ML for vertex and mass reconstruction for DarkLight@ARIEL

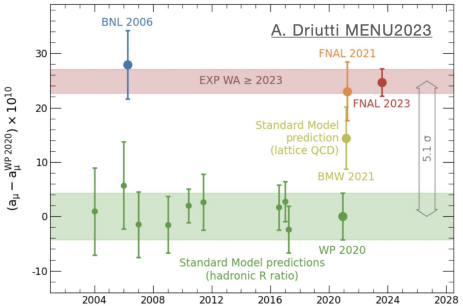
Sid Gupte, Win Lin

Stony Brook University

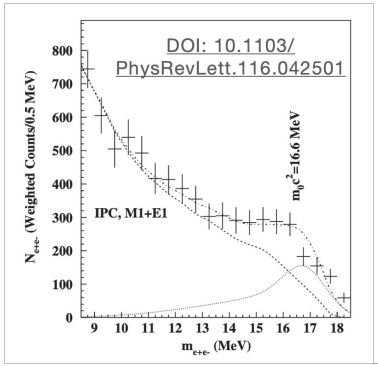
CFNS MLLM meeting 02/07/2025

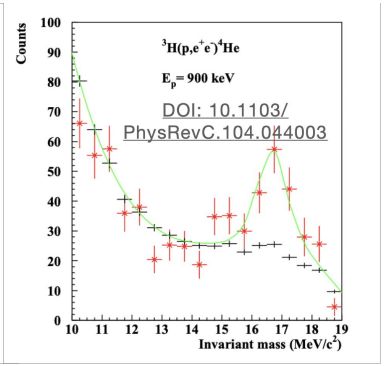
Introduction

- The DarkLight Experiment

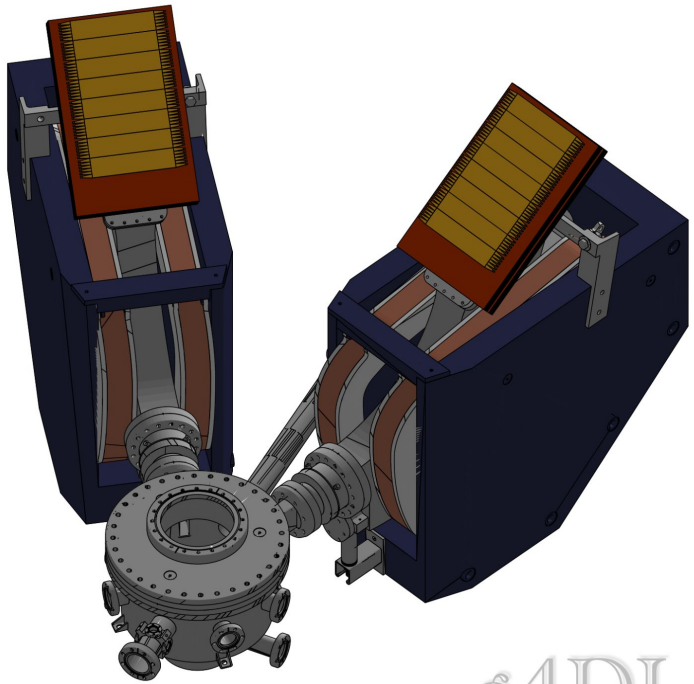


Possible "dark photon"?





