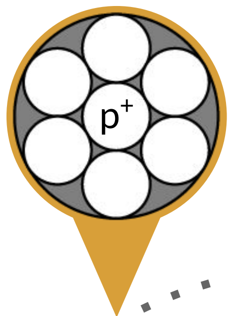
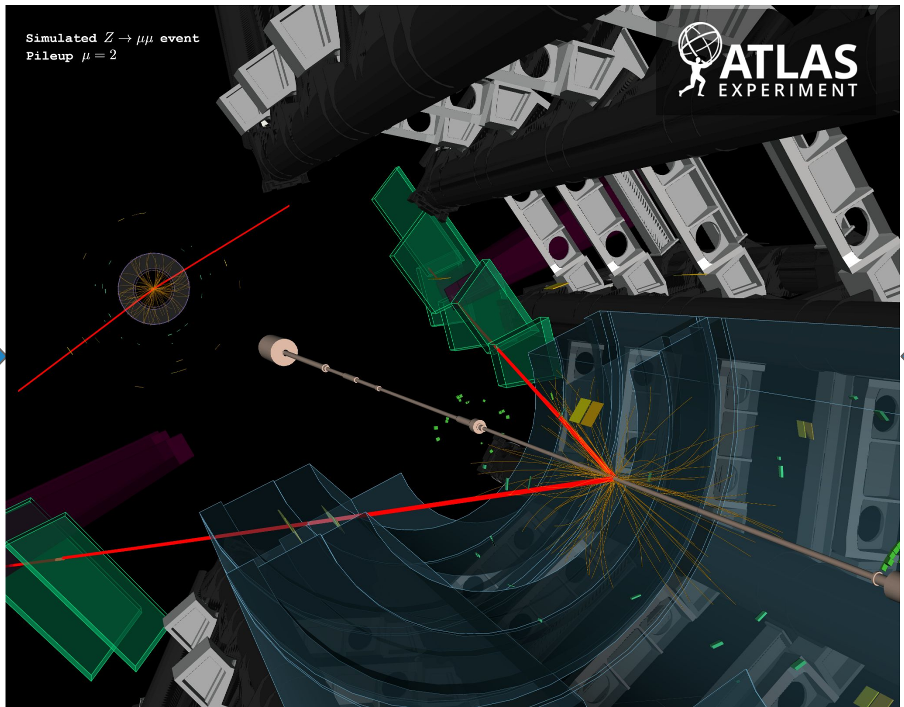
Optimal Transport Pileup Mitigation for Hadron Colliders

