Generation of High-Energy Particle Collisions using Generative Adversarial Particle Transformers

Anni Li
Mentors: Javier Duarte, Raghav Kansal

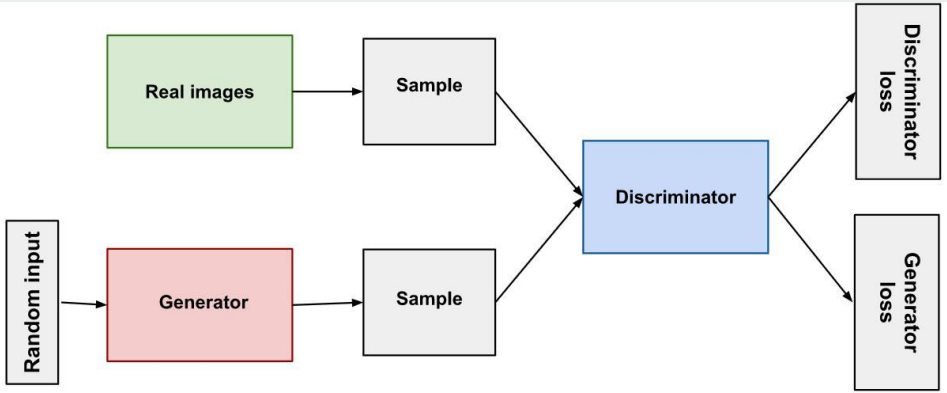
University of California, San Diego
Department of Physics

UC San Diego

Motivation & Introduction

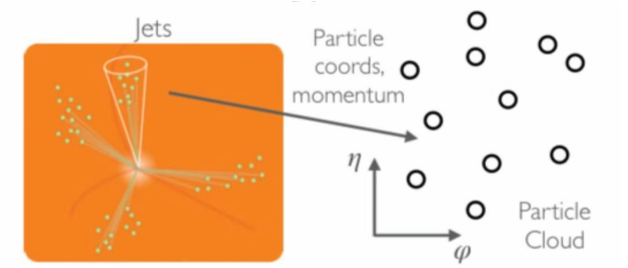
- High-energy particle collisions at the CERN LHC provide rich data, often containing **jets**, for high-energy physics research.
- Jets are **sparse** distributions of particles that have **complex patterns**, which we can use to identify rare particles, such as the Higgs bosons .
- Traditional simulations of such collisions consume a significant and increasingly **unsustainable amount of computational resources** at CERN, hence new machine learning methods are being explored to both improve and speed up these simulations.
- **Graph neural networks** (GNNs) have been found to be successful for a number of computational tasks in high energy physics, including generating realistic jets [1].
- **My project**: We implemented a **transformer-based generative adversarial network**, which improved both the **speed** and the **fidelity** of the generation.

Generative Adversarial Networks (GANs)



- A popular generative model in machine learning, include a generator and a discriminator
- Generator: Produces the simulations
- Discriminator: Tries to distinguish between real and generated simulations
- The generator and discriminator are trained together adversarially. By improving the discriminator's ability to distinguish, we can simultaneously improve the generator's ability to produce models closer to real jets

Graph Neural Networks (GNNs)



• A class of neural network for processing data that best represented by graph data structures. [2]

• Can be directly applied to graphs, and provide an easy way to do **node-level, edge-level, and graph-level prediction tasks**

• In our project, GNNs is used to be **graphical representation describes jets as nodes & edges, and combine them as a network**

• **Less computational complexity, high efficiency with more information**

• Since high energy jets are **sparse** and have **irregular shapes**, using graph neural networks would save both **time** and **space** than using images to feed the GAN model

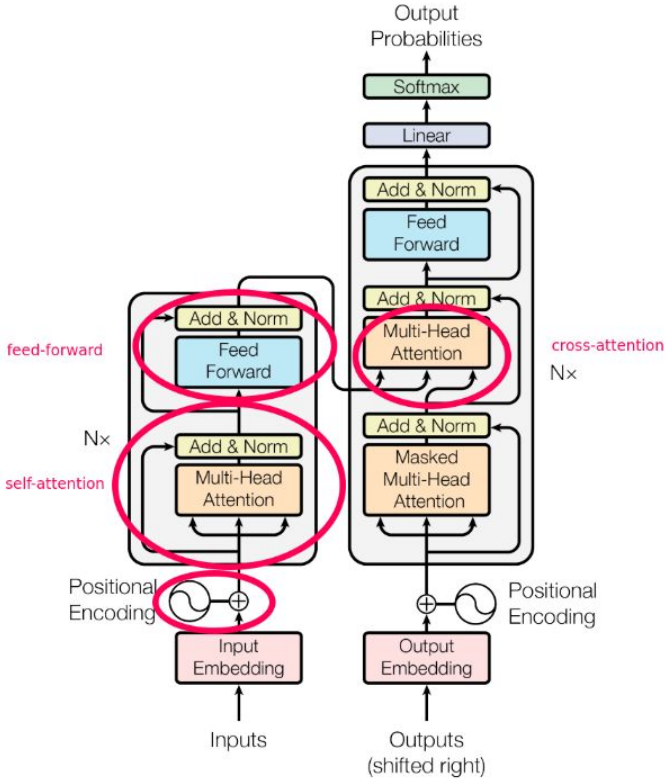
Transformers

• Commonly used in language translation, conversational chatbots, text generation, or search engines, etc

• The attention mechanism enables the transformers to have a **long-time memory**, focussing on all previous tokens that have been generated [3]

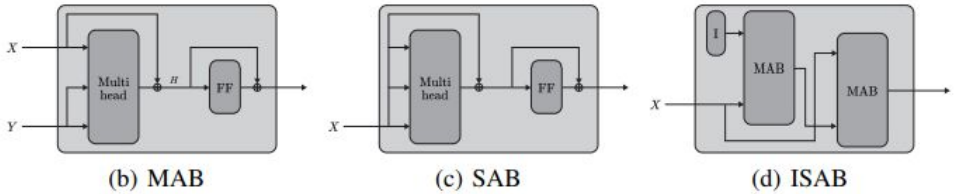
• We mainly takes advantage of the **multihead self-attention** mechanism in the transformers

Multiple self-attention heads composite a layer called multi-head attention layer, allowing each head to focus on a different subspace and modify the corresponding output sequence



https://vaclavkosar.com/ml/transformers-self-attention-mechanism-simplified

Set Transformer & GAPT



•Set transformer[4] uses sets instead of sentence tokens as input, so it can be applied to jets; Compared with other methods for set-structured data, set transformers can reduce computational complexity and have shown better performances.

