Cluster Reconstruction

https://github.com/Joy-whale0321/INTT-EMCAL/tree/main/InttSeedingTrackDev/ParticleGen
https://github.com/Joy-whale0321/INTT-EMCAL/tree/main/InttSeedingTrackDev/InttSeedTrackPerformance

CaloGeomMappingv2.h & CaloGeomMappingv2.cc
CaloGeomMappingv2 is used to load the new, accurate EMCal geometry and store it within RawTowerGeomv5.

RawTowerGeomv5.h & RawTowerGeomv5.cc
These files are used to store the descriptions of the eight vertices of each tower, along with some simple functions to obtain the centers of certain surfaces.

tutorial.h & tutorial.cc
In simulation events, read the information stored in each node, including truth information, track detector information, and EMCal information.

Fun4All_physiTuto.C
Simulate particle interactions in sPHENIX, producing truth information and the reconstruction information of each detector.

CaloInfo.C
Study cluster position reconstruction and correction.
From the previous simulation files, read the first g4hit produced by the primary electron in the EMCal as the truth position.
Read Cluster_innr and Cluster_geom, that is, “cluster reconstruct with tower inner face center” and “cluster reconstruct with tower volume center,” as the reconstructed position.
Compare the difference between the reconstructed position and the truth position, and compute the distance difference.
Then split the distance into radial and tangential directions, mainly study the angular difference in the tangential direction between the reconstructed position and the truth position, study the bias of the reconstructed position of positrons and electrons with different energies relative to the truth position, plot dphi – pT, write the results as TGraph/TF1, and store them in a ROOT file. When performing Cluster_innr reconstruction, the correction can be obtained by interpolating in the corresponding TGraph/TF1 according to its charge and energy
correction TGraph/TF1 named “grPeakVsX” and “fitCurve” on file https://github.com/Joy-whale0321/INTT-EMCAL/tree/main/InttSeedingTrackDev/ParticleGen/output (wC - with Correct; woC -without Correction) with

TruthToSvtxTrack.h & TruthToSvtxTrack.cc
Read G4primary information and, based on it, set up an svtxtrack.

PT func

https://github.com/Joy-whale0321/INTT-EMCAL/tree/main/InttSeedingTrackDev/InttSeedTrackPerformance

GetPtFunc.C

Read simulation data and extract the charged particle's deflection angle Δϕ, the particle's truth η, and the particle's truth pT.

Then compute C(η) = pT·Δϕ to obtain (η, C(η)). Next, use a polynomial fit on these data points to get the relation between pT and C(η). First do a rough fit over a wide range, then based on the rough fit perform a precise fit to obtain the C(η) – η function.

Store the fitted function in a ROOT file.

calc_pt_fromFunc.C

Read the TF1 function from the ROOT file, then compute pT = C(η)/Δϕ, and study the pT performance (resolution) calculated from the function. Store the results in a ROOT file.

At present, referring to Takashi's result, the Δϕ term in the function is $\Delta\phi^{-0.986}$, which is better than $\Delta\phi^{-1}$.

Therefore, the above code also considers using $\Delta\phi^{-0.986}$. That is, use C(η) = pT·($\Delta\phi^{0.986}$) to obtain C(η), and use pT = C(η)/($\Delta\phi^{0.986}$) to compute pT.

How to use the function?

First you need the deflection angle Δϕ and the particle's η (in my simulation the vertex is at (0,0,0), so I directly used the EMCal position η).

Open the ROOT file where I stored the function, obtain the C(η) – η function, and use η to get the corresponding C(η) from the function.

Compute the particle's pT = C(η)/Δϕ, or pT = C(η)/($\Delta\phi^{0.986}$). You can refer to my calc_pt_fromFunc.C file for the related code.

The term of C(η) on Pt func with $\Delta\phi^{1}$ & $\Delta\phi^{0.986}$ results have been stored on root file:
https://github.com/Joy-whale0321/INTT-EMCAL/tree/main/InttSeedingTrackDev/InttSeedTrackPerformance/output

ML 4 RECO

https://github.com/Joy-whale0321/INTT-EMCAL/tree/main/InttSeedingTrackDev/ML4Reco