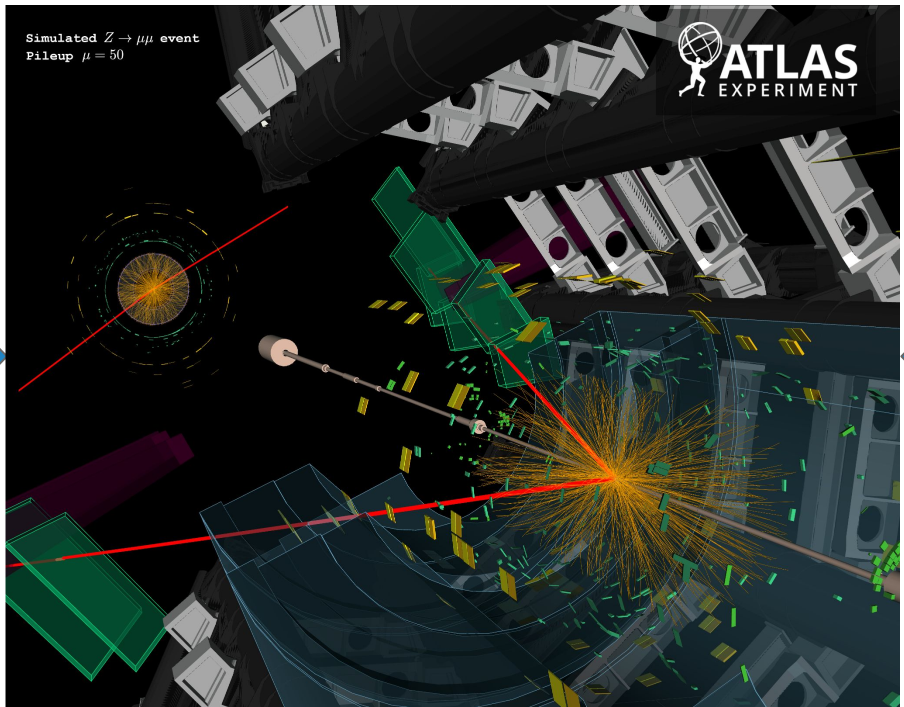
Nathan Suri, Vinicius Mikuni

ML4FP 2024





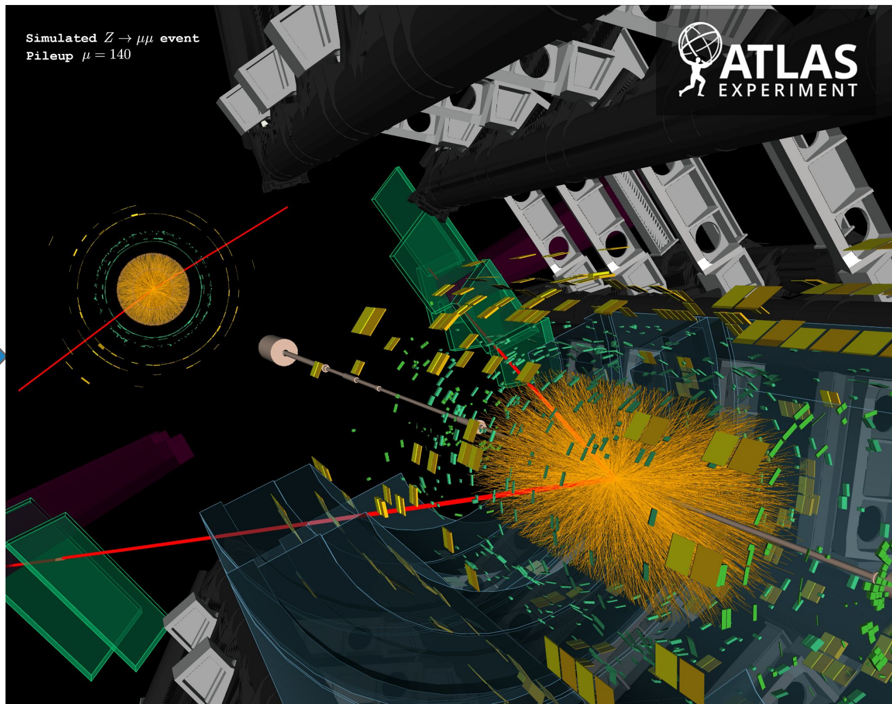
Charged + neutral pileup
In-time + out-of-time pileup