DARKLIGHT @ ARIEL

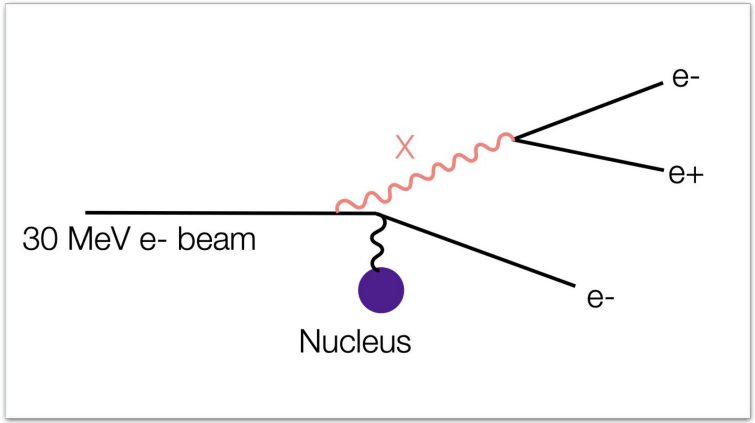


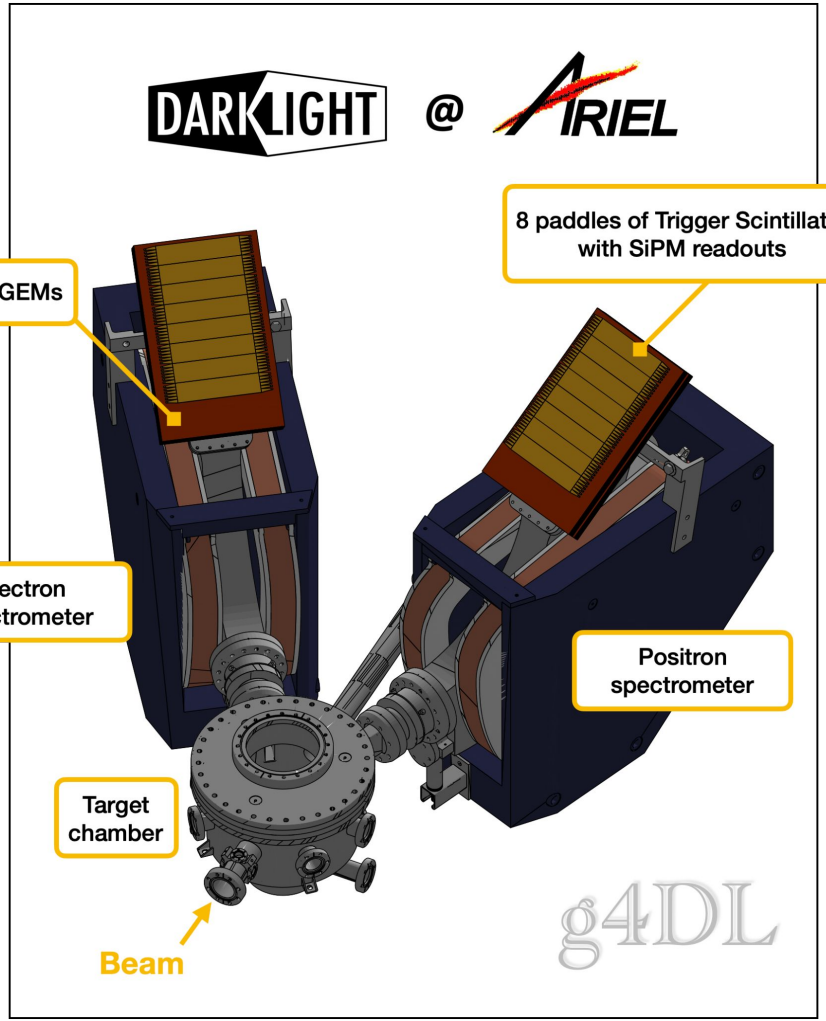
g4DL

Introduction

- The DarkLight Experiment

Physics process of potential “dark photon” signal

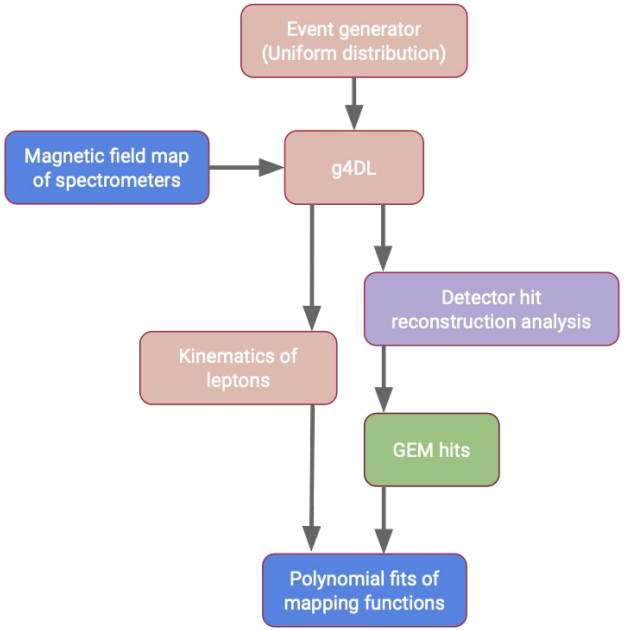




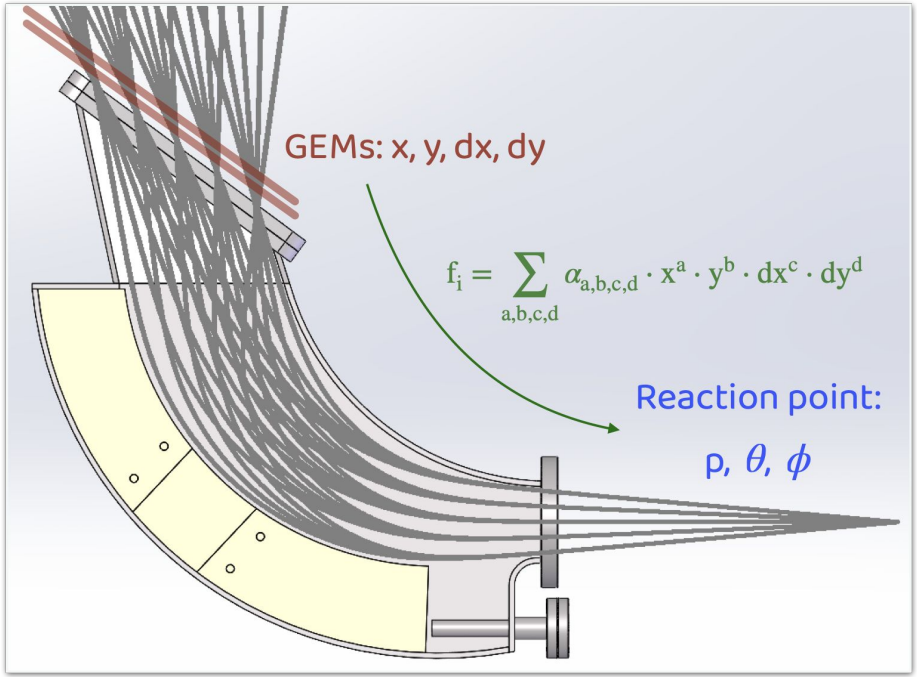
Introduction

- Mass Reconstruction

Calibration Steps



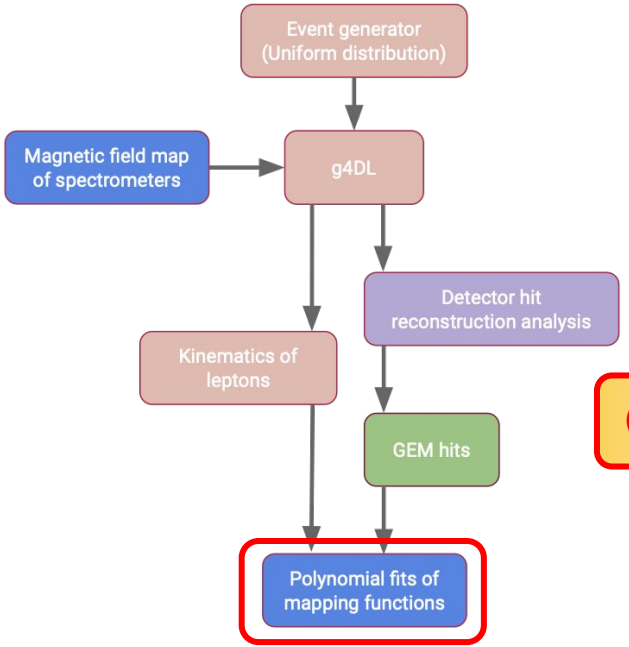
Schematic of spectrometer and particle trajectories



