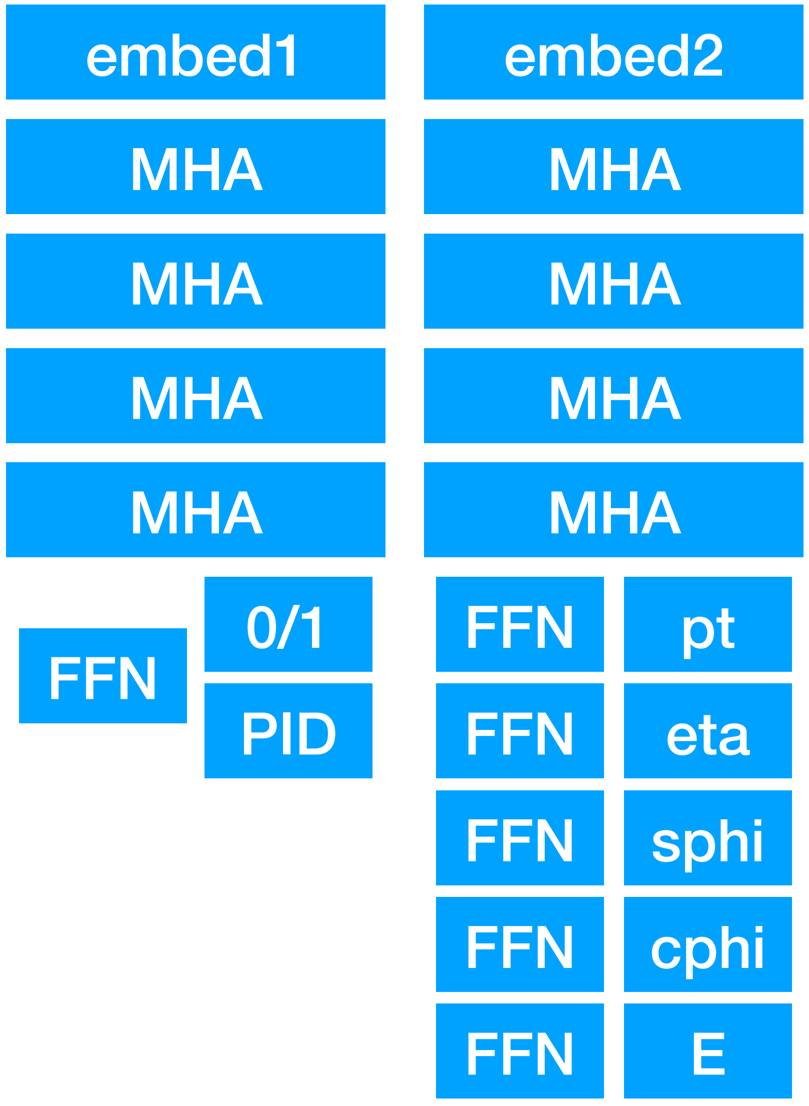
CLIC recap

- **Several breaking changes since the paper**
- **Old target → new target**: new target is better aligned with PF, more physical and harder to reconstruct
- **TF → pytorch**: generally, we saw comparable performance between the two implementations on the old target, but small differences can exist.
- **GNNLSH → Transformer**: Transformer significantly outperforms GNNLSH in terms of final loss and convergence speed in our tests, regardless of the target.
    - Transformer implementation has several degrees of freedom that have not been studied so far
- Changes to model structure have resulted in **limited improvements to loss or final physics performance**, but results only clear after several days of training with multiple GPUs on large samples.
- We **need to provide a new CLIC model that outperforms PF** with the new setup

Model

- Reconstructing or not reconstructing a particle matters more than the specific PID: classification split to binary (ptcl/no ptcl with cross-entropy) and PID multiclass (focal)
- Transform targets as: $E_{tgt}'=\log[E_{tgt}/E_{elem}]$, approx. Gaussian for energy and pt, more stable loss
- Momentum regression predicts pt, eta, sphi/cphi, energy separately, particles can be off shell, restrict to positive mass^2
- Final MHA layer queries can be one of three options: previous MHA output, **initial embeddings**, or trainable queries (a la ParticleTransformer)
- MultiheadAttention uses `key_padding_mask` with math backend (CLIC-size events)
- Initial embeddings and final FFNs can be split according to element type, different weights for tracks & clusters



Momentum regression of separate E, pT, eta components can result in negative mass^2, which silently screws up fastjet.

```diff
@@ -414,7 +425,25 @@ def forward(self, X_features, mask):
         preds_eta = self.nn_eta(X_features, final_embedding_reg, X_features[..., 2:3])
         preds_sin_phi = self.nn_sin_phi(X_features, final_embedding_reg, X_features[..., 3:4])
         preds_cos_phi = self.nn_cos_phi(X_features, final_embedding_reg, X_features[..., 4:5])
-        preds_energy = self.nn_energy(X_features, final_embedding_reg, X_features[..., 5:6])
-        preds_momentum = torch.cat([preds_pt, preds_eta, preds_sin_phi, preds_cos_phi, preds_energy], axis=-1)

+        # ensure created particle has positive mass^2 by computing energy from pt and adding a positive-only correction
+        pt_real = torch.exp(preds_pt.detach()) * X_features[..., 1:2]
+        pz_real = pt_real * torch.sinh(preds_eta.detach())
+        e_real = torch.log(torch.sqrt(pt_real**2 + pz_real**2) / X_features[..., 5:6])
+        e_real[~mask] = 0
+        e_real[torch.isinf(e_real)] = 0
+        e_real[torch.isnan(e_real)] = 0
+        preds_energy = e_real + torch.nn.functional.relu(self.nn_energy(X_features, final_embedding_reg, X_features[..., 5:6]))
+        preds_momentum = torch.cat([preds_pt, preds_eta, preds_sin_phi, preds_cos_phi, preds_energy], axis=-1)
         return preds_binary_particle, preds_pid, preds_momentum
```

Hack in model output to constrain E^2 > pT^2 + pz^2. Doing this with a loss term was not effective. Any better way?

pT and energy highly correlated, can we reparametrize?

25 mlpf/pyg/PFDataset.py

```diff
@@ -70,6 +70,23 @@ def __getitem__(self, item):
             ret["ygen"][:, 0][(ret["X"][:, 0] == 10) & (ret["ygen"][:, 0] == 7)] = 2
             ret["ygen"][:, 0][(ret["X"][:, 0] == 11) & (ret["ygen"][:, 0] == 7)] = 2
 
+            # set pt for HO which would otherwise be 0
+            msk_ho = ret["X"][:, 0] == 10
+            eta = ret["X"][:, 2][msk_ho]
+            e = ret["X"][:, 5][msk_ho]
+            ret["X"][:, 1][msk_ho] = np.sqrt(e**2 - (np.tanh(eta) * e) ** 2)
+
+        # transform pt -> log(pt / elem pt), same for energy
+        ret["ygen"][:, 6] = np.log(ret["ygen"][:, 6] / ret["X"][:, 5])
+        ret["ygen"][:, 6][np.isnan(ret["ygen"][:, 6])] = 0.0
+        ret["ygen"][:, 6][np.isinf(ret["ygen"][:, 6])] = 0.0
+        ret["ygen"][:, 6][ret["ygen"][:, 0] == 0] = 0
+
+        ret["ygen"][:, 2] = np.log(ret["ygen"][:, 2] / ret["X"][:, 1])
+        ret["ygen"][:, 2][np.isnan(ret["ygen"][:, 2])] = 0.0
+        ret["ygen"][:, 2][np.isinf(ret["ygen"][:, 2])] = 0.0
+        ret["ygen"][:, 2][ret["ygen"][:, 0] == 0] = 0
+
```

Log-transform pT and energy with element pt/energy.

Attention backend

- **Math**: default in pytorch, simple N^2 evaluation of attention. Works for CLIC (N<300), does not work for CMS due to memory and speed constraints. Supports `key_padding_mask`. Supports old GPU architectures.

- **Flash**: available since pytorch 2.2, numerically equivalent to math, but much faster for large events (N>500 ptcls) and does not require N^2 memory. Required for CMS training. Does not support `key_padding_mask`. Requires recent GPU architecture: A100, H100, MI250X or similar.