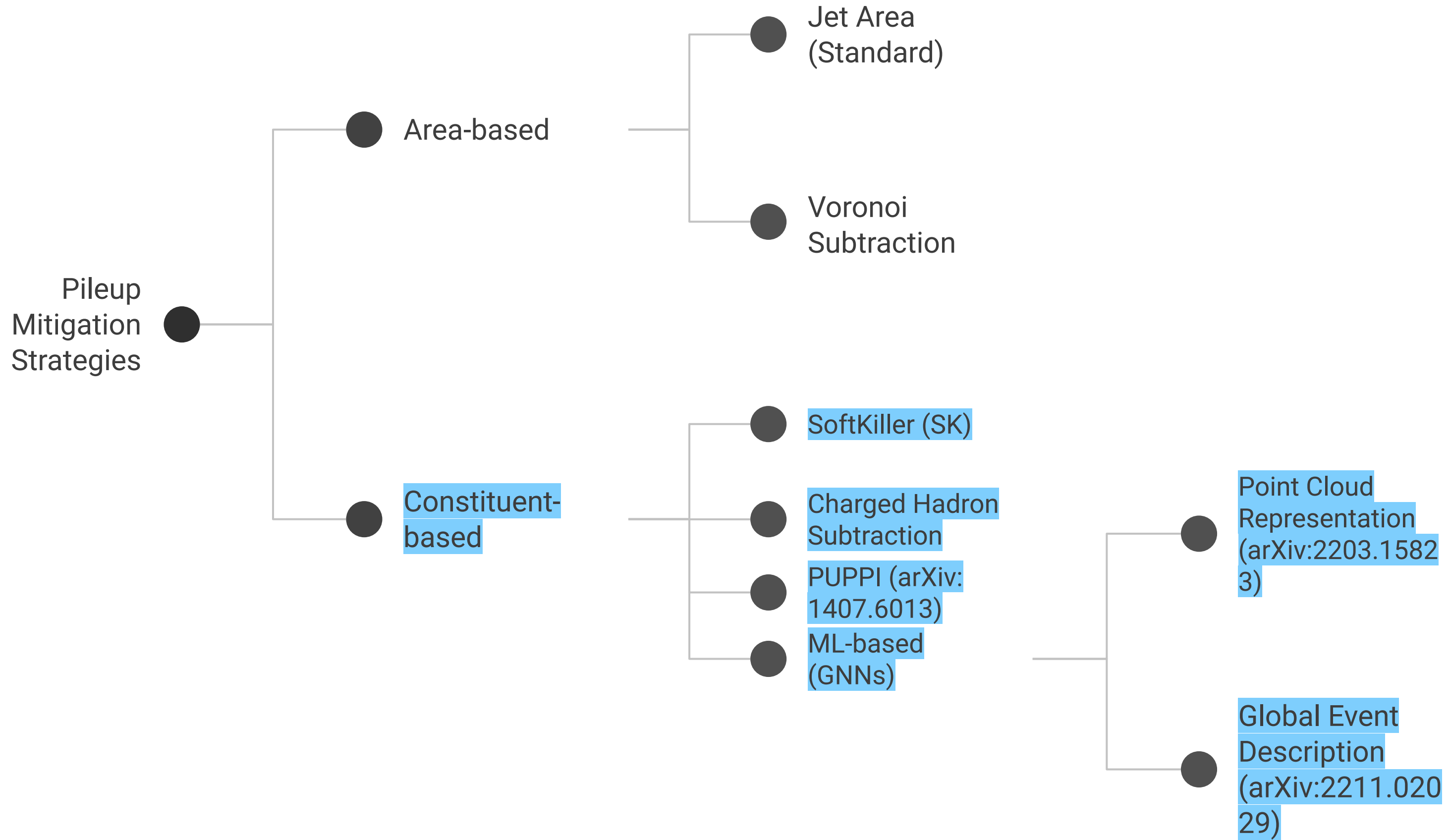
p+

Charged + neutral pileup
In-time + out-of-time pileup

p+



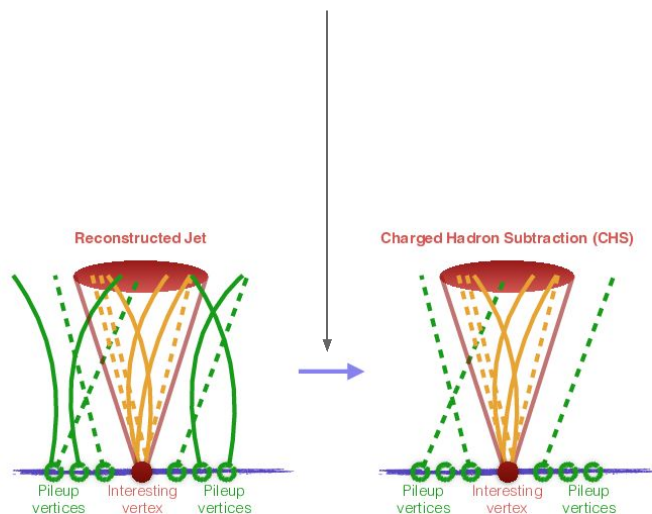
Charged + neutral pileup
In-time + out-of-time pileup

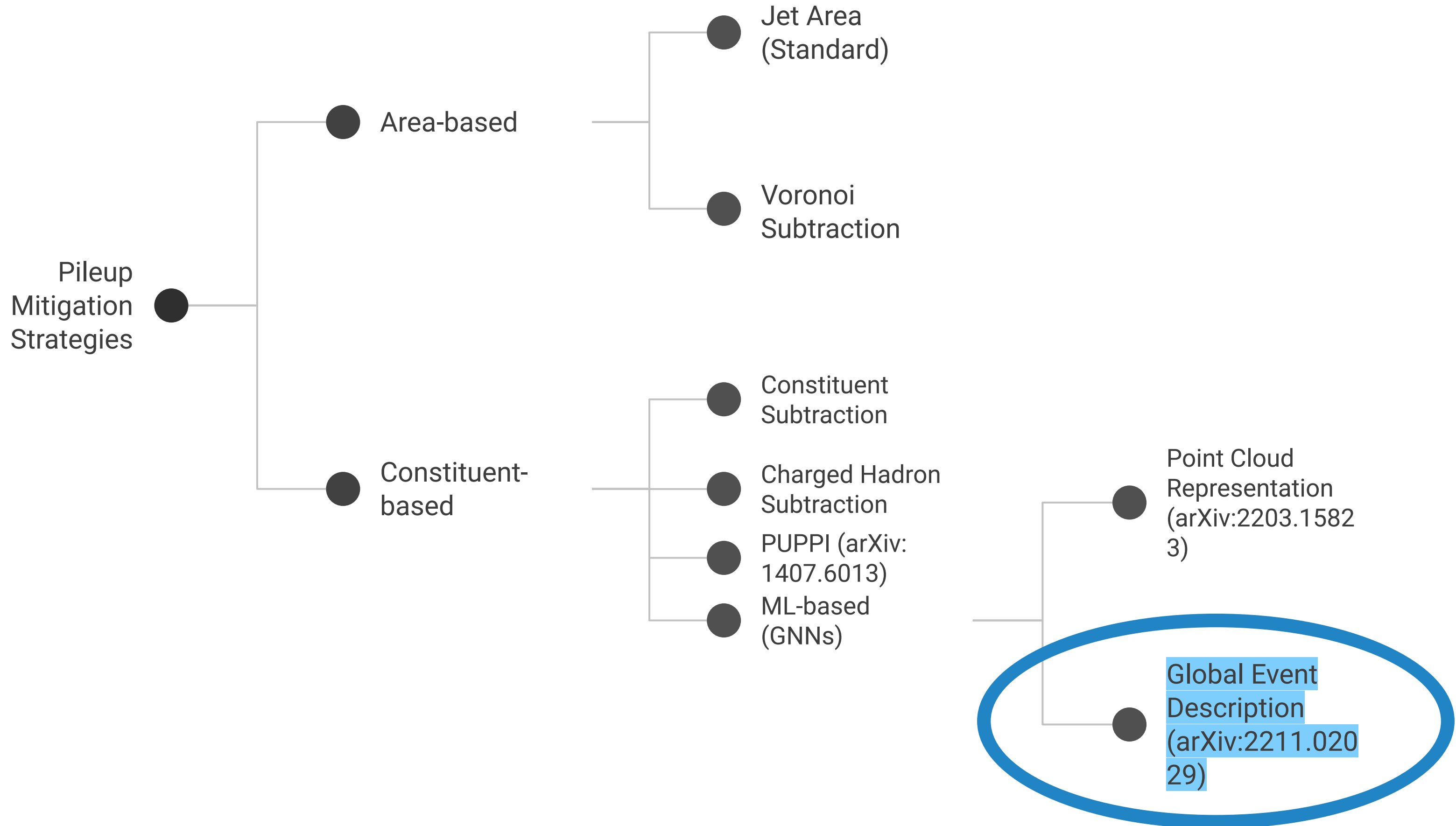

Pileup
Mitigation
Strategies
Area-based
Jet Area
(Standard)
Voronoi
Subtraction
Constituent-
based
SoftKiller (SK)
Charged Hadron
Subtraction
PUPPI (arXiv:
1407.6013)
ML-based
(GNNs)
Point Cloud
Representation
(arXiv:2203.1582
3)
Global Event
Description
(arXiv:2211.020
29)