•We implemented set transformer into our previous GAN model, creating a **generative adversarial particle transformer (GAPT):**

$$\mathrm{ISAB}_m(X) = \mathrm{MAB}(X, H) \in \mathbb{R}^{n \times d},$$
$$\text{where } H = \mathrm{MAB}(I, X) \in \mathbb{R}^{m \times d}.$$

-MAB: Multihead Attention Block, a basic block used for attention-based set operations

-SAB: Set Attention Block, SAB = MAB(X, X) where X is the input n (number of elements in set) × d (dimension) matrix.

-ISAB: Induced Set Attention Block, by defining a induced vector I, it reduces the quadratic time complexity for large sets.

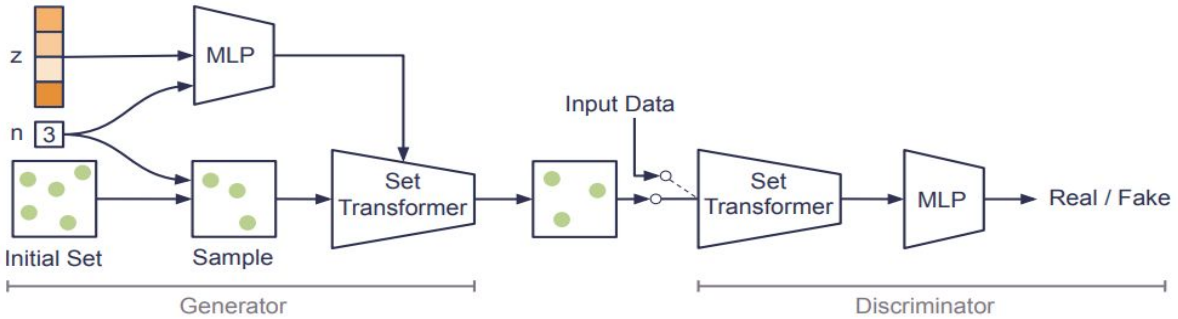
•We used MNIST as dataset to test the time and performance of this algorithm first, then moving to JetNet dataset [5]

Juho Lee et.al, Set Transformer: A Framework for Attention-based Permutation-Invariant Neural Networks

Implementing Set Transformers - Mechanism

•The generator takes the initial set of element, x, the noise z, and the number of elements n. It filters the initial set to a desired n-sized sample, processes z to a multilayer perceptron, then feeds them to our set transformer to produce the output. [6]

•Discriminator takes the output and the input data to its set transformer, then through an MLP to generate the final score.



Karl Stelzner et.al, Set Transformer: A Framework for Attention-based Permutation-Invariant Neural Networks

Implementing Set Transformers - Code

```python
class MAB(nn.Module):
    def __init__(
        self,
        embed_dim: int,
        num_heads: int,
        ff_layers: list = [],
        layer_norm: bool = False,
        dropout_p: float = 0.0,
        final_linear: bool = True,
        linear_args={},
    ):
        super(MAB, self).__init__()

        self.num_heads = num_heads
        self.attention = nn.MultiheadAttention(embed_dim, num_heads, batch_first=True)
        self.ff = LinearNet(
            ff_layers,
            input_size=embed_dim,
            output_size=embed_dim,
            final_linear=final_linear,
            **linear_args,
        )

        self.layer_norm = layer_norm

        if self.layer_norm:
            self.norm1 = nn.LayerNorm(embed_dim)
            self.norm2 = nn.LayerNorm(embed_dim)

        self.dropout = nn.Dropout(p=dropout_p)
```

```python
    def forward(self, x: Tensor, y: Tensor, y_mask: Tensor = None):
        if y_mask is not None:
            # torch.nn.MultiheadAttention needs a mask of shape [batch_size * num_heads, N, N]
            y_mask = torch.repeat_interleave(y_mask, self.num_heads, dim=0)

        x = x + self.attention(x, y, y, attn_mask=y_mask, need_weights=False)[0]
        if self.layer_norm:
            x = self.norm1(x)
        x = self.dropout(x)

        x = x + self.ff(x)
        if self.layer_norm:
            x = self.norm2(x)
        x = self.dropout(x)

        return x
```

https://github.com/juho-lee/set_transformer

Implementing Set Transformers to GAN - Code

```python
class SAB(nn.Module):
    def __init__(self, **mab_args):
        super(SAB, self).__init__()
        self.mab = MAB(**mab_args)

    def forward(self, x: Tensor, mask: Tensor = None):
        if mask is not None:
            # torch.nn.MultiheadAttention needs a mask vector for each target node
            # i.e. reshaping from [B, N, 1] -> [B, N, N]
            mask = mask.transpose(-2, -1).repeat((1, mask.shape[-2], 1))

        return self.mab(x, x, mask)

# Adapted from https://github.com/juho-lee/set_transformer/blob/master/modules.py
class ISAB(nn.Module):
    def __init__(self, num_inds, embed_dim, **mab_args):
        super(ISAB, self).__init__()
        self.I = nn.Parameter(torch.Tensor(1, num_inds, embed_dim))
        nn.init.xavier_uniform_(self.I)
        self.mab0 = MAB(embed_dim=embed_dim, **mab_args)
        self.mab1 = MAB(embed_dim=embed_dim, **mab_args)

    def forward(self, X, mask: Tensor = None):
        H = self.mab0(self.I.repeat(X.size(0), 1, 1), X)
        return self.mab1(X, H)
```

