```
#%% md
### Dataset consists of 4 keys ->

1.) X_jets (Images with 3 channels, and size 125 * 125) <br>
2.) m0 (Mass) <br>
3.) pt (Transverse momentum) <br>
4.) y (Labels) <br>

In state-of-the-art Jet Image Taggers, 4-tuple
#%% md
For this screening task, as asked I am constructing an Autoencoder (Variational) to act upon Jet Images
#%%
from google.colab import drive
drive.mount('/content/drive')
#%%
import h5py
import numpy as np
filename = '/content/drive/MyDrive/quark-gluon_data-set_n139306.hdf5'

def load_h5(file_name, size):
    # Load the dataset from the HDF5 file
    with h5py.File(file_name, 'r') as f:
        print("The keys are : ", list(f.keys()))
        print("The number of images in dataset : ", len(f['X_jets']))
        print("Dimensions of image tensor : ", f['X_jets'].shape[1:])
        X = np.array(f['X_jets'][:size])
        y = np.array(f['y'][:size])
    return X, y
#%%
X, y = load_h5(filename, 10000)
#%%
def data_info_getter(X, y):

    print("Max value of intensity along 1st channel : ", np.max(X[:,:,:,0]), "   Min value : ", np.min(X[:,:,:,0]))
```

```
35      print("Mean intensity value along 1st channel : "
   , np.mean(X[:,:,:,0]))
36      print("Standard Deviation : ", np.std(X[:,:,:,0
   ]), "\n\n")
37
38      print("Max value of intensity along 2nd channel
    : ", np.max(X[:,:,:,1]), "   Min value : ", np.min(X
   [:,:,:,1]))
39      print("Mean intensity value along 2nd channel : "
   , np.mean(X[:,:,:,1]))
40      print("Standard Deviation : ", np.std(X[:,:,:,1
   ]), "\n\n")
41
42
43      print("Max value of intensity along 3rd channel
    : ", np.max(X[:,:,2]), "   Min value : ", np.min(X
   [:,:,:,2]))
44      print("Mean intensity value along 3rd channel : "
   , np.mean(X[:,:,:,2]))
45      print("Standard Deviation : ", np.std(X[:,:,:,2
   ]), "\n\n")
46
47      combined_dataset = X[:,:,:,0] + X[:,:,:,1] + X
   [:,:,:,2]
48      combined_dataset = np.expand_dims(
   combined_dataset, axis= 3)
49
50      print("Max value of intensity in combined channel
    image : ", np.max(combined_dataset[:,:,:,0]), "
   Min value : ", np.min(combined_dataset[:,:,:,0]))
51      print("Mean intensity value in combined channel
    : ", np.mean(combined_dataset[:,:,:,0]))
52      print("Standard Deviation : ", np.std(
   combined_dataset[:,:,:,0]), "\n\n")
53      return
54 #%%
55 data_info_getter(X,y)
56 #%%
57 %matplotlib inline
58 from PIL import Image
59 import matplotlib.pyplot as plt
```

```
60
61
62 #Plotting functions for the images -->
63
64 def plot_fxn(X):
65
66     print("For the first image in the passed data
   batch -> \n")
67
68
69     X_sample = X[0]
70     print("For the first image among the batch
   passed : ")
71     print("Max value of intensity along 1st channel
    : ", np.max(X_sample[:,:,0]), "   Min value : ", np
   .min(X_sample[:,:,0]))
72     print("Max value of intensity along 2nd channel
    : ", np.max(X_sample[:,:,1]), "   Min value : ", np
   .min(X_sample[:,:,1]))
73     print("Max value of intensity along 3rd channel
    : ", np.max(X_sample[:,:,2]), "   Min value : ", np
   .min(X_sample[:,:,2]))
74     og_plot = plt.imshow(X_sample)
75
76
77     fig, axs = plt.subplots(1, 3, figsize=(20, 20))
78
79     im1 = axs[0].imshow(X_sample[:,:,0], cmap='
   viridis', vmin=-0.5, vmax=2.0, interpolation='
   nearest')
80     axs[0].set_title('Track')
81
82     im2 = axs[1].imshow(X_sample[:, :, 1], cmap='
   viridis', vmin=-0.5, vmax=2.0, interpolation='
   nearest')
83     axs[1].set_title('ECAL')
84
85     im3 = axs[2].imshow(X_sample[:, :,2], cmap='
   viridis', vmin=-0.5, vmax=2.0, interpolation='
   nearest')
86     axs[2].set_title('HCAL')
```

```
 87
 88     # Add colorbars
 89     fig.colorbar(im1, ax=axs[0], shrink=0.25)
 90     fig.colorbar(im2, ax=axs[1], shrink=0.25)
 91     fig.colorbar(im3, ax=axs[2], shrink=0.25)
 92
 93     plt.show()
 94
 95     return None
 96 #%% md
 97 Plotting the given dataset, we realize that the
    intensity values (calorimeter hits) across all the
    channels (Track, ECAL, HCAL) vary highly, and that
    there is a need to perform data pre-processing -
    Namely, Normalization of intensity values and
    Standardization of the intensity distribution.