Charged Hadron Subtraction

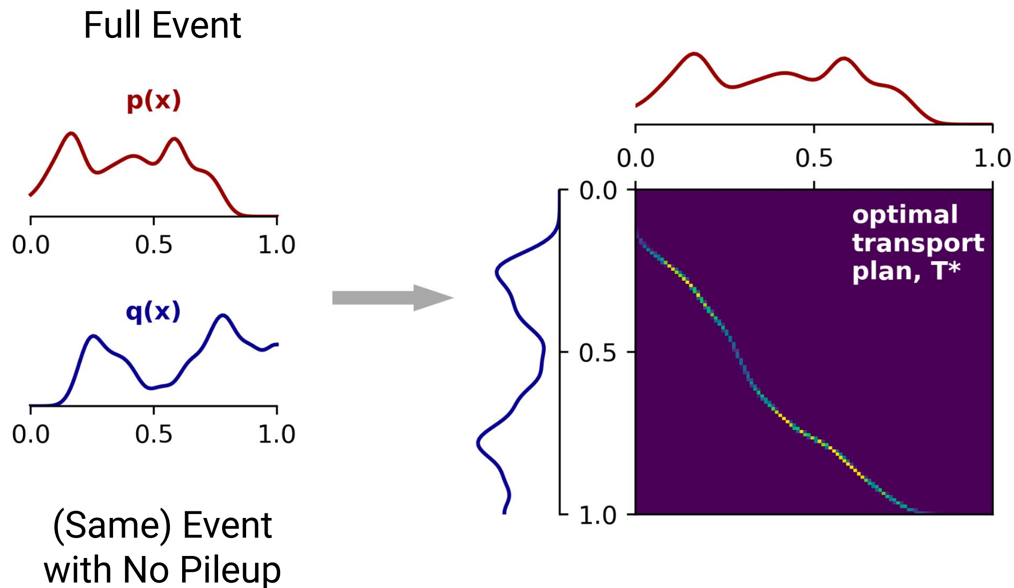
$$\mathrm{CVF} = \frac{\sum_{\mathrm{tracks,HS}} p_{\mathrm{T}}^{\mathrm{track}}}{\sum_{\mathrm{tracks,HS}} p_{\mathrm{T}}^{\mathrm{track}} + \sum_{\mathrm{tracks,PU}} p_{\mathrm{T}}^{\mathrm{track}}},$$

- Benefits
  - Very effective at removing charged pileup due to track information
- Drawbacks
  - Inapplicable to neutral pileup
- (arXiv:2012.06271)




Pileup
Mitigation
Strategies
Area-based
Constituent-
based
Jet Area
(Standard)
Voronoi
Subtraction
Constituent
Subtraction
Charged Hadron
Subtraction
PUPPI (arXiv:
1407.6013)
ML-based
(GNNs)
Point Cloud
Representation
(arXiv:2203.1582
3)
Global Event
Description
(arXiv:2211.020
29)

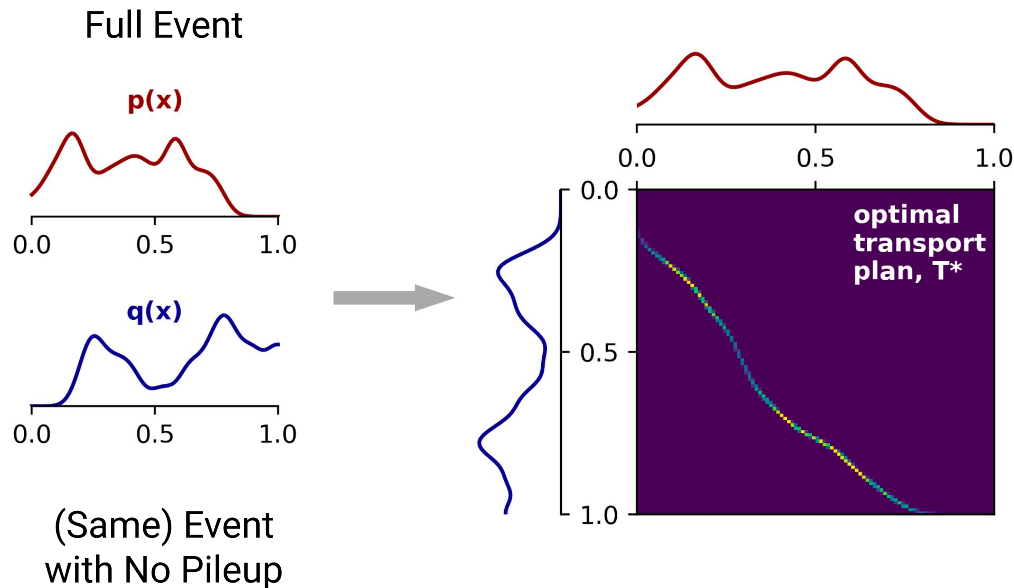
TOTAL: Training Optimal Transport with Attention Learning



$$\mathcal{L} = \mathrm{SWD}(x'_p, x_{np}) + \lambda\,\mathrm{MSE}(p_{\mathrm{T}}^{\mathrm{miss}}(x'_p), p_{\mathrm{T}}^{\mathrm{miss}}(x_{np}))$$

