Reconstructing neutrino events in IceCube

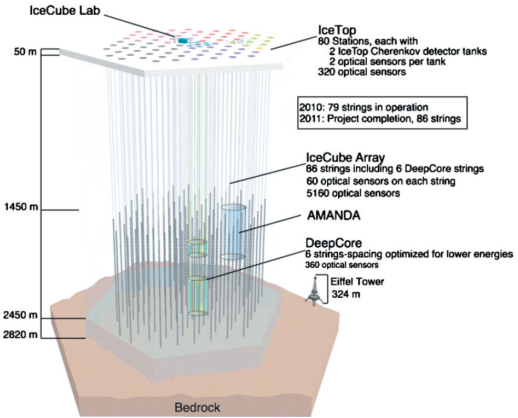
By use of a Graph neural network

Mikkel Lauritzen, Aske Rosted, Anna Tan,
Mikkel Schmidt and Moust Holmes[1]
The Niels Bohr Institute

[1] All group members have contributed evenly to the project

Outline

IceCube - South Pole Neutrino Observatory

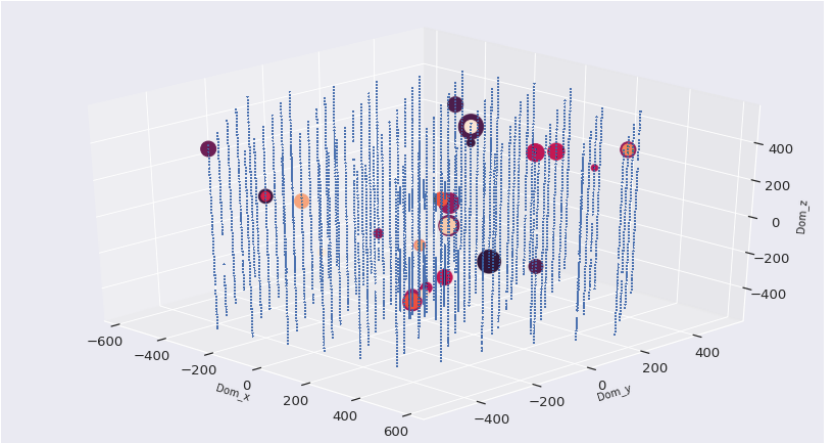


- 1,500 - 2,800 meters under the South Pole
- 5160 digital optical modules (DOMs) equipped with a photomultiplier tube (PMT) able to measure Cherenkov Radiation

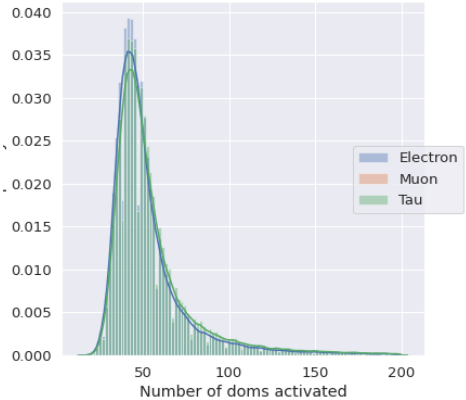
Data

- 100,000 Monte Carlo simulated events per dataset
- 1 dataset for electron neutrinos, 1 for muon neutrinos and 1 for tau neutrinos
- 5 Input variables
  - Charge, Q, of the signal
  - x, y, z coordinates of the DOM
  - time, t, where it received a signal
  - Given per DOM activation, associated with event_no, charge, time and position per activation. (length = 55,852,230)
- 8 Target variables
  - Given per event with true values for energy; x, y, z coordinates and time, t, for first particle interaction; and x, y, z coordinates for particle direction. (length = 100,000)