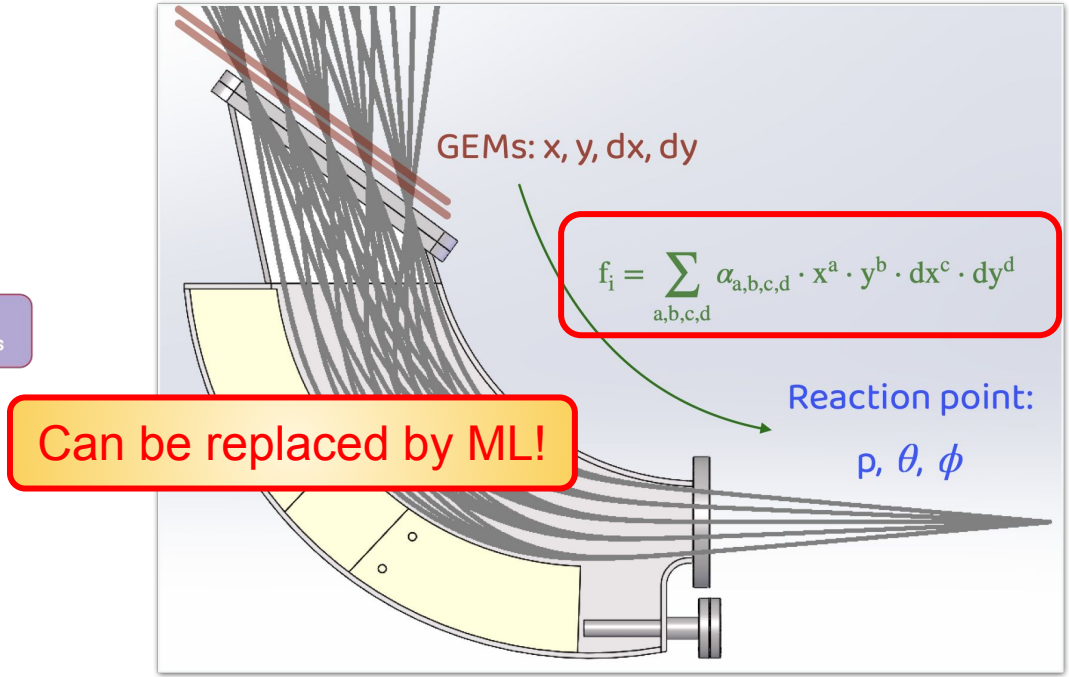
Introduction

- Mass Reconstruction

Calibration Steps



Schematic of spectrometer and particle trajectories

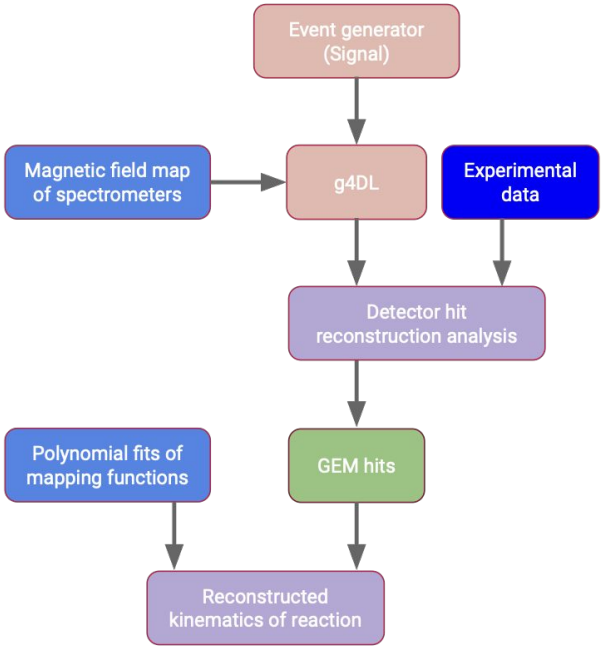


Can be replaced by ML!

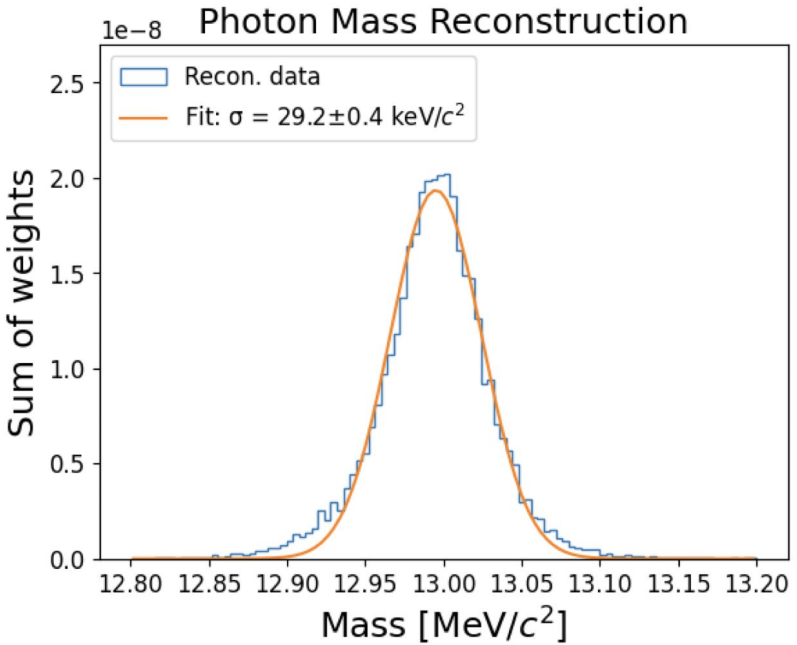
Introduction

- Mass Reconstruction

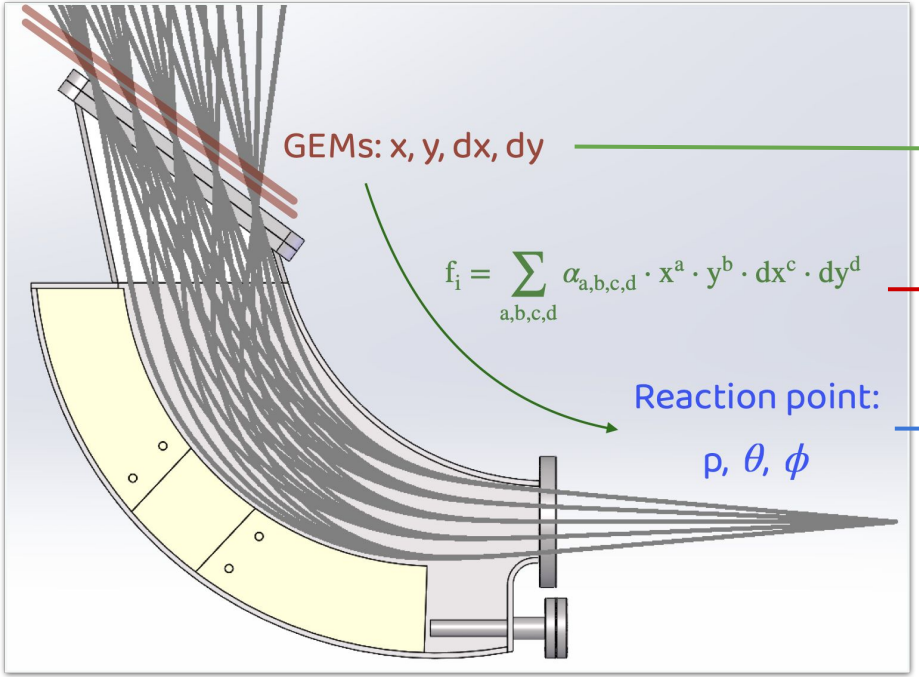
Analysis Steps



Reconstruction for simulated source using current fitting method



Machine learning!



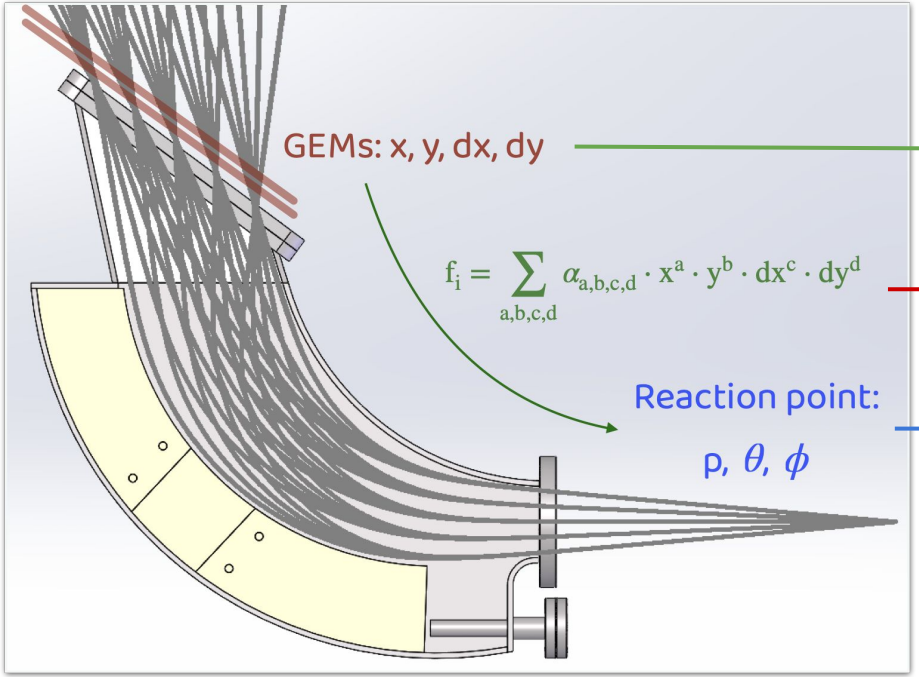
Inputs

ML model

Outputs

- Neural network
- Random forest
- and others ..?

Machine learning!