TOTAL: Training Optimal Transport with Attention Learning

- The probability density is intractable, but we can approximate the density
- Realizations of the density are accessible
- Optimal transport over the space of inputs allows for approximation



$$\mathcal{L} = \mathrm{SWD}(x'_p, x_{np}) + \lambda\, \mathrm{MSE}(p_{\mathrm{T}}^{\mathrm{miss}}(x'_p), p_{\mathrm{T}}^{\mathrm{miss}}(x_{np}))$$

TOTAL: Training Optimal Transport with Attention Learning

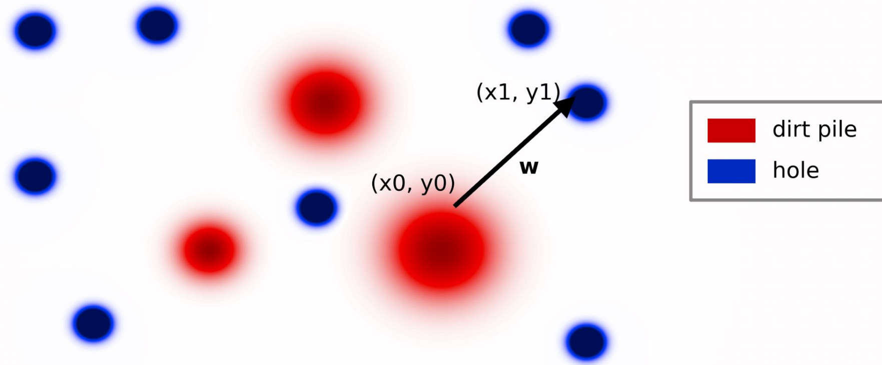
$$\mathcal{L} = \mathrm{SWD}(x'_p, x_{np}) + \lambda\,\mathrm{MSE}(p_{\mathrm{T}}^{\mathrm{miss}}(x'_p), p_{\mathrm{T}}^{\mathrm{miss}}(x_{np}))$$

TOTAL: Training Optimal Transport with Attention Learning

$$\mathcal{L} = \mathrm{SWD}(x'_p, x_{np})$$

- Wasserstein distance (WD): Finds the transport function that keeps hard scattering particles and removes those from simultaneous vertices
- Sliced WD to compensate for poor scaling of computational costs of calculating WD at high dimensions

Earth Mover's Distance = W_1



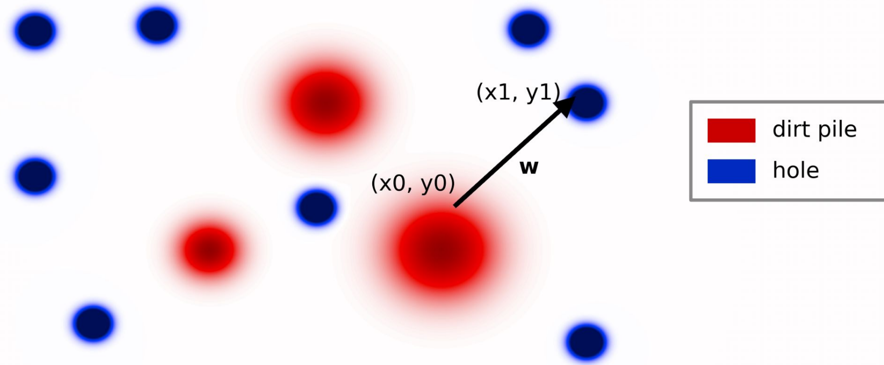
- Assumption: Total volume of the holes = total volume of the dirt piles
- Piles as the probability density function of P and holes as the probability density function of Q
- Per unit transportation cost:

$$C(x_0, y_0, x_1, y_1) = (x_0 - x_1)^2 + (y_0 - y_1)^2$$

- Transportation Plan:

$$T(x_0, y_0, x_1, y_1) = w$$

Earth Mover's Distance = W_1



$$\int\int T(x_0, y_0, x, y)dxdy = p(x_0, y_0)$$

$$\int\int T(x, y, x_1, y_1)dxdy = q(x_1, y_1)$$

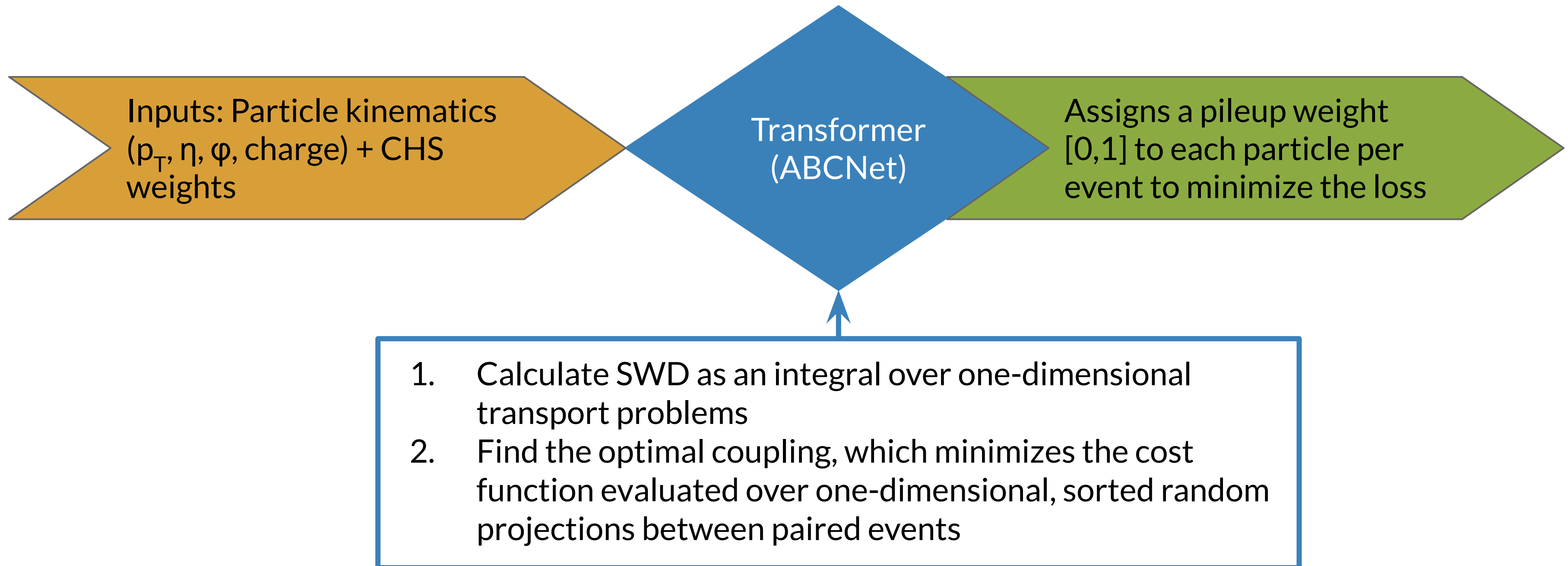
$$\text{Total Cost } = \int\int\int\int C(x_0, y_0, x_1, y_1) \cdot T(x_0, y_0, x_1, y_1)dx_0dy_0dx_1dy_1$$

TOTAL: Training Optimal Transport with Attention Learning

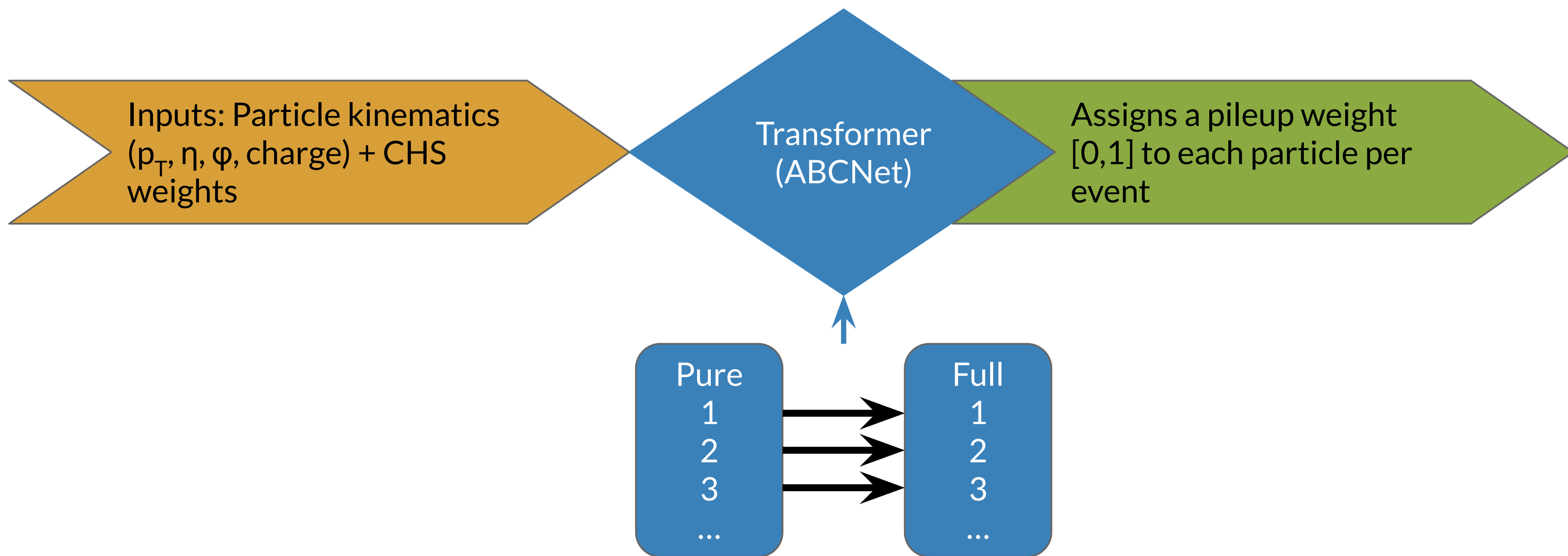
$$\mathcal{L} = \quad + \lambda\,\mathrm{MSE}(p_{\mathrm{T}}^{\mathrm{miss}}(x'_p), p_{\mathrm{T}}^{\mathrm{miss}}(x_{np}))$$

- Scaled Mean Square Error of missing p_T
- Forces energy conservation between the pure and full samples