https://github.com/juho-lee/set_transformer

```python
class GAPT_G(nn.Module):
    def __init__(
        self,
        num_particles: int,
        output_feat_size: int,
        sab_layers: int = 2,
        num_heads: int = 4,
        embed_dim: int = 32,
        sab_fc_layers: list = [],
        layer_norm: bool = False,
        dropout_p: float = 0.0,
        final_fc_layers: list = [],
        use_mask: bool = True,
        use_isab: bool = False,
        num_isab_nodes: int = 10,
        linear_args: dict = {},
    ):
        super(GAPT_G, self).__init__()
        self.num_particles = num_particles
        self.output_feat_size = output_feat_size
        self.use_mask = use_mask

        self.sabs = nn.ModuleList()

        sab_args = {
            "embed_dim": embed_dim,
            "ff_layers": sab_fc_layers,
            "final_linear": False,
            "num_heads": num_heads,
            "layer_norm": layer_norm,
            "dropout_p": dropout_p,
            "linear_args": linear_args,
        }

        # intermediate layers
        for _ in range(sab_layers):
            self.sabs.append(SAB(**sab_args) if not use_isab else ISAB(num_isab_nodes, **sab_args))

        self.final_fc = LinearNet(
            final_fc_layers,
            input_size=embed_dim,
            output_size=output_feat_size,
            final_linear=True,
            **linear_args,
        )

    def forward(self, x: Tensor, labels: Tensor = None):
        if self.use_mask:
            # unnormalize the last jet label - the normalized # of particles per jet
            # (between 1/``num_particles`` and 1) - to between 0 and ``num_particles`` - 1
            num_jet_particles = (labels[:, -1] * self.num_particles).int() - 1
            # sort the particles bythe first noise feature per particle, and the first
            # ``num_jet_particles`` particles receive a 1-mask, the rest 0.
            mask = (
                (x[:, :, 0].argsort(1).argsort(1) <= num_jet_particles.unsqueeze(1))
                .unsqueeze(2)
                .float()
            )
            logging.debug(
                f"x \n {x[:2, :, 0]} \n num particles \n {num_jet_particles[:2]} \n gen mask \n {mask[:2]}"
            )
        else:
            mask = None

        for sab in self.sabs:
            x = sab(x, _attn_mask(mask))

        x = torch.tanh(self.final_fc(x))

        return torch.cat((x, mask - 0.5), dim=2) if mask is not None else x
```

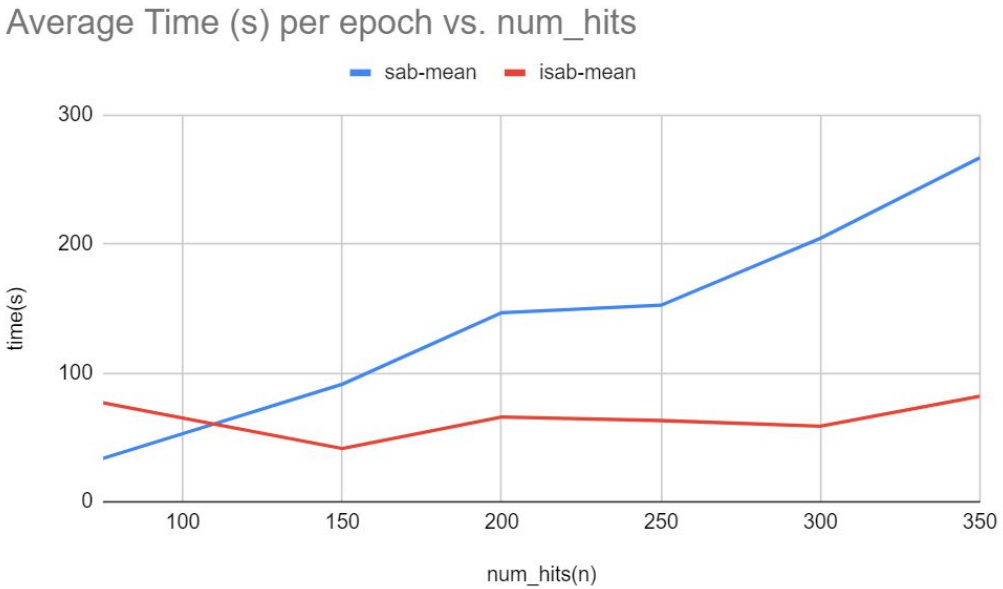
```python
class GAPT_D(nn.Module):
    def __init__(
        self,
        num_particles: int,
        input_feat_size: int,
        sab_layers: int = 2,
        num_heads: int = 4,
        embed_dim: int = 32,
        sab_fc_layers: list = [],
        layer_norm: bool = False,
        dropout_p: float = 0.0,
        final_fc_layers: list = [],
        use_mask: bool = True,
        use_isab: bool = False,
        num_isab_nodes: int = 10,
        linear_args: dict = {},
    ):
        super(GAPT_D, self).__init__()
        self.num_particles = num_particles
        self.input_feat_size = input_feat_size
        self.use_mask = use_mask

        self.sabs = nn.ModuleList()

        sab_args = {
            "embed_dim": embed_dim,
            "ff_layers": sab_fc_layers,
            "final_linear": False,
            "num_heads": num_heads,
            "layer_norm": layer_norm,
            "dropout_p": dropout_p,
            "linear_args": linear_args,
        }

        self.input_embedding = LinearNet(
            [], input_size=input_feat_size, output_size=embed_dim, **linear_args
        )

        # intermediate layers
        for _ in range(sab_layers):
            self.sabs.append(SAB(**sab_args) if not use_isab else ISAB(num_isab_nodes, **sab_args))

        self.pma = PMA(
            num_seeds=1,
            **sab_args,
        )

        self.final_fc = LinearNet(
            final_fc_layers,
            input_size=embed_dim,
            output_size=1,
            final_linear=True,
            **linear_args,
        )

    def forward(self, x: Tensor, labels: Tensor = None):
        if self.use_mask:
            mask = x[..., -1:] + 0.5
            x = x[..., :-1]
        else:
            mask = None

        x = self.input_embedding(x)

        for sab in self.sabs:
            x = sab(x, _attn_mask(mask))

        return torch.sigmoid(self.final_fc(self.pma(x, _attn_mask(mask)).squeeze()))
```

The code is open to public for future use.

