the DOM locations (xyz) as its key and returns a 6 element array containing the icedata for that DOM.

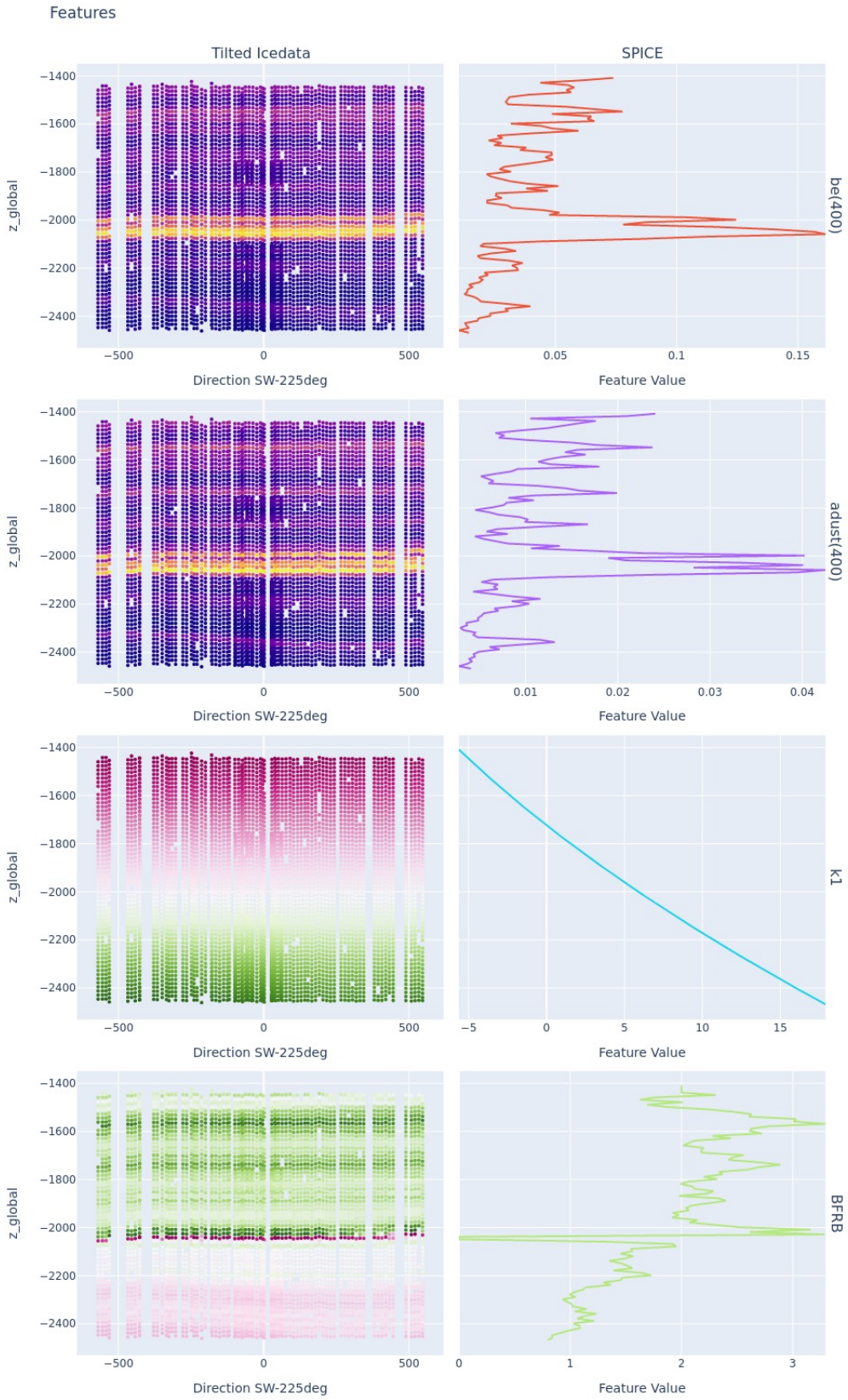

Figure 17: A visualisation of icedata that has been transformed/interpolated to all DOM locations.

