Transforming IceCube

Energy and Angular Regression of IceCube Data using a NN, GNN, and Transformer Architecture

Laurits Møberg, Franciszek Nawrocki
& David Johansen
11.06.2025

(All parts contributed equally)

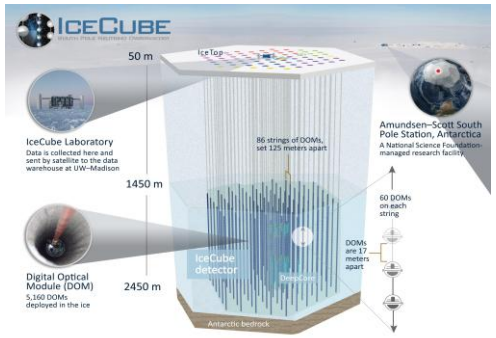
UNIVERSITY OF COPENHAGEN

Outline

- Data
- Tools
- Neural Network
- GNN
- Transformer
  - Energy
  - Direction
- Conclusion

Data

Source:



https://icecube.wisc.edu/science/icecube/

Simulated Data – No outliers/NaNs
Energy range: 10^4 - 10^6 GeV

PMT-fication

124/dom Multiple pulses ->
32/dom Variables

Pulses -> List

Events: ~200k per file

1	[Receipt]22011_1.json	truth_1.parquet
2	[Receipt]22011_2.json	truth_2.parquet
3	[Receipt]22011_3.json	truth_3.parquet
4	[Receipt]22011_4.json	truth_4.parquet
5	[Receipt]22011_5.json	truth_5.parquet
6	[Receipt]22011_6.json	truth_6.parquet
7	[Receipt]22011_7.json	truth_7.parquet
8	[Receipt]22011_8.json	truth_8.parquet
9	[Receipt]22011_9.json	truth_9.parquet
10	[Receipt]22011_10.json	truth_10.parquet

Data

- Dataloader

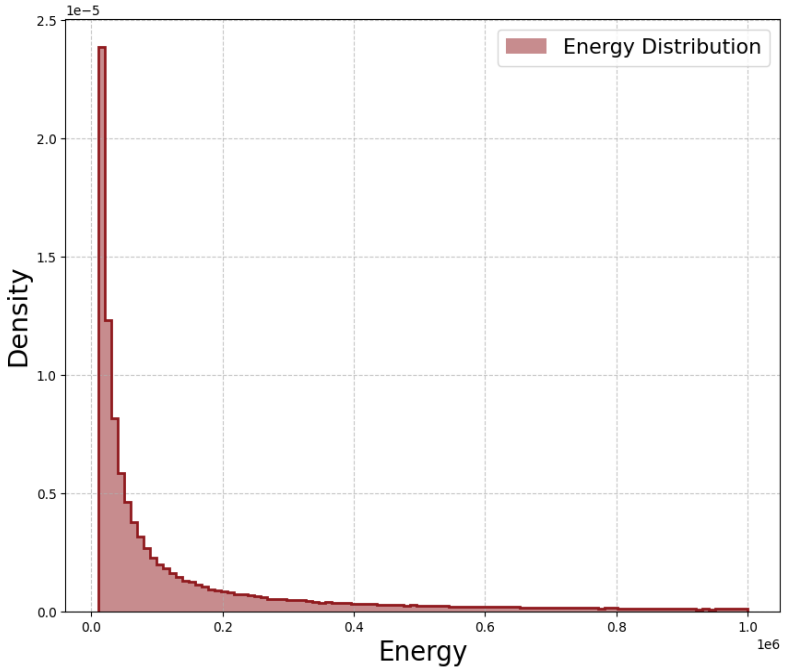
```
class PMTfiedDatasetPyArrow(Dataset):
    def __init__(
            self,
            truth_paths,
            selection=None,
            transform=feature_preprocessing,
    ):
```

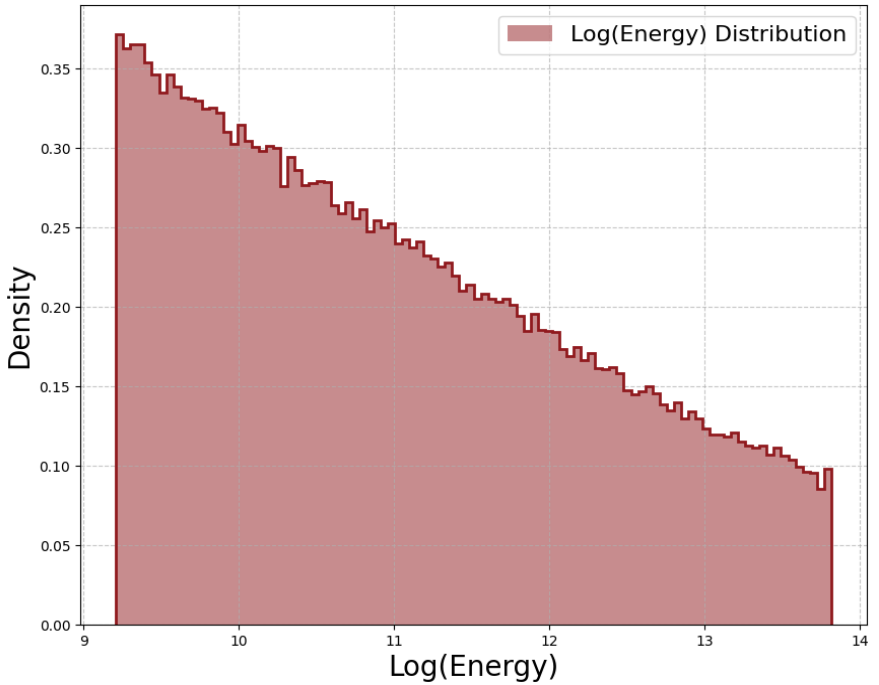
- Feature preprocessing

```
def make_dataloader_PMT(
        root_dir,
        dataset_id,
        training_parts,
        validation_parts,
        batch_size=config['training_params']['batch_size'],
        num_workers=config['input_data']['num_workers'],
):
```

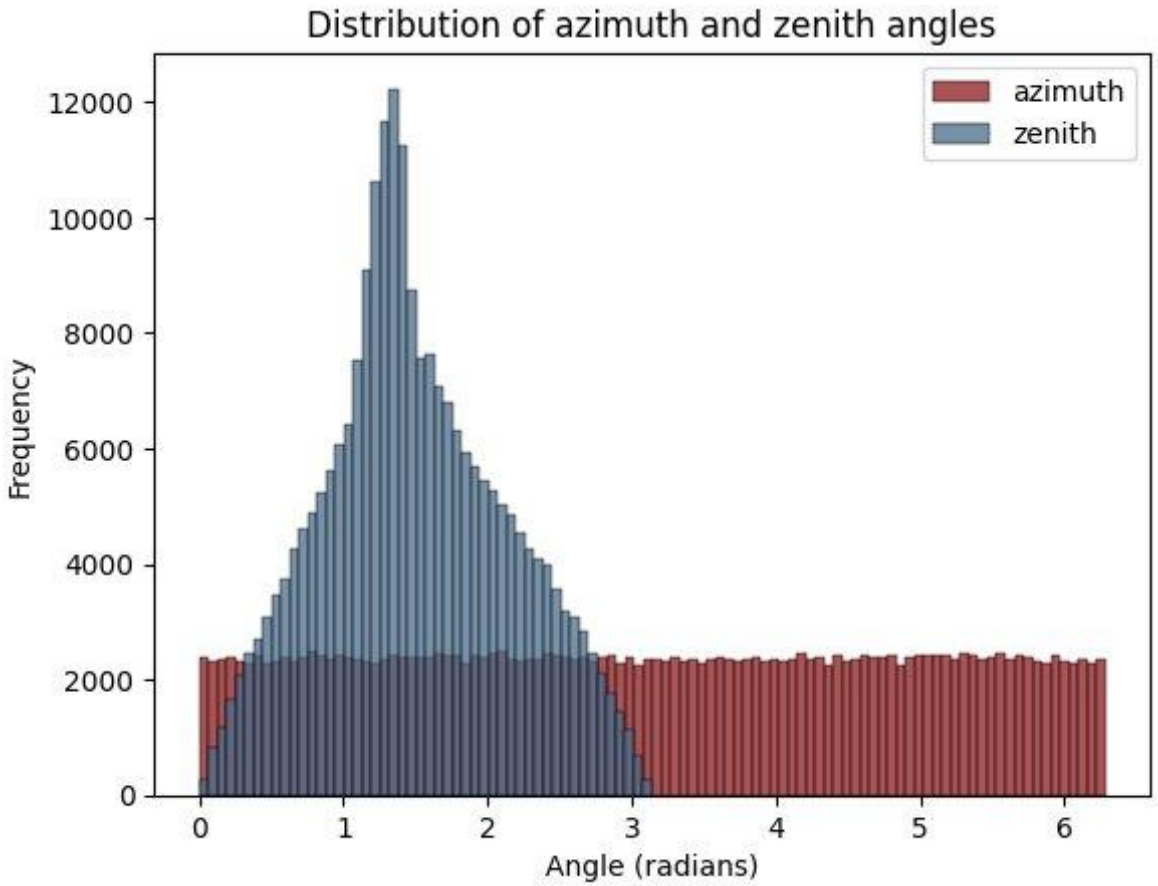
```
if col_name in ['dom_x', 'dom_y', 'dom_z', 'dom_x_rel', 'dom_y_rel', 'dom_z_rel']:
    value = value / 500
elif col_name in ['rde']:
    value = (value - 1.25) / 0.25
elif col_name in ['pmt_area']:
    value = value / 0.05
elif col_name in ['q1', 'q2', 'q3', 'q4', 'q5', 'Q25', 'Q75', 'Qtotal']:
    mask = value > 0
    value[mask] = np.log10(value[mask])
elif col_name in ['t1', 't2', 't3','t4', 't5']:
    mask = value > 0
    value[mask] = (value[mask] - 1.0e04) / 3.0e04
elif col_name in ['T10', 'T50', 'sigmaT']:
    mask = value > 0
    value[mask] = value[mask] / 1.0e04
```

Data





Data



Tools

Need GPU for efficient training

Solution:

Workflow:

Code locally -> Push to Github

-> Clone and run on Colab

```
[ ]  !git clone https://github.com/Moebergs/AppML_FinProject
     %cd AppML_FinProject/training_and_inference/
```

```
[ ]  !pip install -r requirements.txt -q
```

```
[ ]  from google.colab import drive
     drive.mount('/content/drive')

     import zipfile
     zip_ref = zipfile.ZipFile("/content/drive/MyDrive/22011_1T4.zip", 'r')
     zip_ref.extractall("/content/dataset")
     zip_ref.close()
```

```
[ ]  !python -m train
```

Tools

Weights & Biases

Even with Colab, training the models takes a lot of **time.**

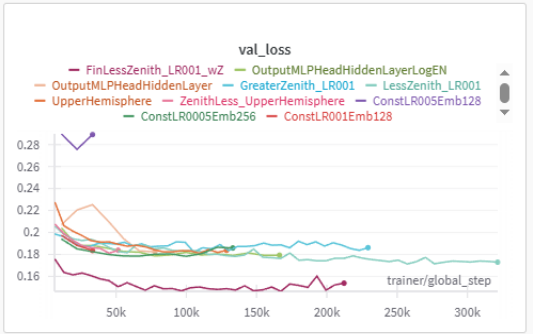
Total time for testing and training:

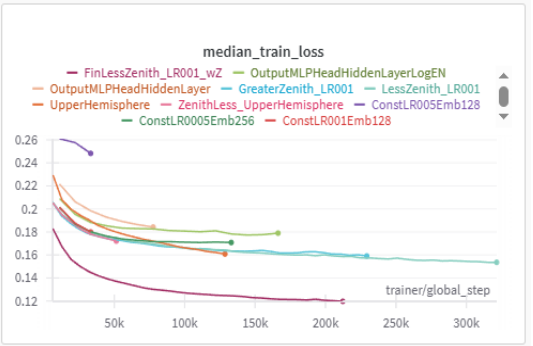
~ 250 Hours

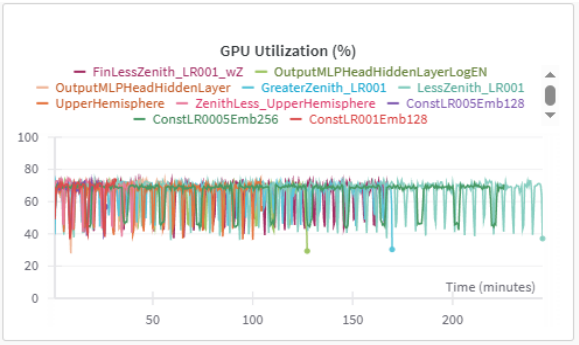
Need way to monitor training

Solution:

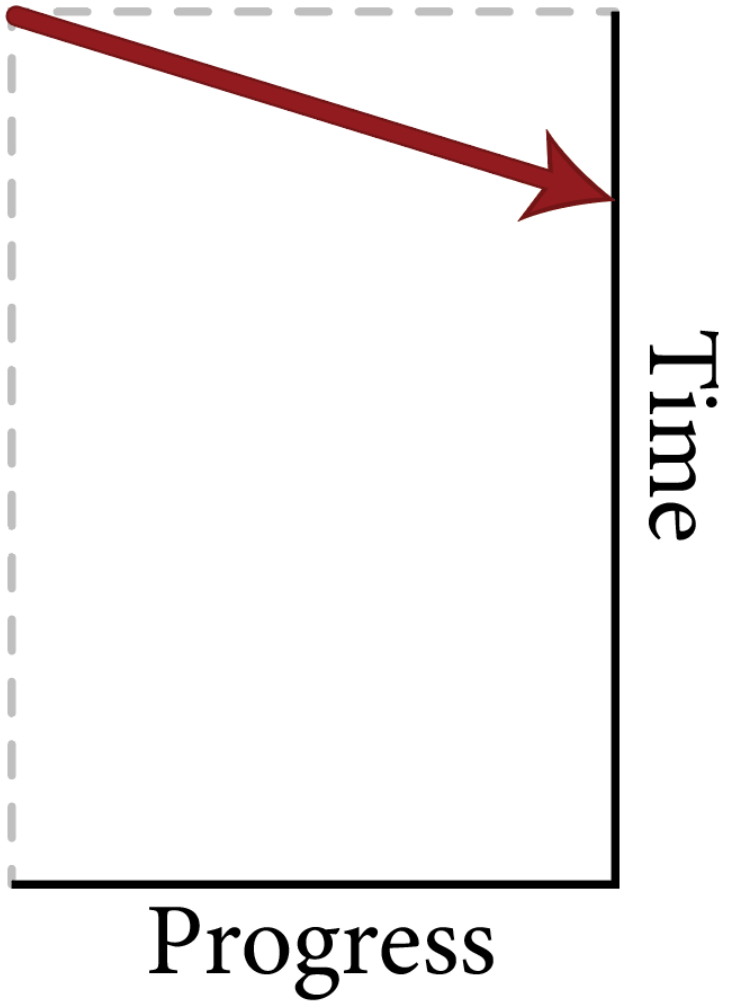
Weights & Biases (wandb.ai)