 98 #%%
 99 plot_fxn(X)
100 #%% md
101 * The idea behind jet images is to treat the energy
    deposits in a calorimeter as intensities in a 2D
    image.
102 * A jet image is formed by taking the constituents
    of a jet and discretizing its energy into pixels in
     (n,φ), with the intensity of each pixel given by
    the sum of the energy of all constituents of the jet
     inside that (n,φ) pixel.
103
104 * The precise procedure followed for pre-processing
    of jet images involves rotations, re-pixelizations
    etc. I am building this Autoencoder while treating
    these jet images as a normal instance of classical
    images, hence I simply used the standard
    normalization and standardization processes as are
    used on images.
105
106 * I assumed that the intricacies of data handling (
    being more theoretically motivated) are BEYOND THE
    SCOPE OF THIS EVALUATION TASK.
107 #%%
108 from skimage.transform import resize
```

```
109 from sklearn.preprocessing import normalize
110 
111 def data_preprocess(X_jets):
112     #Normalizing the images
113 
114     # Resizing images from (125, 125, 3) to (128,
    128, 3)
115     resized_images = np.zeros((X_jets.shape[0], 128
    , 128, 3), dtype=np.float32)
116     for i in range(X_jets.shape[0]):
117         resized_images[i] = resize(X_jets[i], (128,
    128), anti_aliasing=True)
118 
119     X_jets = resized_images
120     del resized_images
121 
122     # Standardizing the distribution across all
    channels
123     mean = np.mean(X_jets)
124     std = np.std(X_jets)
125     X_jets = (X_jets - mean) / std
126 
127     # Clipping negative intensity values to 0...
128     X_jets = np.clip(X_jets, 0, None)
129     return X_jets
130 #%% md
131 
132 ### Image Resizing
133 
134 The function `data_preprocess()` begins by resizing
    the input images. This resizing step is essential
    for standardizing the dimensions of all images in
    the dataset. In this case, the original images are
    of size (125, 125, 3), where 125x125 represents the
    width and height of the image, and 3 represents the
    number of channels (RGB). However, VAEs typically
    work with fixed-size inputs. Therefore, the images
    are resized to a common size of (128, 128, 3) using
    bilinear interpolation. This resizing ensures that
    all images have consistent dimensions, which is
    crucial for feeding them into the neural network
```

```
model later.

### Standardization

After resizing, the function performs standardization on the images. Standardization involves transforming the data such that it has a mean of 0 and a standard deviation of 1 across each channel. This step helps in stabilizing the learning process by making the input data more amenable to optimization algorithms. By ensuring that the distribution of pixel intensities across different images is consistent, standardization can improve the convergence and performance of the VAE during training.

### Clipping Negative Intensity Values

Finally, the function clips any negative intensity values in the images to zero. This step is commonly performed in image processing tasks to ensure that all pixel values are within a valid range. Since negative pixel intensities do not make sense in the context of most image data, clipping them to zero helps in removing noise or inconsistencies in the dataset.

#%%
X_jets = data_preprocess(X)
#%%
data_info_getter(X_jets, y)
#%% md
Note, in the following plot, we are yet to normalize each image channel-wise.