A. From [INTT_R,Z + calo_R,Z,E] to pt

version2/data.py

```
Receive the root file and read the branch from the tree
branches_to_load = [
            "trk_system", "trk_layer", "trk_X", "trk_Y", "trk_Z",
            "caloClus_system", "caloClus_X", "caloClus_Y", "caloClus_Z", "caloClus_edep",
            "caloClus_innr_X", "caloClus_innr_Y", "caloClus_innr_Z", "caloClus_innr_edep",
            "PrimaryG4P_Pt"
        ]
Then analyze it and extract iINTT oINTT and EMCal Cluster Position R, Z and Energy
trk_feat = np.array([
                        r34, p34[-1],    # iINTT 3/4 layer → r, z
                        r56, p56[-1]     # oINTT 5/6 layer → r, z
                    ])
calo_innr_feat = np.array([
                        r_innr,
                        calo_innr_z[0],
                        calo_innr_e[0]
                    ])
Then they are spliced together
feat = np.concatenate([trk_feat, calo_innr_feat])
X_data.append(feat)
As the input feature of the model, this input includes the R, Z values of the two positions provided by
INTT, as well as the R, Z values and energy of the position of EMCal.
Y_data.append(Truth_Pt[0])
Extract the "truth pt" of electrons as the Y target of the model.
```

version2/model.py

```
self.net = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, 1) )
The MLP network consists of an input-output layer and three hidden layers. The input layer receives 7
standardized features (iINTT R, iINTT Z, oINTT R, oINTT Z, EMCal Cluster R, EMCal Cluster Z, EMCal Cluster
Energy), and each hidden layer node has 256 nodes. The output prediction value is used as the reconstructed
point.
```

version2/train_ptbin.py

```
The training parameters are set as follows: The training data is divided into a 70% training set and a
30% validation set. In each epoch, the training set is shuffled to eliminate the influence of sequence.
def train(list_file, pt_min=0.0, pt_max=2.0, batch_size=1024, epochs=300, lr=5e-5, val_ratio=0.3,
device="cuda" if torch.cuda.is_available() else "cpu"):

The loss function is defined as follows
xb, yb = xb.to(device), yb.to(device)
pred = model(xb)
pt_reso = (pred - yb) / (yb)   ### Relative resolution as main term
weights = (pt_reso.abs() < 0.2).float() * 2.0 + 1.0   ### Increase the weight inside the peak,
reduce the influence of outlier data points
loss = ((pt_reso) ** 2 * weights).mean()   ### Use squared values instead of absolute values to make the
peak position closer to zero.

if val_loss < best_val_loss:
            best_val_loss = val_loss
            torch.save(model.state_dict())
            print(f"✓ Saved best model (val loss = {val_loss:.4f})")   ### Saved best model
```

version2/test_ptbin.py
Call the previously trained model, use it in the test set, and study the "performance (pt resolution)"

B. From [$\Delta\Phi$] to pt

version4/data.py

```
Receive the root file and read the branch from the tree.
branches_to_load = [
            "trk_system", "trk_layer", "trk_X", "trk_Y", "trk_Z",
            "caloClus_system", "caloClus_X", "caloClus_Y", "caloClus_Z", "caloClus_edep",
            "caloClus_innr_X", "caloClus_innr_Y", "caloClus_innr_Z", "caloClus_innr_edep",
            "PrimaryG4P_Pt"
        ]
Then, conduct an analysis on it, extract the x and y positions of iINTT and oINTT, as well as the x and
y positions of the EMCal Cluster. Connect iINTT and oINTT, then connect EMCal and oINTT. Calculate the
angle between the two lines, and take the 1/delta-phi as feature.
proxy_trans = 1/dphi
feat = np.array([proxy_trans, 0])
As the input feature of the model
X_data.append(feat)

Extract the "truth pt" of electrons as the Y target of the model
Y_data.append(Truth_Pt[0])
```

version4/model.py

```
self.net = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, 1)  # 回归输出
        )
The MLP network consists of an input-output layer and three hidden layers. The input layer receives the
reciprocal of the deflection angle (1/delta-phi), and the nodes in each hidden layer are all 256. The
output prediction value serves as the reconstructed point.
```