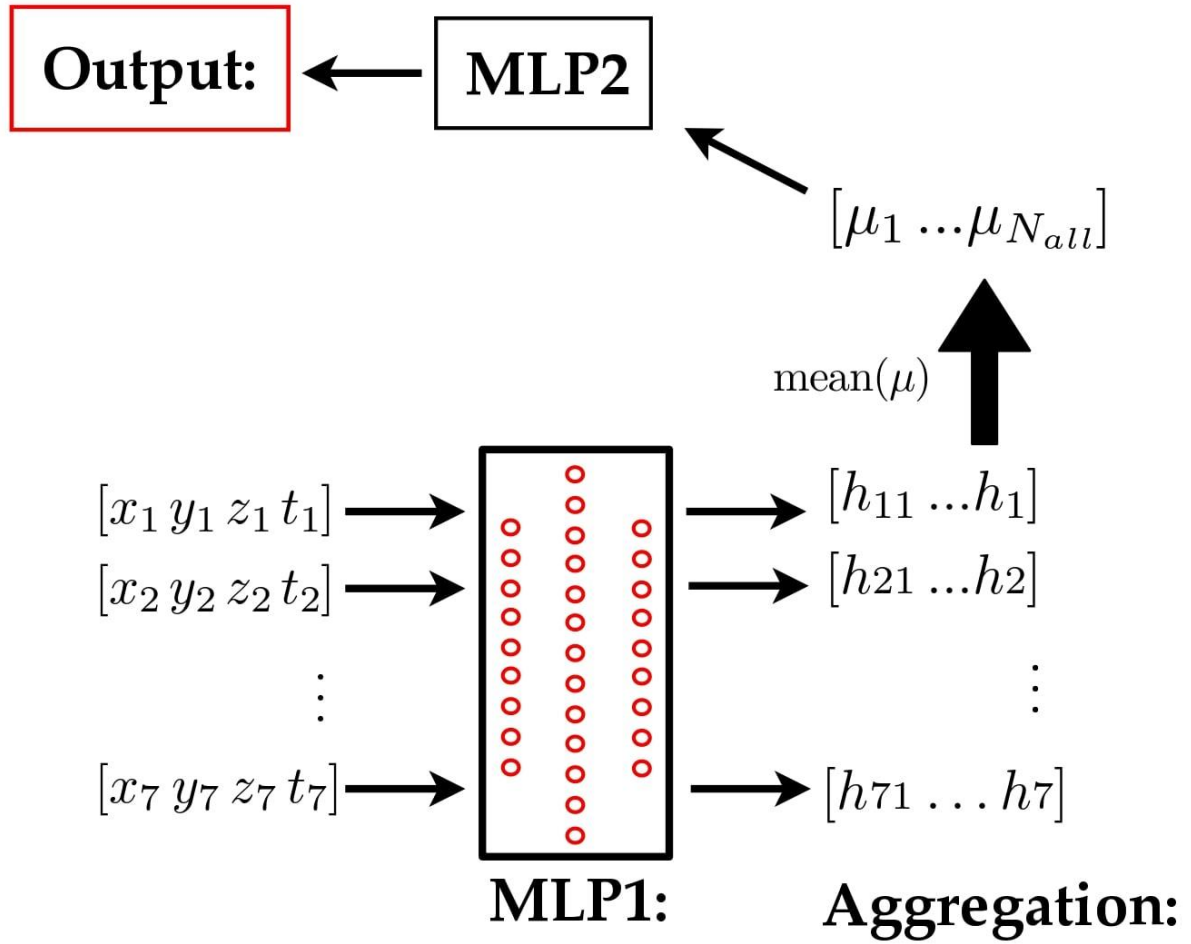




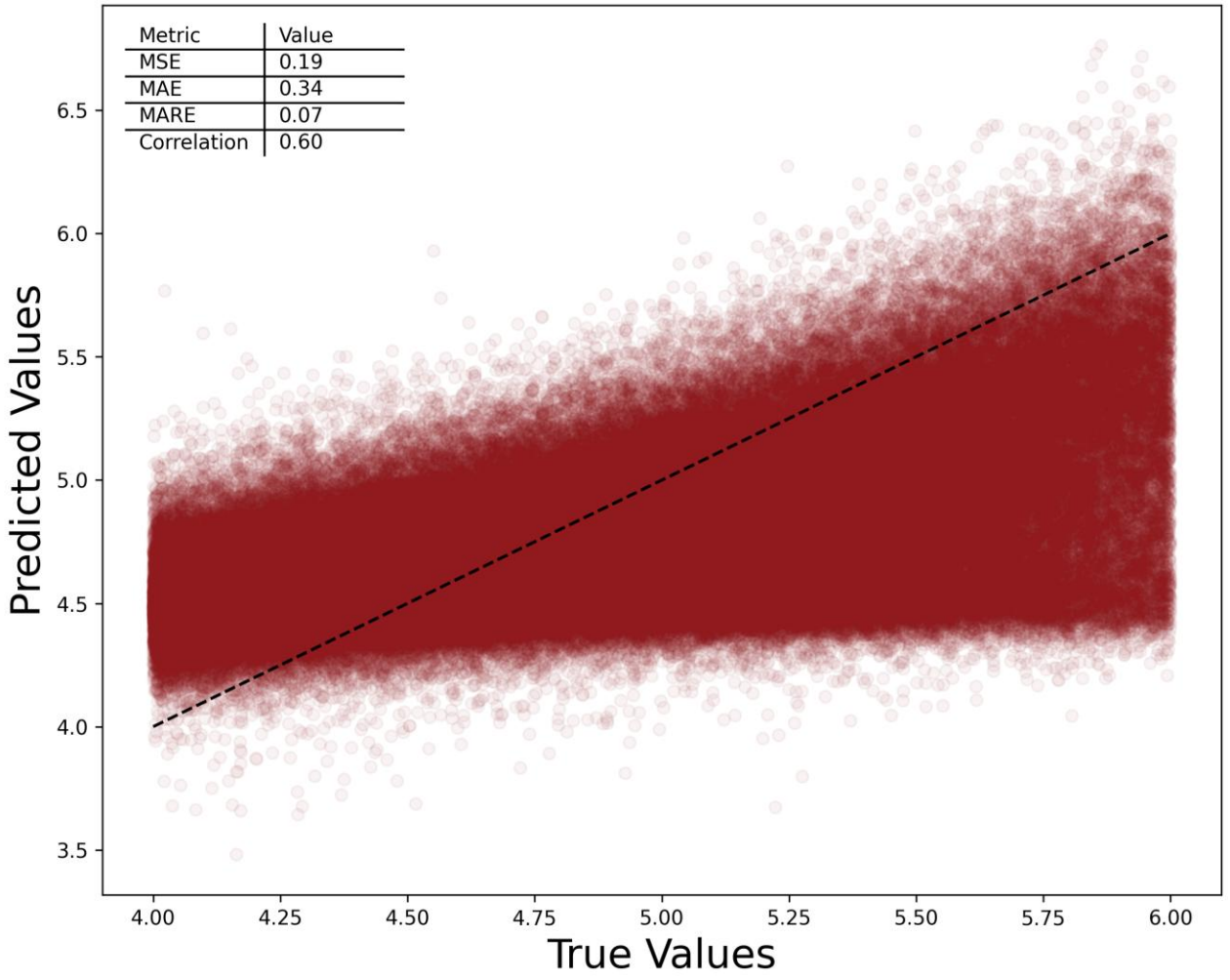


Neural Network

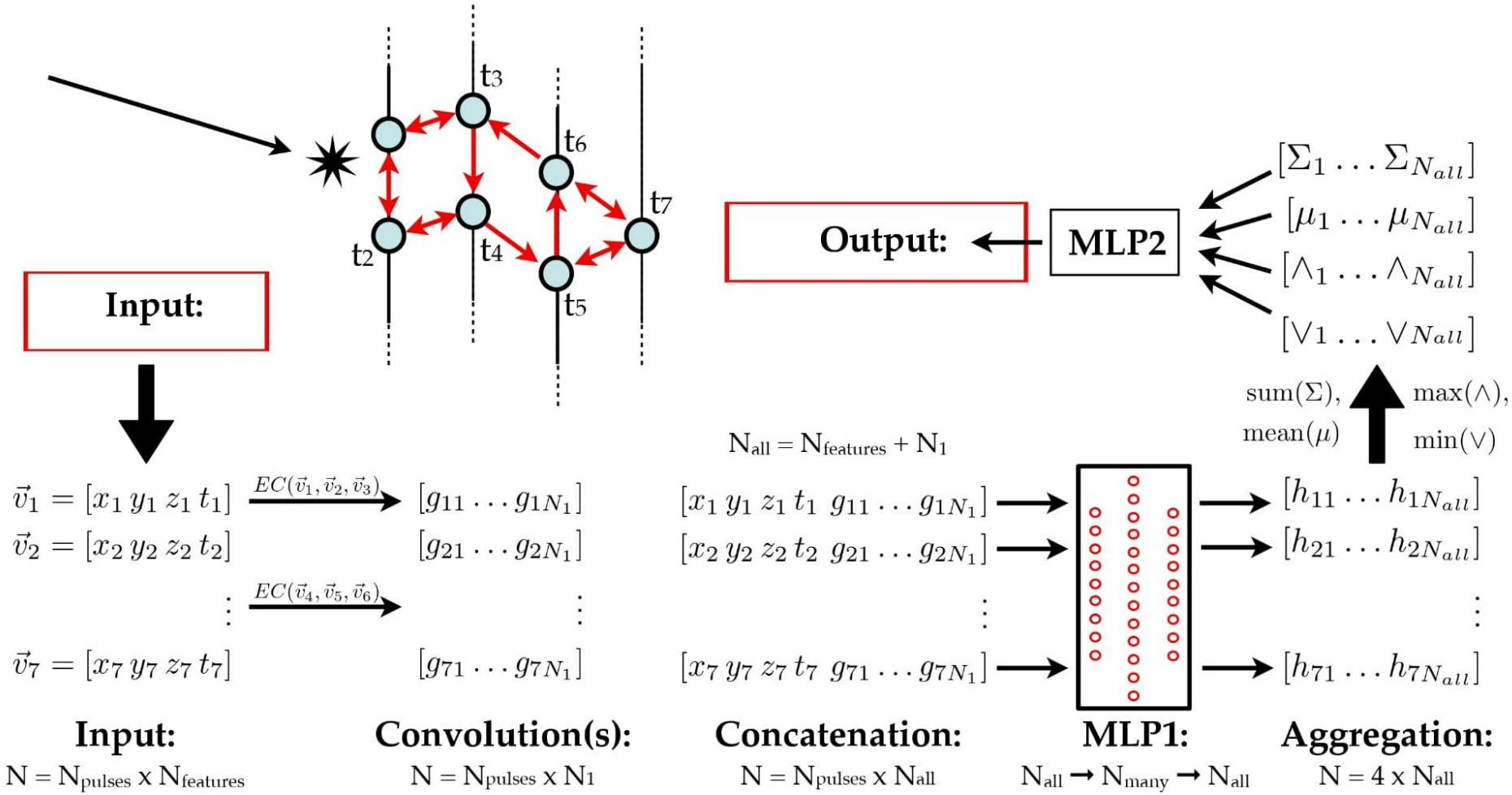




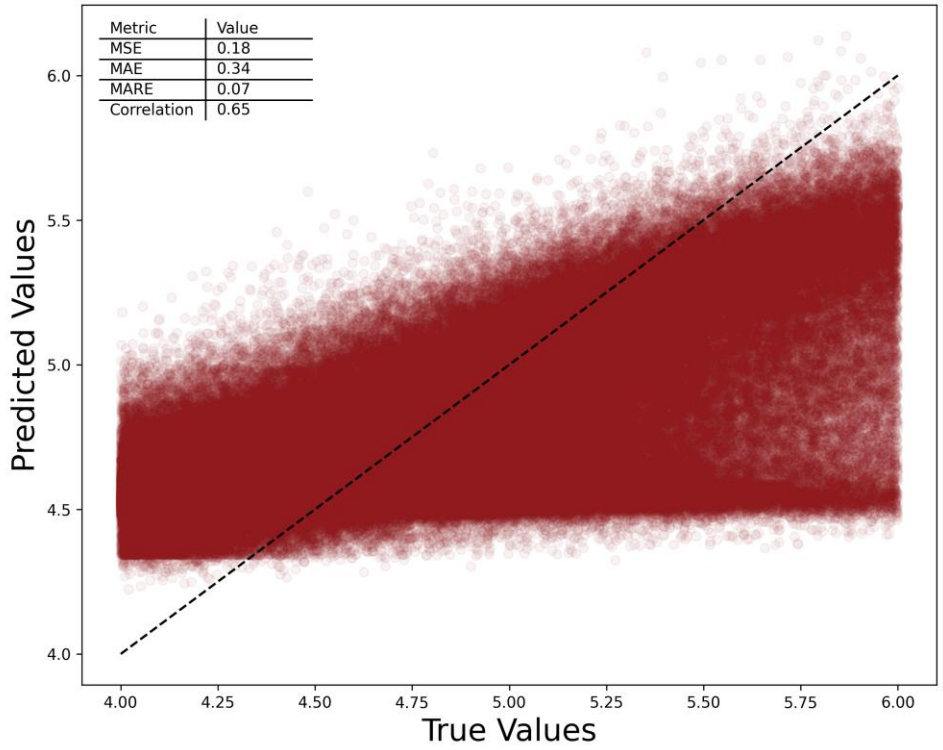
It is learning...

