Enhancing Geant4 Monte Carlo Simulations through Machine Learning Integration

G.A.P Cirrone, S.Fattori, G. Gallo, V. Ientile, A. Sciuto, A. Tricomi

Use Case Expected Activities

MS7	**m13-m18** [… - Feb 2024]	Conduct a thorough examination of modern technologies and methodologies. Keep up with the latest advancements in hardware and software that are relevant to the project. Define a first simple test case on the utilization of ML to speed up a typical Monte Carlo Simulation, with the definition of Geant4 part to be modified.	Template MS
MS8	[March 2024 – June 2024]	If the Flagship has no activities in MS8, an intermediate report must still be produced.	Spoke2_modello_rapporto_revisori_MS8
MS9	[July 2024 – Oct 2024]	Put the chosen technologies into action, create machine learning models, and integrate them with Monte Carlo simulations. Perform testing and validation on the chosen datasets. Construct a proof-of-concept (PoC) system. Make the codebase available in a public repository.	Spoke2_modello_rapporti_revisori_MS9
MS10	m31-m36	Optimize the proof-of-concept system, perform rigorous testing, and prepare for wider deployment. Investigate new techniques (such as Deep Learning) and expand on existing use cases. Present results to at least one conference.	Spoke2_modello_rapporti_revisori_MS10

KPIs

KPI ID	Description	Acceptance threshold	Check
KPI2.6.1.1	Publications	1	✗
KPI2.6.1.2	Presentation at conferences	1	✗
KPI2.6.1.3	Publicly available Code repository	1	⚠
KPI2.6.1.4	Use case Test Datasets defined	1	☑
KPI2.6.1.5	Geant4 Algorithms to be used as targets for a ML optimization	1	⚠
KPI2.6.1.6	Efficiency Gain on the same hardware: The improved simulation, when run on the same hardware as the standard simulation, should achieve at least a 20% reduction in the time taken to generate predictions.	20% in time reduction, with acceptable physics performance	⚠

Preliminary Info

- runbeamOn: 1e6
- File: ‘Let_1-1.out’ -> cut_in_um: 1.0,
  voxel_in_um: 1.0,
  entries: 40,000

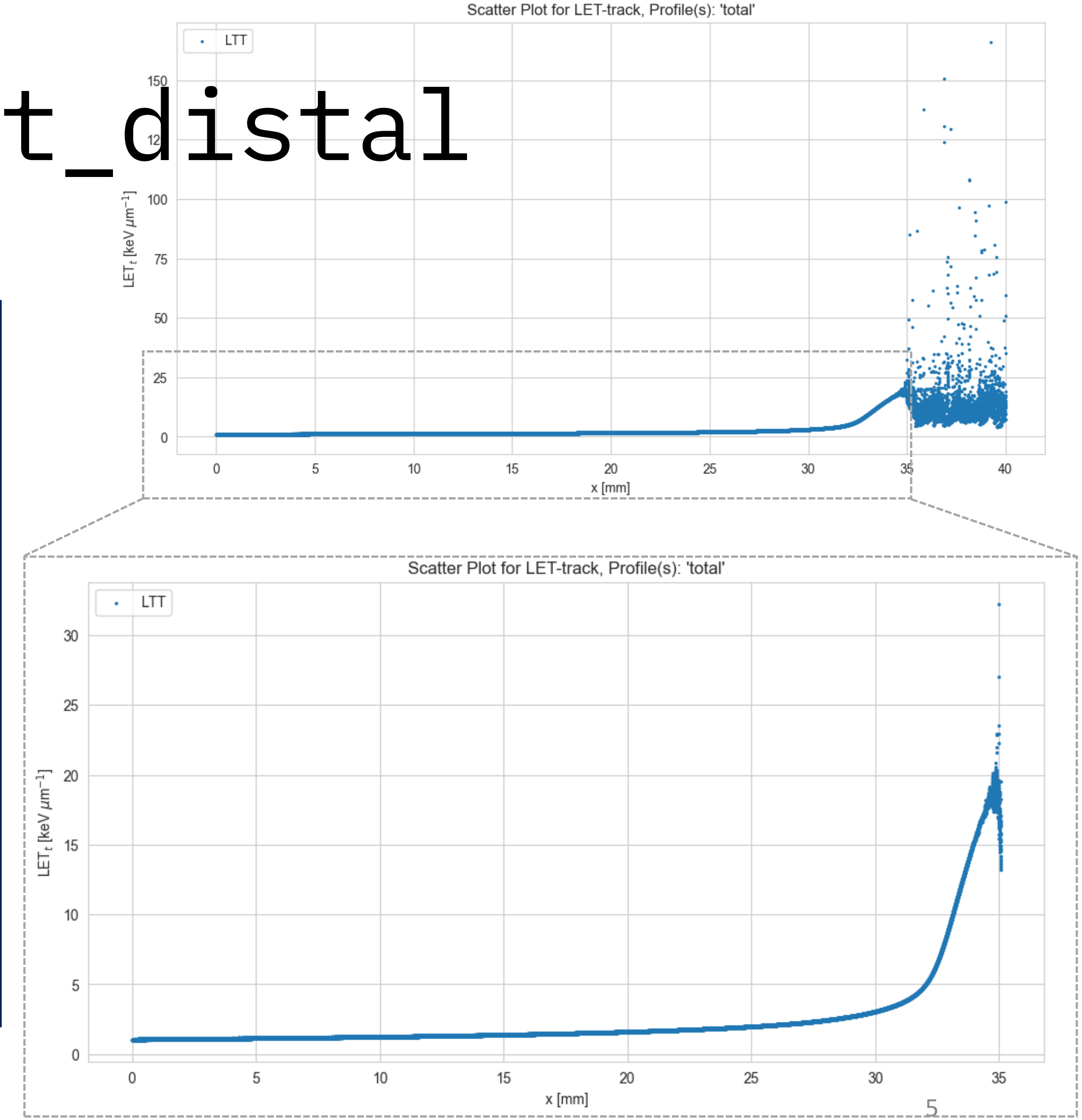
DataFrame with loaded data from file: runbeam_1e6/Let_1-1.out