6 The GNN & Training Procedure

6.1 GraphNeT

`GraphNeT` is a machine learning library that makes applying a Graph Neural Network to data from PMT-based neutrino telescopes (such as IceCube or KM3NeT) easy. It is build on top of PyTorch, one of the most popular open source machine learning frameworks. Its modular approach allows for easy use and customization, GraphNeT uses `PyTorch` as a basis and implements modules and adapters which interface with the IceCube data. Because PyTorch itself does not implement the use of graphs, an additional library `PyTorch Geometric` is used for the generation and learning on graphs.

As previously touch on in section 4.9, graphs are generated for each event with DOMs as nodes and edges created by the k-nearnest-neighbours algorithm.

6.2 dynedge

The most widely used GNN implemented in GraphNeT is `dynedge` designed by Rasmus Ørsøe. It consists of five EdgeConv layers, the first layer operates on the graph generated by the k-nearest-neighbours algorithm with all nodes containing the unmodified DOM data from section 3.6. After the first layer the number of features is increased, by being scaled up by a MLP. The results of all five EdgeConv layers are then concatenated into an even wider shape. Afterwards another MLP is applied, the data is aggregated and finally a last final MLP brings the data into the output shape.

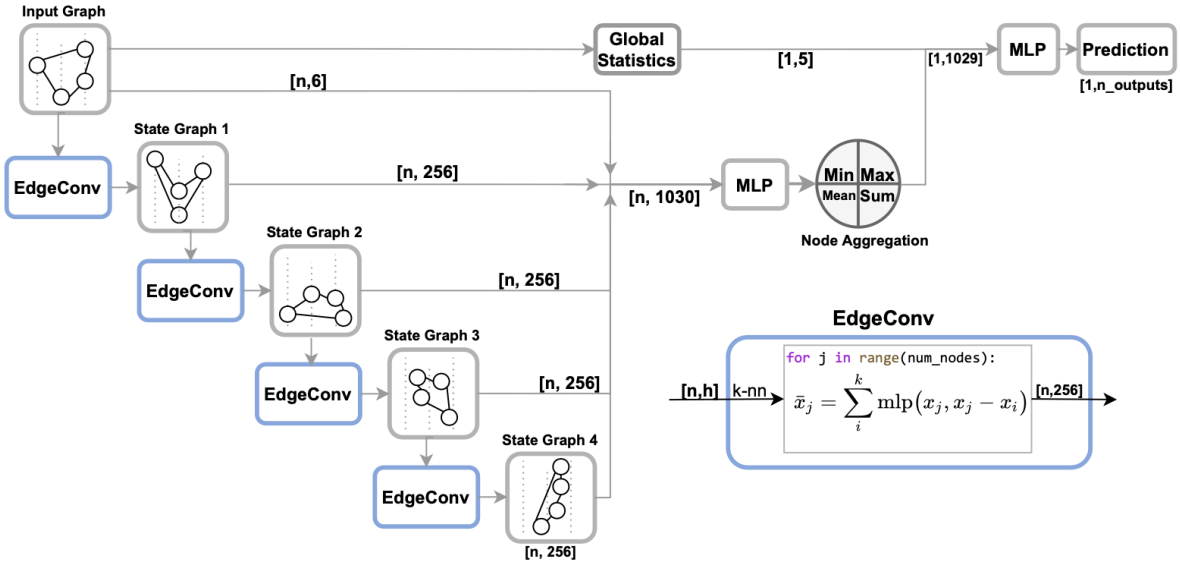

Figure 18: A diagram of the model dynedge.

6.3 Inserting Icedata

There exist several sensible approaches of inserting the icedata into training. One could do calculations on each graph edge and calculate probabilities of absorption based on the absorption rate and distance and let this value influence message passing. One could calculate optical distance based on the refractive index between any two DOMs and modify the k-nearest-neighbours algorithm to use it instead of real distance. One could also weigh graph edges based on a combination of the values above. For simplicity and in order to test whether the icedata gives any benefit to the network at all, it was decided to attach the local icedata to the features of each node. That means that the feature variables from section 3.6 have been extended to also contain all ice properties from section 5. As touched on before, a lookup table was generated before training and the code loading the data for each DOM was modified to look up icedata based on the DOMs position and attach that data to the node's features.