#%%
plot_fxn(X_jets)
#%% md
Relevant data classes are `X_jets` and `y` ->

Loading these datasets using standard PyTorch dataloader ->
```

```
#%%
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
#%%
from torch.distributions import Normal


class Sampling(nn.Module):
    def forward(self, z_mean, z_log_var):
        # Get the shape of the tensor for the mean and log variance.
        batch, dim = z_mean.shape
        # Generate a normal random tensor (epsilon) with the same shape as z_mean
        # This tensor will be used for reparameterization trick
        epsilon = Normal(0, 1).sample((batch, dim)).to(z_mean.device)
        # Apply the reparameterization trick to generate the samples in the
        # latent space
        return z_mean + torch.exp(0.5 * z_log_var) * epsilon
#%%

# Define the Encoder
class Encoder(nn.Module):
    def __init__(self, embedding_dim):
        super(Encoder, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=2, padding=1)
        self.bn1 = nn.BatchNorm2d(64)
        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=1)
        self.bn2 = nn.BatchNorm2d(128)
        self.conv3 = nn.Conv2d(128, 256, kernel_size=3, stride=2, padding=1)
```

```
188         self.bn3 = nn.BatchNorm2d(256)
189         self.conv4 = nn.Conv2d(256, 512, kernel_size
    =3, stride=2, padding=1)
190         self.bn4 = nn.BatchNorm2d(512)
191         self.sampling = Sampling()
192         # Calculate the size of the flattened tensor
193         # Assuming the image size is 128x128
194         # After the first conv layer: 64x64
195         # After the second conv layer: 32x32
196         # After the third conv layer: 16x16
197         # After the fourth conv layer: 8x8
198         # Flattened size: 8*8*512 = 32768
199         self.fc_mu = nn.Linear(32768, embedding_dim)
200         self.fc_logvar = nn.Linear(32768,
    embedding_dim)
201 
202     def forward(self, x):
203 
204 #         print(x.shape)
205         # Assuming `x` is the input tensor with
    dimensions (N, H, W, C)
206 #         x = x.permute(0, 3, 1, 2) # Permute to (N,
    C, H, W)
207 
208         # Apply the convolutional layers with batch
    normalization
209         x = self.bn1(self.conv1(x))
210         x = nn.LeakyReLU(0.2)(x)
211         x = self.bn2(self.conv2(x))
212         x = nn.LeakyReLU(0.2)(x)
213         x = self.bn3(self.conv3(x))
214         x = nn.LeakyReLU(0.2)(x)
215         x = self.bn4(self.conv4(x))
216         x = nn.LeakyReLU(0.2)(x)
217 
218         # With this line
219         x = x.reshape(x.size(0), -1)
220 
221         # Apply the fully connected layers to get
    the mean and log variance
222         mu = self.fc_mu(x)
```

```
223         logvar = self.fc_logvar(x)
224 
225         # Sample a latent vector using
    reparameterization trick
226         z = self.sampling(mu, logvar)
227 
228         return mu, logvar, z
229 
230 
231 # class Encoder(nn.Module):
232 #     def __init__(self, latent_dim, img_size):
233 #         super(Encoder, self).__init__()
234 #
235 #         self.latent_dim = latent_dim
236 #         self.img_size = img_size
237 #         self.channels = 3
238 #
239 #         self.conv1 =  nn.Conv2d(3, 32, kernel_size
    =3, stride=2, padding=1)
240 #         self.conv2 = nn.Conv2d(32, 64, kernel_size
    =3, stride=2, padding=1)
241 #         self.conv3 = nn.Conv2d(64, 128,
    kernel_size=3, stride=2, padding=1)
242 #     #    self.conv4 = nn.Conv2d(128, 256,
    kernel_size = 3, stride = 2, padding = 1)
243 #         self.flatten = nn.Flatten()
244 #
245 #         # Parameters for the latent space
246 #         self.fc_mean = nn.Linear(128 * (img_size//
    8) * (img_size//8), latent_dim)
247 #         self.fc_log_var = nn.Linear(128 * (
    img_size//8) * (img_size//8), latent_dim)
248 #