Link to repo: https://github.com/rkansal47/MPGAN/tree/development

Results - MNIST, time

- Running GAPT with both SAB and ISAB to compare the time



- Try to vary number of induced nodes for ISAB

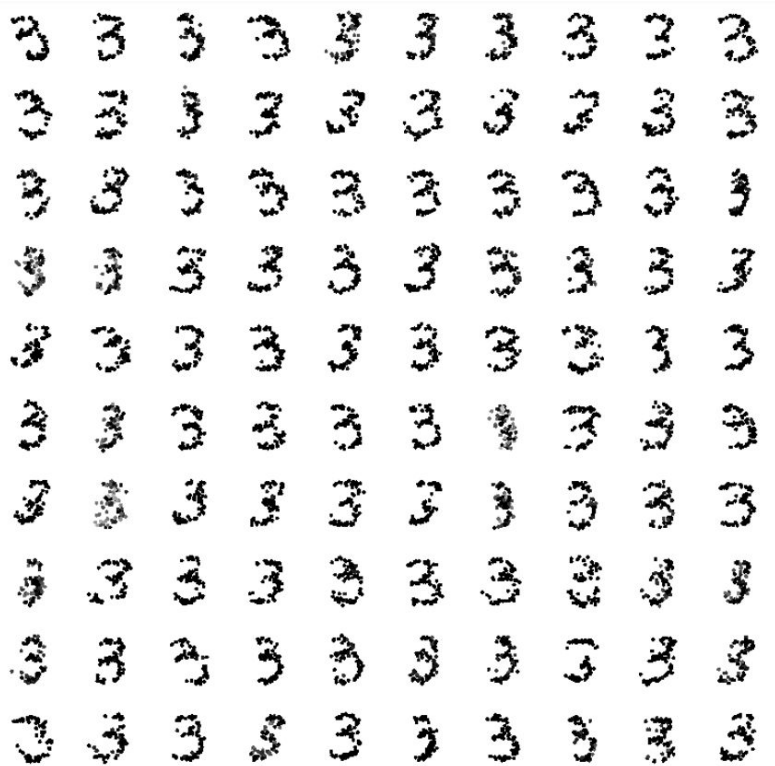
n=75, isab		
m	isab-mean	isab-total
10	9.8	8hr57min15s
20	3.6	4hr58s
30	3.6	4hr8s
40	5.4	6hr15min2s
50	7.6	6hr52min32s

Results - MNIST, performance

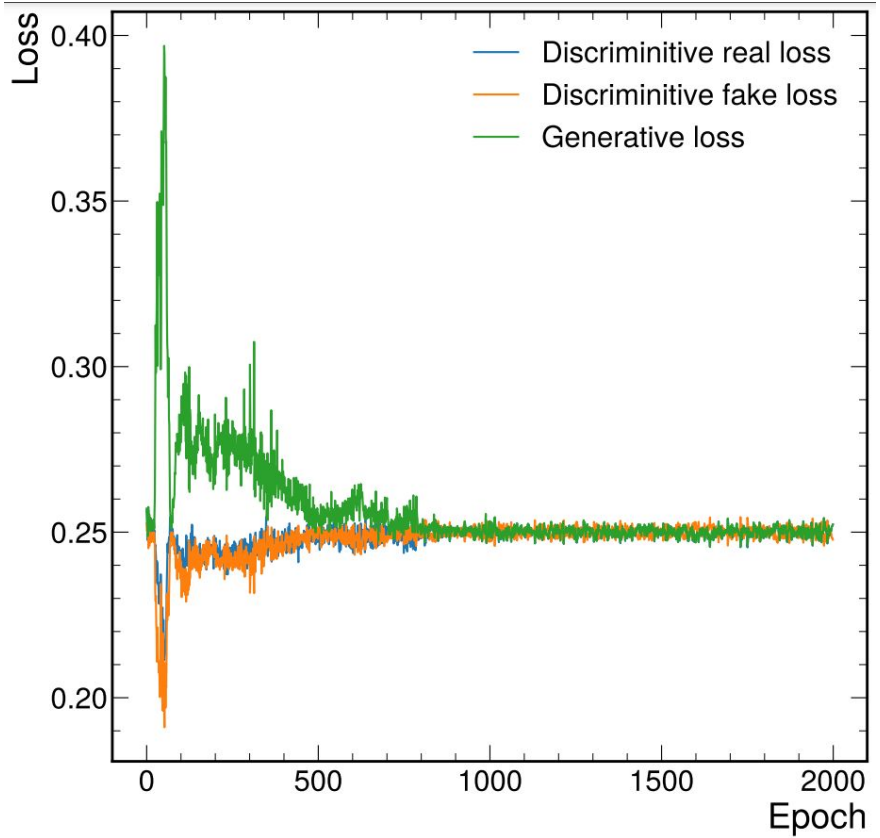
- Try to vary number of induced nodes for ISAB

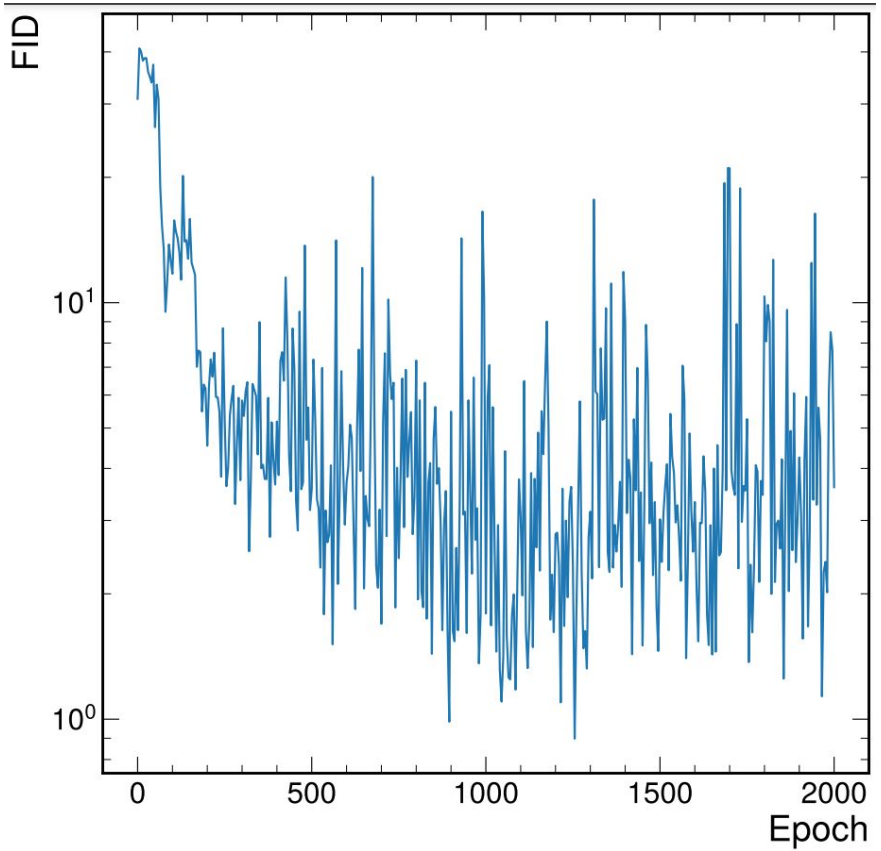
n=75			
isab_num_nodes	Best epoch fid	Best epoch loss_D	Best epoch loos_G
m=10	0.335	0.497	0.259
m=20	0.634	0.499	0.247
m=30	0.898	0.5	0.248
m=40	0.605	0.491	0.269
m=50	1.113	0.499	0.249