- Training on CLIC: use math and whatever GPU you have

- Training on CMS: use flash and a recent GPU

Datasets

- Pythia, CLICdet Geant4 model, Marlin, Pandora, 380 GeV, Key4HEP, ~4M events in each sample. No PU-like overlay, simple ee.
- Samples used in training so far: ttbar, qq
- **New**, not used so far: WW → full hadronic, ZH→$\tau\tau$, Z→$\tau\tau$

EDMHEP + HEPMC

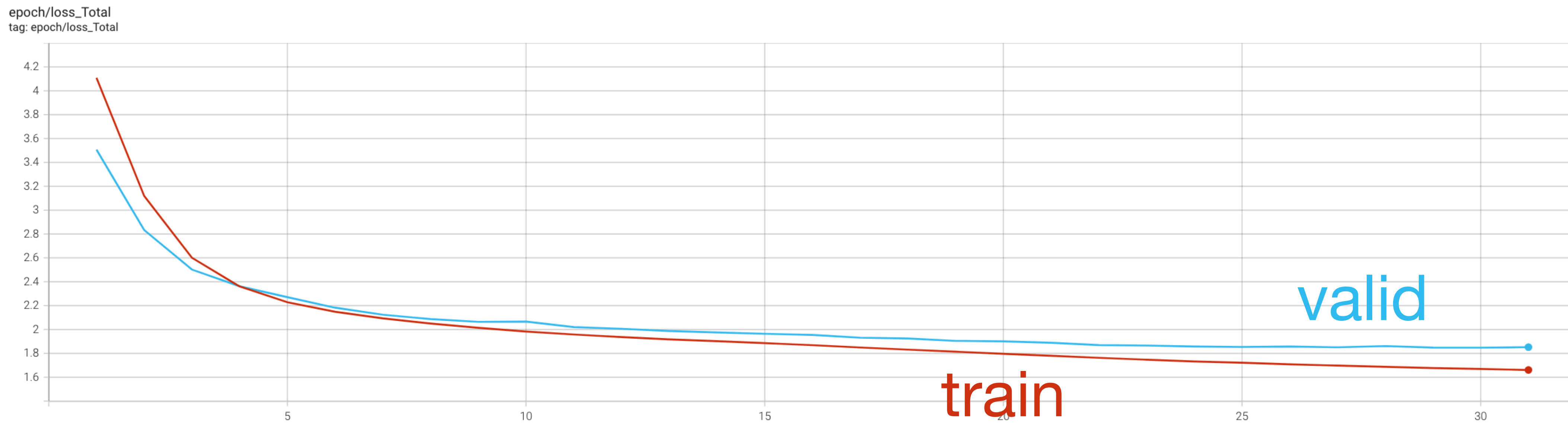
```
1.1T    /local/joosep/clic_edm4hep/2024_07/p8_ee_qq_ecm380
1.5T    /local/joosep/clic_edm4hep/2024_07/p8_ee_tt_ecm380
1.5T    /local/joosep/clic_edm4hep/2024_07/p8_ee_WW_fullhad_ecm380
790G    /local/joosep/clic_edm4hep/2024_07/p8_ee_ZH_Htautau_ecm380
443G    /local/joosep/clic_edm4hep/2024_07/p8_ee_Z_Ztautau_ecm380
```

MLPF features and targets

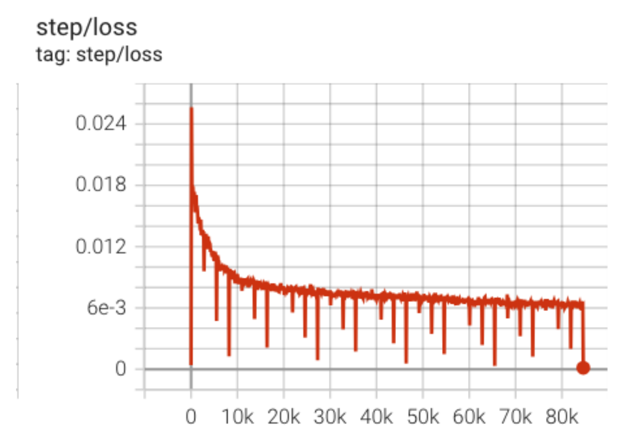
```
32G     /local/joosep/mlpf/clic_edm4hep/p8_ee_qq_ecm380
63G     /local/joosep/mlpf/clic_edm4hep/p8_ee_tt_ecm380
51G     /local/joosep/mlpf/clic_edm4hep/p8_ee_WW_fullhad_ecm380
24G     /local/joosep/mlpf/clic_edm4hep/p8_ee_ZH_Htautau_ecm380
5.7G    /local/joosep/mlpf/clic_edm4hep/p8_ee_Z_Ztautau_ecm380
```

TFDS, available on EOS

```
23G /eos/user/j/jpata/mlpf/tensorflow_datasets/clic/clic_edm_qq_pf
36G /eos/user/j/jpata/mlpf/tensorflow_datasets/clic/clic_edm_ttbar_pf
37G /eos/user/j/jpata/mlpf/tensorflow_datasets/clic/clic_edm_ww_fullhad_pf
18G /eos/user/j/jpata/mlpf/tensorflow_datasets/clic/clic_edm_zh_tautau_pf
4.3G/eos/user/j/jpata/mlpf/tensorflow_datasets/clic/clic_edm_z_tautau_pf
```

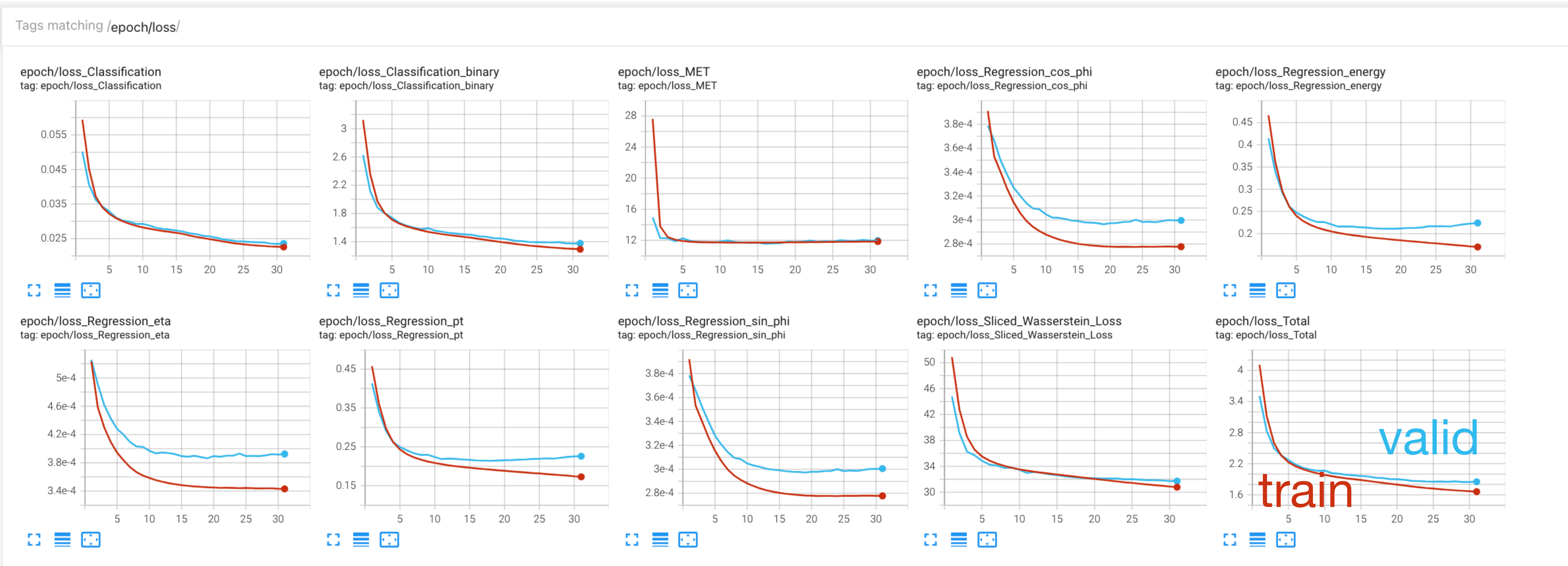


30 epochs (~1h/epoch) on 8x MI250X (LUMI HPC). Stable convergence and no numerical issues.