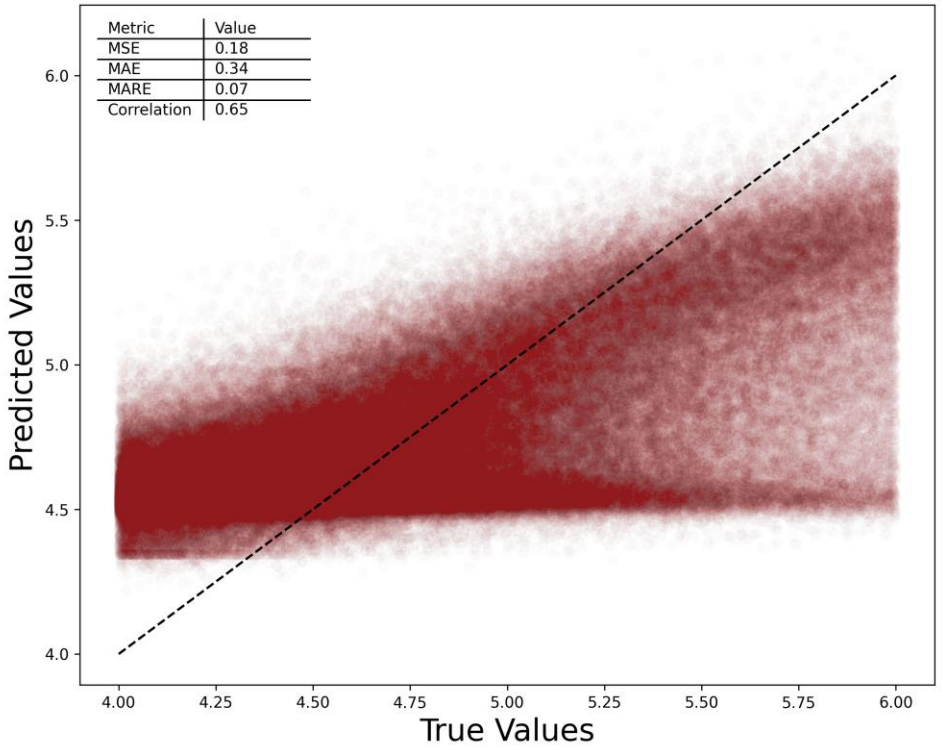


GNN



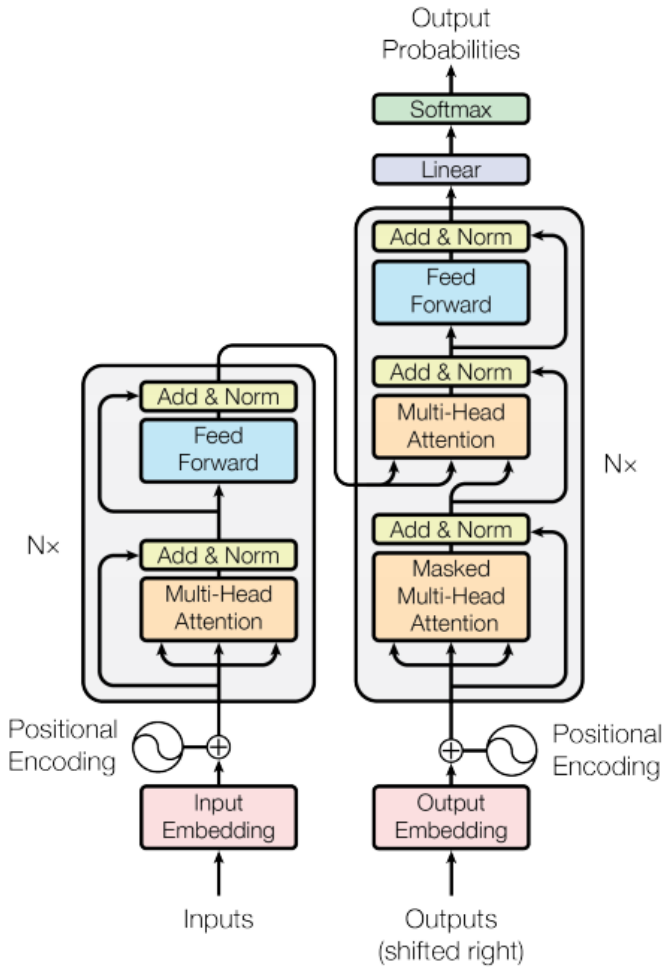
GNN



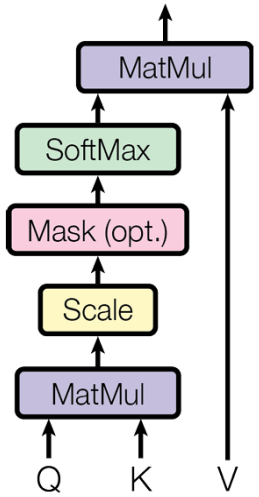


Transformer

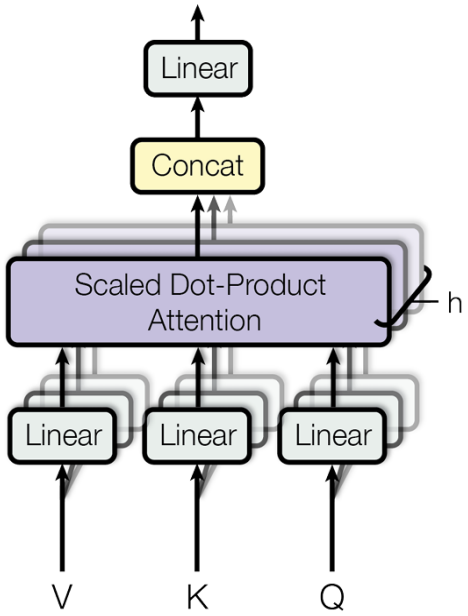
Attention Is All You Need



Scaled Dot-Product Attention



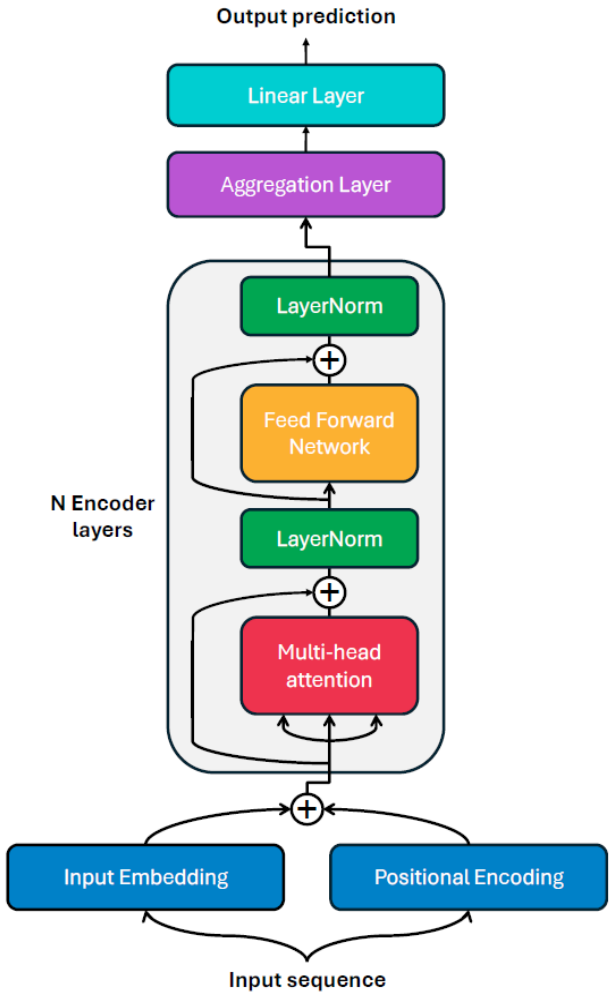
Multi-Head Attention



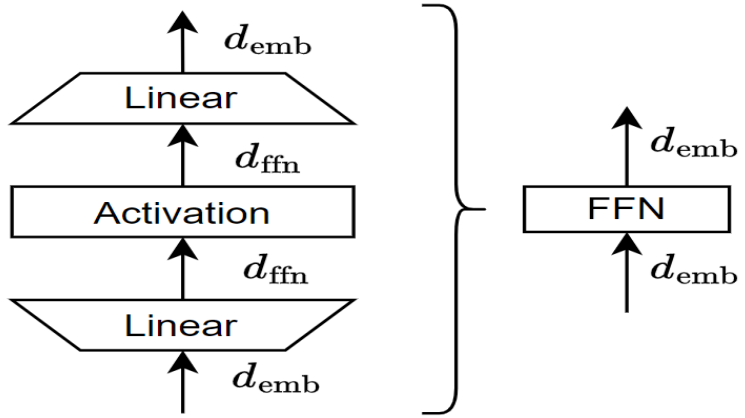
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Transformer

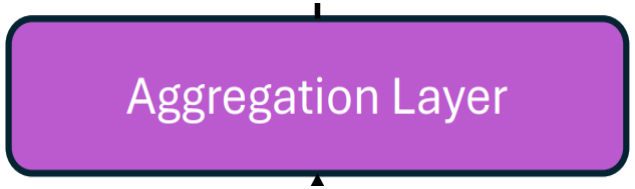
Transformer encoder model made by Luc Voorend (https://github.com/lucvoorend/IceCubeTransformer)



From *Angular reconstruction of high energy neutrinos using machine learning* by L. Voorend, 2025, University of Copenhagen.



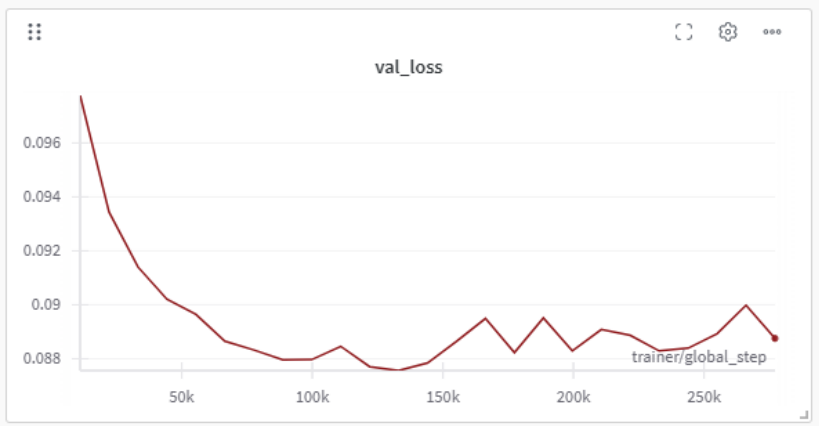
From *wikipedia.org/wiki/Transformer_(deep_learning_architecture)*

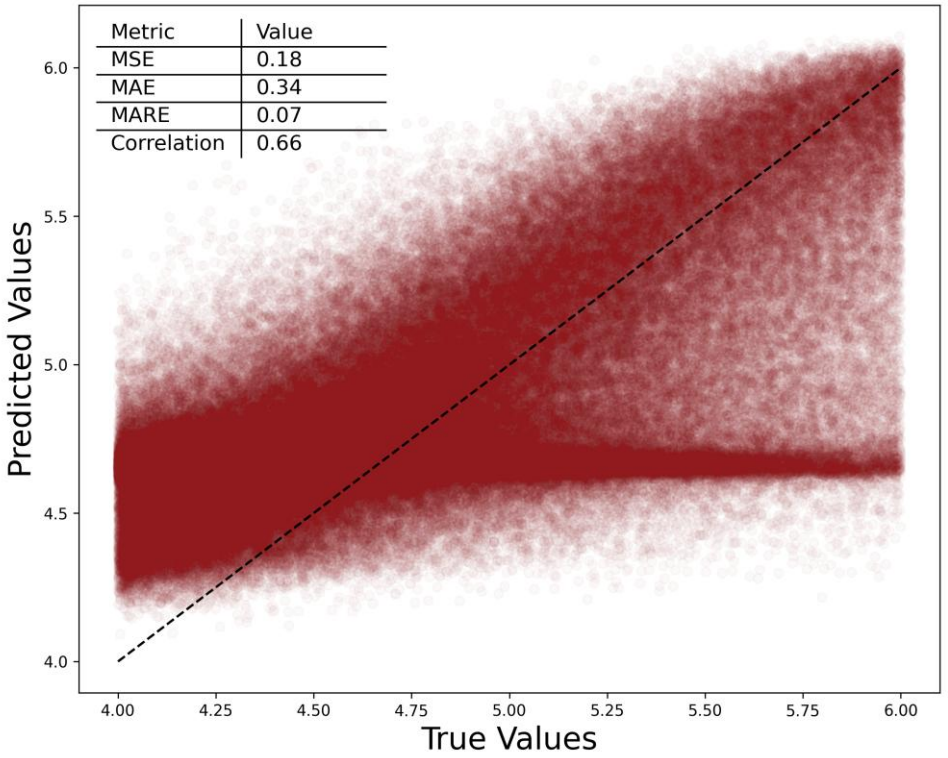


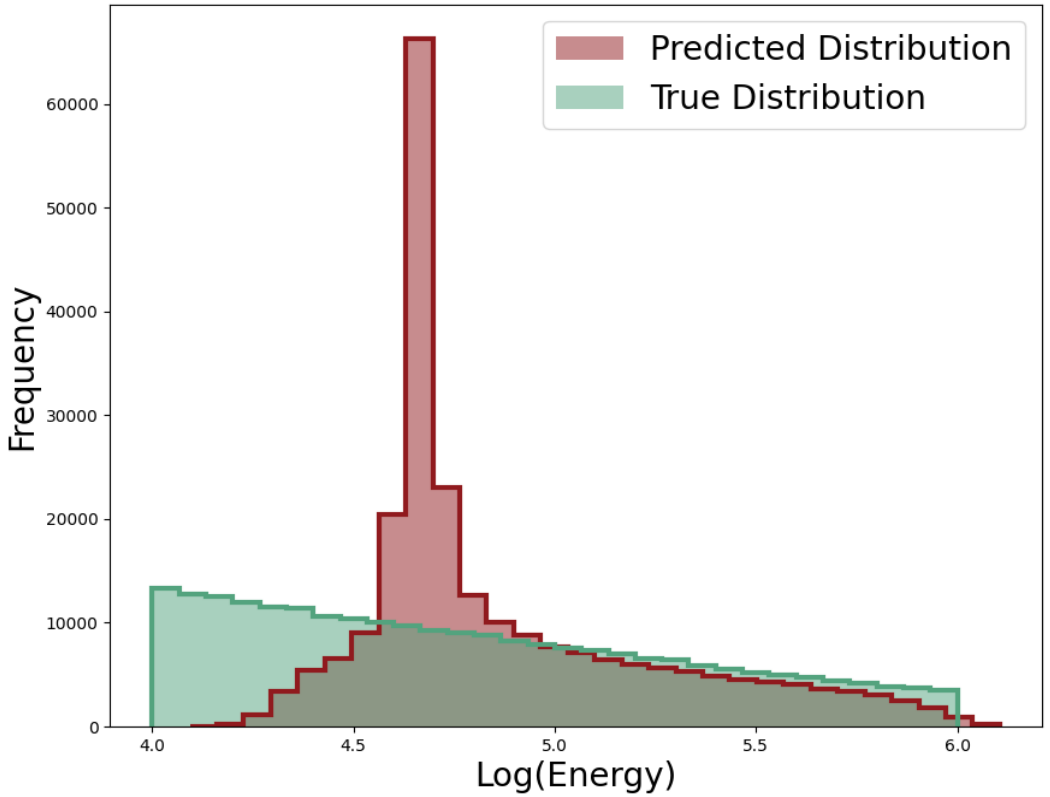
Need to mask to remove padded sequence elements
Mostly used just mean pooling

Transformer – Energy Regression

Trained in log domain
Model is learning, but it is exhibiting a strong bias towards the mean





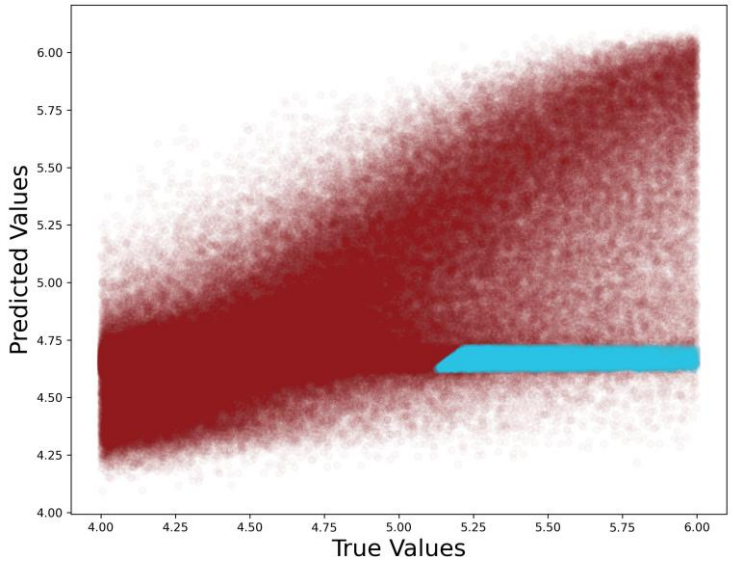


Transformer – Energy Regression

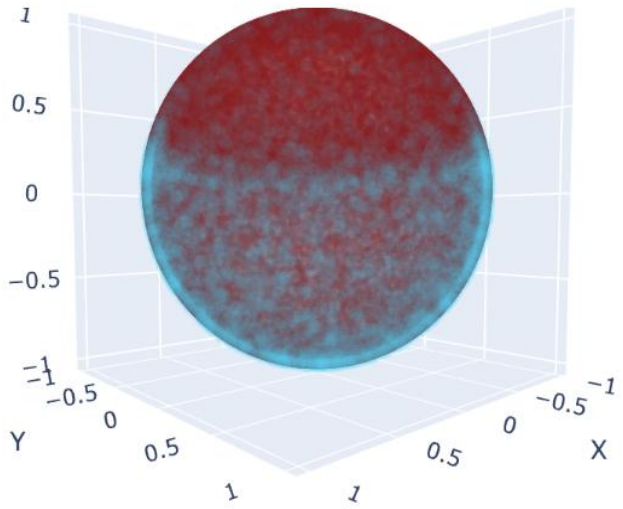
Mean predicted points clearly correlated with zenith and number of domes.