TOTAL: Training Optimal Transport with Attention Learning

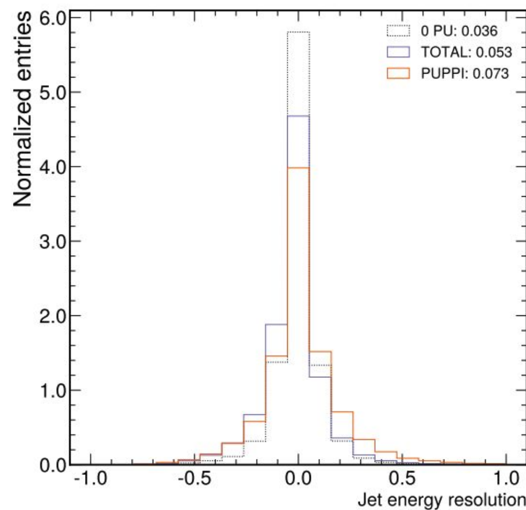


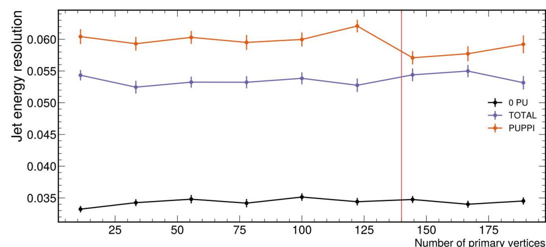
TOTAL: Training Optimal Transport with Attention Learning



TOTAL: Training Optimal Transport with Attention Learning

+ Outperforms traditional and ML-based alternatives
+ Relies on global event descriptions
+ Robustly learns pileup characteristics as a transport function



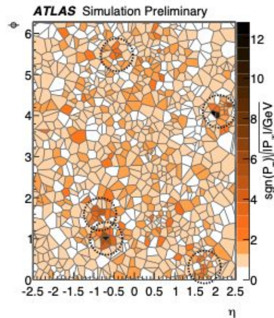


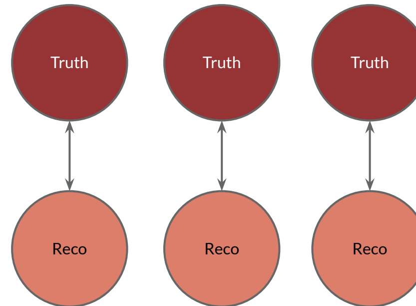
(arXiv:2211.02029)

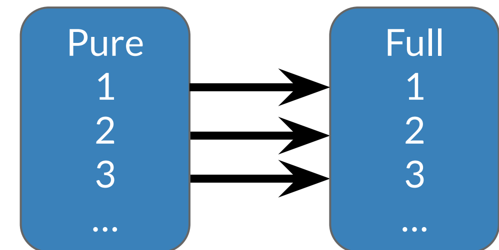
- Requires direct matching of events
- Overall limited due to supervision

TOTAL: Training Optimal Transport with Attention Learning

Rule-based Competitors (PUPPI, SK, etc.)	ML Competitors	TOTAL
• Reliant on an assumptions of pileup nature (MC correction)	• Matching between truth and reco at particle level (MC correction)	• Matching between pileup events and same event without pileup vertices (data-driven*)

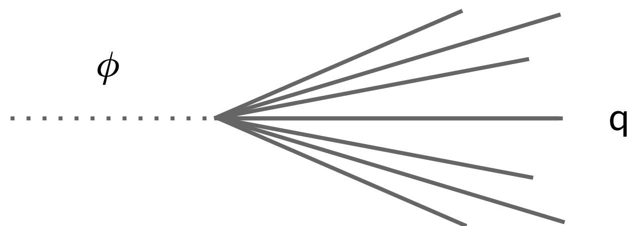






For more information, check out:
https://arxiv.org/pdf/2211.02029

Tutorial Example: High p_T Jets



- PUMML Dataset: https://zenodo.org/records/2652034
- Datasets
  - *mH_Mu140: Set pileup vertex count, varied scalar mass*
  - **Mu_mH500: Varied pileup vertex (PV) count, set scalar mass**

Tutorial directions (OT)

- Hop into https://jupyter.nersc.gov/
- Request a "Configurable Job" and choose:
  a. Account: **ntrain1**
  b. Reservation: **ml4fp2024_day4**
  c. **cpus=32, gpus=1, ntasks-per-node=1**
  d. Make sure you use QOS = shared
- **Hit start**
- Select kernel: tensorflow-2.9.0

Server Options

Account ("_g" suffix will be added as needed):

ntrain1

Constraint:

gpu

QOS:

shared

cpus-per-task (node has 128 cpus):

32

gpus-per-task (node has 4 GPUs):

1

nodes (maximum of 4 for jupyter QOS):

1

ntasks-per-node:

1

Reservation:

ml4fp2024_day3

time (time limit in minutes):

300

Backup Slides

Constituent Subtraction

- Benefits
  - Efficient at identifying potential pileup clusters
- Drawbacks
  - Reliant on an assumption of uniform pileup energy density

1. Apply ghost association on jet input objects ($|\eta| < 2.0$) with $p_T^g = A_g \times \rho$ where

   p_T^g: Expected pileup radiation contribution in a $\Delta\eta\Delta\varphi = 0.01$ region
   $A_g = 0.01$: Area of the ghosts
   ρ: Pileup energy density estimated as the median of the p_T/A distribution of the $R = 0.4 k_t$ jets

2. Calculate the distance between each i cluster and k ghost

$$\Delta R_{i,k} = \sqrt{(\eta_i - \eta_k)^2 + (\varphi_i - \varphi_k)^2}$$

3. Sort cluster-ghost pairs in ascending $\Delta R_{i,k}$ order and update p_T values appropriately

$$\begin{aligned} \text{If } p_{T,i} \geq p_{T,k} : p_{T,i} &\rightarrow p_{T,i} - p_{T,k} \\ p_{T,k} &\rightarrow 0 \\ \text{Else: } p_{T,k} &\rightarrow p_{T,k} - p_{T,i} \\ p_{T,i} &\rightarrow 0 \end{aligned}$$