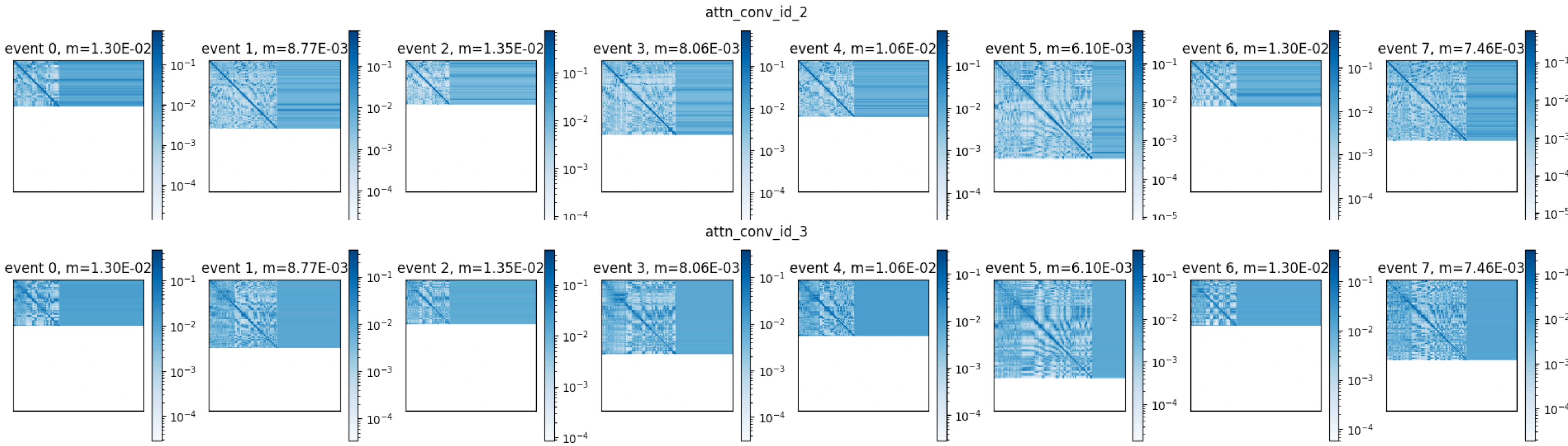


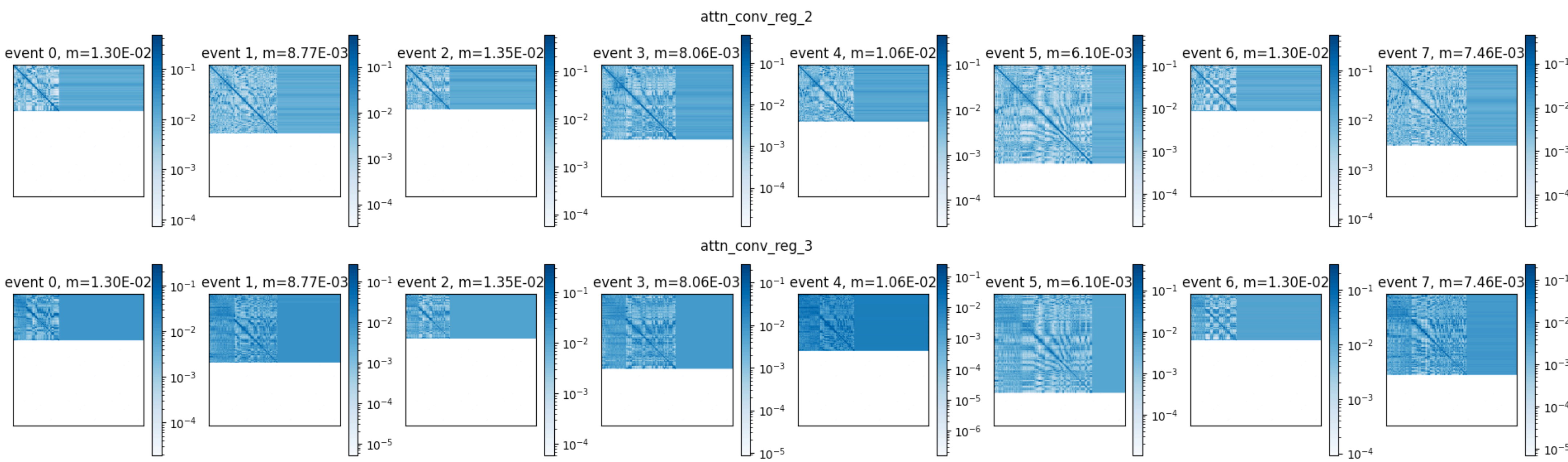
Stepwise loss is decreasing stably.

Energy & pt regression start to overtrain.

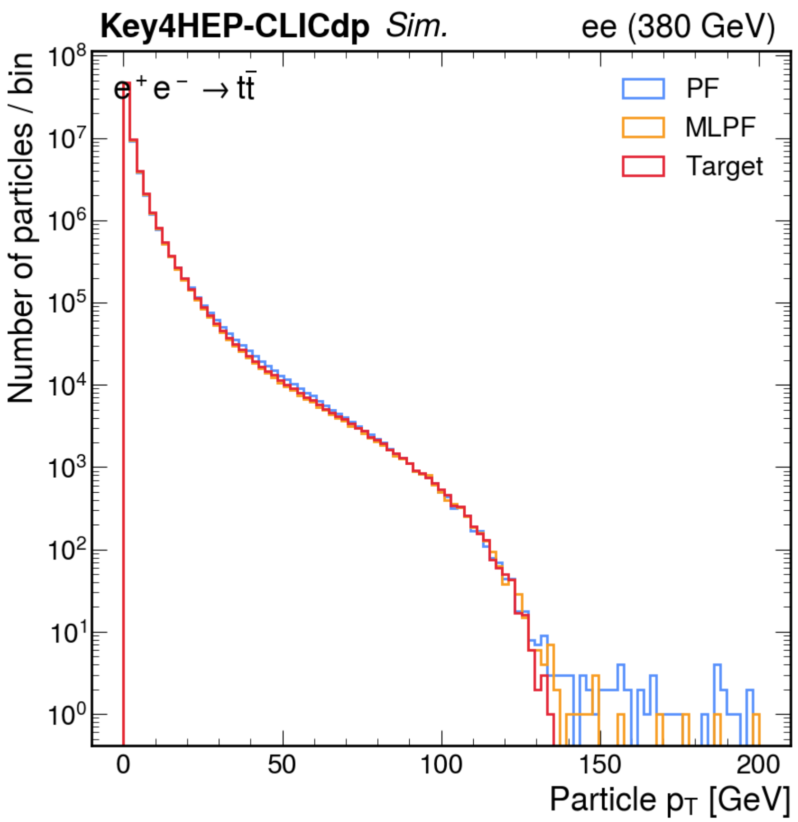


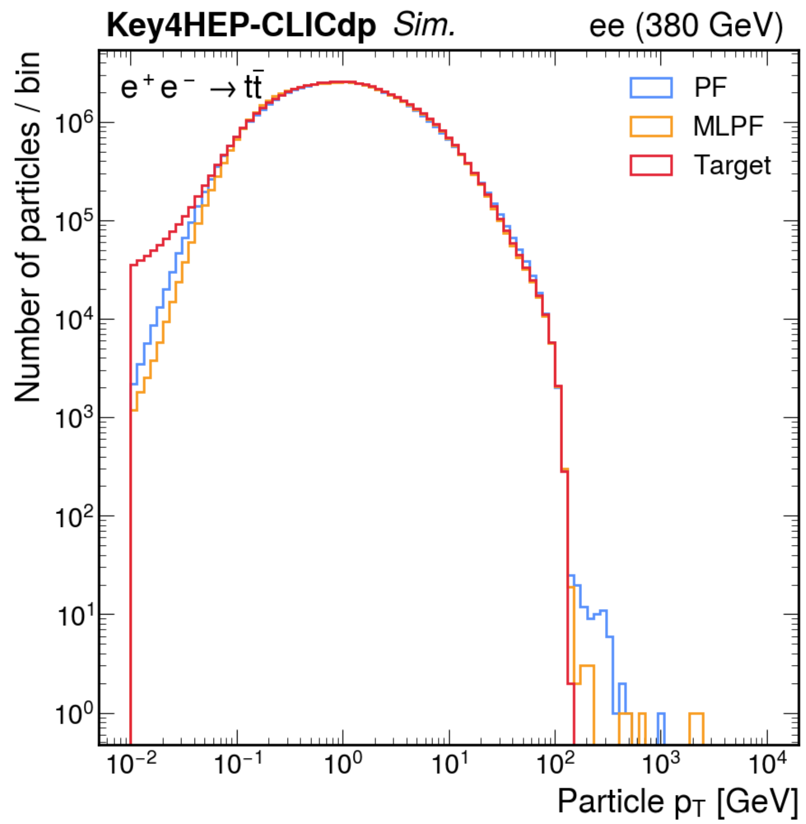
MET and SWD losses are for monitoring, no gradient propagation.
No significant improvement in MET after initial convergence.