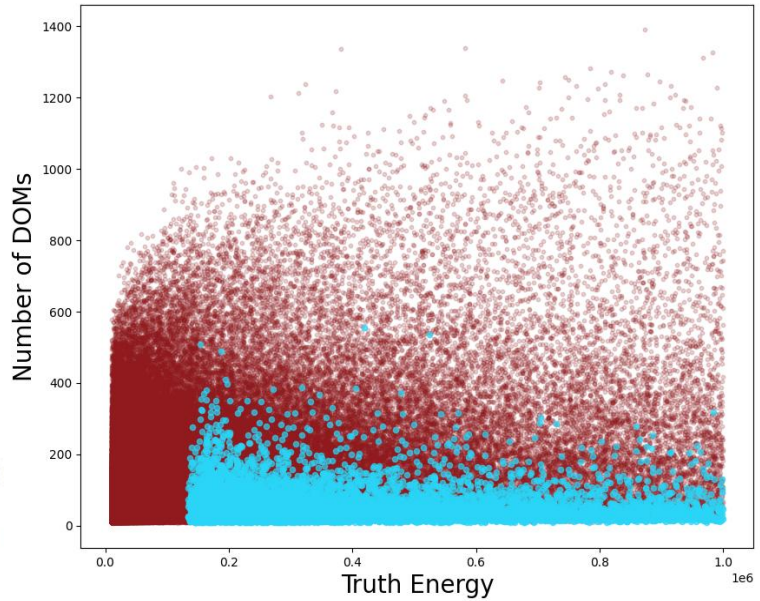
Quick test: Add number of domes as input to final linear layer. Little to no improvement

Idea: Train model for two hemispheres separately

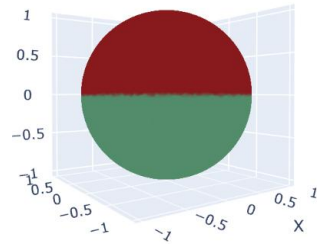


Direction

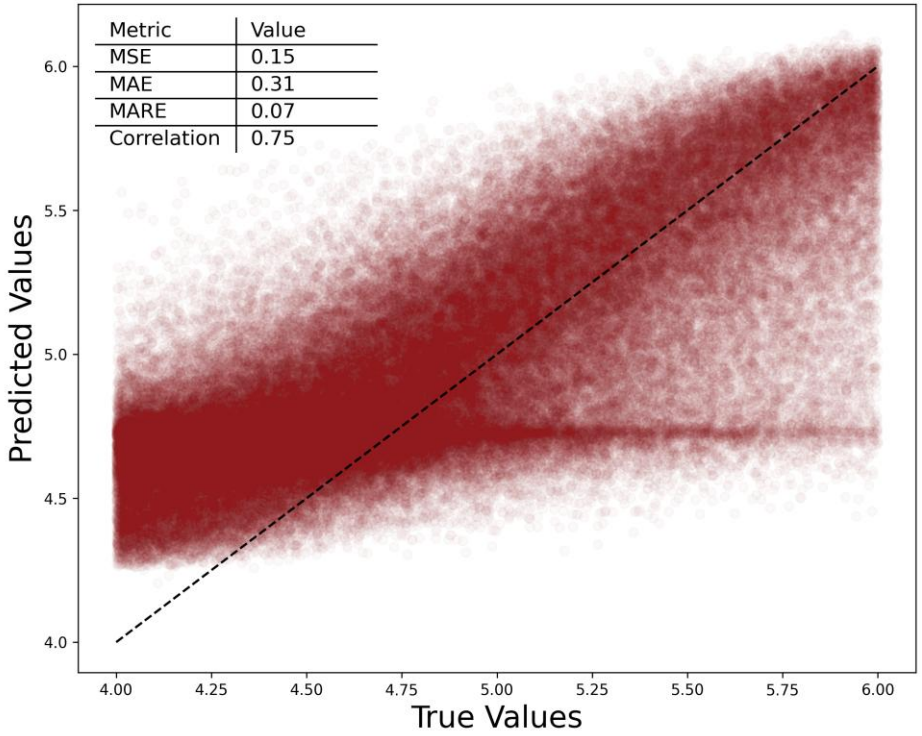




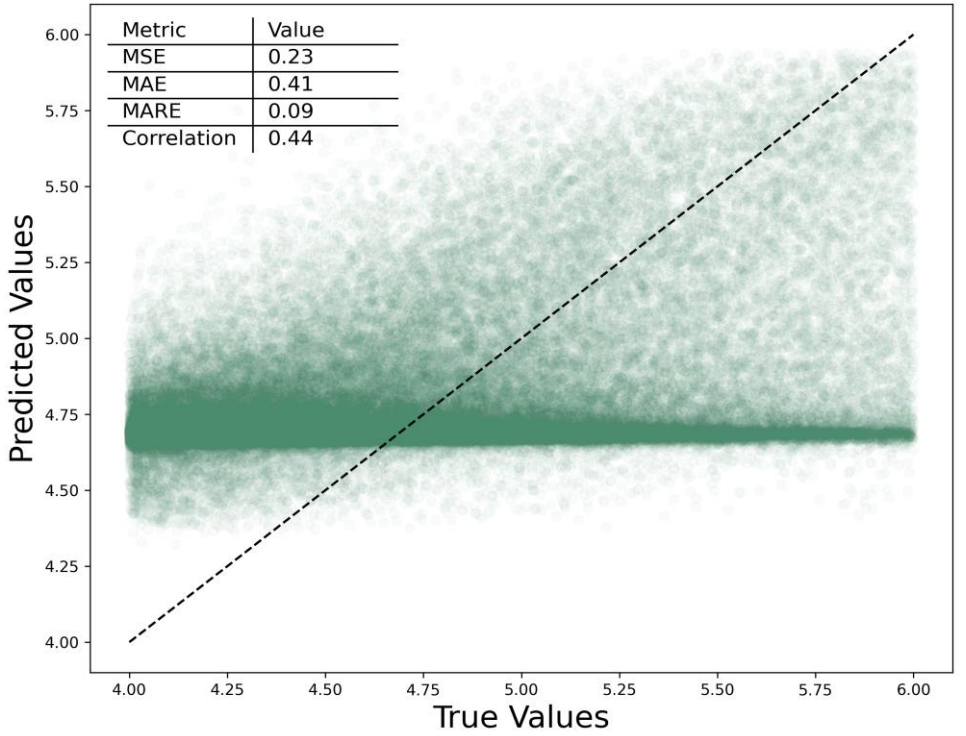
Transformer – Energy Regression



$$\text{Zenith} < \frac{\pi}{2}$$

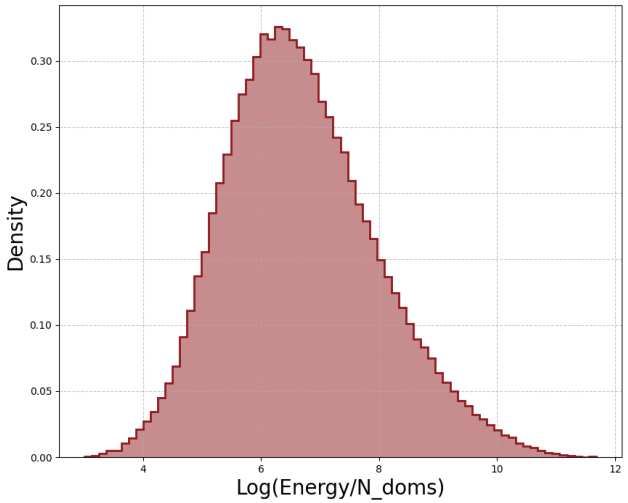


$$\text{Zenith} > \frac{\pi}{2}$$



Transformer – Energy Regression – Further work

Domain

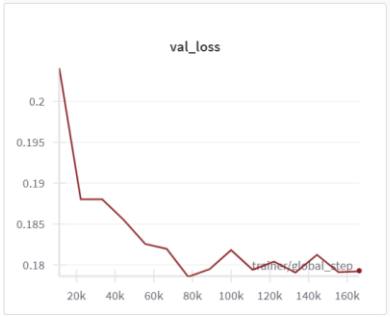


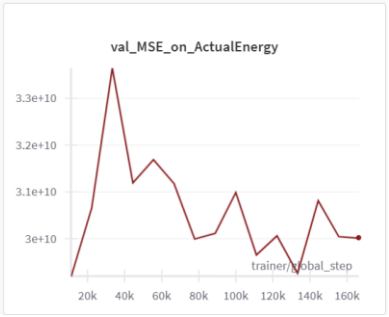
Aggregation

Mean, Sum, Min and Max instead of just Mean

Final regression

Replace final linear layer with more complex MLP.





Implemented both in model with no increase in performance (See Appendix). However, no optimization was done.

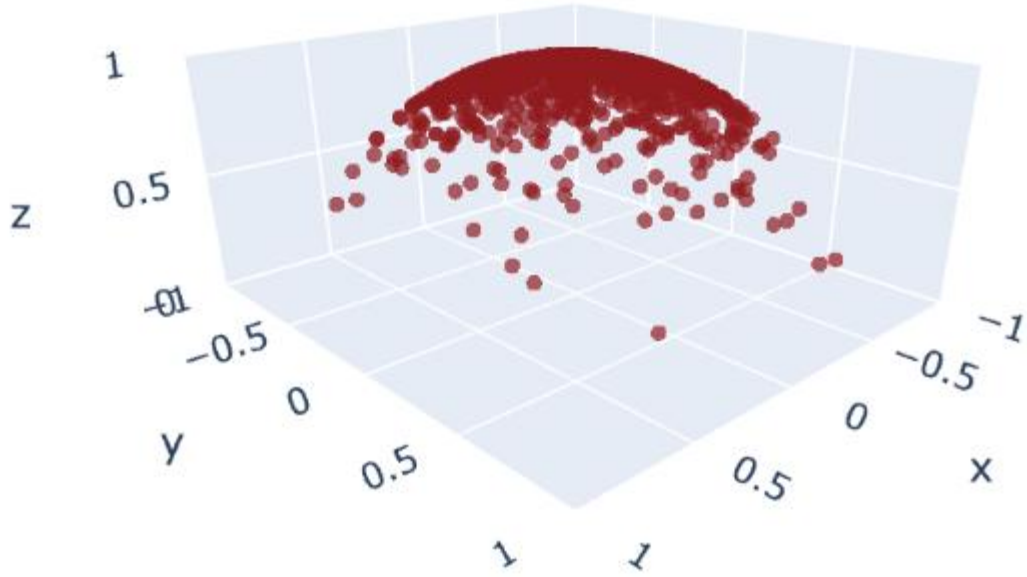
Transformer – Direction

- x, y, z $\rightarrow \theta, \phi$

- Von Mises Fisher Loss

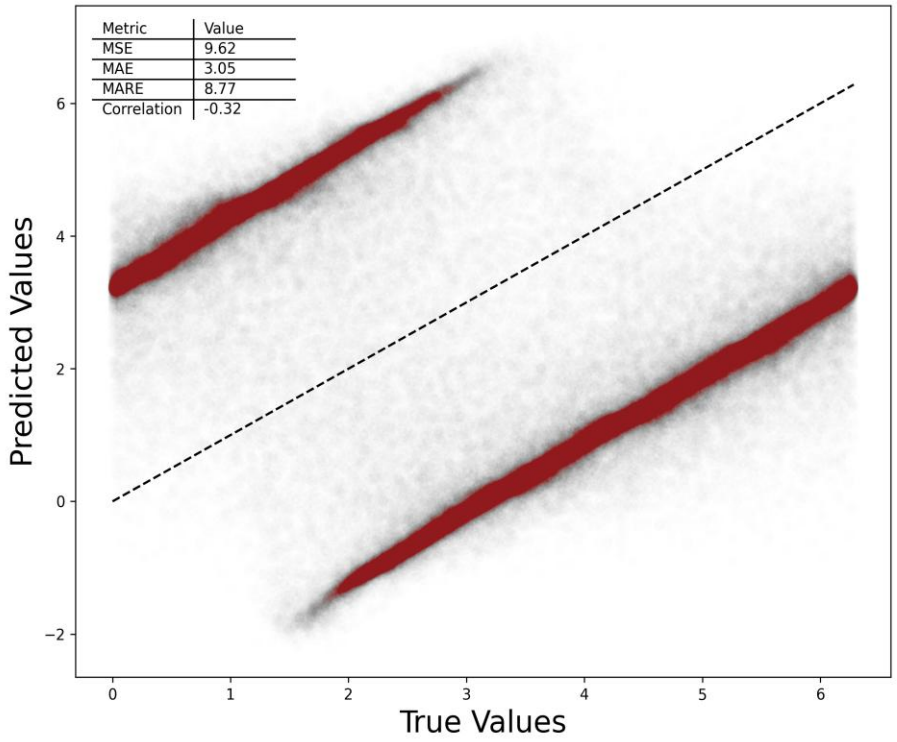
$$\log \mathrm{prob}(x_{\mathrm{truth}} | x_{\mathrm{predicted}}, \kappa)$$