Inputs

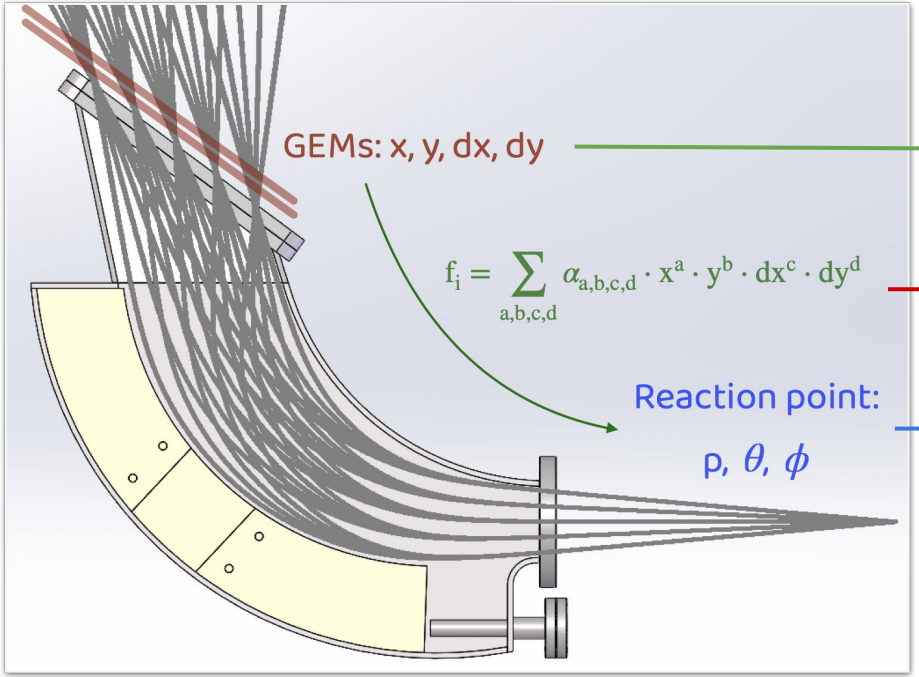
ML model

Outputs

- Neural network
- Random forest
- and others ..?

All training, verification and testing data are generated by simulation

Machine learning!



Inputs

ML model

Outputs

- Neural network
- Random forest
- and others ..?

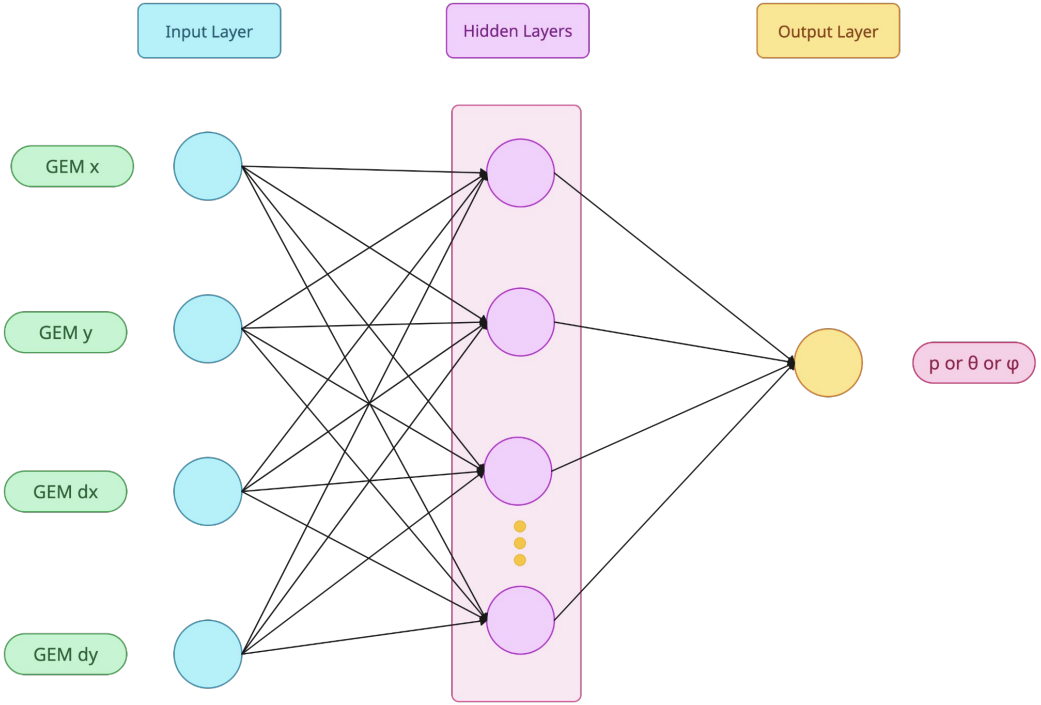
All training was done on Google CoLab.

- SBU does not give CoLab access :(
- SeaWulf queue is long …

Neural network:

- simple testing model 1

- linear network
- individual model for each output
- Use optimizer to find layers and neurons
- Activation function: Leaky ReLu



Choice of Activation Function

- During backpropagation, we perform gradient descent (or some variation of it) which looks as following:

$$W^{(l)} = W^{(l)} - \eta \cdot \frac{\partial L}{\partial W^{(l)}}$$

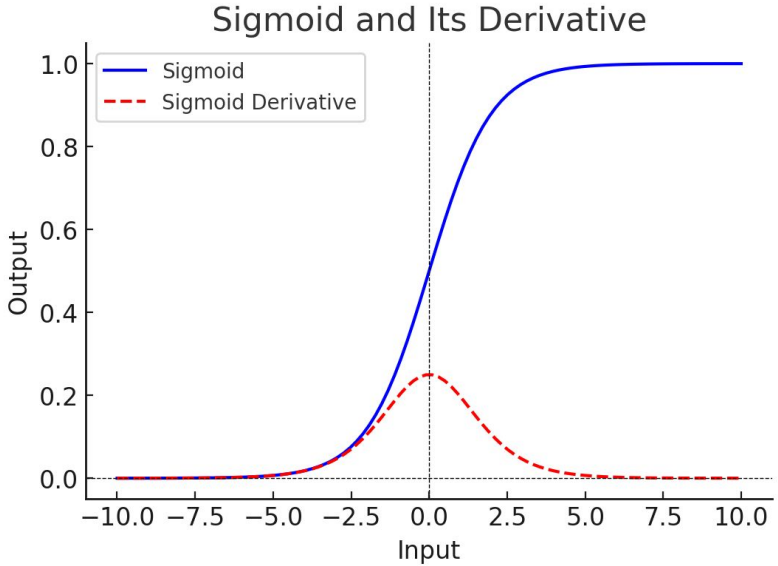
- The gradient wrt weights depends on derivative of the activation function:

$$\frac{\partial L}{\partial W^{(l)}} = \frac{\partial L}{\partial a^{(l)}} \cdot \boxed{\frac{\partial a^{(l)}}{\partial z^{(l)}}} \cdot \frac{\partial z^{(l)}}{\partial W^{(l)}}$$