In order to reach a generally well performing network, Early Stopping was utilized during the training process. As the optimizer the choice was made to use Adam, since it allows for quicker learning on average. Especially in this case, because the addition of icedata will drastically and unpredictably complicate the loss landscape, Adam is a great choice, since it automatically adapts its learning rate in order to navigate the local loss landscape more appropriately. In addition to Adam a piecewise linear Learning Rate Scheduler was used, which scales the base learning rate of Adam 10^{-3} by a factor of 10^{-2} in the beginning, to 1 after the first half epoch and then back down again to 10^{-2} after 50 epochs. This allows Adam to get to know the loss landscape a bit better after starting out and improves its accuracy in hitting a minima at the end.

6.4 Applying Weighted Loss

Since the majority of events happen in the energy interval from 0 to 1,5 log10GeV it is sensible to focus our attention to these events, because the networks performance is largely influenced by this set of data. In order to improve the networks performance for these kinds of events, a weighted loss was created which flattens the density per energy within this interval, while providing an inverse decrease outside the upper limit of the interval. Usage of this weighted loss will cause the network to treat these energies as equally important and not specialise to the energy with the highest density (see section 4.8).

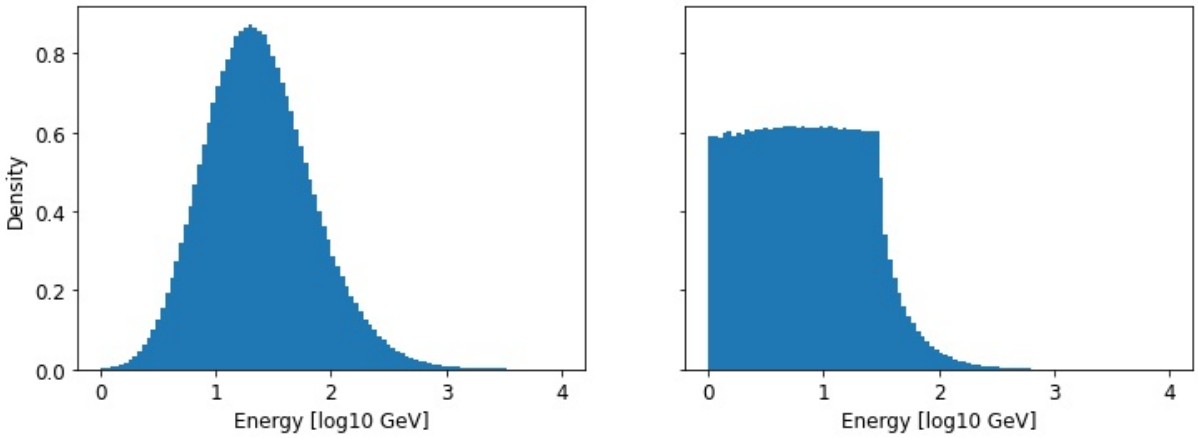

Figure 19: A histogram comparing the standard (left) vs. weighted (right) event energy distribution.

7 Results & Conclusion

7.1 Performance with Icedata

7.2 Performance with Icedata and Weighted Loss

7.3 Repeatability

First, one important insight, that was made and has to be held in mind when investigating and testing with several variants of neural networks, was about reproducibility/repeatability. As unually any neural network is initialized with random parameters before training, it is not guaranteed that the optimizer will find the same loss minimum each time. This makes comparisons between different training runs more difficult. It was indeed observed, that the performance of training runs containing icedata and a weighted loss, and therefore having a way more complicated loss landscape, varies much more than without the icedata (see figure 21). Because of this issue, the best run of each category was selected for the final comparisons in the next sections.