- Best epoch output for m=20



- Loss curves and fidelity for GAPT with isab_num_nodes = 20





Results - Jets, time

- Running GAPT and MPGAN [5] on Jets dataset with num_hits = 30, num_isab_nodes = 10

- Message Passing GAN (MPGAN) is a model developed by our groups previously.

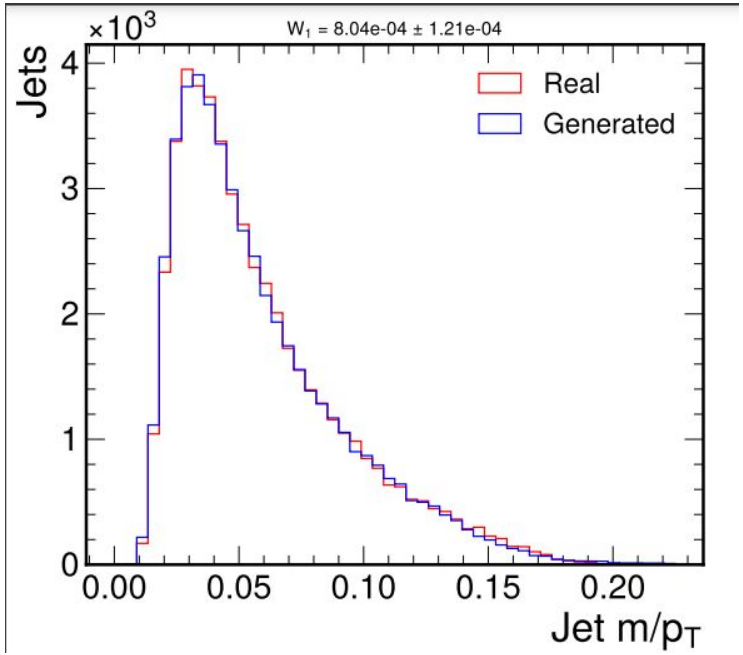
- Link to MPGAN paper: https://arxiv.org/abs/2106.11535

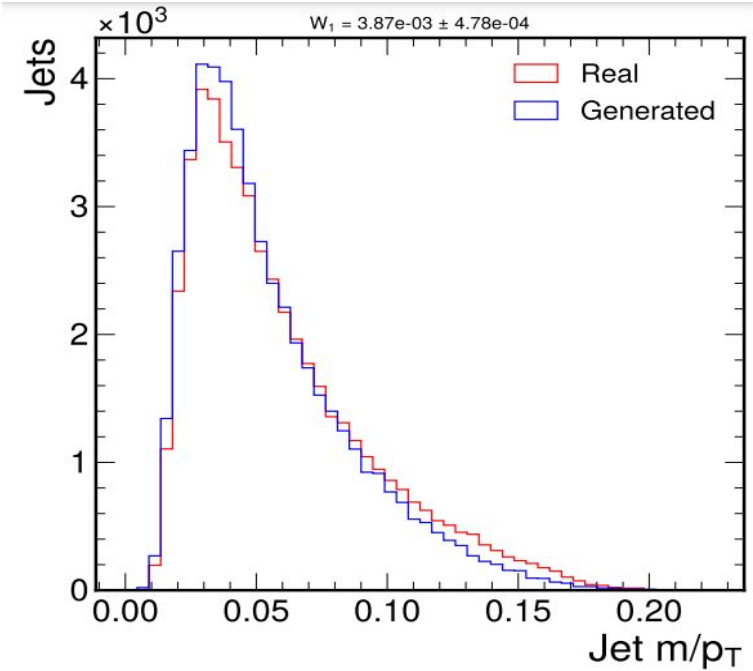
R. Kansal, J. Duarte, H. Su et.al, Particle Cloud Generation with Message Passing Generative Adversarial Networks

	GAPT	MPGAN
Average time per epoch (s)	28	190
Best epoch loss_D	0.488	0.491
Best epoch loss_G	0.284	0.269