Sigmoid Function

$$a(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$

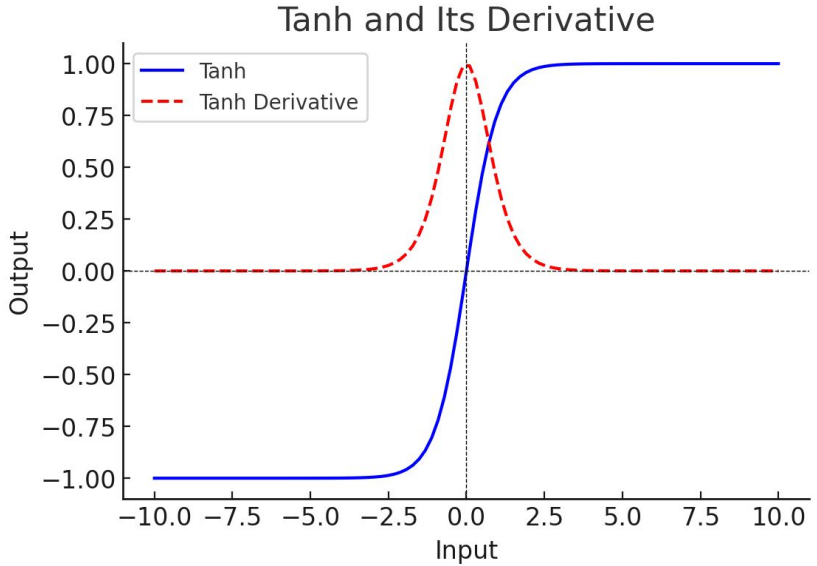
$$\implies \frac{\partial a}{\partial z} = \sigma(z)\big(1 - \sigma(z)\big)$$



Hyperbolic Tangent Function

$$a(z) = \mathrm{th}(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

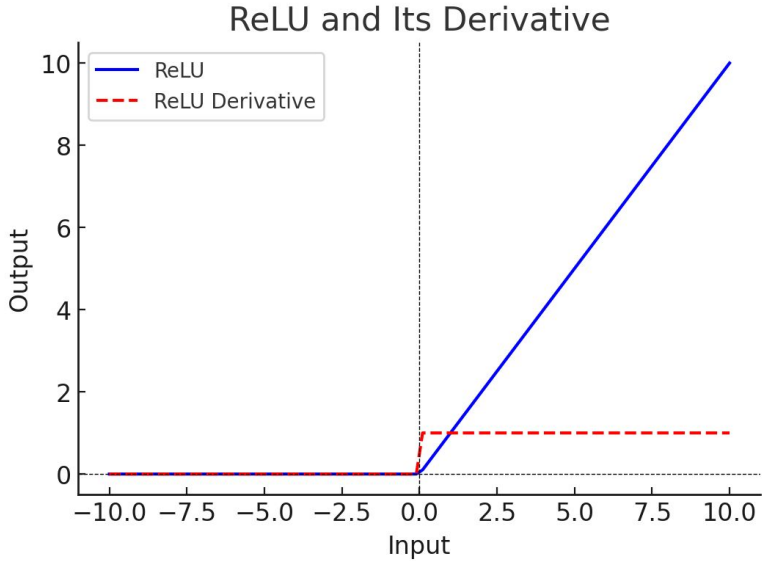
$$\implies \frac{\partial a}{\partial z} = 1 - \mathrm{th}^2(z)$$



ReLU (Rectified Linear Unit) Function

$$\mathrm{ReLU}(x) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{if } x \leq 0 \end{cases}$$

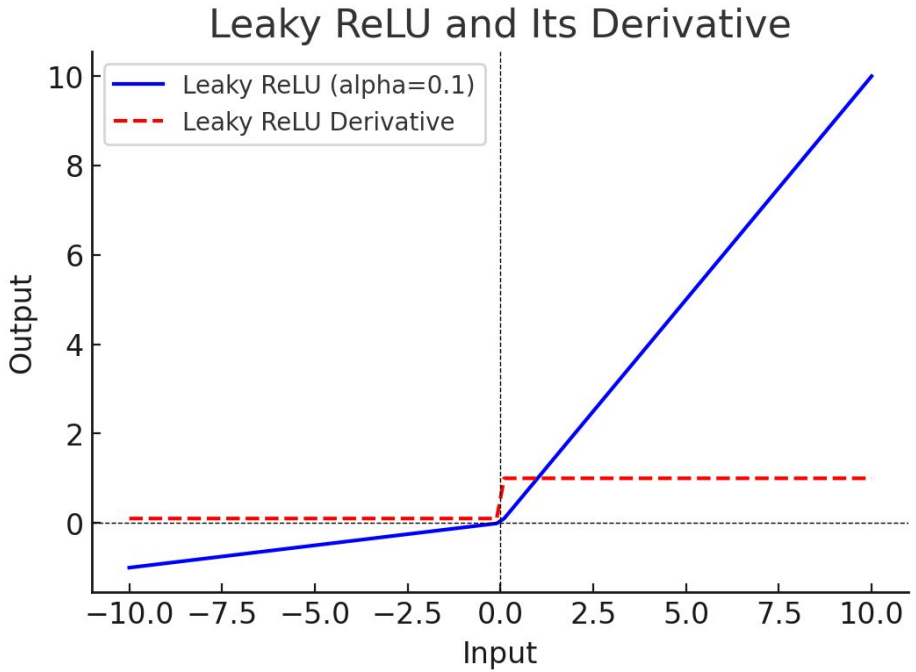
- Output Range: 0 to infinite
- Advantage: fast; good for pattern recognition
- Disadvantage: dying ReLU



https://www.geeksforgeeks.org/activation-functions-neural-networks/

Leaky ReLU Function

$$\text{LeakyReLU}(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha x, & \text{if } x \leq 0 \end{cases}$$



Setting up for using GPU

```
import torch
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Data

```
import pandas as pd
df_e = pd.DataFrame(electron_data, columns=['P', 'ip', 'oop', 'X', 'Y', 'dX', 'dY'])
df_e.head()
```

	P	ip	oop	X	Y	dX	dY
0	11.2456	-0.811484	-3.34895	-17.6243	-17.3677	-23.6235	0.895071
1	10.5401	0.617530	-3.46603	-54.8449	11.6648	-21.3299	-0.561832
2	12.8834	0.496969	1.99661	69.9853	15.2980	-41.2681	-1.252310
3	12.5593	1.272000	-1.75516	50.7739	32.0927	-30.5484	-1.969000
4	12.0571	1.098930	2.23481	24.3938	32.0202	-38.7435	-2.639890

Pandas DataFrames

Transforming the Data for using in the Neural Networks

```
X = df_e.drop(['P', 'ip', 'oop'], axis=1)
Y = df_e[['P']]

X = torch.tensor(X.values, dtype=torch.float32, device=device)
Y = torch.tensor(Y.values, dtype=torch.float32, device=device)

from sklearn.model_selection import train_test_split
X_train, X_val, Y_train, Y_val = train_test_split(X, Y, test_size=0.2, random_state=42, shuffle=False)
X_train = X_train.to(device)
Y_train = Y_train.to(device)
X_val = X_val.to(device)
Y_val = Y_val.to(device)
```

Hyperparameter Optimization

```python
from torch import nn
class MomentumNetwork(nn.Module):
    def __init__(self, num_layers, hidden_sizes, lr):
        super().__init__()
        layers = []

        layers.append(nn.Linear(4, hidden_sizes[0]))
        layers.append(nn.ReLU())

        for i in range(num_layers - 1):
            layers.append(nn.Linear(hidden_sizes[i], hidden_sizes[i + 1]))
            layers.append(nn.ReLU())

        layers.append(nn.Linear(hidden_sizes[-1], 1))
        self.linear_relu_stack = nn.Sequential(*layers)
        self.lr = lr

    def forward(self, x):
        return self.linear_relu_stack(x)
```

Load Neural Network Module from PyTorch

Base structure of Network