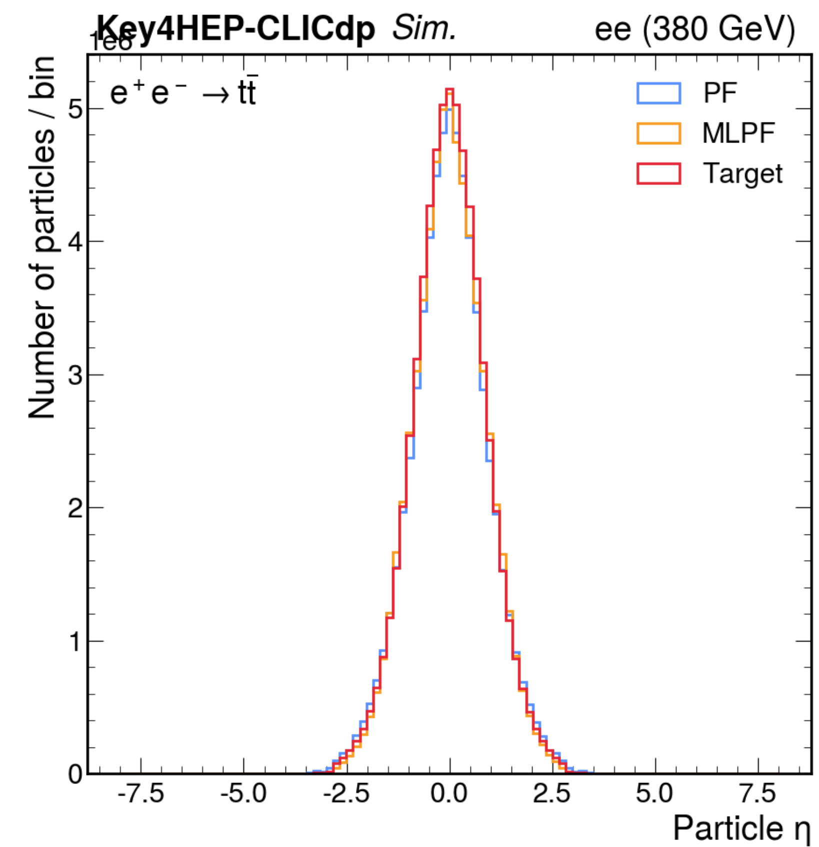


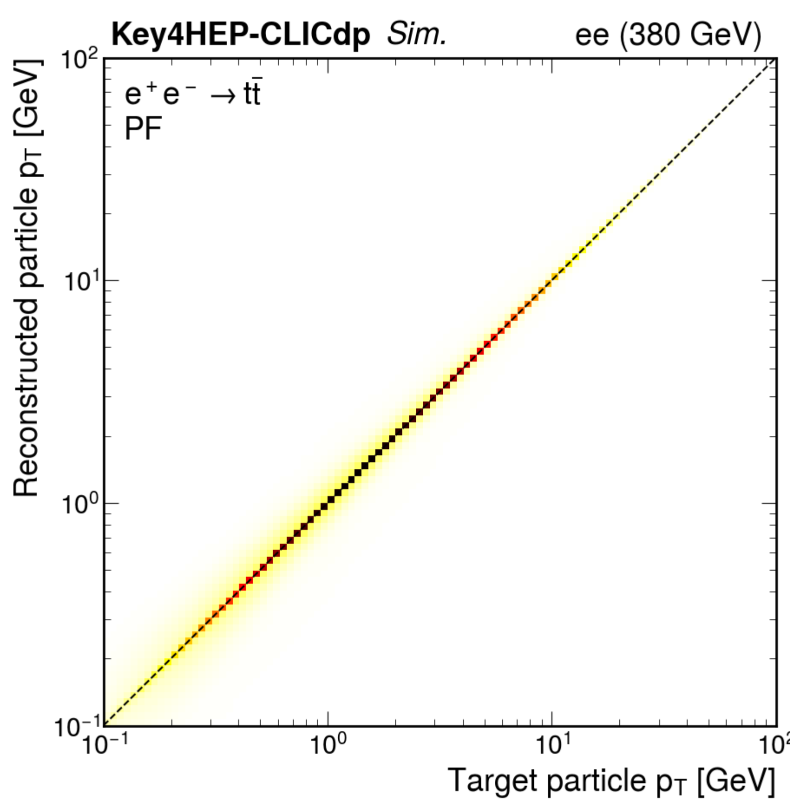


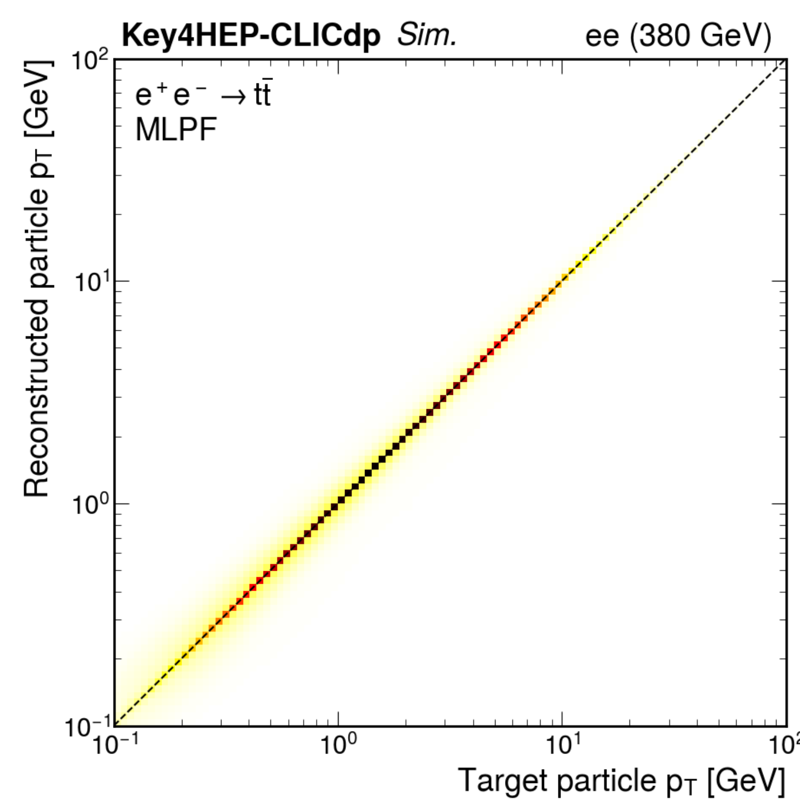
Learned attention matrix is nontrivial in all layers. ID / reg attention matrices visually rather similar: redundant?