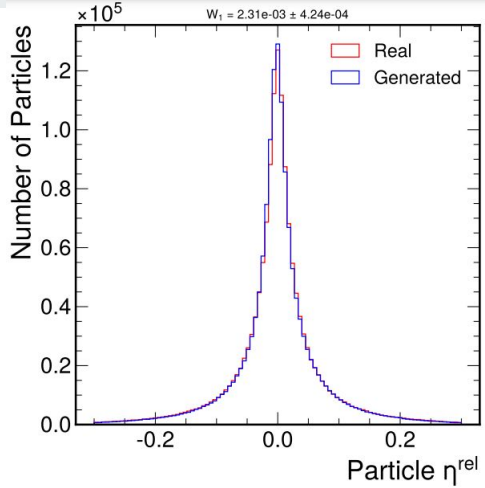
Results - Jets, performance

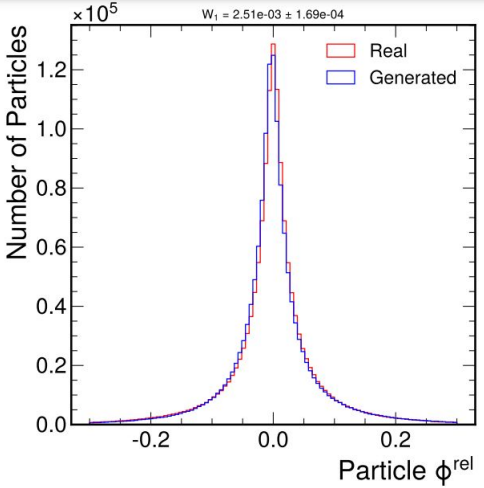
Generation of Jets

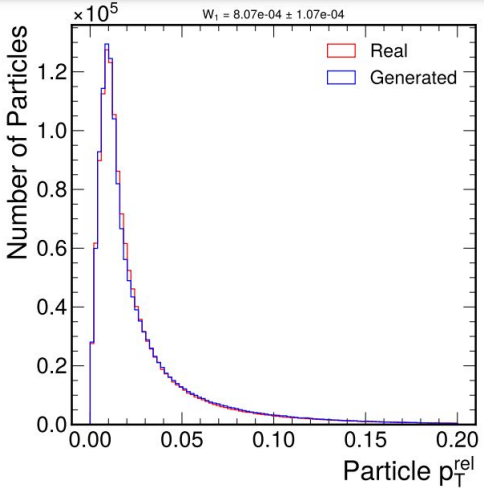


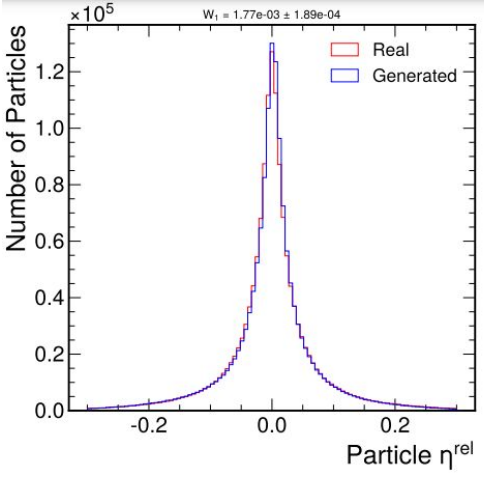


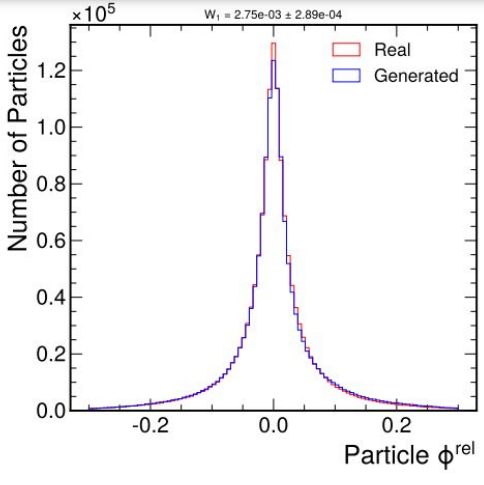
Generation of particles

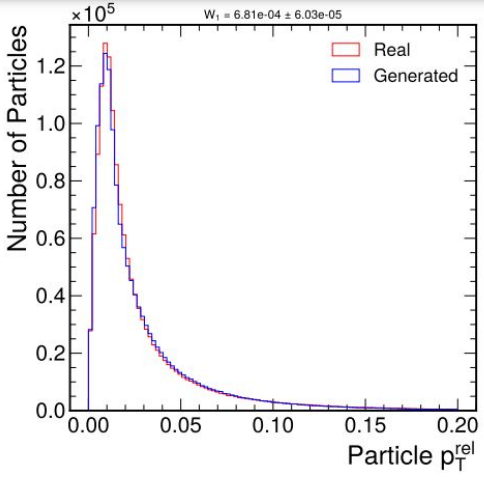




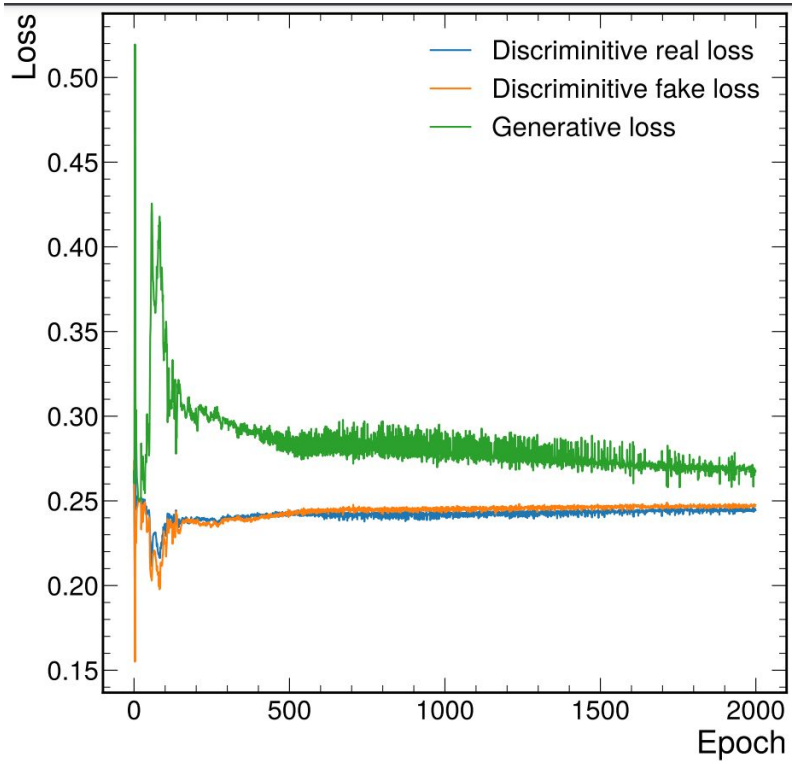


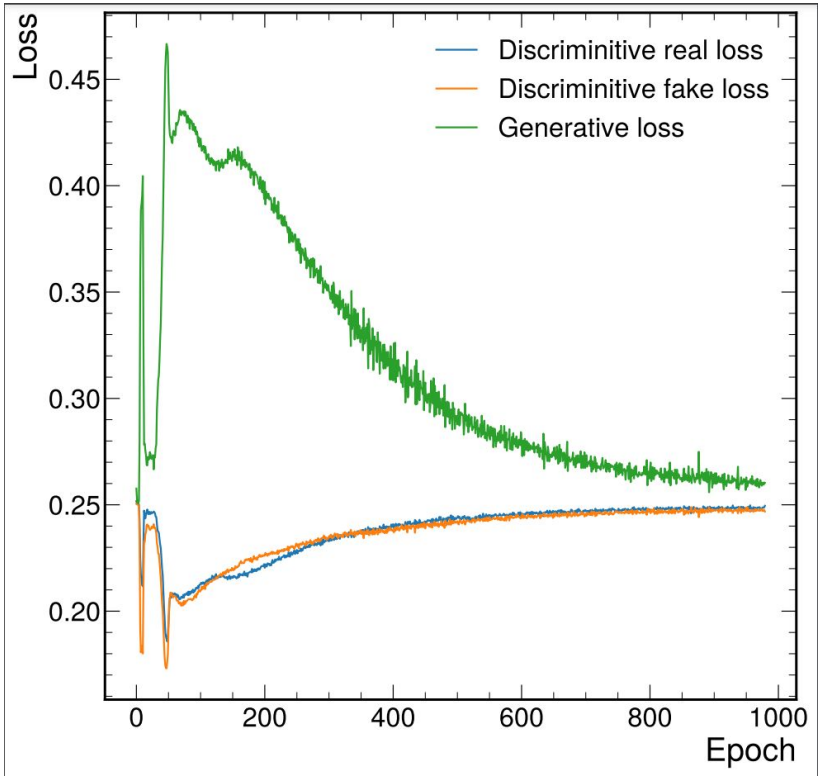




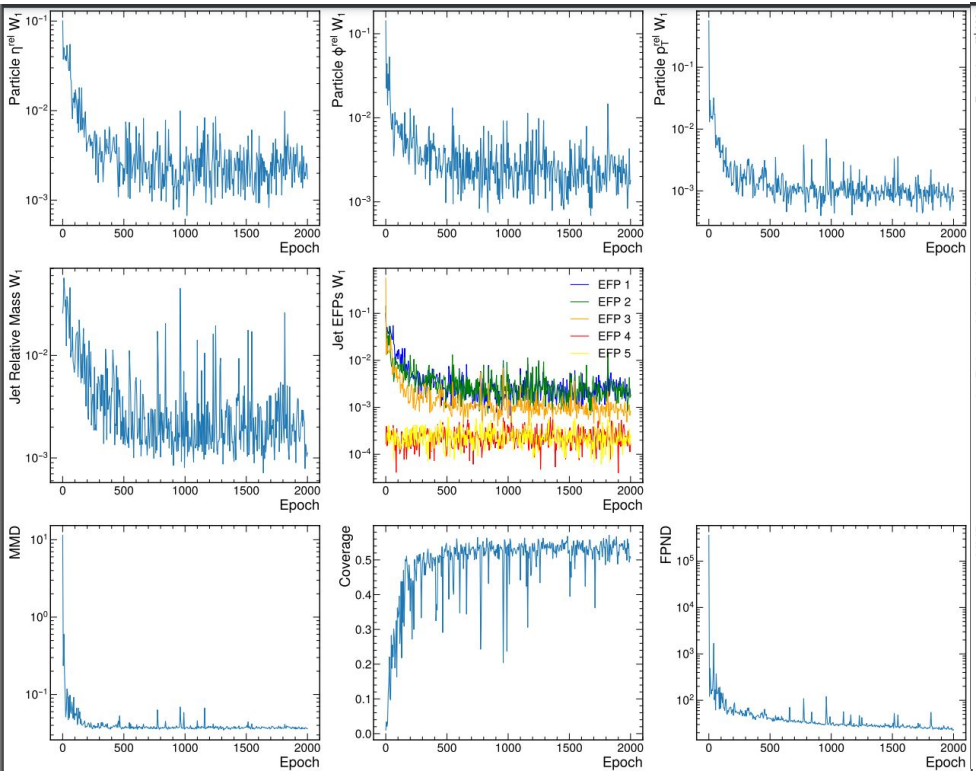


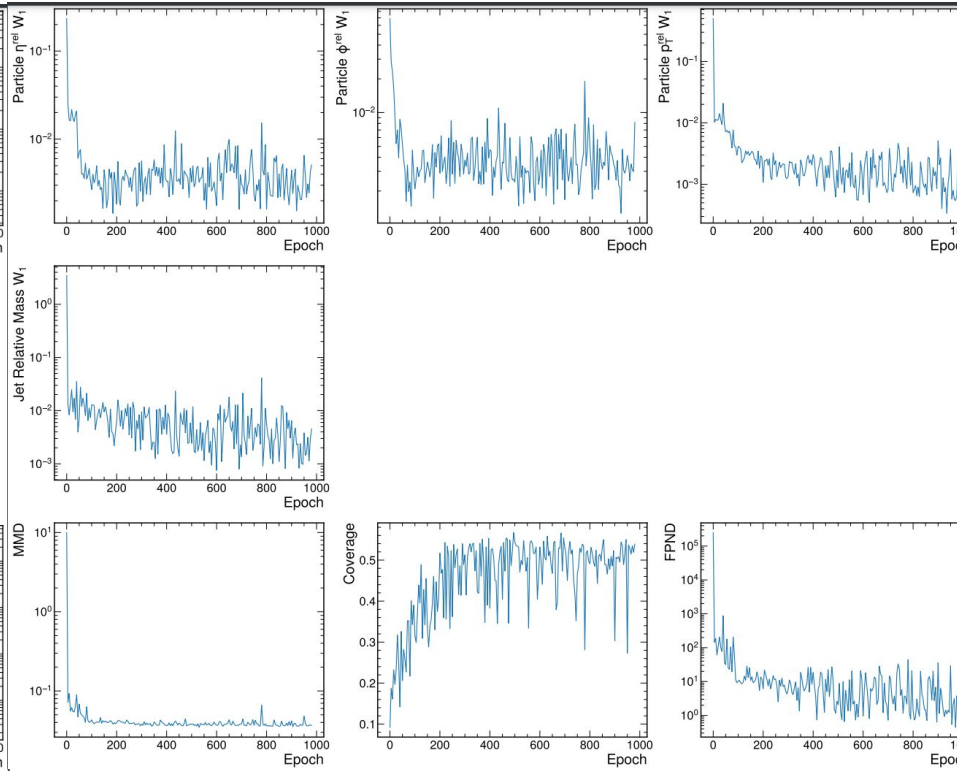
Loss curves for MPGAN (total epoch=2000) and GAPT (total epoch=1000)





Evaluation for MPGAN and GAPT





Conclusion

- In this project, we implemented set transformers architecture in generative adversarial networks to generate jets, constructing a promising model called **GAPT (Generative Adversarial Particle Transformer)**. Based on MNIST dataset, GAPT **reduces the time** it takes as well as maintains a **good performance**. For our tests on jets dataset, GAPT is also **significantly faster** than our previous models and has a **reliable quality** of the output.
- **Some future steps:**
  - Finish tests varying num_hits and isab_num_nodes to explore deeper on jets generation
  - Adjust codes and tune the model to further improve the overall quality of result
  - Combine GAPT with conditional GAN, which can conditionally generate jets with desired features
  - Refining the repo and add necessary instructions or comments in codes for public use in experimental collaborations

Thank You!