```
num_layers  = [2, 10]
num_neurons = [4, 100]
```

Parameter Space for Hyperparameter Optimization

```
def objective(trial):
  num_layers = trial.suggest_int("num_layers", num_layers[0], num_layers[1])
  hidden_sizes = [trial.suggest_int(f"hidden_size_{i}", num_neurons[0], num_neurons[1], step=16) for i in range(num_layers)]
  lr = trial.suggest_loguniform("lr", 1e-5, 1e-2)

  model = MomentumNetwork(num_layers, hidden_sizes, lr).to(device)
  criterion = nn.MSELoss()
  optimizer = optim.Adam(model.parameters(), lr=lr)

  epochs = 100
  for epoch in range(epochs):
      outputs = model(X_train)
      loss = criterion(outputs, Y_train)
      optimizer.zero_grad()
      loss.backward()
      optimizer.step()

  with torch.no_grad():
      val_outputs = model(X_val)
      val_loss = criterion(val_outputs, Y_val)

  return val_loss.item()
```

Setup the objective function for optimizer

Using OPTUNA for Optimization

```
!pip install optuna
import optuna
```

Run OPTUNA

```
num_trials = 30

study = optuna.create_study(direction="minimize")
study.optimize(objective, n_trials=num_trials)
print("Best hyperparameters:", study.best_params)
```

Instantiating the Neural Network

```
num_layers = study.best_params['num_layers']
hidden_sizes = list(study.best_params.values())[1:-1]
lr = list(study.best_params.values())[-1]
```

Example setup for network:

```python
class OptimizedMomentumNetwork(nn.Module):
    def __init__(self, input_size=4, hidden_sizes, dropout_prob=0.0):
        super(OptimizedMomentumNetwork, self).__init__()

        # hidden layer #1
        self.fc1 = nn.Linear(input_size, hidden_sizes[0])
        self.act1 = nn.LeakyReLU()
        self.drop1 = nn.Dropout(dropout_prob)

        # hidden layer #2
        self.fc2 = nn.Linear(hidden_sizes[0], hidden_sizes[1])
        self.act2 = nn.LeakyReLU()
        self.drop2 = nn.Dropout(dropout_prob)

        # Output layer
        self.fc_out = nn.Linear(hidden_sizes[1], 1)

    def forward(self, x):
        x = self.fc1(x)
        x = self.act1(x)
        x = self.drop1(x)

        x = self.fc2(x)
        x = self.act2(x)
        x = self.drop2(x)

        x = self.fc_out(x)
        return x
```

Setup linear NN with x layer ...

Define activation function

Network architecture

```
OptimizedMomentumNetwork(
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=4, out_features=20, bias=True)
    (1): LeakyReLU(negative_slope=0.01)
    (2): Linear(in_features=20, out_features=20, bias=True)
    (3): LeakyReLU(negative_slope=0.01)
    (4): Linear(in_features=20, out_features=68, bias=True)
    (5): LeakyReLU(negative_slope=0.01)
    (6): Linear(in_features=68, out_features=84, bias=True)
    (7): LeakyReLU(negative_slope=0.01)
    (8): Linear(in_features=84, out_features=84, bias=True)
    (9): LeakyReLU(negative_slope=0.01)
    (10): Linear(in_features=84, out_features=20, bias=True)
    (11): LeakyReLU(negative_slope=0.01)
    (12): Linear(in_features=20, out_features=100, bias=True)
    (13): LeakyReLU(negative_slope=0.01)
    (14): Linear(in_features=100, out_features=1, bias=True)
  )
)
```

Training the Network

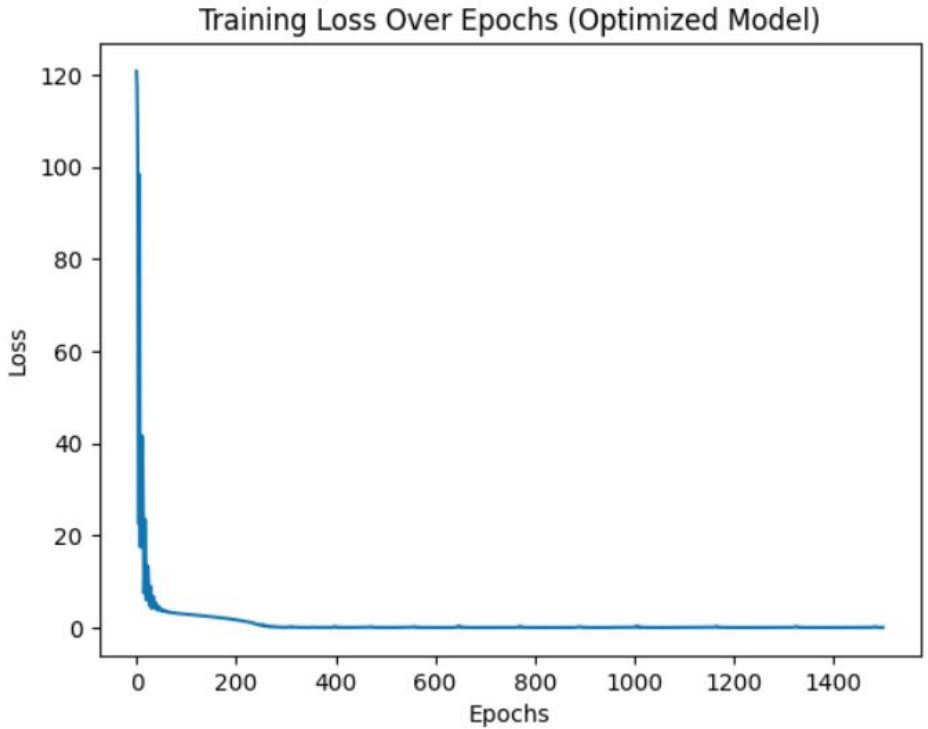
```
model = OptimizedMomentumNetwork().to(device)
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=lr)

epochs = 1500
losses = []

for epoch in range(epochs):
    outputs = model(X_train)

    loss = criterion(outputs, Y_train)
    losses.append(loss.item())

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```


Training Loss Over Epochs (Optimized Model)
Loss
120
100
80
60
40
20
0
0
200
400
600
800
1000
1200
1400
Epochs

Save the Model!

```
torch.save(model.state_dict(), 'model_1.pth')
```

Load the Model

```
loaded_model = OptimizedMomentumNetwork().to(device)
loaded_model.load_state_dict(torch.load('model_1.pth'))
loaded_model.eval()
```

Running the Trained Model

```
with torch.no_grad():
    Y_pred = model(X_val)
    loss = criterion(Y_pred, Y_val)
```