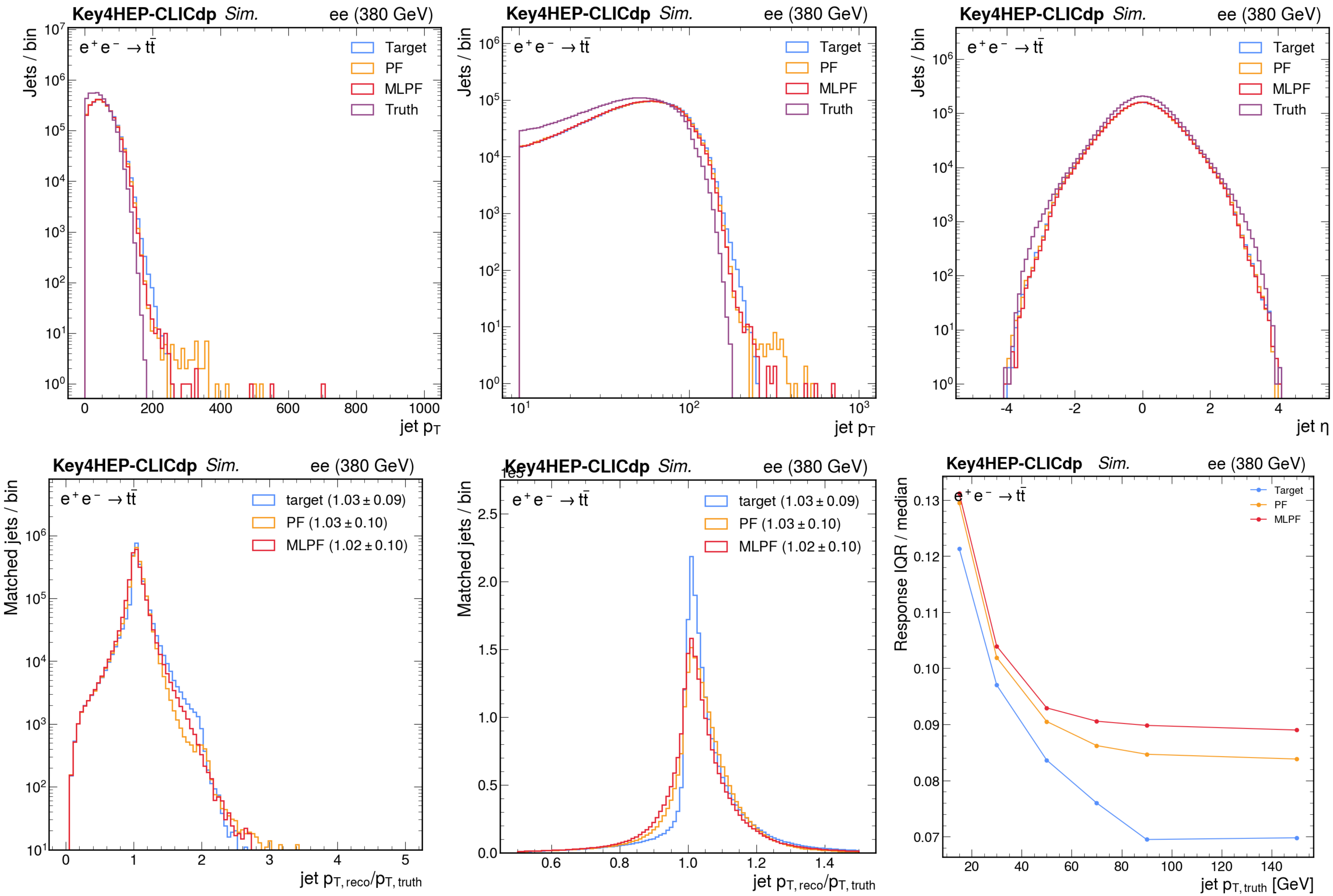




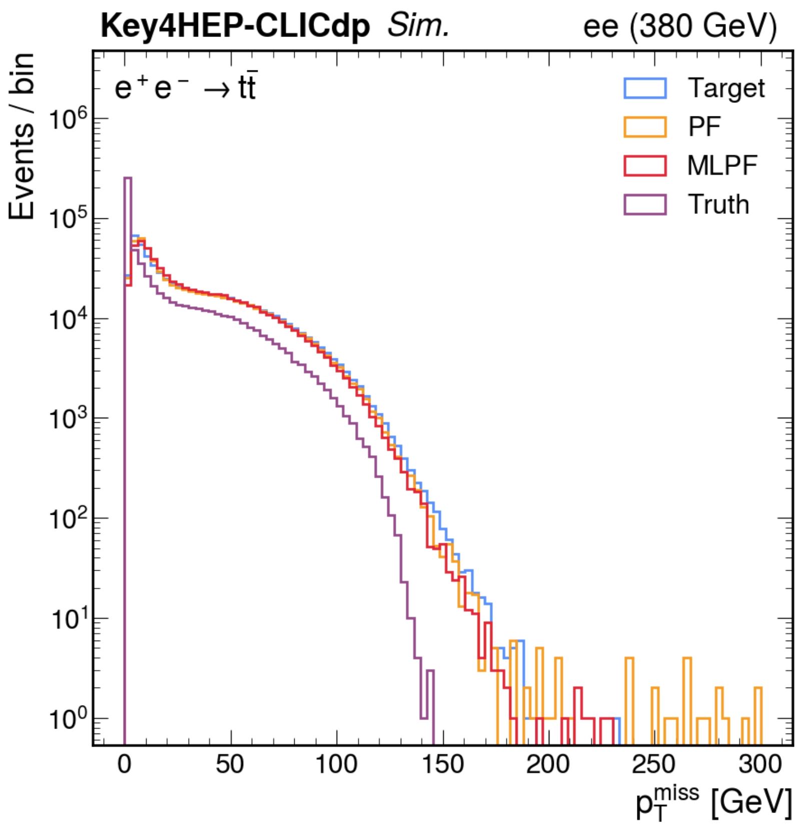


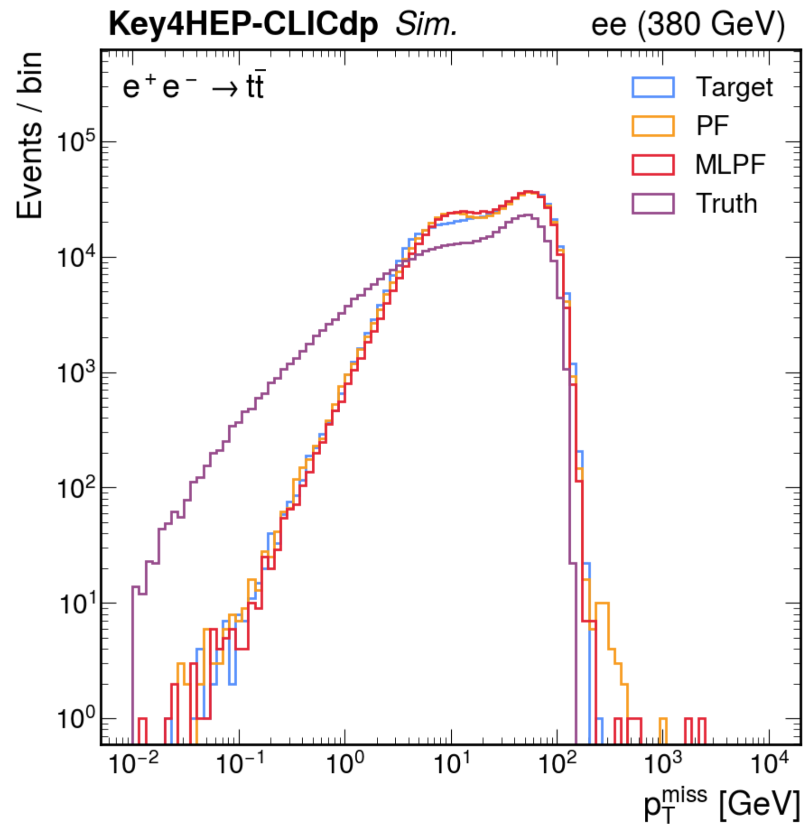


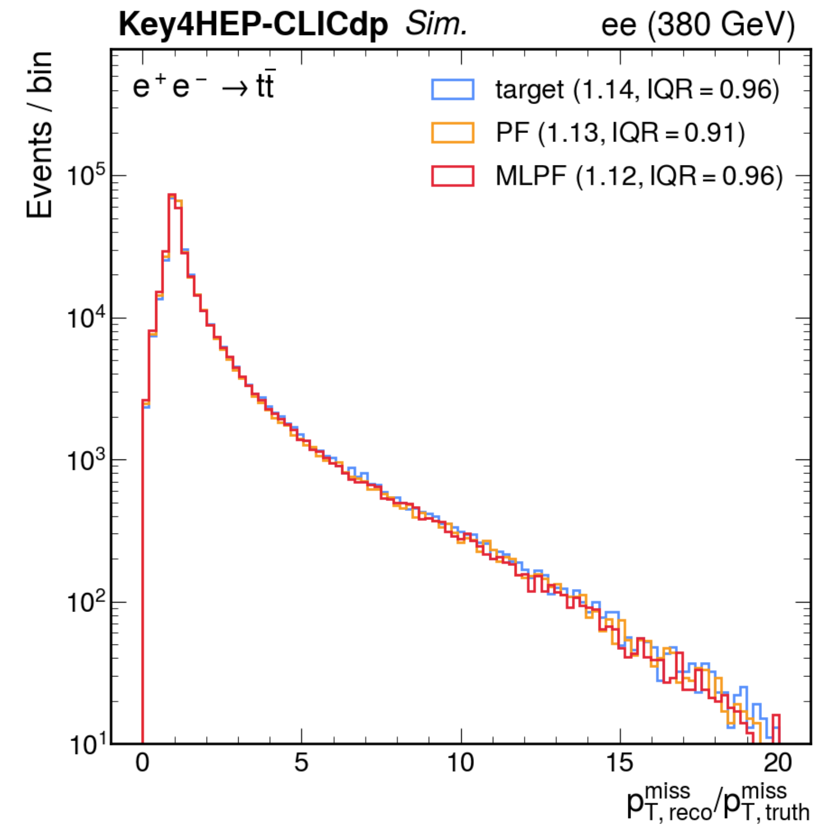
Performance is on par with PF

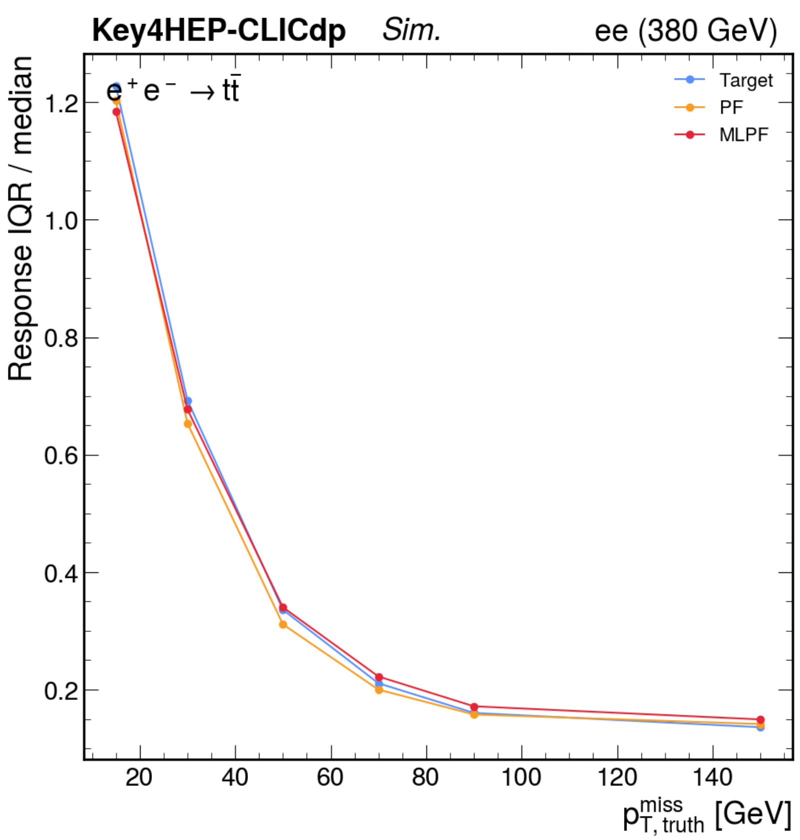


Performance is on par / slightly worse than PF

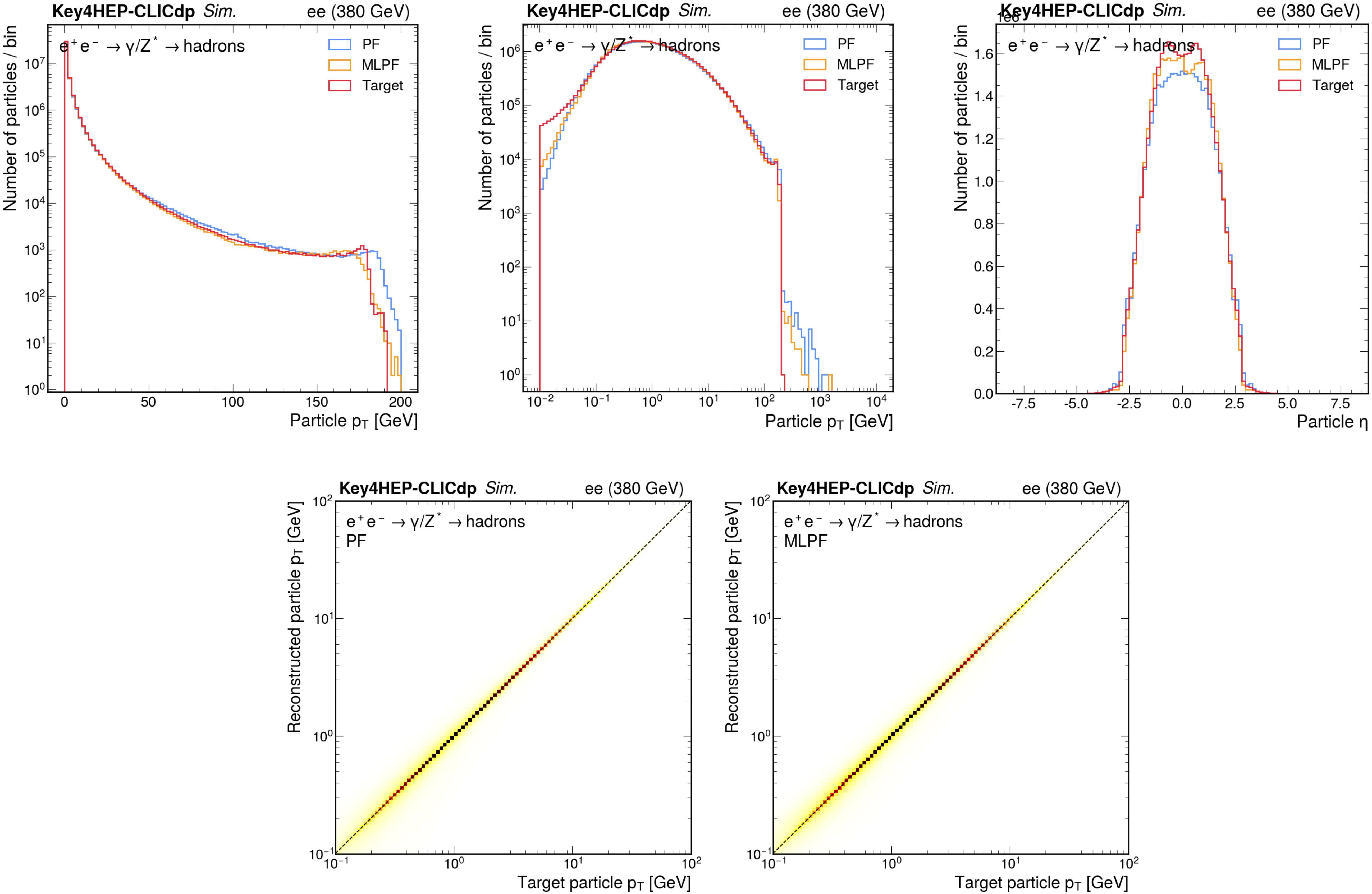