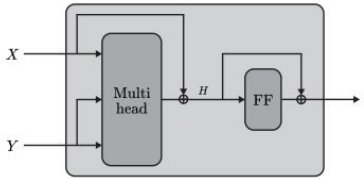
- Special thanks to my mentors Javier Duarte and Raghav Kansal.
- All IRIS HEP members
- HSF training team

References

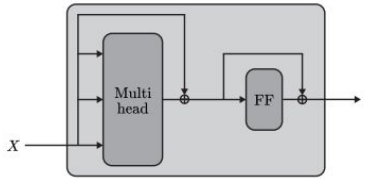
1. J. Duarte and J.-R. Vlimant, “Graph neural networks for particle tracking and reconstruction”, in Artificial Intelligence for High Energy Physics, P. Calafiura, D. Rousseau, and K. Terao, eds. 11 World Scientific Publishing, 12, 2020. arXiv:2012.01249. Submitted to Int. J. Mod. Phys. A. doi:10.1142/12200.
2. J. Shlomi et al., Graph Neural Networks in Particle Physics, 2020, arXiv:2007.13681
3. https://towardsdatascience.com/illustrated-guide-to-transformers-step-by-step-explanation-f74876522bc0
4. Juho Lee, Yoonho Lee, Jungtaek Kim, Adam R. Kosiorek, Seungjin Choi, Yee Whye Teh, Set Transformer: A Framework for Attention-based Permutation-Invariant Neural Networks
5. R. Kansal et al., Particle Cloud Generation with Message Passing Generative Adversarial Networks, 2021, arXiv:2106.1153
6. Karl Stelzner, Kristian Kersting, Adam R. Kosiorek, Set Transformer: A Framework for Attention-based Permutation-Invariant Neural Networks
7. https://github.com/rkansal47/MPGAN/blob/development
8. https://github.com/juho-lee/set_transformer
9. https://github.com/DLii-Research/tf-gast

$$\mathrm{SAB}(X) := \mathrm{MAB}(X, X)$$

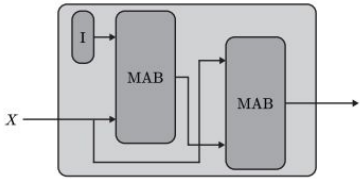
$$\mathrm{ISAB}_m(X) = \mathrm{MAB}(X, H) \in \mathbb{R}^{n \times d},$$
$$\text{where } H = \mathrm{MAB}(I, X) \in \mathbb{R}^{m \times d}.$$



(b) MAB



(c) SAB



(d) ISAB

Backup - Explanation for Set Transformers in Detail

MAB: Multihead Attention Block, a basic block used for attention-based set operations (nxd X,Y)

$$\mathrm{MAB}(X, Y) = \mathrm{LayerNorm}(H + \mathrm{rFF}(H)) \text{ where } H = \mathrm{LayerNorm}(X + \mathrm{Multihead}(X, Y, Y; \omega))$$

ω is the parameter of the NN block, rFF is any row-wise feedforward layer (i.e., it processes each instance independently and identically), and LayerNorm is layer normalization. The MAB is an adaptation of the encoder block of the Transformer without positional encoding and dropout.

SAB: Set Attention Block, SAB = MAB(X, X) where X is the input n (number of elements in set) × d (dimension) matrix. SAB takes a set and performs self-attention between the elements in the set, resulting in a set of equal size. Since the output of SAB contains information about pairwise interactions among the elements in the input set X, we can stack multiple SABs to encode higher order interactions. Potential problem: quadratic time complexity.

ISAB: Induced Set Attention Block, reduces the quadratic time complexity for large sets. Along with the set n×d X, additionally we defined a m×d vector I, which we call inducing points. The ISAB first transforms I into H by attending to the input set. The set of transformed inducing points H, which contains information about the input set X, is again attended to by the input set X to finally produce a set of n elements. The time complexity then is reduced to linear.

http://proceedings.mlr.press/v97/lee19d/lee19d.pdf

Backup - JetNet Evaluation Metrics

`Jetnet.evaluation.cov_mmd:` Calculate coverage and MMD between real and generated jets, using the Energy Mover's Distance as the distance metric.

`Jetnet.evaluation.fpnd:` Calculates the Frechet ParticleNet Distance, as defined in https://arxiv.org/abs/2106.11535

`Jetnet.evaluation.w1efp:` Get 1-Wasserstein distances between Energy Flow Polynomials (Komiske et al. 2017 https://arxiv.org/abs/1712.07124)

`Jetnet.evaluation.w1m:` Get 1-Wasserstein distance between masses of `jets1` and `jets2`.

`Jetnet.evaluation.w1p:` Get 1-Wasserstein distances between particle features of `jets1` and `jets2`.

https://jetnet.readthedocs.io/en/latest/pages/metrics.html

Backup - Terminology

MLP: multilayer perceptron is a fully connected class of feedforward artificial neural network (ANN). The term MLP is used ambiguously, sometimes loosely to mean any feedforward ANN, sometimes strictly to refer to networks composed of multiple layers of perceptrons (with threshold activation).

MMD: Maximum Mean Discrepancy is a distance on the space of probability measures which has found numerous applications in machine learning and nonparametric testing. This distance is based on the notion of embedding probabilities in a reproducing kernel Hilbert space.

https://en.wikipedia.org/wiki/Multilayer_perceptron