249 #         # Initializing the sampling layer as well,
250 #         self.sampling = Sampling()
251 #
252 #%%
253 # Decoder
254 import torch.nn.functional as F
255 
256 import torch
```

```
257 import torch.nn as nn
258 
259 class Decoder(nn.Module):
260     def __init__(self, embedding_dim):
261         super(Decoder, self).__init__()
262 
263         # Fully connected layer to map the latent
    vector back to the size of the flattened tensor
264         self.fc = nn.Linear(embedding_dim, 32768)
265 
266         # Transposed convolutional layers to upscale
     the tensor
267         self.deconv1 = nn.ConvTranspose2d(512, 256,
    kernel_size=3, stride=2, padding=1, output_padding=1
    )
268         self.bn1 = nn.BatchNorm2d(256)
269         self.deconv2 = nn.ConvTranspose2d(256, 128,
    kernel_size=3, stride=2, padding=1, output_padding=1
    )
270         self.bn2 = nn.BatchNorm2d(128)
271         self.deconv3 = nn.ConvTranspose2d(128, 64,
    kernel_size=3, stride=2, padding=1, output_padding=1
    )
272         self.bn3 = nn.BatchNorm2d(64)
273         self.deconv4 = nn.ConvTranspose2d(64, 3,
    kernel_size=3, stride=2, padding=1, output_padding=1
    )
274 
275         # Final activation function
276         self.tanh = nn.Tanh()
277 
278     def forward(self, z):
279         # Fully connected layer
280         x = self.fc(z)
281         x = x.view(z.size(0), 512, 8, 8)
282 
283         # Transposed convolutional layers
284         x = self.bn1(self.deconv1(x))
285         x = nn.LeakyReLU(0.2)(x)
286         x = self.bn2(self.deconv2(x))
287         x = nn.LeakyReLU(0.2)(x)
```

```
288         x = self.bn3(self.deconv3(x))
289         x = nn.LeakyReLU(0.2)(x)
290         x = self.deconv4(x)
291 
292         # Final activation
293         x = (self.tanh(x) + 1)/2.
294 
295         return x
296 
297 #     def forward(self, x):
298 #         # Pass the latent vector through the fully connected layer
299 #         x = self.fc(x)
300 #         # Reshape the tensor
301 #         x = self.reshape(x)
302 #         # Apply transposed convolutional layers with relu activation function
303 # #         x = F.relu(self.deconv1(x))
304 #         x = F.relu(self.deconv2(x))
305 #         x = F.relu(self.deconv3(x))
306 #         # Apply the final transposed convolutional layer with a sigmoid
307 #         # activation to generate the final output
308 #         x = 0.5 * (torch.sigmoid(self.deconv4(x)) + 1)
309 #
310 #
311 #         return x
312 #%%
313 #!pip install pytorch-lightning
314 #%%
315 
316 #import pytorch_lightning as pl
317 import torch
318 import torch.nn as nn
319 import torch.nn.functional as F
320 
321 class VAE(nn.Module):
322     def __init__(self, embedding_dim):
323         super(VAE, self).__init__()
324         self.encoder = Encoder(embedding_dim)
```

```
325         self.decoder = Decoder(embedding_dim)
326 
327     def forward(self, x):
328         mu, logvar, z = self.encoder(x)
329         x_reconstructed = self.decoder(z)
330         return x_reconstructed, mu, logvar
331 
332     def vae_gaussian_kl_loss(self, mu, logvar):
333         KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2
    ) - logvar.exp(), dim=1)
334         return KLD.mean()
335 
336     def reconstruction_loss(self, x_reconstructed, x
    ):
337         bce_loss = nn.BCELoss()
338         return bce_loss(x_reconstructed, x)
339 
340     def vae_loss(self, x_reconstructed, x, mu,
    logvar):
341         recon_loss = self.reconstruction_loss(
    x_reconstructed, x)
342         kld_loss = self.vae_gaussian_kl_loss(mu,
    logvar)
343         return 500 * recon_loss + kld_loss
344 
345 #%%
346 # # Define loss function
347 # #KL Divergence is computed between the learned
    latent variable distribution and a standard normal
    distribution.