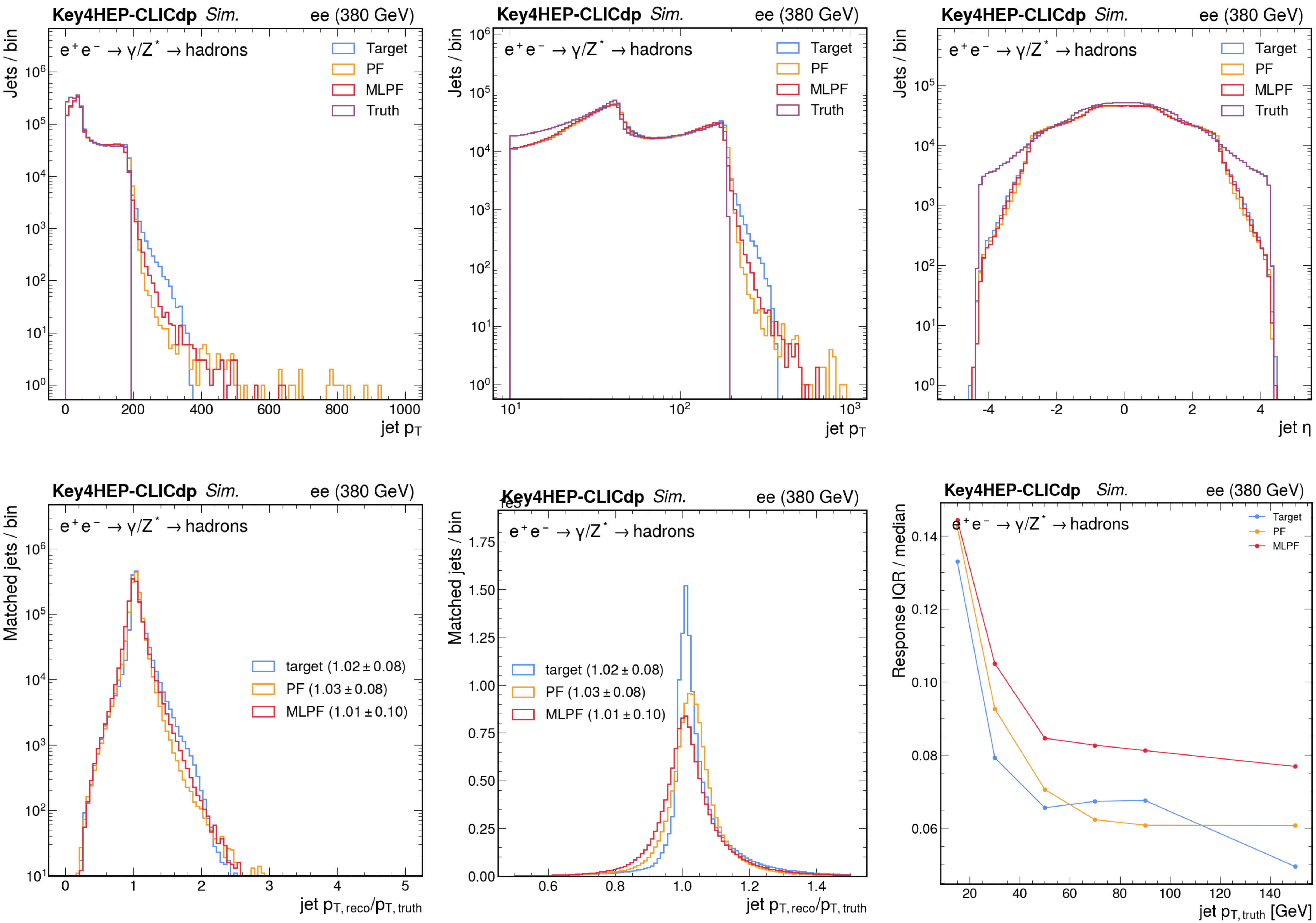




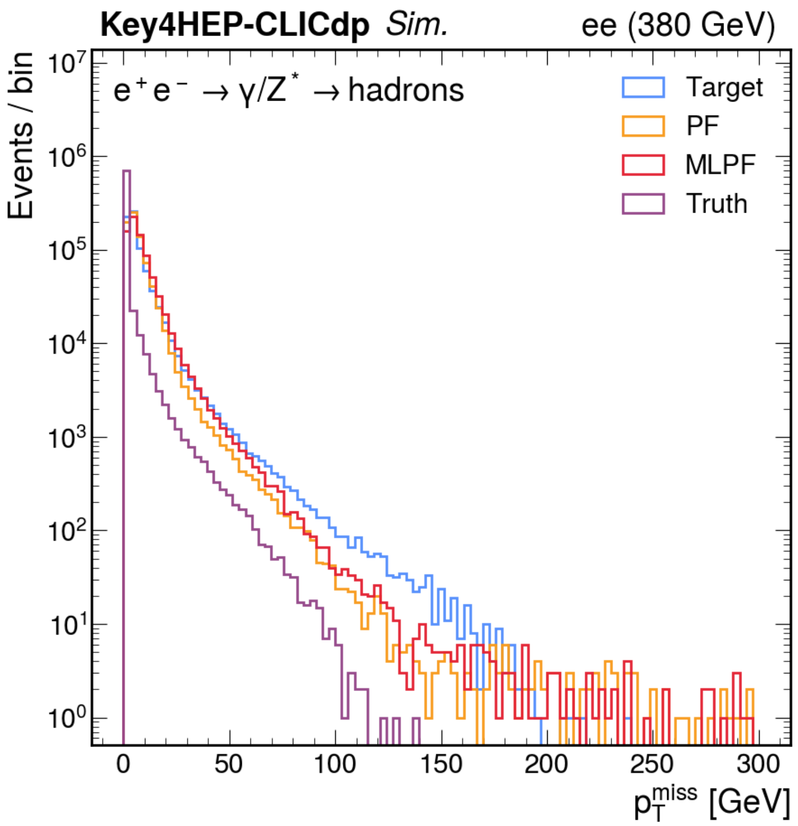
Performance is on par with PF

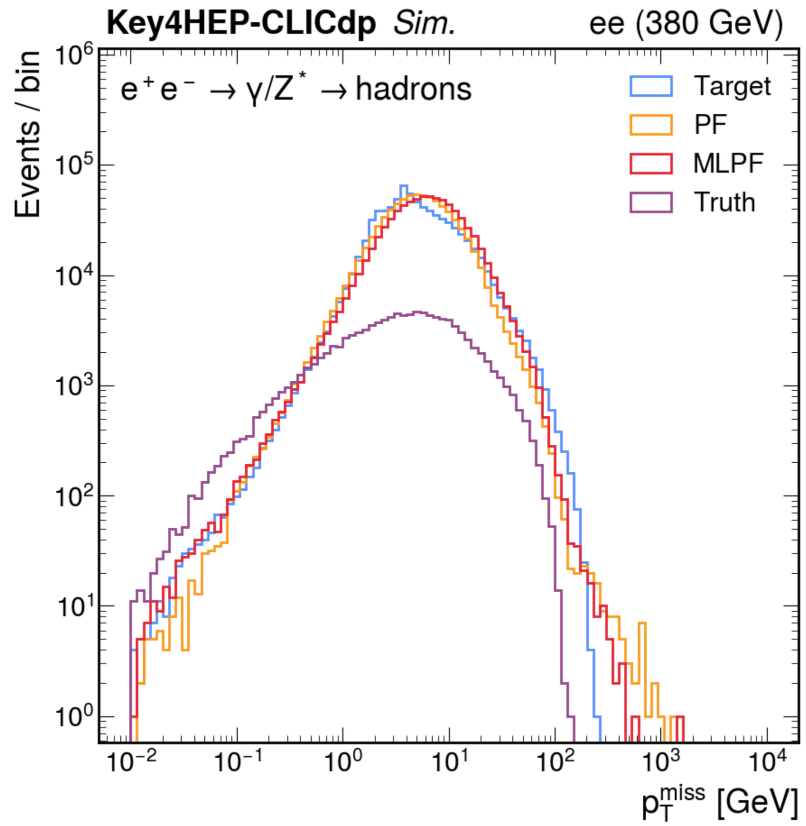


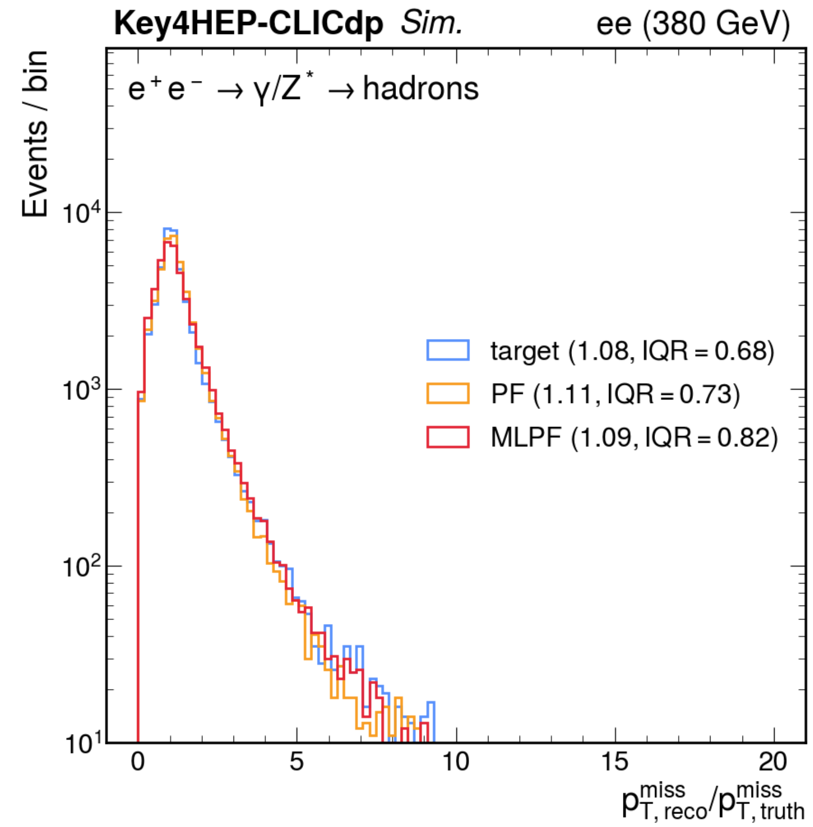
Performance is on par with PF

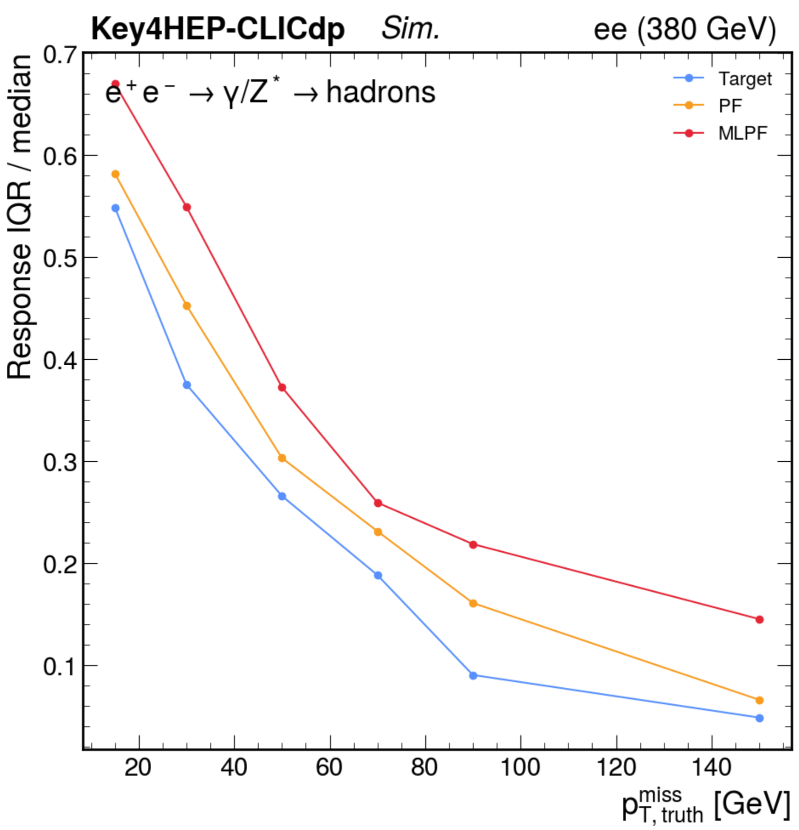


Performance is somewhat worse than PF