- κ Concentration/certainty parameter
- Calculated using 3vectors with norm 1

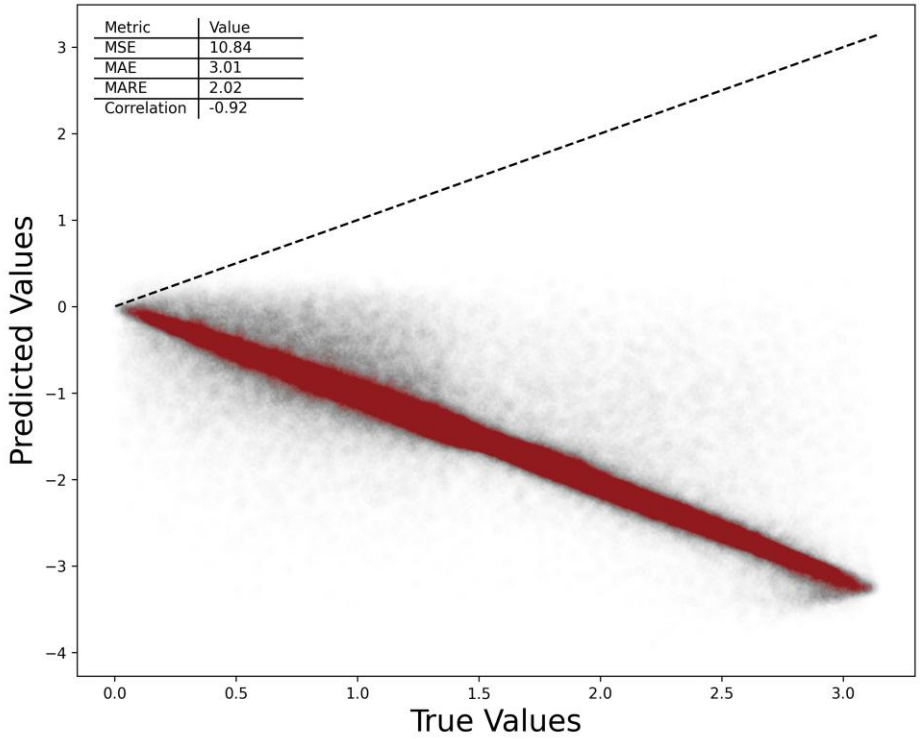


Transformer – Direction

Azimuthal Angle



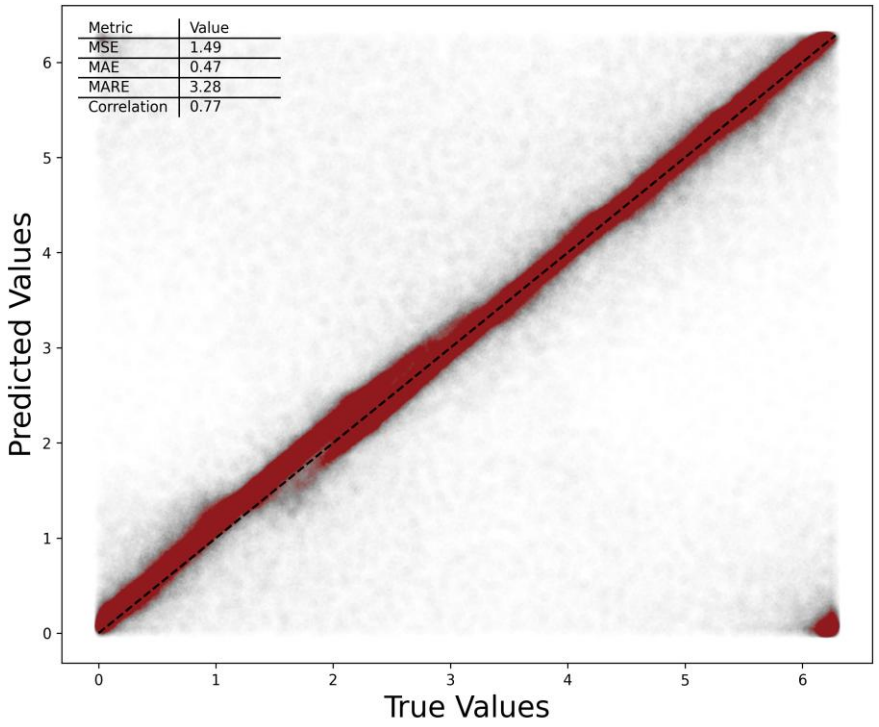
Zenith Angle



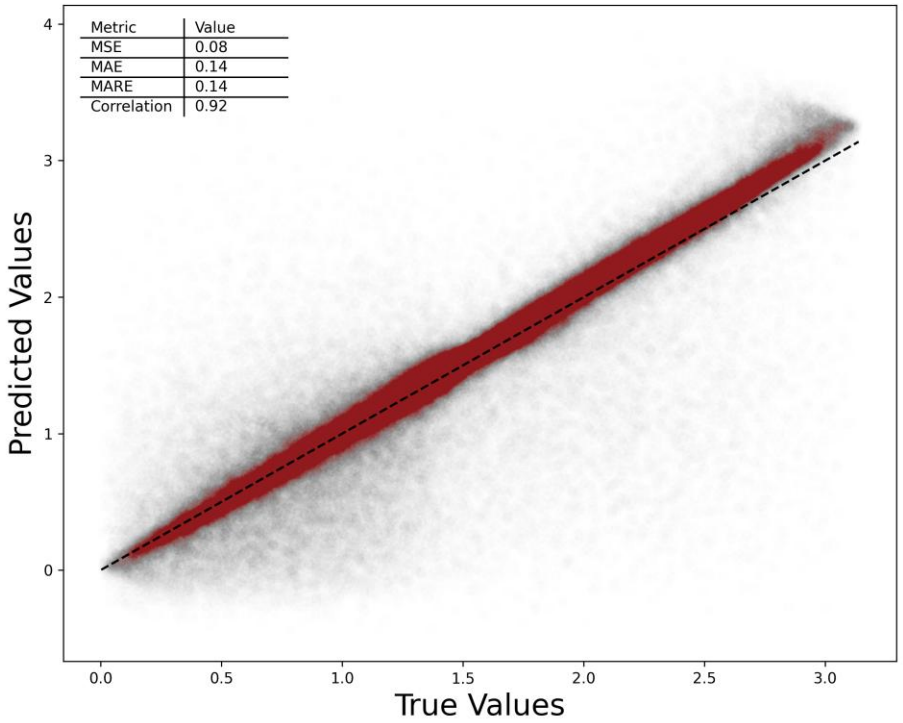
Periodic Domains – Transformation of predicted values to account for physical interpretation

Transformer – Direction

Azimuthal Angle



Zenith Angle



Curiosity: Attention Weights - A window into the black box

Conclusion

- Positive experience with ‘the Troels model’
- Multiple models to compare
- Room for optimization with energy
- Significant difference in prediction power for different targets

Thank you for listening!

Special thanks to Luc for code and assistance

(All parts contributed equally)

Links to our github pages:
https://github.com/DavidJohansen/IceCubeTransformer
https://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@github.com/mister-fran/FinalProjectFran
https://github.com/Moebergs/AppML_FinProject

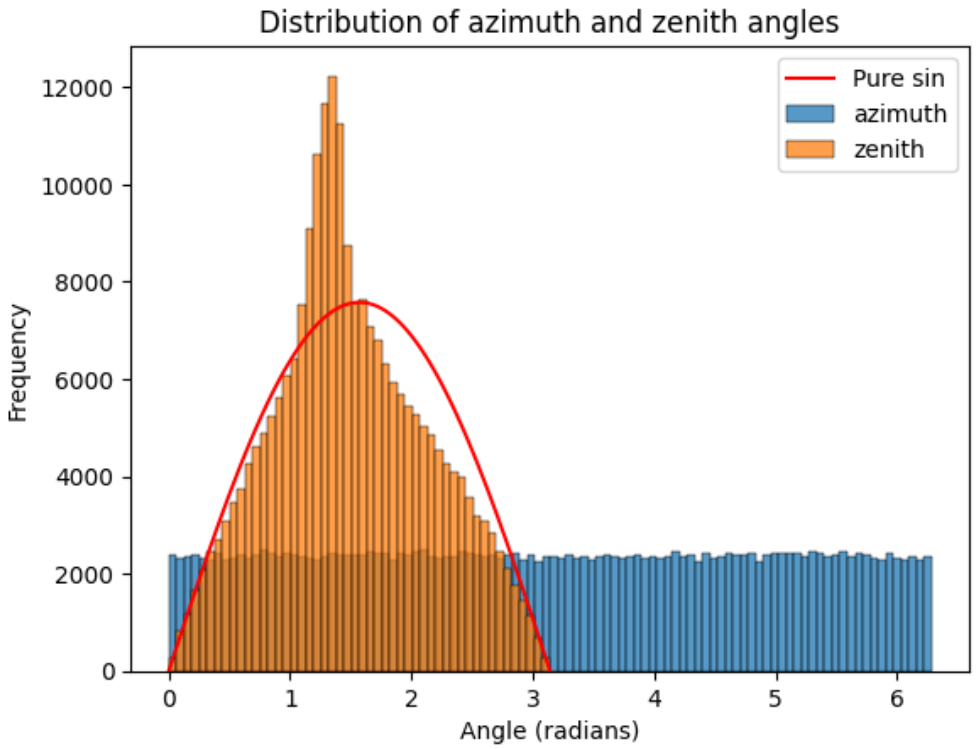
Appendix

Appendix A – Data

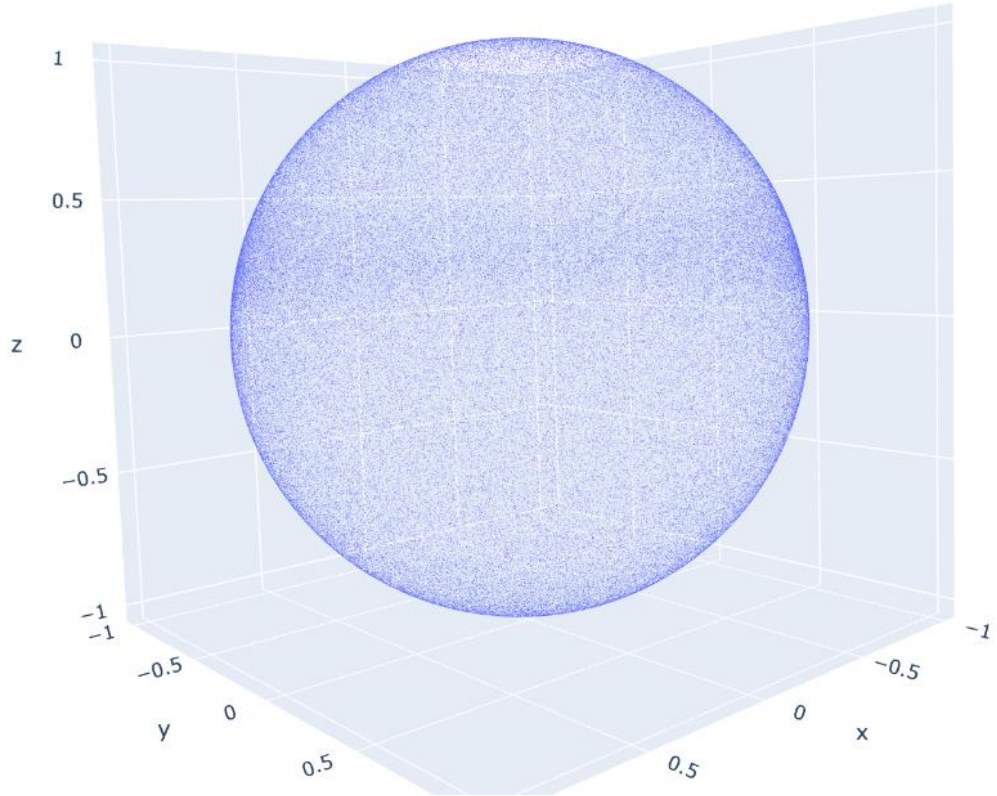
All data is generated through simulations. This means that it contains no NaNs, outliers or any other difficulties.
This does however give certain limitations to the applicability of the models. Domain-shifts will be expected upon transition from simulated to real-world data.
A few parameters are only simulated in a narrow parameter space. The parameter bad_dom_status is only simulated with a value of 1, meaning that any deviation from this value can give rise to unexpected predictions. This is also the case for a few other parameters.

Appendix A – Data

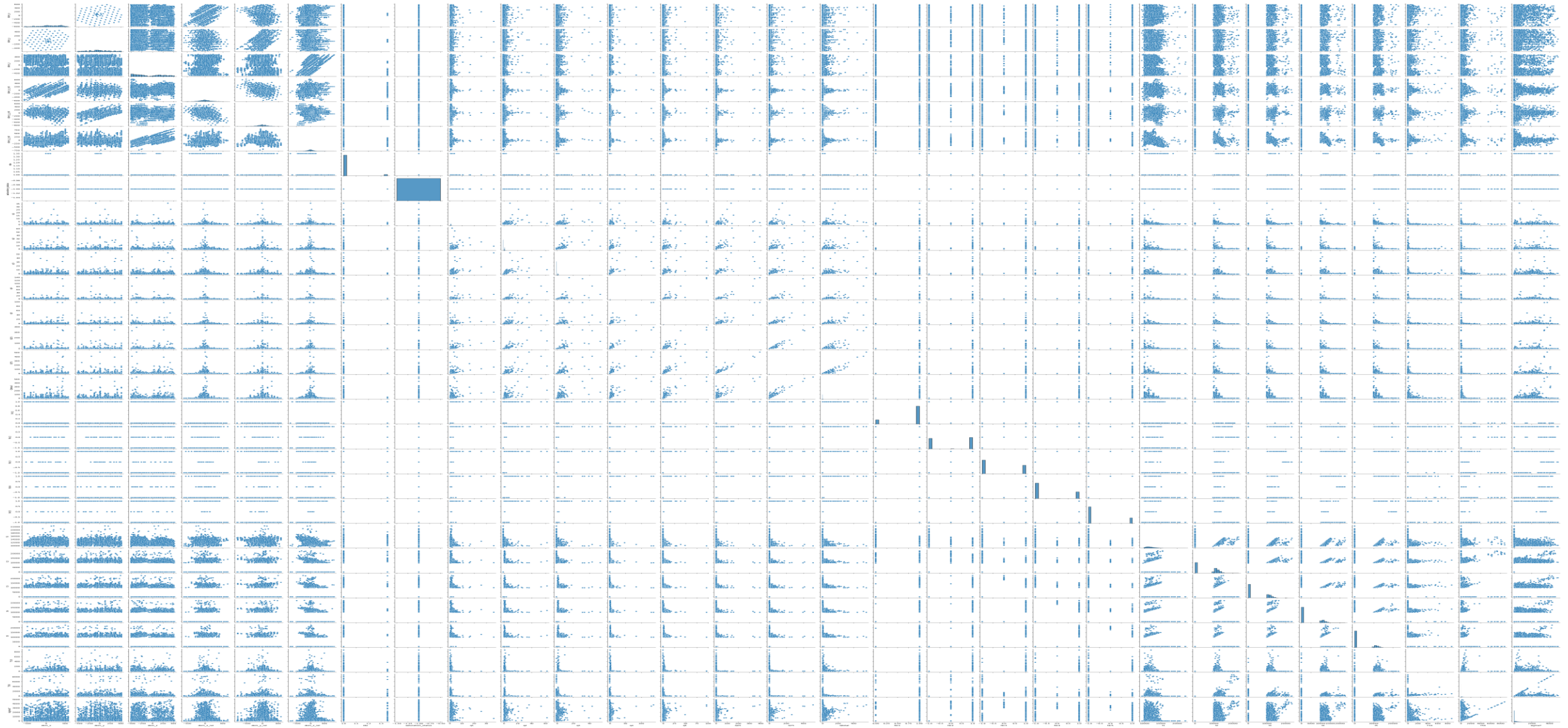
The simulated data does not have a spherical distribution, due to the neutrinos' interaction with the Earth and the atmosphere. We do, however, have uniform azimuth distribution, giving rotational symmetry.



3d scatter plot of spherical distribution of data points



Appendix A – Data (feature correlation and distribution)



Appendix B – Tools

The Colab file used for the energy regression part is:

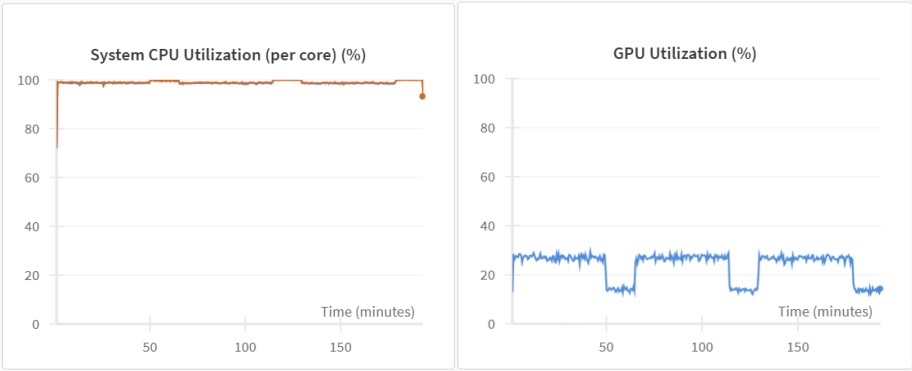
https://colab.research.google.com/drive/1wbdo8p09oRSELITFzBZrL8zmzLuplh7J?usp=sharing

For the other tasks, other Colab files were used but one could simply replace the initial git clone link

It was mostly run with the L4 GPU, which gave the best efficiency compared to the cost. One can also run it on the free version, but importantly the paid version gives access to more CPU cores, that are used to increase the number of workers in the dataloader greatly improving the GPU utilization.