348 # def vae_gaussian_kl_loss(mu, logvar):
349 #     # see Appendix B from VAE paper:
350 #     # Kingma and Welling. Auto-Encoding
    Variational Bayes. ICLR, 2014
351 #     # https://arxiv.org/abs/1312.6114
352 #     KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2
    ) - logvar.exp(), dim=1)
353 #     return KLD.mean()
354 #
355 # def reconstruction_loss(x_reconstructed, x):
356 #     bce_loss = nn.BCELoss()
```

```
357 #     return bce_loss(x_reconstructed, x)
358 #
359 #
360 # def vae_loss(y_pred, y_true):
361 #     mu, logvar, recon_x = y_pred
362 #     recon_loss = reconstruction_loss(recon_x,
    y_true)
363 #     kld_loss = vae_gaussian_kl_loss(mu, logvar)
364 #     return 500 * recon_loss + kld_loss
365 # # def vae_loss(reconstructed, original, mu,
    log_var):
366 # #     reconstruction_loss = F.binary_cross_entropy
    (reconstructed, original, reduction='sum')
367 # #     kl_divergence = -0.5 * torch.sum(1 + log_var
     - mu.pow(2) - log_var.exp())
368 # #     return reconstruction_loss + kl_divergence
369 #%%
370 trackMax = np.max(X_jets[:,:,:,0])
371 ecalMax = np.max(X_jets[:,:,:,1])
372 hcalMax = np.max(X_jets[:,:,:,2])
373
374
375 X_jets[:,:,:,0] = X_jets[:,:,:,0]/trackMax
376 X_jets[:,:,:,1] = X_jets[:,:,:,1]/ecalMax
377 X_jets[:,:,:,2] = X_jets[:,:,:,2]/hcalMax
378 #%%
379 data_info_getter(X_jets, y)
380 #%%
381 from sklearn.model_selection import train_test_split
382
383 # Assuming X_jets is your dataset and y are the
    labels
384 X_jets_train, X_jets_test, y_train, y_test =
    train_test_split(X_jets, y, test_size=0.3,
    random_state=42)
385 X_jets_train, X_jets_val, y_train, y_val =
    train_test_split(X_jets_train, y_train, test_size=0.
    25, random_state=42)
386 #%%
387 import torch
388 from torch.utils.data import Dataset
```

```
389
390 class JetDataset(Dataset):
391     def __init__(self, X, y):
392         self.X = X
393         self.y = y
394
395     def __getitem__(self, index):
396         # X is in the format (h, w, channels)
397         # Permute the dimensions to (channels, h, w)
398         image = self.X[index].permute(2, 0, 1)
399         return image, self.y[index]
400
401     def __len__(self):
402         return len(self.X)
403
404 # Convert your data to PyTorch tensors
405 train_dataset = JetDataset(torch.from_numpy(
    X_jets_train), torch.from_numpy(y_train))
406 val_dataset = JetDataset(torch.from_numpy(X_jets_val
    ), torch.from_numpy(y_val))
407 test_dataset = JetDataset(torch.from_numpy(
    X_jets_test), torch.from_numpy(y_test))
408 #%%
409 from tqdm import tqdm
410 #%%
411 torch.cuda.is_available()
412 #%%
413 from torch.utils.data import DataLoader
414 import torch.optim as optim
415
416
417 device = torch.device("cuda" if torch.cuda.
    is_available() else "cpu")
418 model = VAE(embedding_dim= 2048).to(device) # Adjust
     embedding_dim as needed
419 optimizer = optim.Adam(model.parameters(), lr=0.0001
    )
420 criterion = model.vae_loss
421
422 # Data loaders
423 train_loader = DataLoader(train_dataset, batch_size=
```

```
128, shuffle=False)
val_loader = DataLoader(val_dataset, batch_size=128, shuffle=False)

# Training loop
num_epochs = 100
train_losses = []
val_losses = []

# for epoch in range(num_epochs):
#     model.train()
#     train_loss = 0
#     for batch_idx, (data, _) in enumerate(tqdm(train_loader, desc=f"Epoch {epoch}")):
#         optimizer.zero_grad()
#         recon_batch, mu, logvar = model(data)
#         loss = criterion(recon_batch, data, mu, logvar)
#         loss.backward()
#         train_loss += loss.item()
#         optimizer.step()
#     train_losses.append(train_loss / len(train_loader))

#     model.eval()
#     val_loss = 0
#     with torch.no_grad():
#         for batch_idx, (data, _) in enumerate(val_loader):
#             recon_batch, mu, logvar = model(data)
#             loss = criterion(recon_batch, data, mu, logvar)
#             val_loss += loss.item()
#     val_losses.append(val_loss / len(val_loader))
for epoch in range(num_epochs):
    model.train()
    train_loss = 0
    for batch_idx, (data, _) in enumerate(tqdm(train_loader, desc=f"Epoch {epoch}")):
        data = data.to(device) # Move data to device
        optimizer.zero_grad()
```

```
457         recon_batch, mu, logvar = model(data)
458         loss = criterion(recon_batch, data, mu,
    logvar)
459         loss.backward()
460         train_loss += loss.item()
461         optimizer.step()
462     train_losses.append(train_loss / len(
    train_loader))
463 
464     model.eval()
465     val_loss = 0
466     with torch.no_grad():
467         for batch_idx, (data, _) in enumerate(
    val_loader):
468             data = data.to(device) # Move data to
    device
469             recon_batch, mu, logvar = model(data)
470             loss = criterion(recon_batch, data, mu,
    logvar)
471             val_loss += loss.item()
472     val_losses.append(val_loss / len(val_loader))
473 
474     print(f'Epoch: {epoch+1}, Train Loss: {
    train_losses[-1]}, Val Loss: {val_losses[-1]}')
475 #%%
476 plot_fxn(X_jets*255.)