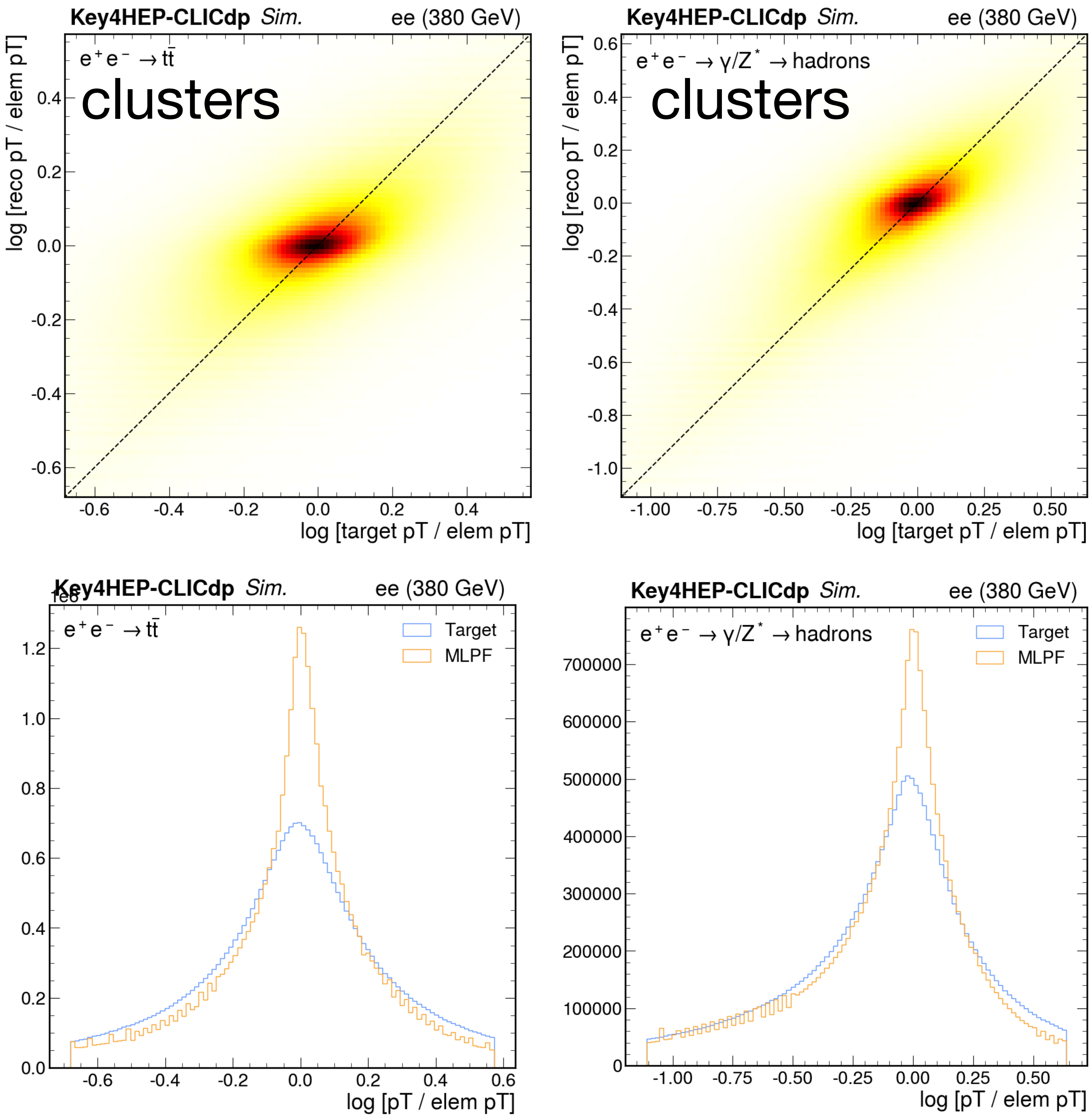




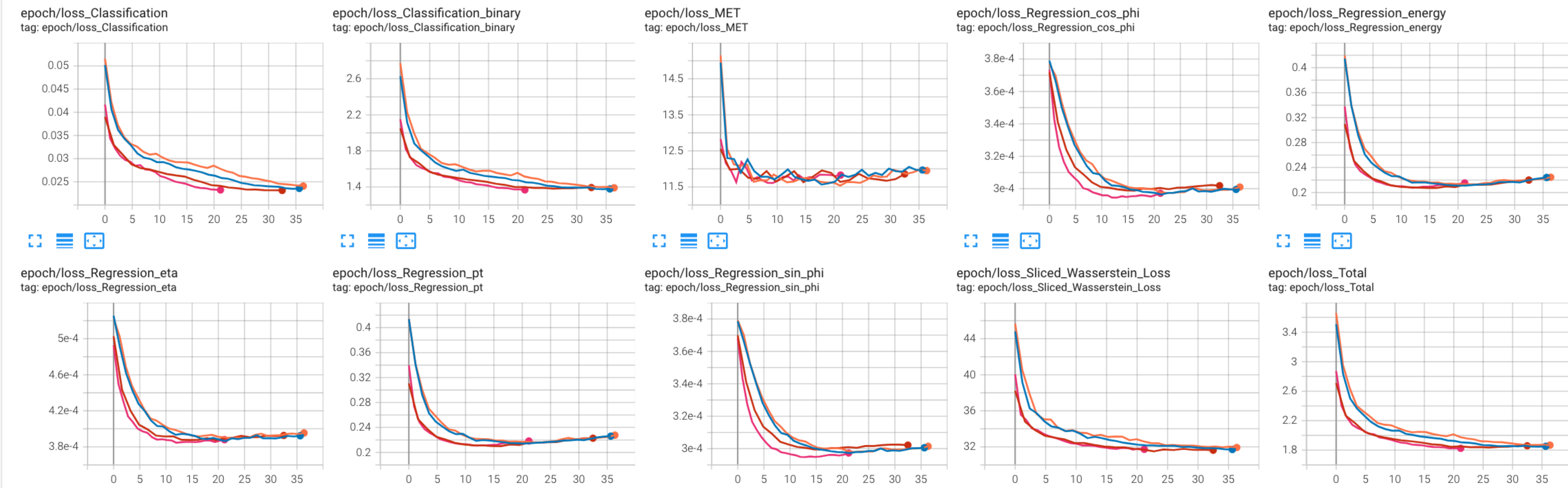




Performance is somewhat worse than PF



pT / energy regression is challenging, model output is too narrow.



default: 4x layers, bs256, initial embeddings as final queries

v1: 4x layers, bs256, trainable final queries **(no effect)**

v2: 4x layers, bs128, initial embeddings as final queries **(improved)**

v3: 8x layers, bs128, initial embeddings as final queries **(no effect)**

Decreasing batch size 256 to 128 has a small positive effect on convergence speed. Increasing the number of layers did not improve the result.