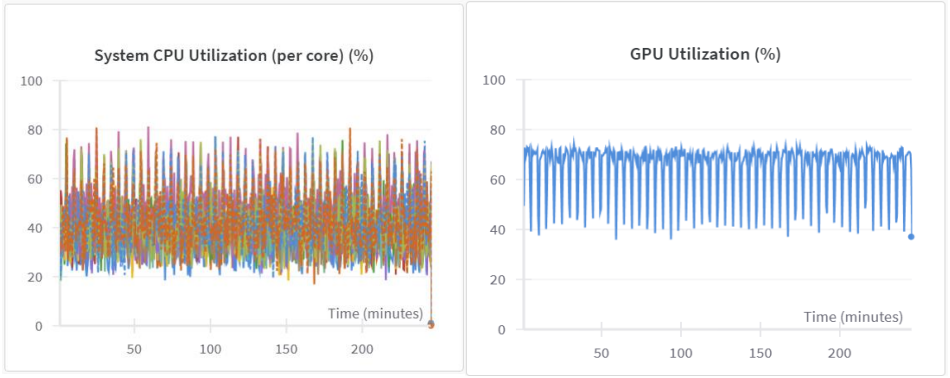
Free Colab Run

T4 GPU – 2 CPU cores.
CPU utilization is maxed out and GPU utilization is around 25%



Paid Colab Run

L4 GPU – 12 CPU cores.
CPU utilization is moderate and GPU utilization is around 75% for a beefier GPU



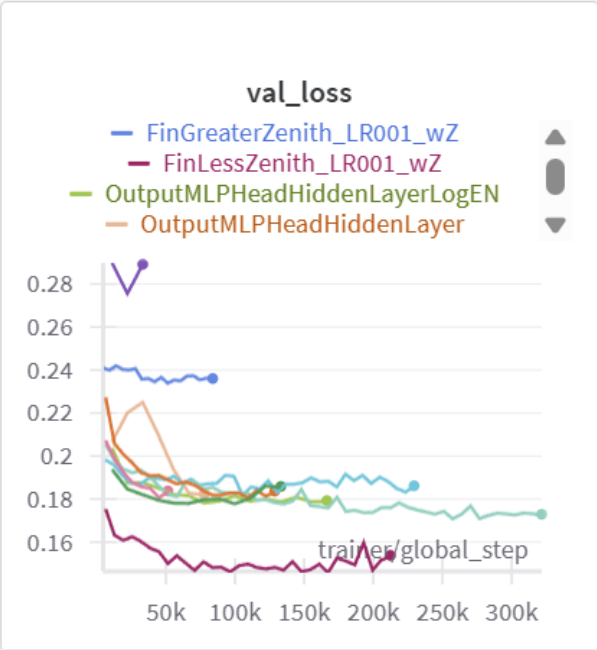
Appendix B – Tools

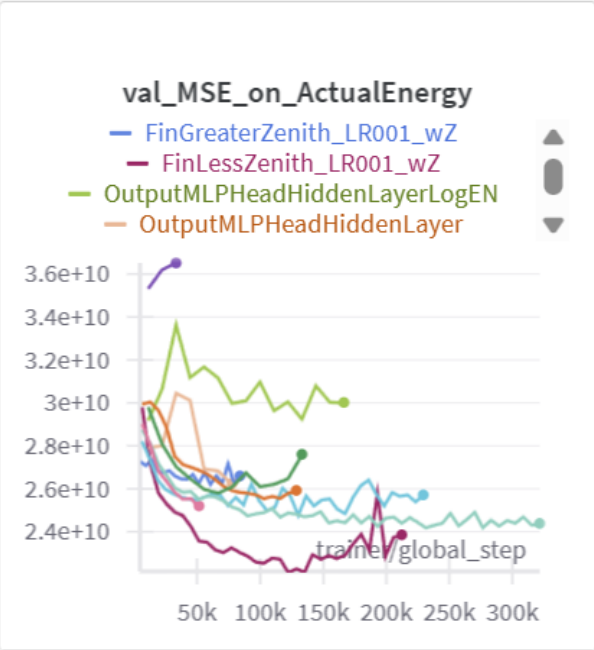
In our Weights & Biases workspace we mainly tracked these following metrics:

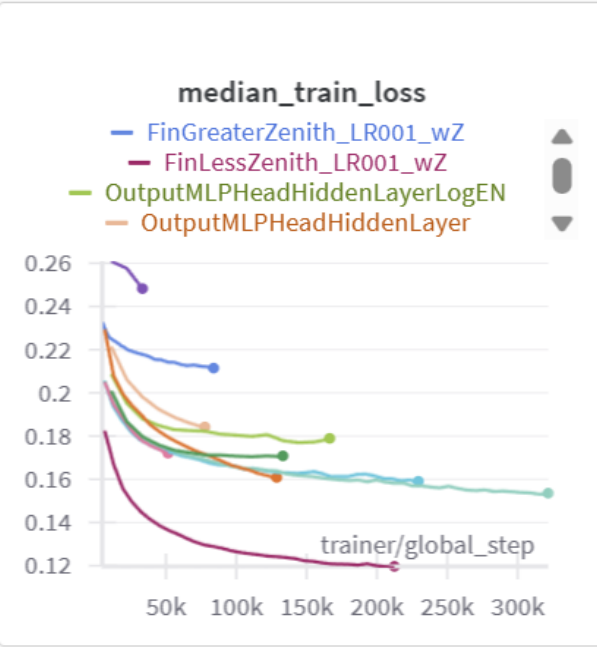
Train_loss: Training loss in the domain we are training

Val_loss: Validation loss in the domain we are training

Val_loss_on_ActualEnergy: Validation loss calculated after transforming prediction back to normal energy







Appendix C – Neural network

The neural network was able to make simple but poor predictions. Not much time was spent on any form of optimization.

The goal of the NN was to overcome all obstacles related to data loading, running the code, and inference.

Gave great insight into the classes and data structures present in the code and data.

MLP1 - HP	Value
N_hiddenLayers	1
n_1	128
dropout	0.1

MLP2-HP	Value
N_hiddenLayers	0

Appendix D – GNN

During data loading, all events are padded or truncated to an equal length.

The GNN model does not require equal-sized input data, so training with and without padding was performed.

Removing padding meant that the input data could not be collected in a single tensor forcing us to implement a for-loop over all events, seriously affecting the runtime of the algorithm.

Removing padding did, however, improve loss in the long run

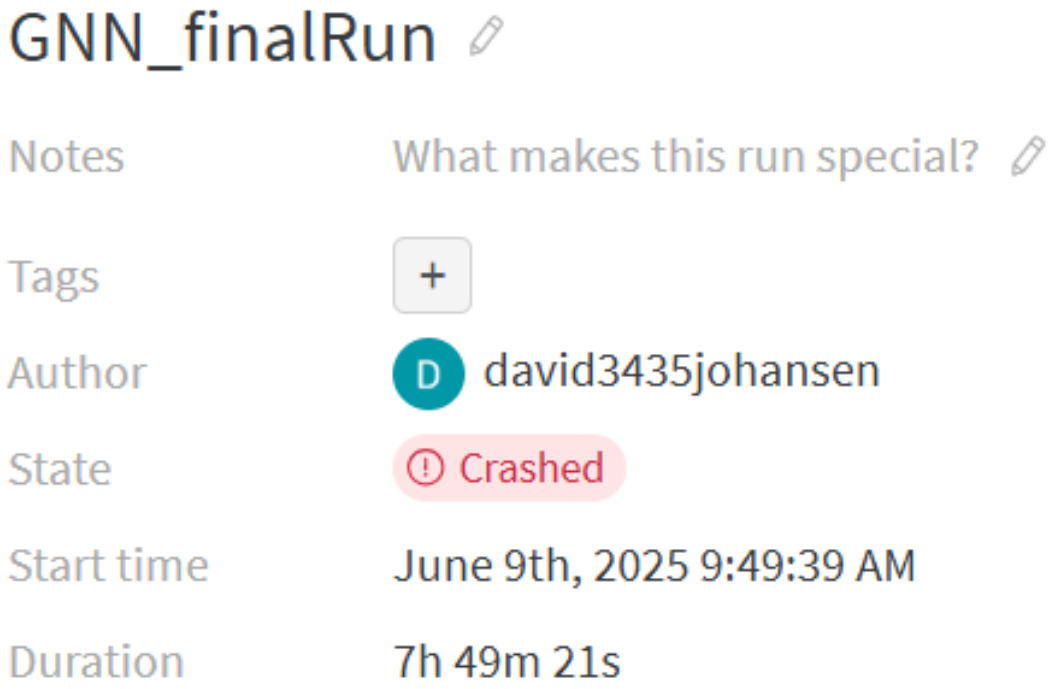
```python
# Apply each convolution
aggregated_list = []
for b in range(batch_size):
    batch_x = x[b]  # Shape: (seq_len, feat_dim)

    # Create mask for non-zero rows (valid events)
    valid_mask = ~torch.all(batch_x == 0, dim=1).to(device)
    # Filter out zero padding
    valid_x = batch_x[valid_mask]  # Shape: (valid_len, feat_dim)

    # Get edge indices for valid events
    edge_index = self.get_edge_index(valid_x).to(device)

    # Apply GATv2Conv
    x_ = self.convs(valid_x, edge_index)

    x_ = torch.cat([x_, valid_x], dim=1).to(device)  # Concatenate GNN output wit

    x_ = self.mlp1(x_)

    min_pool = torch.min(x_, dim=0)[0]
    max_pool = torch.max(x_, dim=0)[0]
    sum_pool = torch.sum(x_, dim=0)
    mean_pool = torch.mean(x_, dim=0)

    aggregated_list.append(torch.cat([min_pool, max_pool, sum_pool, mean_pool]))
```

Appendix D – GNN

Only ‘small’ runs with 1 training dataset were finalized, due to unexpected termination of runtimes for larger runs.

Without any error messages, resolving the issue proved difficult.

Due to time constraints and limited Colab tokens, the issues did not get resolved.



Appendix D – GNN

Due to long runtimes, only manual HP optimization has been performed. The parameters of the best run are shown in the table.

Hyper parameter	Value
K_neighbours	8
N_convs	4
Dropout	0.1
N_hidden_mlp1	1 (640)
N_hidden_mlp2	2 (64, 64)

Appendix E – Transformer

We built our investigations of the transformer architecture for IceCube data regression on the work of Luc Voorhend. If one wants to learn more about or just use the model architecture, we encourage you to visit his github (https://github.com/lucvoorend/IceCubeTransformer) which gives a wonderful overview of the architecture. Additionally, our Githubs (Linked in the last slide of the presentation) show our changes to the model for the different use cases.

Here is an overview of the most important parts of the model we utilize also showing the structure of the files:

```
├─── training_and_inference/ # Core scripts for the transformer model
│   ├─── src/
│   │   ├─── model.py # Script defining the model structure. (We used this to change the architecture)
│   │   ├─── dataset.py # Script with Dataset Class (We used to change the domain and filtering process)
│   │   ├─── dataloader.py # Dataloader implementation (Used to make changes to the variables that were propagated to the dataset)
│   │   ├─── loss.py # Loss functions (Used to try out different loss functions)
│   │   ├─── utils.py # Assertion functions for config file (We didn't change this)
│   ├─── train.py # Script to train the model
│   ├─── inference.py # Script to run inference and evaluate the model
│   └─── config.yaml # Config file controlling settings for training and inference
```

Appendix E – Transformer

```python
class regression_Transformer(nn.Module):
    """
    Regression transformer class:
    - contains an input embedding layer, position embedding layer, transformer layers, and output layers
    - applies the transformer to a sequence of embeddings
    - returns a single predicted target value
    """
    def __init__(self,embedding_dim=96,n_layers=6,n_heads=6,
                 input_dim=32,seq_dim=256,dropout=0.1,output_dim=1,):
        super(regression_Transformer, self).__init__()

        self.input_embedding = nn.Linear(input_dim, embedding_dim)
        self.position_embedding = nn.Embedding(seq_dim, embedding_dim)
        self.layers = nn.ModuleList([EncoderBlock(embedding_dim, n_heads, dropout) for _ in range(n_layers)])
        self.layer_norm = nn.LayerNorm(embedding_dim)
        self.linear_regression = nn.Linear(embedding_dim, output_dim)
```

```python
    def forward(self, x, target=None, event_lengths=None, original_event_n_doms=None):
        seq_dim_x = x.shape[1]
        device = x.device

        input_emb = self.input_embedding(x).to(device)
        pos_emb = self.position_embedding(torch.arange(seq_dim_x, device=device))
        pos_emb = pos_emb.unsqueeze(0).expand(x.shape[0], -1, -1)   # Shape: (batch_size, seq_dim, emb_dim)

        x = input_emb + pos_emb

        for layer in self.layers:
            x = layer(x, event_lengths=event_lengths)

        # Feed the output of the transformer to the pooling layer
        batch_dim, seq_dim_x, emb_dim = x.shape[0], x.shape[1], x.shape[2]

        # Aggregate x over the sequence dimension to the event length
        row_indices = torch.arange(seq_dim_x).view(1, -1, 1)  # Shape: (1, seq_dim, 1)
        row_indices = row_indices.expand(batch_dim, -1, emb_dim)
        row_indices = row_indices.to(device)

        mask = row_indices < event_lengths.view(-1, 1, 1).to(device) # Shape: (batch_size, seq_dim, emb_dim)

        # Apply mask to x and do mean pooling
        x = x.masked_fill(mask == 0, 0)
        x_mean = x.sum(dim=1) / event_lengths.view(-1, 1) # Shape: (batch_size, emb_dim)
        y_pred = self.linear_regression(x_mean)
```

On the left is the part of Luc's code defining the regression model. One can here rather easily follow what the model does which, for us, made the architecture very approachable and is why we include it explicitly here.

It starts with the embedding of both the simple input embedding and the position embedding being summed.

This is then input to the encoding layers which we usually repeat 4 times (n_layers=4). Here, a few things are hidden under the hood namely the attention mechanism, FFN, as well as the LayerNorms and skip connections for better training and backpropagation.

After the encoding blocks the padded sequence elements are removed and mean aggregation is done.

Finally a linear layer does the final prediction mapping to the output dimension

Appendix F – Transformer - Energy

Here we present our approach to doing energy regression using the Transformer architecture focusing on optimization steps and more fundamental changes we made to the model.

To start off we made a Github repository focusing on the energy regression. Here we made a model for energy regression by implementing and changing parts of Luc's code which is designed and hard coded for angular regression.

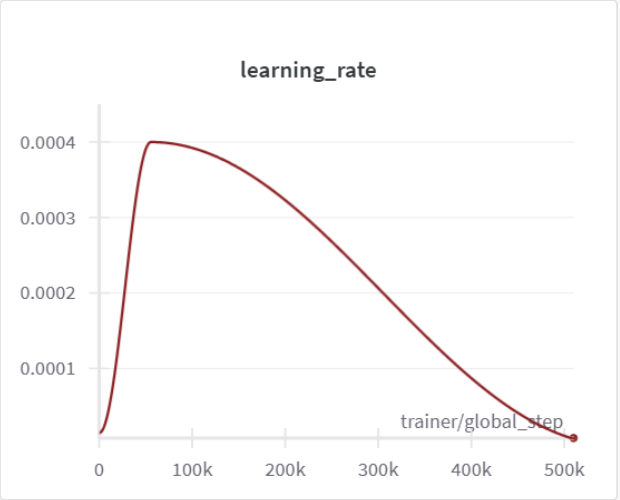
With the model running as intended we started the cruelling task of optimization. To optimize the transformer model for energy regression we just tried different configurations of hyperparameters (Grad student search).

With long training times, we did our best to do this in a structured manner focusing mostly on learning rate, sequence length, embedding dimension and batch size.

Appendix F – Transformer - Energy

Learning rate

With learning rate being as important as it is we tried a few different things with this. Namely, we tried using a dynamic learning rate, as shown on the right. The idea here is to let the model explore the loss landscape in the beginning with a later fine tuning.
However, we found that a simple constant learning rate would approach the general minimum validation loss we saw faster and more consistently. This was therefore used for the final models with values around 0.001.
It would be interesting to explore learning rate even more and maybe trying with a simple decaying learning rate.