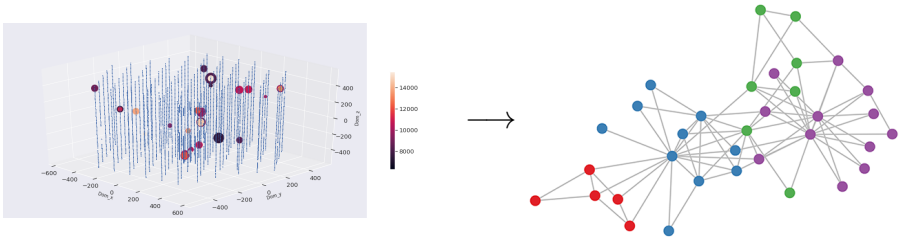
Data


Dom_x
−600
−400
−200
0
200
400
600
Dom_y
−400
−200
0
200
400
Dom_z
400
200
0
−200
−400
14000
12000
10000
8000

Data

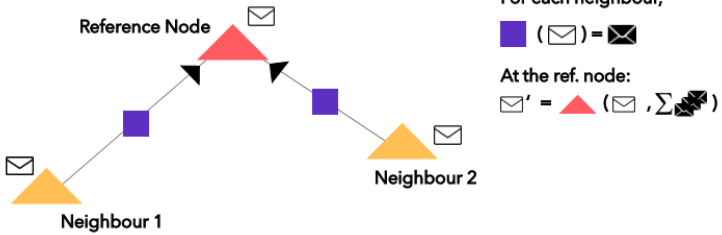

0.040
0.035
0.030
0.025
0.020
0.015
0.010
0.005
0.000
50
100
150
200
Number of doms activated
Electron
Muon
Tau

Transforming data to graphs



- Many different sized data structures can be model by graphs $G(V, E)$
- Each event was made into a graph
- Pulses $\longrightarrow$ nodes
- Many ways to connect the nodes

Graph convolution

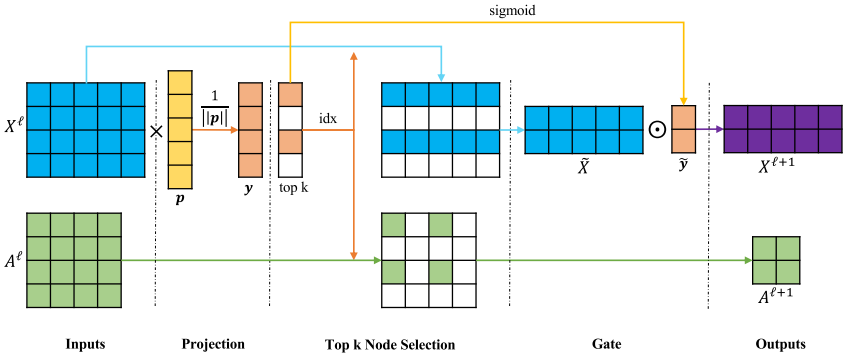


Neighborhood aggregation

$$x_i' = \sigma\left(x_i W_1 + \sum_{j \in \mathcal{N}(i)} x_j W_2\right)$$

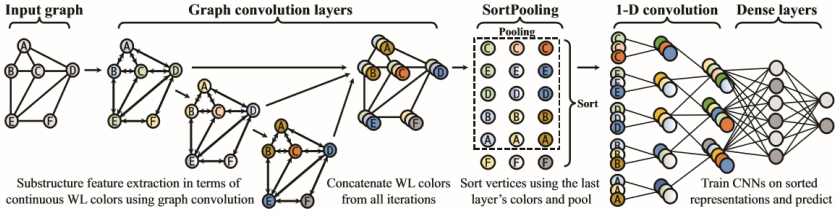
- Size invariant
- Doesn't change structure

Pooling layer



- Transforming the graphs to same size

Graph Convolutional Neural Network Architecture



- Can be combined with usual MLP, giving an overall feedforward network from which the backpropagation can be done to update weights

Our Network Architecture

- Network structure bases of a example of protein classification
- 1-4 Convolutional layer followed by an topK pooling
- 2-4 regular fully connected layers(Activation: ReLU)
  (+ batch normalization in the regression case)
  - **Regression output:** 1 output node no activation
    loss function: Mean square error (mse)
  - **Classification output:** 3 output nodes softmax activation
    loss function: negative log likelihood (nnl)