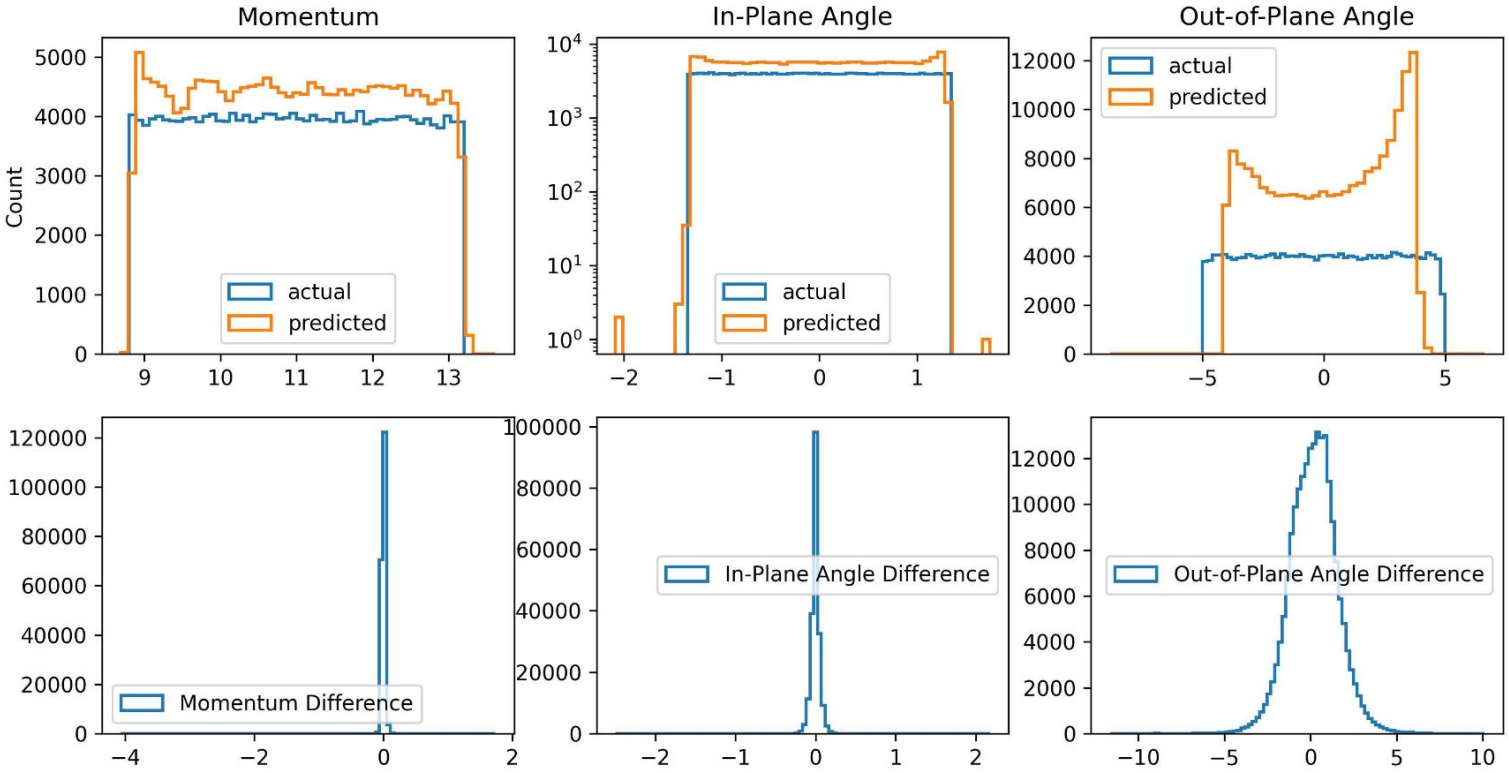
Running the Loaded Model

```
with torch.no_grad():
    Y_pred = loaded_model(X_val)
    loss = criterion(Y_pred, Y_val)
```

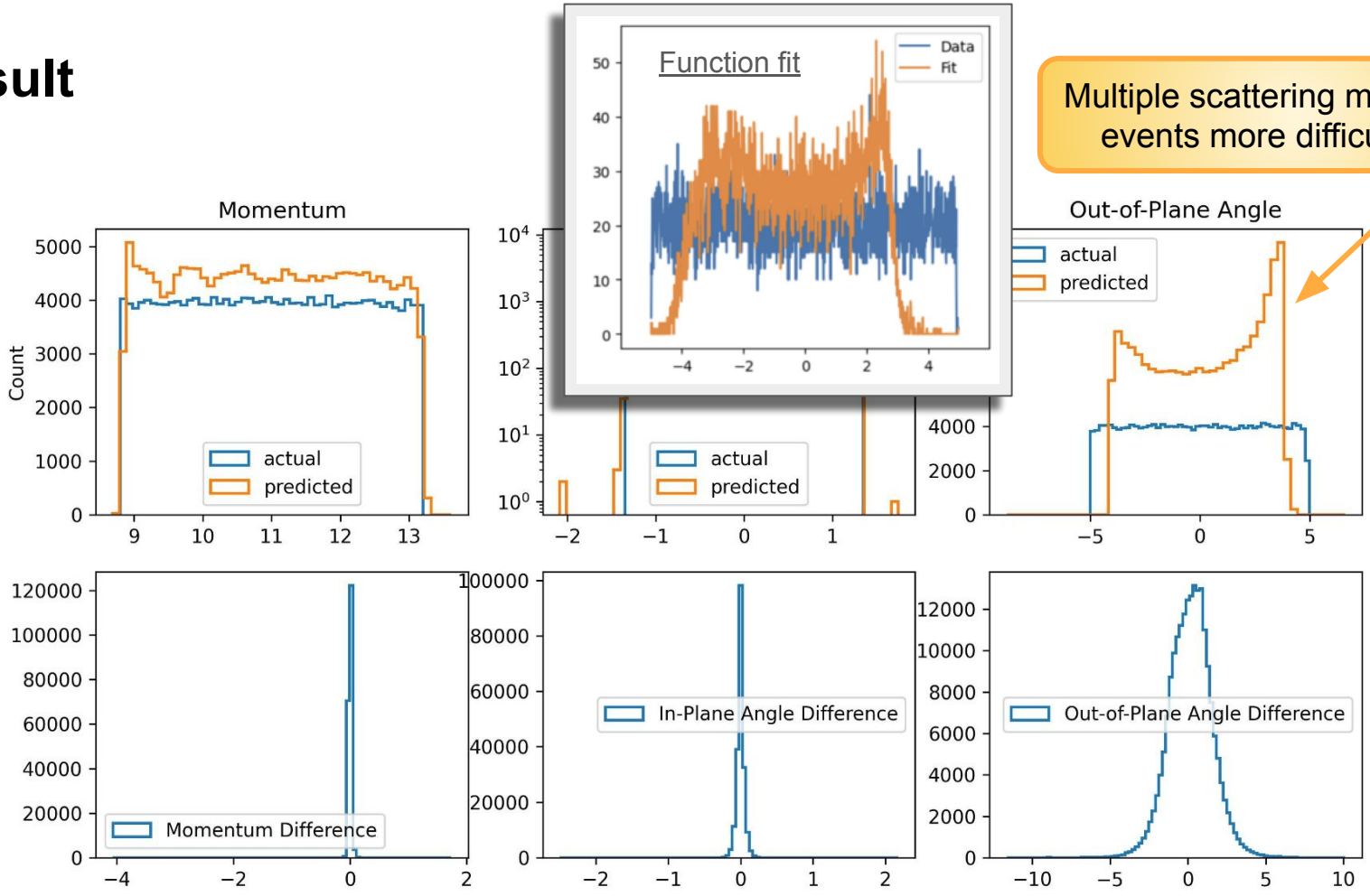
Prepare Data for Results

```
X_train_p = X_train.to('cpu')
Y_train_p = Y_train.to('cpu')
X_val_p = X_val.to('cpu')
Y_val_p = Y_val.to('cpu')
Y_pred_p = Y_pred.to('cpu')
diff_p = [Y_pred_p[i].item() - Y_val_p[i].item() for i in range(len(Y_val_p))]
```