version4/train_ptbin.py

```
The training parameters are set as follows: The training data is divided into a 70% training set and a
30% validation set. In each epoch, the training set is shuffled to eliminate the influence of sequence.
def train(list_file, pt_min=0.0, pt_max=2.0, batch_size=1024, epochs=500, lr=5e-5, val_ratio=0.2,
device="cuda" if torch.cuda.is_available() else "cpu"):

The loss function is defined as follows:
The relative resolution is the main component.pt_reso = (pred - yb) / (yb)
weights = (pt_reso.abs() < 0.2).float() * 2.0 + 1.0
main_loss = ((pt_reso) ** 2 * weights).mean()

# == boundary loss ==
Add boundary conditions outside the data range to ensure that the pt estimate is sufficiently elevated
at small angles.
x1 = np.array([0, 0.5, 1, 2, 10, 15, 25, 50, 100, 200])
x2 = np.zeros_like(x1)
x_boundary_np = np.stack([x1, x2], axis=1)
x_boundary = torch.tensor(x_boundary_np, dtype=torch.float32).to(device)
y_boundary_target = torch.tensor([0, 0.0961, 0.1922, 0.3844, 1.922, 2.883, 4.805, 9.61, 19.22, 38.44],
dtype=torch.float32).unsqueeze(1).to(device)
y_boundary_pred = model(x_boundary)
boundary_loss = nn.MSELoss()(y_boundary_pred, y_boundary_target)
# Dynamic weighting to prevent affecting data learning in the early stages.
lambda_boundary = min(0.005 * epoch, 0.2)

# == monotonic penalty ==
requires pt to decrease as the angle increases; avoid oscillations occur.
x_sorted, indices = torch.sort(xb[:,0])
pred_sorted = pred[indices]
mono_penalty = monotonic_loss(pred_sorted)
```

```
# === sum loss ===
loss = main_loss + lambda_mono * mono_penalty + lambda_boundary * boundary_loss

if val_loss < best_val_loss:
            best_val_loss = val_loss
            torch.save(model.state_dict())
            print(f"✓ Saved best model (val loss = {val_loss:.4f})")   ### Saved best model
```

version4/test_ptbin.py
Call the previously trained model, use it in the test set, and study the "performance (pt resolution)"

Fusion A and B

combine2_gate/data_combined.py

```
First, load the previously trained weights, and then construct the datasets required by each of the previous
two models.
ds_dphi   = dphiDataset(list_file, pt_min=pt_min, pt_max=pt_max)
ds_energy = energyDataset(list_file, scaler=scaler_energy, pt_min=pt_min, pt_max=pt_max)

Obtain the predicted values of the two models separately and use them as the input
pt_pred1 = model_dphi(x1.unsqueeze(0).to(device)).cpu().item()
pt_pred2 = model_energy(x2.unsqueeze(0).to(device)).cpu().item()
X_fusion.append([pt_pred1, pt_pred2])

target is truth pt
Y_fusion.append(truth)
```

combine2_gate/model_combined.py
Here, a gate net is used to train an MLP to determine the weights by which to combine [p1, p2] reconstructed from the previous two models to obtain pt_pred.

combine2_gate/train_combined.py

```
xb, yb = xb.to(device), yb.to(device)
pred = model(xb)
pt_reso = (pred - yb) / (yb)
weights = (pt_reso.abs() < 0.2).float() * 2.0 + 1.0
loss = ((pt_reso) ** 2 * weights).mean()

The definition of the loss function is similar to that mentioned earlier
```

combine2_gate/test_combined.py
Call the previously trained model, use it in the test set, and study the performance.

How to use them?

Training:

For Models A and B (on ***version2*** and ***version4***), go into each model's directory and run ***train_ptbin.py***. Before running, check that your training dataset includes the tree and branches defined in ***data.py***; data format follows the class modules in ***ParticleGen/physiTuto/tutorial.h*** and ***ParticleGen/physiTuto/tutorial.cc*** on that write data into those branches.

For the Fusion model in ***combine2_gate***, you need the prediction outputs from Models A and B. Copy their ***data.py*** and ***model.py*** into the ***combine2_gate*** directory and rename them to match each architecture: ***data_dphi.py, model_dphi.py*** for the ΔΦ-based model, and ***data_energy.py, model_energy.py*** for the energy-based model. Then, in ***data_combined.py***, load the pretrained weights for Models A and B and set the correct path to Model A's normalization scaler (change the absolute paths to where your weights are stored). After that, run ***train_combined.py***.

I have uploaded my training weights. They seem quite a lot files, but you can see which one is currently named in the "train" section or which one is loaded in the "test" section, and then you will know which one to use.

Test:

In the ***version2*** and ***version4*** directories, run ***test_ptbin.py***. In the ***combine2_gate*** directory, run ***test_combined.py***. Each script will load the saved model weights and report the reconstruction performance.

You can reference my data struction from example file on:

https://github.com/Joy-whale0321/INTT-EMCAL/tree/main/InttSeedingTrackDev/ML4Reco/Filelist

Performance I have: Maybe its better cause I plot them by a rough fitting