Implementation

- Google Colab
  - Free GPU for everybody!
  - Difficulties with runtime restart and collaboration
- GPU
- Weights and Biases (`WandB.com`)
  - Visualize training
  - Hyperparameter optimization and debugging

Regression

- Many problems few answers
  - The NN thinks the average is a nice guess
  - offset by a factor?
  - unsurprisingly different targets require different solutions.
- Some attempted solutions
  - No activation layer in final layer
  - targets natural log transformed
  - batch normalizing

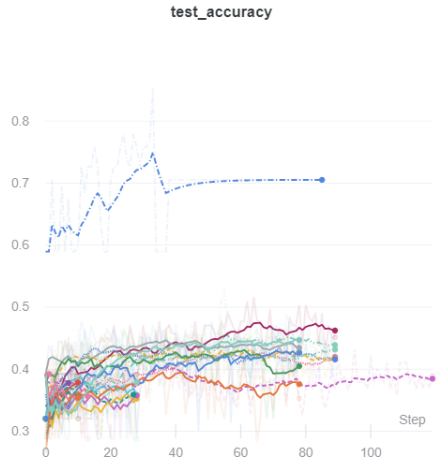
Regression Results

Energy RMSE: 0.295 with a mean range of 2.843. (RMSE $\sim 10\%$ of full range.) time RMSE: 152 with a mean range of 1454. (RMSE $\sim 9\%$ of full range.) From a validation set of 15% of the full set.

Classification

- Our solutions
  - Network Architecture from the protein classification got $\approx 40\%$ accuracy
  - Best model got 81% accuracy at its best
  - Hard to reproduce (only broken the 50% threshold twice)
  - Unstable: Small changes could result in failing to learn

Classification Results



Optimization

- Hard to optimize
  - Different ways of constructing graphs
    (fully connected, one way in time or N nearest neighbors)
  - Many kinds of convolution and pooling layers
  - Exponentially more hyper-parameters
  - Big graphs = slow training
  - Bayesian optimization with WandB (with problem)

Further discussion and outlook

- Graphs take up a lot of RAM, crashes at 50,000 events
- Fully connected, size scales like $O(n^2)$
- Network structure, number and type of layers
- Extremely volatile

Outlook

- Full reconstruction of event
- *Testing various Graph structures (forward edges, neighbor edges, etc.)*
- *More hyperparameter optimization*

Conclusion

- Graph Neural Networks are useful for different sized structures
- We were able to implement working Graph Neural Networks for regression and classification of simulated IceCube data
- Requires the use of a GPU, $\sim$40 times faster than a CPU
- We achieve accuracies of up to 80% for classification while regression is still in progress

Thank you for listening!

Conclusion

- Graph Neural Networks are useful for different sized structures
- We were able to implement working Graph Neural Networks for regression and classification of simulated IceCube data
- Requires the use of a GPU, $\sim$40 times faster than a CPU
- We achieve accuracies of up to 80% for classification while regression is still in progress

Thank you for listening!

Thank you for listening

References

[1] https://arxiv.org/pdf/1905.05178.pdf

[2] https://pytorch-geometric.readthedocs.io/en/latest/

[3] https://arxiv.org/abs/1810.02244

[4] https://tkipf.github.io/graph-convolutional-networks/

[5] https://medium.com/crim/deep-learning-applied-to-graphs-586ce63bb28e

[6] https://github.com/rusty1s/pytorch_geometric/blob/master/examples/geniepath.py

Appendix A - Load data

```
1  url = 'https://sid.erda.dk/share_redirect/bSoeotINL9/140000_00.db
       '
2  r = requests.get(url, allow_redirects=True)
3  open('140000_00.db', 'wb').write(r.content)
4
5  db_file = '/content/140000_00.db'
6  with sqlite3.connect(db_file) as con:
7      query = 'select * from sequential'
8      sequential_4 = pd.read_sql(query, con)
9      query = 'select * from scalar'
10     scalar_4 = pd.read_sql(query, con)
```