Nathan Suri, 2024

PUPPI

- Assigns a likelihood of candidate particle originating from a pileup interaction based on kinematic properties and proximity to charged hard-scatter particles from the PV
- (CERN-EP-2020-134)

$$\alpha_i = \log\left(\sum_j \frac{p_T^j}{\Delta R^{ij}} \times \Theta(R_{min} \leq \Delta R_{ij} \leq R_0)\right) \tag{1}$$

where

j: Charged inputs matched to the PV
$R_0 \approx 0.3$: Maximum radial distance at which inputs may be matched to each other
$R_{min} \approx 0.001$: Minimum radial distance of matching
ΔR^{ij}: Angular distance between an input and a charged hard-scatter particle

$$\chi_i^2 = \Theta(\alpha_i - \overline{\alpha}_{PU}) \times \frac{(\alpha_i - \overline{\alpha}_{PU})^2}{\sigma_{PU}^2} \tag{2}$$

where

$\overline{\alpha}_{PU}$: Mean value of α for all charged pileup input objects in the event
$\sigma_{PU} \approx 0.3$: RMS of aforementioned distribution

$$w_i = F_{\chi^2,\mathrm{NDF}=1}(\chi_i^2) \tag{3}$$

where

F_{χ^2}: Cumulative distribution of χ^2, eliminating all neutral inputs i where $\alpha_i < \overline{\alpha}_{PU}$

Nathan Suri, 2024

Data assumed to be more than one-dimensional, necessitating projections

For data more than one-dimensional, perform multiple random projection to 1-D

```
def SWD(y_true, y_pred,nprojections=128):
    pu_pfs = y_true[:,:,:y_true.shape[2]//2]
    nopu_pfs = y_true[:,:,y_true.shape[2]//2:]

    nopu_pfs = nopu_pfs
    pu_pfs = pu_pfs*y_pred



    def _getSWD(pu_pf,nopu_pf):
        proj = tf.random.normal(shape=[tf.shape(pu_pf)[0],tf.shape(pu_pf)[2], nprojections])
        proj *= tf.math.rsqrt(tf.reduce_sum(tf.square(proj), 1, keepdims=True))

        p1 = tf.matmul(nopu_pf, proj) #BxNxNPROJ
        p2 = tf.matmul(pu_pf, proj) #BxNxNPROJ
        p1 = sort_rows(p1, tf.shape(pu_pf)[1])
        p2 = sort_rows(p2, tf.shape(pu_pf)[1])

        wdist = tf.reduce_mean(tf.square(p1 - p2),-1)
        return wdist

    def _getMET(particles):
        px = tf.abs(particles[:,:,2])*tf.math.cos(particles[:,:,1])
        py = tf.abs(particles[:,:,2])*tf.math.sin(particles[:,:,1])
        met = tf.stack([px,py],-1)
        # print(met)
        return met

    #MET Loss, not used atm
    met_pu = tf.reduce_sum(_getMET(pu_pfs),1)
    met_nopu = tf.reduce_sum(_getMET(nopu_pfs),1)
    met_mse = tf.reduce_sum(tf.square(met_pu[:,:2] - met_nopu[:,:2]),-1)


    wdist = _getSWD(pu_pfs,nopu_pfs)
    return 1e3*tf.reduce_mean(wdist)
```

The same projection is applied for each particle in an event within a batch

The particles are sorted based on the projection and then the non-pileup and pileup versions are compared using the defined cost function

```python
def SWD(y_true, y_pred,nprojections=128):
    pu_pfs = y_true[:,:,:y_true.shape[2]//2]
    nopu_pfs = y_true[:,:,y_true.shape[2]//2:]

    nopu_pfs = nopu_pfs
    pu_pfs = pu_pfs*y_pred



    def _getSWD(pu_pf,nopu_pf):
        proj = tf.random.normal(shape=[tf.shape(pu_pf)[0],tf.shape(pu_pf)[2], nprojections])
        proj *= tf.math.rsqrt(tf.reduce_sum(tf.square(proj), 1, keepdims=True))

        p1 = tf.matmul(nopu_pf, proj) #BxNxNPROJ
        p2 = tf.matmul(pu_pf, proj) #BxNxNPROJ
        p1 = sort_rows(p1, tf.shape(pu_pf)[1])
        p2 = sort_rows(p2, tf.shape(pu_pf)[1])

        wdist = tf.reduce_mean(tf.square(p1 - p2),-1)
        return wdist

    def _getMET(particles):
        px = tf.abs(particles[:,:,2])*tf.math.cos(particles[:,:,1])
        py = tf.abs(particles[:,:,2])*tf.math.sin(particles[:,:,1])
        met = tf.stack([px,py],-1)
        # print(met)
        return met

    #MET Loss, not used atm
    met_pu = tf.reduce_sum(_getMET(pu_pfs),1)
    met_nopu = tf.reduce_sum(_getMET(nopu_pfs),1)
    met_mse = tf.reduce_sum(tf.square(met_pu[:,:2] - met_nopu[:,:2]),-1)


    wdist = _getSWD(pu_pfs,nopu_pfs)
    return 1e3*tf.reduce_mean(wdist)
```