477 #%%
478 def reconstruct_image(original_image, model):
479     # Ensure the image is a PyTorch tensor and on
    the correct device
480     original_image = torch.from_numpy(original_image
    ).float().to(device)
481     # Reshape the image to match the input shape
    expected by the model
482     original_image = original_image.unsqueeze(0) #
    Assuming the model expects a batch of images
483 
484     # Pass the image through the model to get the
    reconstructed image
485     with torch.no_grad():
486         reconstructed_image = model(original_image)
```

```
487
488     # Convert the reconstructed image back to a
    NumPy array
489     reconstructed_image = reconstructed_image[0]
490     reconstructed_image = reconstructed_image.
    squeeze(0)
491 #     print(reconstructed_image.shape)
492     reconstructed_image = reconstructed_image.cpu().
    numpy().astype(np.float32)
493
494     reconstructed_image = np.transpose(
    reconstructed_image, (1, 2, 0))
495 #     print(reconstructed_image.shape)
496     # # Reshape the reconstructed image to its
    original shape (128x128 for our case)
497     # reconstructed_image = reconstructed_image.
    reshape(128, 128)
498
499     return reconstructed_image
500 #%%
501
502 #%%
503 import matplotlib.pyplot as plt
504 import numpy as np
505
506 # # Now that the model is trained, and we have
    already loaded a  DataLoader for the test data.
507 # test_loader = DataLoader(test_dataset, batch_size=
    128, shuffle=False)
508
509 # import matplotlib.pyplot as plt
510 # import numpy as np
511
512 # # Assuming the model is trained and you have a
    DataLoader for test data
513 # #test_loader = DataLoader(test_dataset, batch_size
    =1, shuffle=False)
514
515 # # Get a single image from the test dataset
516 # dataiter = iter(test_loader)
517 # images, _ = next(dataiter)
```

```
518 images = X_jets[0]
519 images[:,:,0] = (images[:,:,0] - np.min(images[:,:,0
    ]))/(np.max(images[:,:,0]) - np.min(images[:,:,0]))
520 images[:,:,1] = (images[:,:,1] - np.min(images[:,:,1
    ]))/(np.max(images[:,:,1]) - np.min(images[:,:,1]))
521 images[:,:,2] = (images[:,:,2] - np.min(images[:,:,2
    ]))/(np.max(images[:,:,2]) - np.min(images[:,:,2]))
522 # Plot original image
523 # plt.figure(figsize=(10, 5))
524 # plt.subplot(1, 2, 1)
525 # plt.imshow(images * 255.)