Appendix B - Sequential

	index	event_no	pulse_no	dom_x	dom_y	dom_z	dom_charge	dom_time	SplitInIcePulses	SRTInIcePulses
0	0	9219761	0	248.149994	-111.870003	-472.149994	0.175	5727	1	0
1	1	9219761	1	361.000000	-422.829987	414.410004	0.575	5840	1	0
2	2	9219761	2	41.599998	35.490002	-234.990005	1.475	6478	1	0
3	3	9219761	3	41.599998	35.490002	-234.990005	0.525	6509	1	0
4	4	9219761	4	-10.970000	6.720000	140.360001	0.375	6911	1	0
...	...	...	...	...	...	...	...	...	...	...
95	95	9094606	32	-111.510002	159.979996	-10.090000	0.975	12234	1	0
96	96	9094606	33	505.269989	257.880005	438.149994	0.625	13168	1	0
97	97	9094606	34	-392.380005	334.239990	-450.369995	0.575	13796	1	0
98	98	9094606	35	-166.399994	-287.790009	140.220001	1.025	13836	1	0
99	99	9094606	36	-189.979996	257.420013	-230.539993	1.325	14653	1	0

100 rows × 10 columns

Figure: Every pulse_no, position in x/y/z, dom_charge and dom_time

Appendix C - Scalar

	index	event_no	true_primary_energy	true_primary_time	true_primary_position_x	true_primary_position_y	true_primary_position_z
0	0	9046728	1.039659	9724.702595	84.007514	54.627137	-224.842781
1	1	9046743	2.127483	9603.571564	-42.488789	-183.541941	-485.287729
2	2	9046747	1.205234	9790.872049	16.707779	-79.376813	-398.508239
3	3	9046748	0.820345	9878.454570	114.190375	-64.239228	-310.199591
4	4	9046752	1.298331	9842.994170	31.476315	34.067668	-423.010471

Figure: Target/true values for every event.

Appendix D - Sort events

Listing 1: Sorting after events

```
1 # Sorting by event number
2 features = sequential_4.sort_values(by=['event_no'])
3 target = scalar_4.sort_values(by=['event_no'])
4
5 unique_events = np.unique(features['event_no'].values) # list of
      unique events
6 unique_sample = unique_events[0:(128*24)] # choose subsample of
      unique events
```

We sort in order to make sure that we get our graphs and target values in the same order after events.

Appendix E - Making the graphs

```
1  n=subsample.shape[0]#number of events in the subsmple
2  events=np.array(subsample['event_no']) #event numbers
3  dataset=[] #empty list for the graphs
4  for i in tqdm(range(n)):
5    pulses = seqcat[seqcat['event_no']==events[i]]
6    if 30 < len(pulses) < 100: #excluding events with many or few
7      ntype = torch.tensor([int(subsample[subsample['event_no']==
           events[i]]['type'])],dtype=torch.long) #neutrino type
8      perm = list((permutations(range(pulses.shape[0]),2)))
9      edge_index = torch.tensor( [list(perm[j]) for j in range(len(
           perm))], dtype=torch.long) #fully connected
10     x = torch.tensor(np.array(pulses.drop(columns='event_no')),
           dtype=torch.float) #features
11     dataset.append(Data(x=x, edge_index=edge_index.t().contiguous
           (), y=ntype)) #appending graph
```

Appendix F - Network

```
1   # Split dataset
2   dataset = edge_index_list
3   print("dataset type:",type(dataset))
4
5   train_dataset = dataset[:len(dataset)//10*7] # 70%
6   valid_dataset = dataset[len(dataset)//10*7:len(dataset)
        //10*7+(len(dataset)-len(dataset)//10*7)//2] # 15%
7   test_dataset = dataset[len(dataset)//10*7+(len(dataset)-len(
        dataset)//10*7)//2:] # 15%
8
9   batch_size= 2**(6)
10  train_loader = DataLoader(train_dataset, batch_size)
11  val_loader = DataLoader(valid_dataset, batch_size)
12  test_loader = DataLoader(test_dataset, batch_size)
```