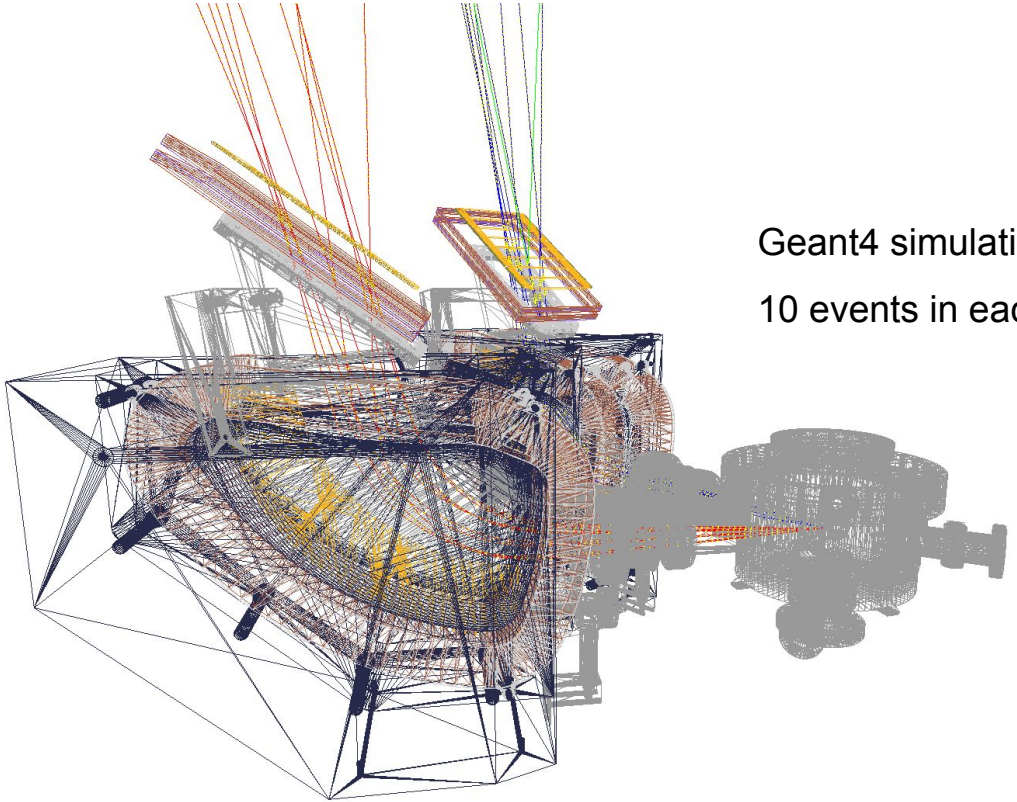
Result


Momentum
In-Plane Angle
Out-of-Plane Angle
Count
actual
predicted
Momentum Difference
In-Plane Angle Difference
Out-of-Plane Angle Difference

Result


Function fit
Data
Fit
Multiple scattering makes edge events more difficult to fit?
Momentum
Count
actual
predicted
Out-of-Plane Angle
Momentum Difference
In-Plane Angle Difference
Out-of-Plane Angle Difference

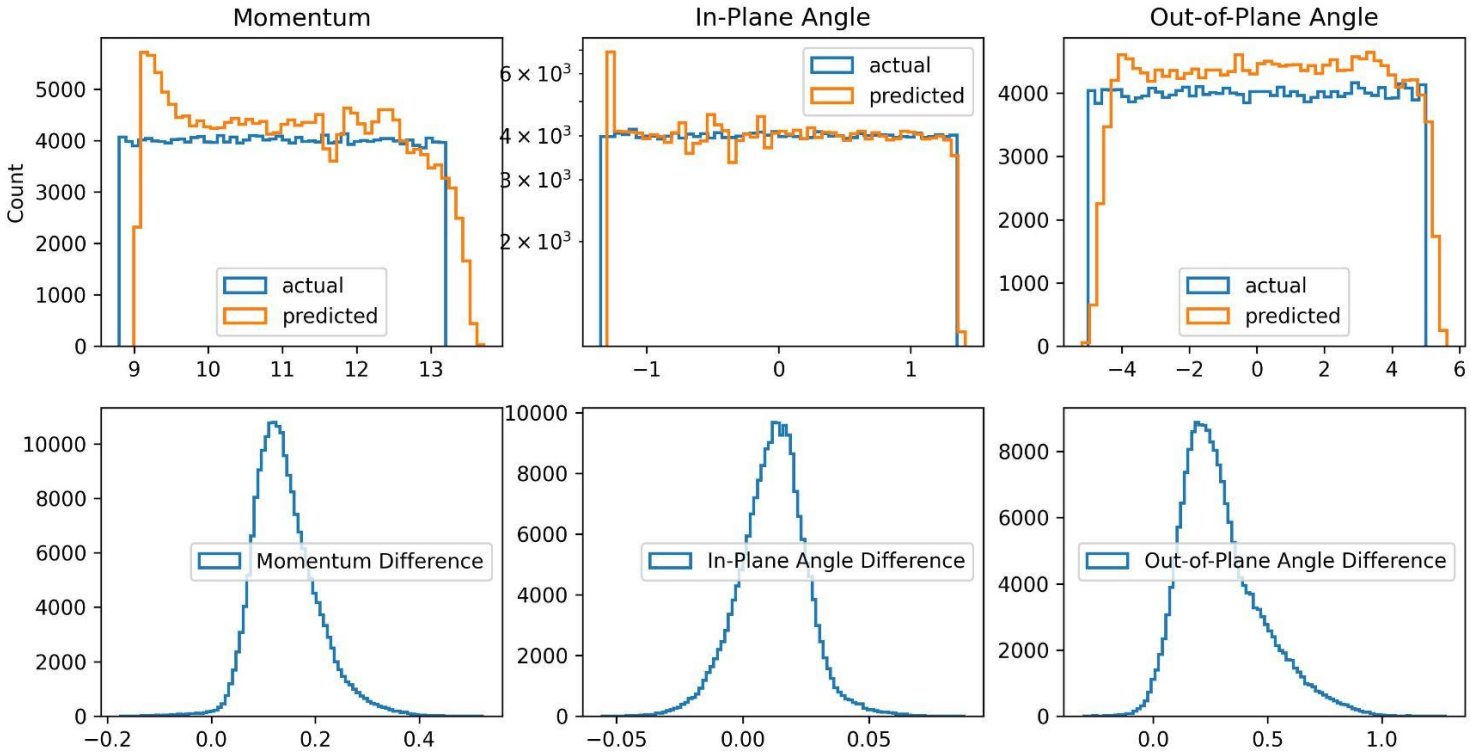
Tried with less physics effect and smearing



Geant4 simulation

10 events in each spectrometer arm

Tried with less physics effect and smearing



Conclusion and next step

- We are testing using ML for the vertex and mass reconstruction for DarkLight
- Will continue to investigate and try more complicated models and combine fit for all three outputs

Conclusion and next step

- We are testing using ML for the vertex and mass reconstruction for DarkLight
- Will continue to investigate and try more complicated models and combine fit for all three outputs