7.4 Neutrino Oscillation Contours

TODO:

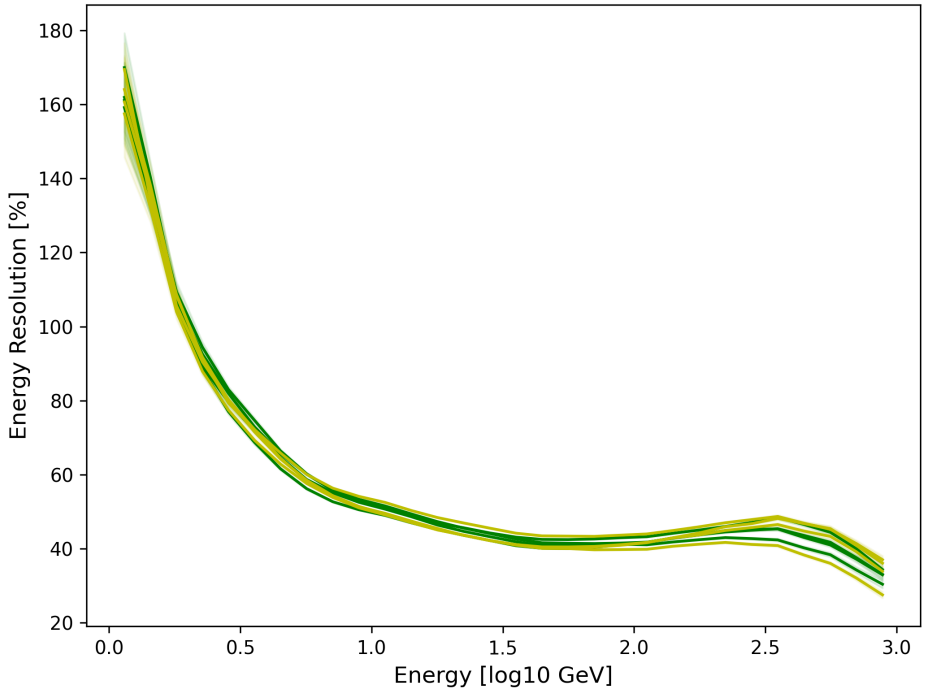

Figure 20: Energy resolution of different training runs. Runs with standard features are colored green, while runs with attached icedata are colored yellow.

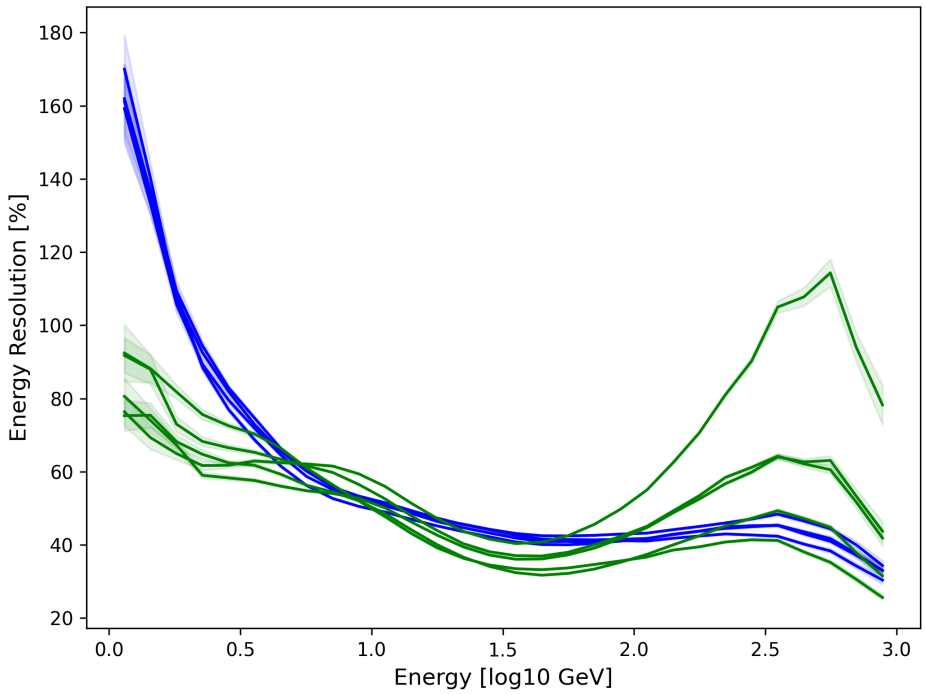

Figure 21: Energy resolutions of different training runs. Runs with standard features are colored green, while runs with attached icedata and weighted loss are colored blue. All runs were trained with the same hyperparameters.