Appendix G - Regression code

```
1  scaling = 2
2  class Net(torch.nn.Module):
3      def __init__(self):
4          super(Net, self).__init__()
5
6          self.conv1 = GraphConv(dataset[0].num_features, 32*
               scaling)
7          self.pool1 = TopKPooling(32*scaling, ratio=0.8)
8          self.conv2 = GraphConv(32*scaling, 32*scaling)
9          self.pool2 = TopKPooling(32*scaling, ratio=0.8)
10         # self.conv3 = GraphConv(32*scaling, 32*2)
11         # self.pool3 = TopKPooling(32*scaling, ratio=0.8)
12
13         self.lin1 = torch.nn.Linear(64*scaling, 32*scaling)
14         self.lin2 = torch.nn.Linear(32*scaling, 16*scaling)
15         self.lin3 = torch.nn.Linear(16*scaling, 1)
```

```
1       def forward(self, data):
2           x, edge_index, batch = data.x, data.edge_index, data.
                batch
3
4           x = F.relu(self.conv1(x, edge_index))
5           x, edge_index, _, batch, _, _ = self.pool1(x, edge_index,
                None, batch)
6           x1 = torch.cat([gmp(x, batch), gap(x, batch)], dim=1)
7
8           x = F.relu(self.conv2(x, edge_index))
9           x, edge_index, _, batch, _, _ = self.pool2(x, edge_index,
                None, batch)
10          x2 = torch.cat([gmp(x, batch), gap(x, batch)], dim=1)
```

```
        # x = F.relu(self.conv3(x, edge_index))
        # x, edge_index, _, batch, _, _ = self.pool3(x,
            edge_index, None, batch)
        # x3 = torch.cat([gmp(x, batch), gap(x, batch)], dim=1)

        x = x1 + x2 # + x3

        x = F.relu(self.lin1(x))
        x = F.dropout(x, p=0.5, training=self.training)
        x = F.relu(self.lin2(x))
        x = self.lin3(x)

        return x
```

```
1 device = torch.device('cuda' if torch.cuda.is_available() else '
      cpu')
2 model = Net().to(device)
3 optimizer = torch.optim.Adam(model.parameters(), lr=0.005)
```

```python
def train(epoch):
    model.train()
    loss_all = 0
    for data in train_loader:
        data = data.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.mse_loss(output.flatten(), data.y)
        loss.backward()
        loss_all += loss.item()
        optimizer.step()
    return loss_all / len(train_loader)
```

```
1  def test(loader):
2      model.eval()
3      mse_all = 0
4      for data in loader:
5          data = data.to(device)
6          pred = model(data)
7          mse = F.mse_loss(pred.flatten(), data.y)
8          mse_all += mse.item()
9          if epoch > 98: print(pred.flatten(),data.y)
10     return mse_all / len(loader)
```

```
1  for epoch in range(1, 100):
2      loss = np.sqrt(train(epoch))
3      train_RMSE = np.sqrt(test(train_loader)) #train_NMAE
4      test_RMSE = np.sqrt(test(test_loader)) #test_NMAE
5
6      print('Epoch: {:03d}, Loss: {:.5f}, Train RMSE: {:.5f}, Test
           RMSE: {:.5f},'.
7              format(epoch, loss, train_RMSE, test_RMSE))
8      metrics = {'loss rmse':loss, 'Train rmse':train_RMSE, 'Test
           rmse':test_RMSE}
```

Appendix H - Regression code

Code:

Time : https://colab.research.google.com/drive/1FeMWLdeunf11_D3llkWzzE_52oPHRs4T?usp=sharing

energy : https://colab.research.google.com/drive/1wdPuDf7OtRvORbyPGZzcBD_amJ9Jc9yv?usp=sharing

Weights and biases

Time : https://app.wandb.ai/maskel/time-regression-icecube?workspace=user-maskel

energy : https://app.wandb.ai/maskel/regression-graph-icecube?workspace=user-maskel

Appendix I - Classification code

Classification code : `https://colab.research.google.com/drive/1zUtHlK6LTyqTNoCzyHJ9oNQyIKJKNNM9?usp=sharing`

Classification WandB: `https://app.wandb.ai/moust/IceCubeClassification?workspace=user-moust`

Appendix J - Weights and biases

Figure: W&B