Appendix F – Transformer - Energy

Loss function

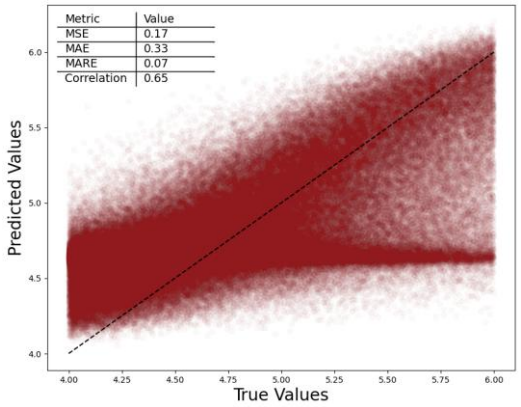
We tried using a few different loss functions as seen on the right. Generally, we used MSE loss, but few runs used MAE. Huber loss was used for some initial trials in the true energy domain to try to mitigate the effects from the skewed values by evaluating errors on smaller values quadratically (MSE) and larger values linearly (MAE). We however quickly realized that the idea of using the true energy domain had to be abandoned.

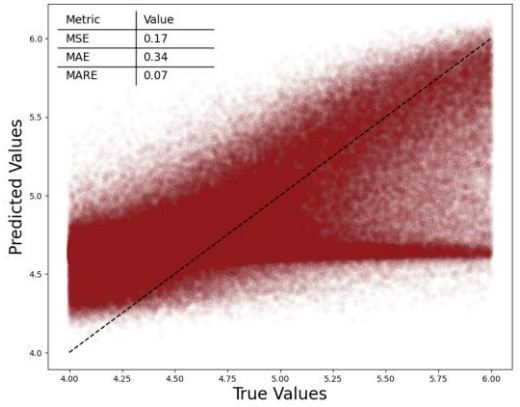
```python
def MSE_loss(y_pred, target):
    y_pred = y_pred.squeeze()
    target = target.squeeze()
    loss = torch.mean((y_pred - target) ** 2)
    return loss

def MAE_loss(y_pred, target):
    y_pred = y_pred.squeeze()
    target = target.squeeze()

    loss = torch.mean(torch.abs(y_pred - target))
    return loss

def Huber_loss(y_pred, target, delta=0.1e6):
    y_pred = y_pred.squeeze()
    target = target.squeeze()

    loss = F.huber_loss(y_pred, target, delta=delta, reduction='mean')
    return loss
```

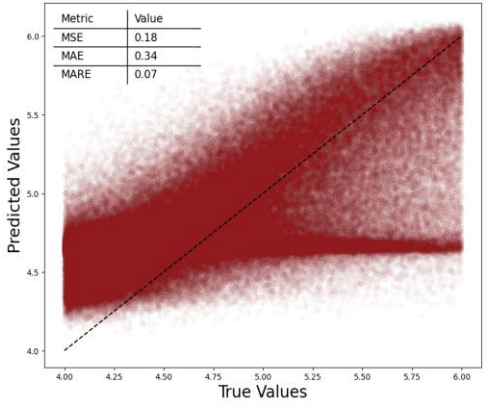
Appendix F – Transformer - Energy

Trials

Here we present a few of the later trials with their distinct hyperparameters. Generally, we see that they approach what we have denoted the 'Hand pattern'.







Common HP

Parameter	Value
Batch Size	64
Num Layers	4
Num Heads	2
Dropout (FFN)	0.1

There are naturally many more runs with other HP that we tried. These often also approach the 'Hand' or something worse.

Parameter	Value
Learning rate	Dynamic
Sequence length	256
Embedding Dim	256

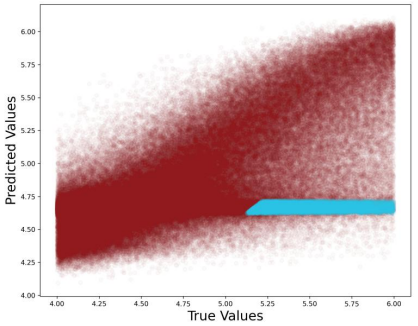
Parameter	Value
Learning rate	Dynamic
Sequence length	512
Embedding Dim	256

Parameter	Value
Learning rate	0.0005
Sequence length	256
Embedding Dim	128

**Dynamic learning rate follows the curve on the earlier plot*

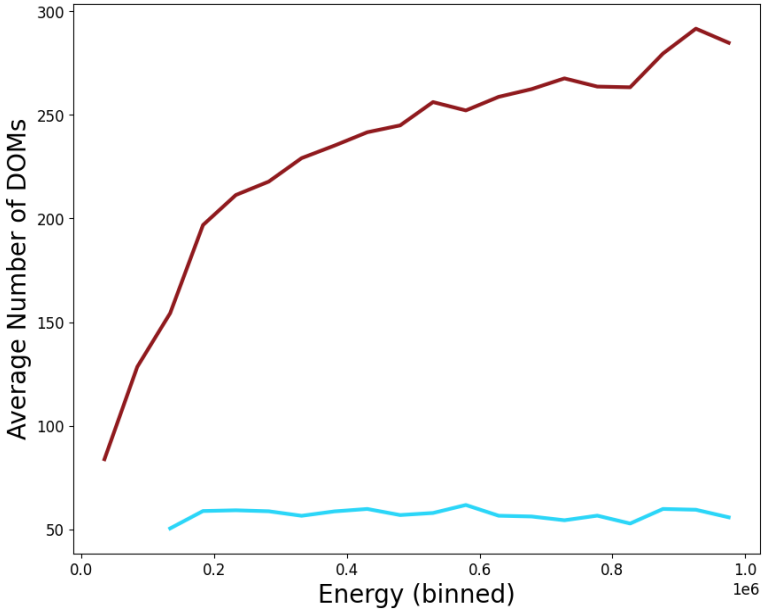
Appendix F – Transformer - Energy

Investigating the outliers



As we have seen beyond multiple model architectures and HP parameter choices the energy predictions tend towards the mean. We turned our attention to these outliers to see whether the points are random or fall into some specific category.

As shown in the presentation, there is a clear correlation between predictive power and the zenith angle. We've seen that the model is not able to learn from particles going through the earth before hitting the detector. Additionally, we've seen a connection to the number of DOMs.



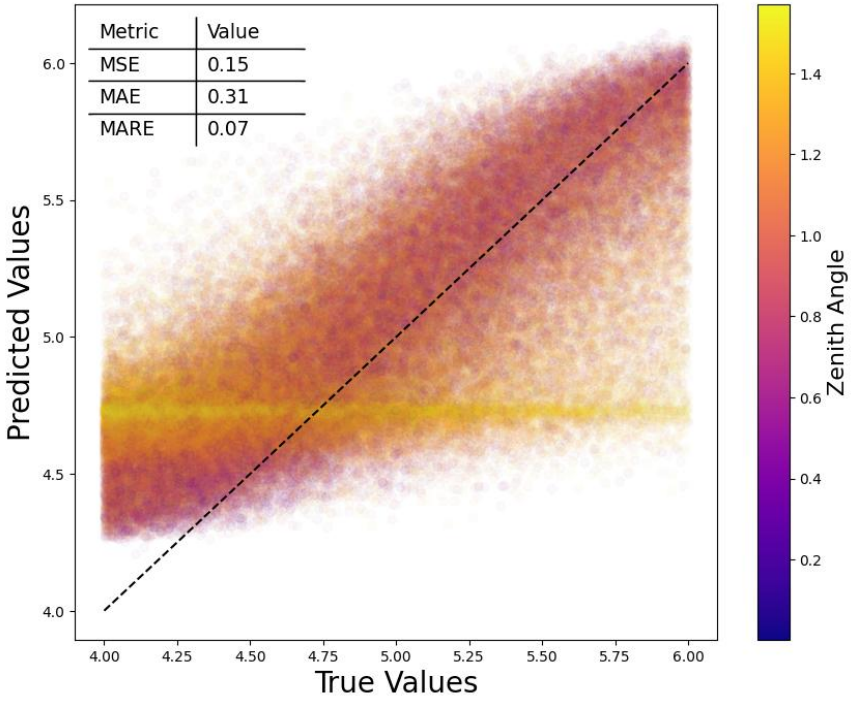
The plot on the left shows the average number of DOMs for the outliers and normal points for different energies. Interestingly, it is very constant for the outliers
For this reason, we tried adding the number of DOMs to the final linear layer thinking it could learn to understand this behavior better. This changed basically nothing, but there was no time to do optimization.

The next idea was to train the model on the particles coming from above and below the earth separately. The idea was to see whether the problem was the combination of the data. Also to see how good the model could learn without the clear outliers present.

Appendix F – Transformer - Energy

Zenith filter

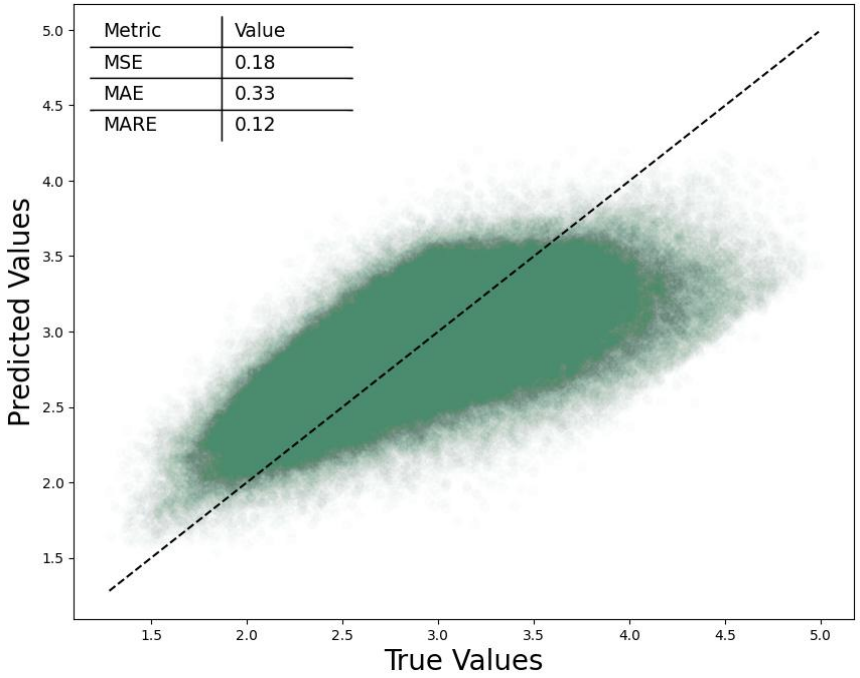
To do the zenith specific training a zenith filter was introduced in the dataset class. For the middle cut at pi/2 presented in the slides, we saw that there is clearly little to no learning happening for the particles going through the earth with some learning for those that don't.



On the left is shown the plot for the model trained exclusively on particles above the horizon. Here, a colormap determined by the zenith angles shows that the horizontal line is dominated by higher zenith values. We therefore expect a more aggressive cut to get rid of this, but didn't have time to try this out

Appendix F – Transformer - Energy

Further Work



The scatter plot on the left is the result from an initial attempt at running the model in the log(E/N) domain with no optimization. One can maybe optimize this to work better than we've seen for the log(E) domain. More work can be focused on this.

The code below shows the NN we quickly tried at the last step before the output of the model. An initial test showed no improvement to earlier instances, but it would be interested to try working more with this.

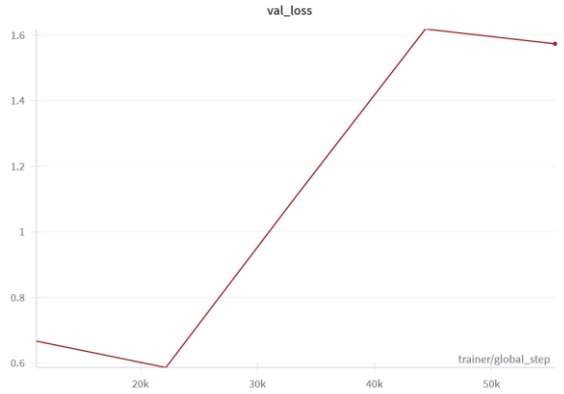
A similar situation occurred for the Mean, Sum, Min and Max aggregation test.

```
self.output_mlp_head = nn.Sequential(
    nn.Linear(embedding_dim+1, (embedding_dim+1)*4),
    nn.ReLU(),
    nn.Dropout(dropout),
    nn.Linear((embedding_dim+1)*4, output_dim))
```

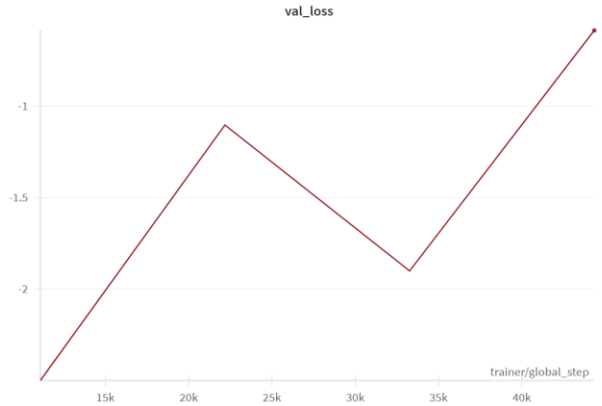
Appendix G – Transformer – Direction

While grad student optimizing these runs, multiple different VMF loss functions were implemented. These included:

- Constant Penalty Terms
- Variable Penalty terms
- No Penalty Terms



Constant



Variable

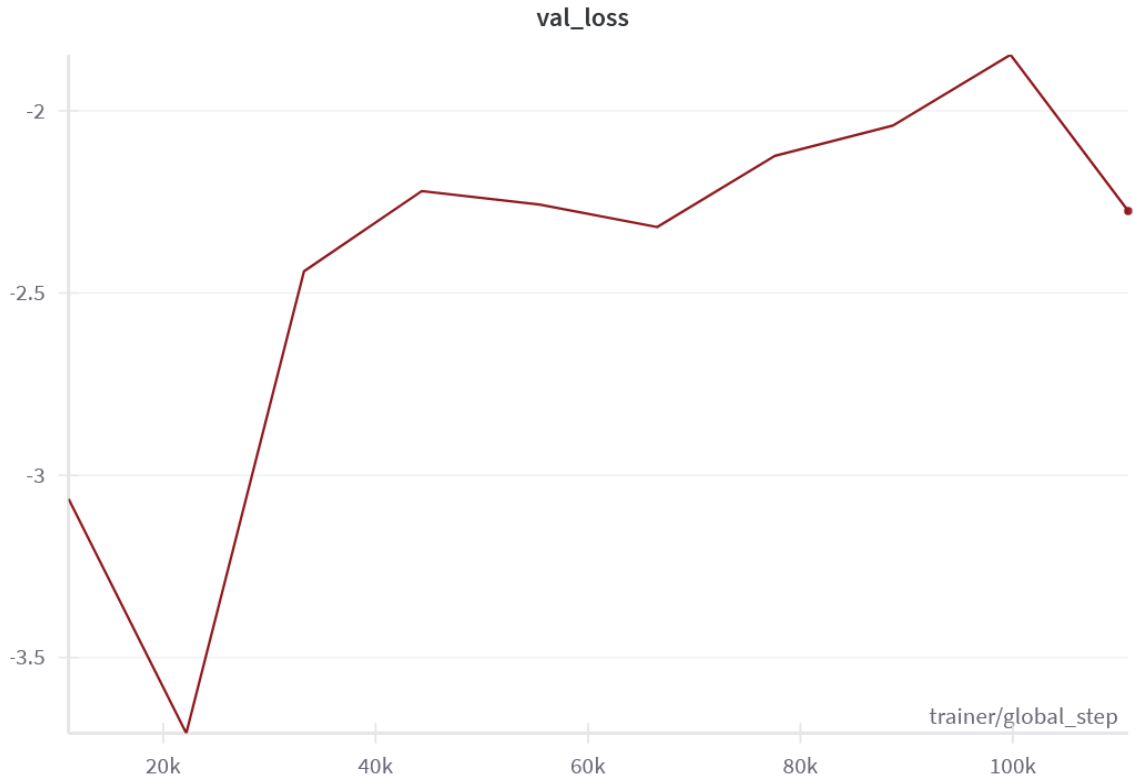
Ultimately having no penalty terms yielded the best result as it seemed like the penalty needed tuning to not overrule the regular gradient descent. As this was very time consuming it was deemed outside the scope of the project.