526 # plt.title('Original Image')
527 # plt.axis('off')
528
529 images = np.transpose(images, (2, 1, 0))
530 #images.to(device)
531 reconstructed_image = reconstruct_image(images,
    model)
532
533 #reconstructed_image = np.clip(reconstructed_image,
    0, None)
534 # Scale the reconstructed image by 255.0 and convert
     to numpy
535 reconstructed_image_scaled = reconstructed_image
536
537 #%% md
538
539 #%%
540 X_sample = X_jets[0]
541 #X_sample = (X_sample - np.min(X_sample)) / (np.max(
    X_sample) - np.min(X_sample))
542 # X_sample[:,:,0] = X_sample[:,:,0] * trackMax
543 # X_sample[:,:,1] = X_sample[:,:,1] * ecalMax
544 # X_sample[:,:,2] = X_sample[:,:,2] * hcalMax
545
546 print("For the first image among the batch passed
     : ")
547 print("Max value of intensity along 1st channel : "
    , np.max(X_sample[:,:,0]), "   Min value : ", np.min
    (X_sample[:,:,0]))
548 print("Max value of intensity along 2nd channel : "
```

```
548 , np.max(X_sample[:,:,1]), "   Min value : ", np.min
    (X_sample[:,:,1]))
549 print("Max value of intensity along 3rd channel : "
    , np.max(X_sample[:,:,2]), "   Min value : ", np.min
    (X_sample[:,:,2]))
550 og_plot = plt.imshow(X_sample)
551
552
553 fig, axs = plt.subplots(1, 3, figsize=(20, 20))
554
555 im1 = axs[0].imshow(X_sample[:,:,0], cmap='viridis'
    , vmin=-0.5, vmax=2.0, interpolation='nearest')
556 axs[0].set_title('Track')
557
558 im2 = axs[1].imshow(X_sample[:, :, 1], cmap='viridis
    ', vmin=-0.5, vmax=2.0, interpolation='nearest')
559 axs[1].set_title('ECAL')
560
561 im3 = axs[2].imshow(X_sample[:, :,2], cmap='viridis'
    , vmin=-0.5, vmax=2.0, interpolation='nearest')
562 axs[2].set_title('HCAL')
563
564 # Add colorbars
565 fig.colorbar(im1, ax=axs[0], shrink=0.25)
566 fig.colorbar(im2, ax=axs[1], shrink=0.25)
567 fig.colorbar(im3, ax=axs[2], shrink=0.25)
568
569 #%%
570 recon_img = reconstructed_image_scaled
571 # recon_img[:,:,0] = (recon_img[:,:,0] - np.min(
    recon_img[:,:,0]))/(np.max(recon_img[:,:,0]) - np.
    min(recon_img[:,:,0]))
572 # recon_img[:,:,1] = (recon_img[:,:,1] - np.min(
    recon_img[:,:,1]))/(np.max(recon_img[:,:,1]) - np.
    min(recon_img[:,:,1]))
573 # recon_img[:,:,2] = (recon_img[:,:,2] - np.min(
    recon_img[:,:,2]))/(np.max(recon_img[:,:,2]) - np.
    min(recon_img[:,:,2]))
574 #%% md
575 ### The reconstructed image is plotted as shown.
576 #%%
```

```
X_sample = recon_img
print("For the first image among the batch passed
 : ")
print("Max value of intensity along 1st channel : "
, np.max(X_sample[:,:,0]), "   Min value : ", np.min
(X_sample[:,:,0]))
print("Max value of intensity along 2nd channel : "
, np.max(X_sample[:,:,1]), "   Min value : ", np.min
(X_sample[:,:,1]))
print("Max value of intensity along 3rd channel : "
, np.max(X_sample[:,:,2]), "   Min value : ", np.min
(X_sample[:,:,2]))
# X_sample[:,:,0] = X_sample[:,:,0] * trackMax
# X_sample[:,:,1] = X_sample[:,:,1] * ecalMax
# X_sample[:,:,2] = X_sample[:,:,2] * hcalMax

og_plot = plt.imshow(X_sample)


fig, axs = plt.subplots(1, 3, figsize=(20, 20))

im1 = axs[0].imshow(X_sample[:,:,0], cmap='viridis'
, vmin=-0.5, vmax=2.0, interpolation='nearest')
axs[0].set_title('Track')

im2 = axs[1].imshow(X_sample[:, :, 1], cmap='viridis
', vmin=-0.5, vmax=2.0, interpolation='nearest')
axs[1].set_title('ECAL')

im3 = axs[2].imshow(X_sample[:, :,2], cmap='viridis'
, vmin=-0.5, vmax=2.0, interpolation='nearest')
axs[2].set_title('HCAL')

# Add colorbars
fig.colorbar(im1, ax=axs[0], shrink=0.25)
fig.colorbar(im2, ax=axs[1], shrink=0.25)
fig.colorbar(im3, ax=axs[2], shrink=0.25)
#%%
plt.figure(figsize=(10, 5))
plt.plot(train_losses, label='Training Loss')
plt.plot(val_losses, label='Validation Loss')
```

```
608
609 plt.title('Training and Validation Loss')
610 plt.xlabel('Epochs')
611 plt.ylabel('Loss')
612 plt.legend()
613 plt.show()
614 #%%
615
616 #%%
617
```