*This is likely also an artefact of not optimizing enough.

Appendix G – Transformer – Direction

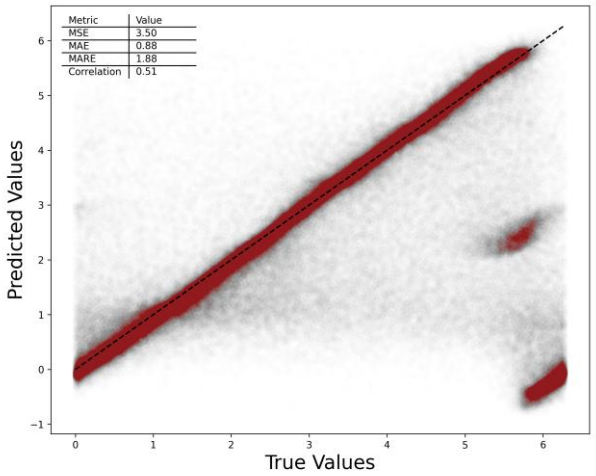
Also, we experimented with the learning rate of the models.

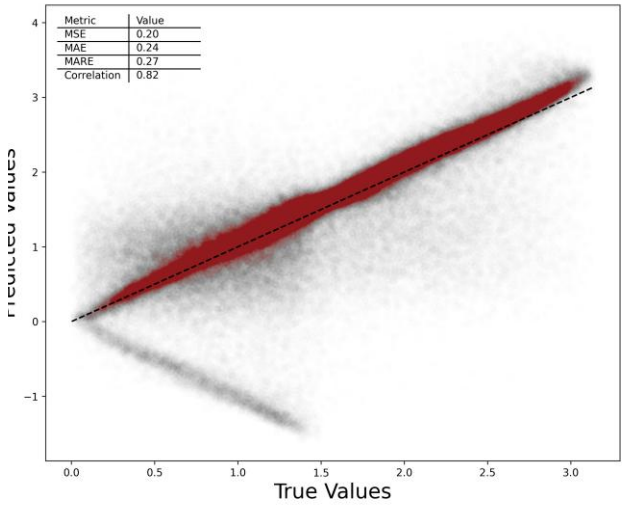
Many of the early models had a validation plot like the one on the right. This could potentially be due to a high learning rate; the models could not settle in a minimum but kept jumping out.
Unexpectedly the models consistently yielded better results for the checkpoint of the last epoch instead of the lowest loss epoch.



Appendix G – Transformer – Direction

Predictions for the last epoch of this model were seen in the powerpoint. The lowest epoch can be seen below.





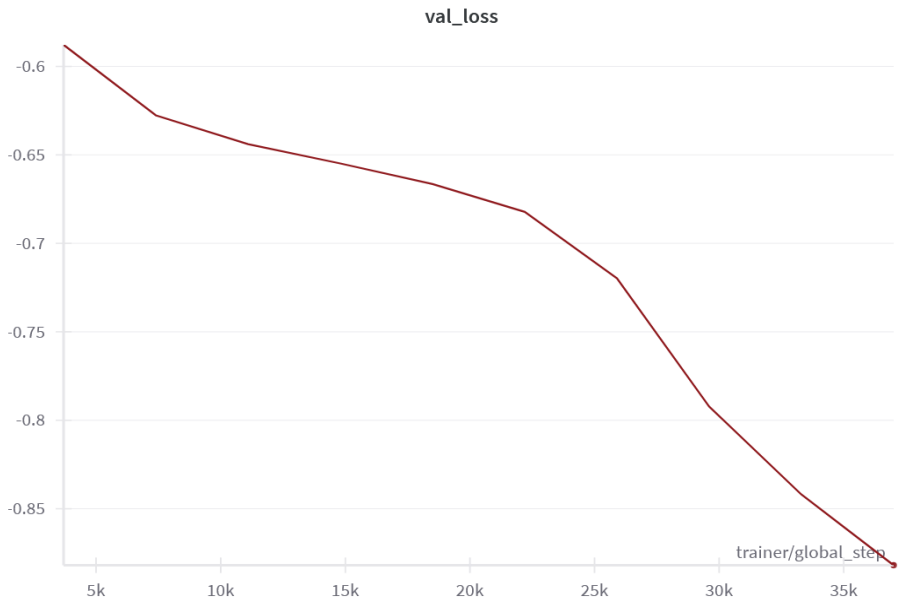
On the azimuth plot, the lowest blob is to be expected as the domain is periodic, and the predictions are shifted to match the physical interpretation. The middle blob, however, is because when the zenith angle has opposite sign, this corresponds to the azimuthal angle being rotated by π.

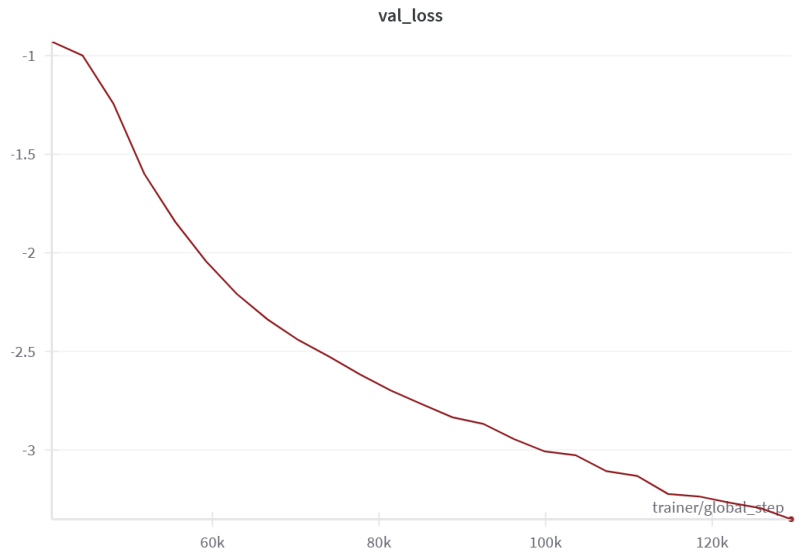
This effect tends to fade as the model continues running.

Appendix G – Transformer – Direction

Work on attempting lower learning rates was initiated as the project was coming to its end. The final, yet intermediate, results can be seen below.

This run is split across two graphs due to technical difficulties. Both can be seen below. The left one contains the first 10 epochs. The right the last 25.



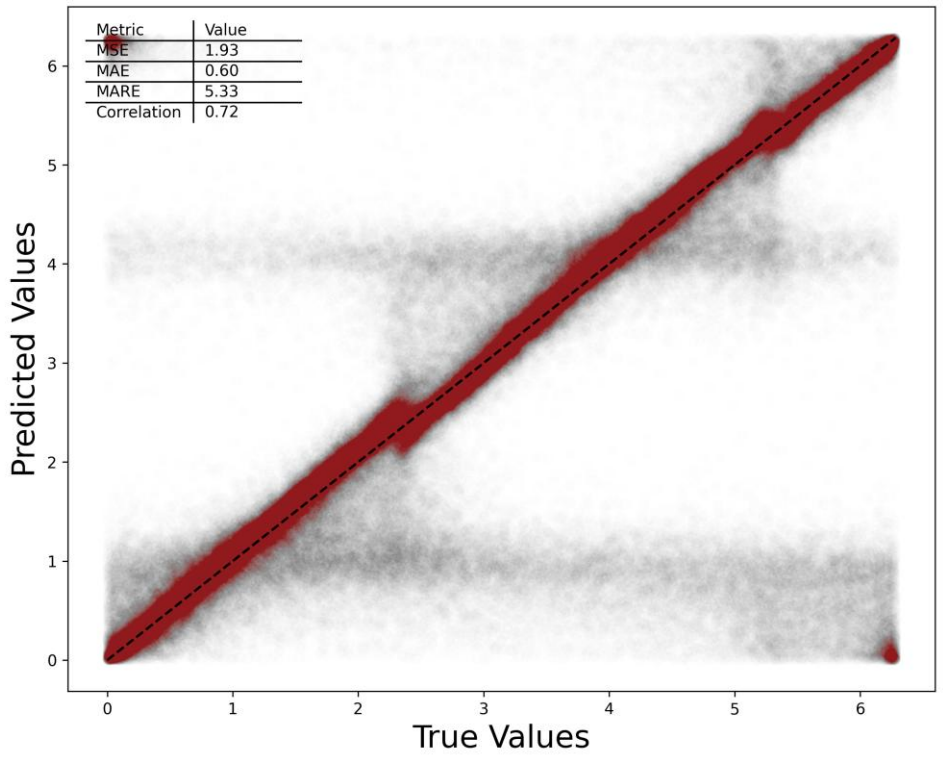


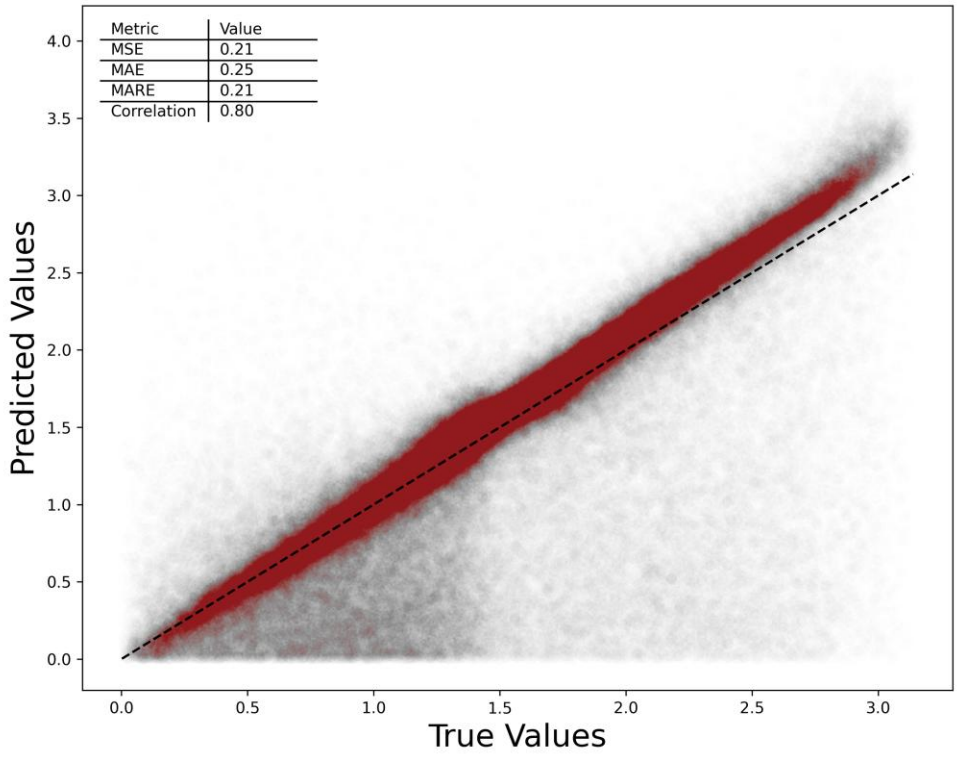
Parameter	Value
Batch Size	64
Num Layers	4
Num Heads	2
Seq dim	256
Learning rate	0.00001

It is clear we do not see the same jumping behaviour as before, but the loss does not go lower (at least for our training time) than it did for a higher learning rate. It does however go steadily down which is why there is merit to this method. Potentially one could implement a learning rate that slowly decreases with epochs, yet it has to stay rather small from the beginning. The predictions for this are on the next slide.

Appendix G – Transformer – Direction

Predictions for the low learning rate models.





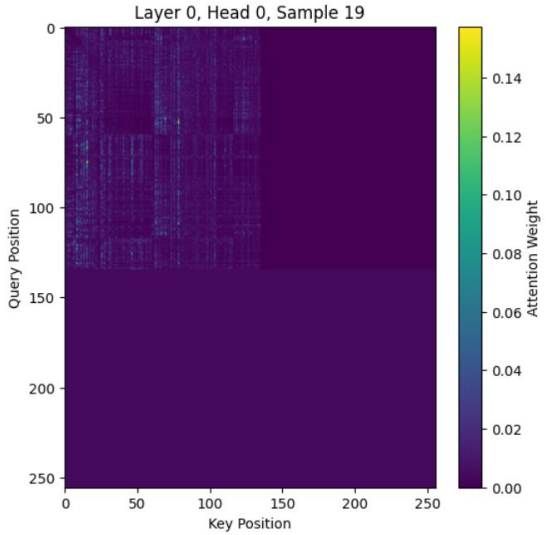
The models have similar performance as the ones with higher learning rate; thus, no considerable improvement is made. The results did need more patching up, that is they originally emerged in different clusters due to the periodicity of the domain.

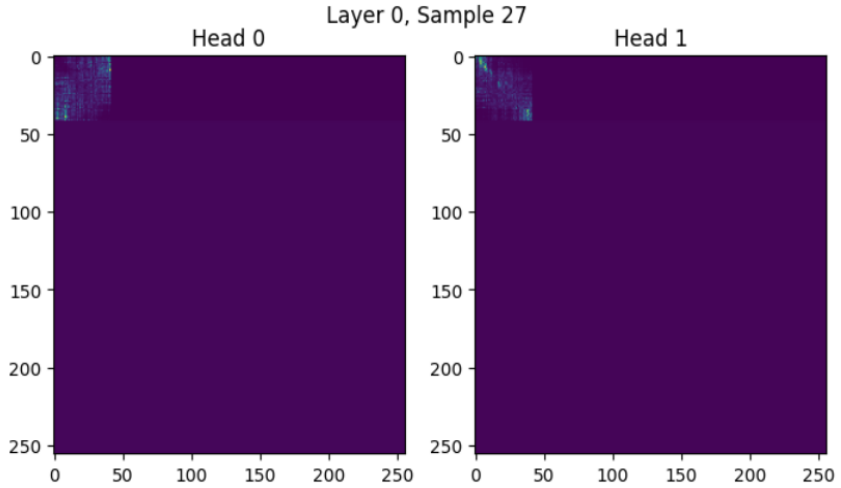
Playing with the learning rate is an interesting and obvious place for further work.

Note the interesting brick pattern on the left stemming from the transformations.

Appendix G – Transformer – Direction

- We tried also to visualize the attention weights in the model training of angular regression. The relevant parameters for this model can be seen in the table.





Parameter	Value
Attention Layers	4
Attention Heads	2

All three plots stem from a model which has two attention heads and had 4 attention layers. On the left is a single head. In it we see clear vertical lines which are indications that certain specific keys (and therefore DOMs) are important. Some of the corresponding querys also light up a lot indicating these have strong connections to each other and that the model should take these into account when analyzing the keys. Also, the pattern only appears in a square in the top left corner. This is due to the specific event, for which the attention is visualized, only had a certain number of DOMs triggering, thus leaving the rest of the sequence length to be padded. Also, this padding is seen on the right-hand plot for which the actual event was even smaller.

On the right, we see both attention heads visualized at the same time. We see both have different attention patterns which agrees with the fact that different heads analyze (catch) different patterns. Here, the left one tends to connect events far (*) from each other whereas the right-most one puts emphasis on the singular events themselves. Note also the squares on opposite sides (*) light up.

(*) We do not know which square in the attention weight plot corresponds to which DOM. Since the order is unknown to us, we cannot in principle relate this to the physical situation. Intuitively though, one could think that, say, pulses at opposite sides of the event would have increased value in determining the direction. One could then imagine a plot like the right-most one to represent this situation.