	i	LDT	LTT	proton_1_D	proton_1_T	proton_1_D.1	proton_1_T.1	proton_D	proton_T	proton_1_D.2	...	O18_D	O18_T	F17_D	F17_T	F18_D	F18_T	F19_D	F19_T	Ne20_D	Ne20_T
0	0	2.63090	1.04760	0.0	0.0	1.04081	1.04065	11.7570	6.01532	0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	1	1.95003	1.04636	0.0	0.0	1.04082	1.04070	12.9513	6.09598	0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	2	2.41322	1.04725	0.0	0.0	1.04092	1.04075	13.8869	6.15894	0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	3	2.15345	1.04629	0.0	0.0	1.04089	1.04075	13.5346	6.05996	0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	4	4.35478	1.04927	0.0	0.0	1.04090	1.04074	12.2653	5.99763	0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
5	5	4.31605	1.05018	0.0	0.0	1.04091	1.04077	11.9284	5.86857	0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
6	6	4.44597	1.05099	0.0	0.0	1.04095	1.04077	14.2888	6.16782	0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
7	7	3.77041	1.04961	0.0	0.0	1.04090	1.04078	13.0829	6.10534	0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
8	8	2.90049	1.04771	0.0	0.0	1.04094	1.04082	11.4468	5.95522	0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
9	9	3.81664	1.04859	0.0	0.0	1.04095	1.04082	10.9422	5.84279	0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

10 rows × 129 columns

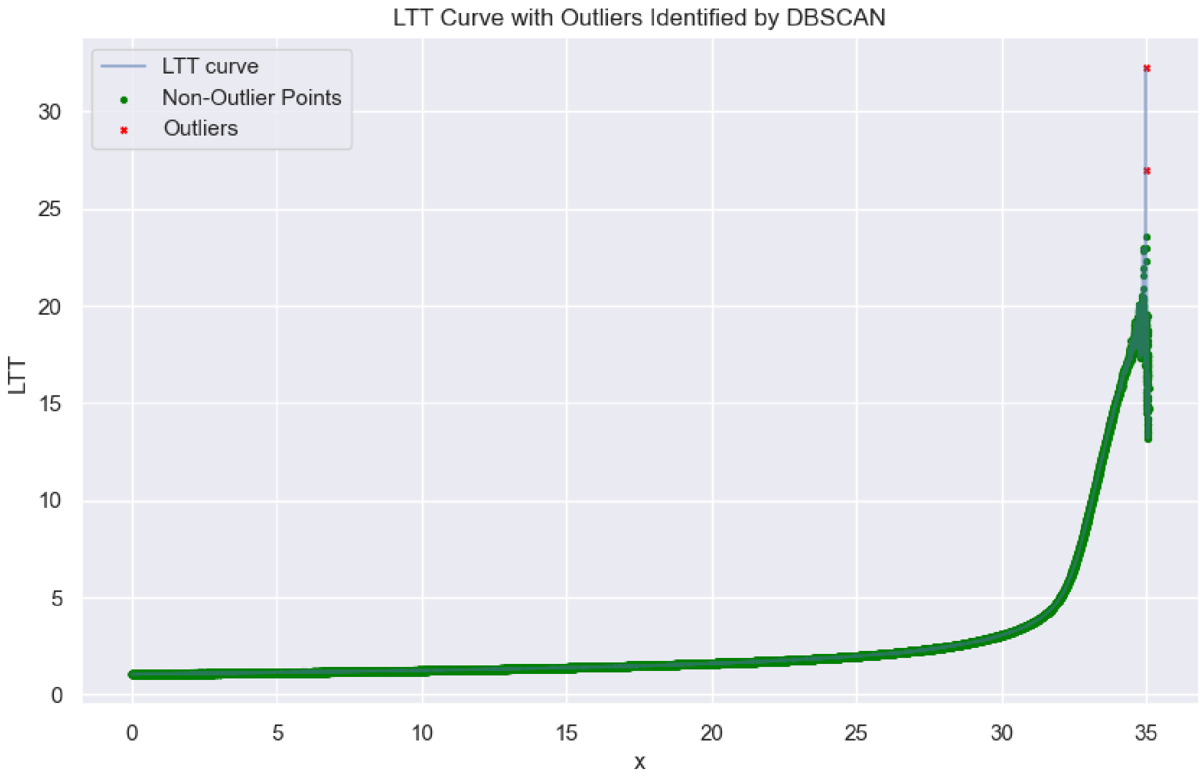
cut_dataframe_at_distal

```python
def cut_dataframe_at_distal(df, column_type='track', std_threshold=3):
    """
    Cut a DataFrame at the point where abrupt variations occur in the LET
total
    column.

    Parameters:
    - df (DataFrame): The input DataFrame.
    - column_type (str, optional): Type of columns to include ('track' or
      'dose'). Defaults to 'track'.
    - std_threshold (int, optional): Standard deviation threshold for
detecting
      abrupt variations. Defaults to 3.

    Returns:
    - DataFrame: DataFrame cut at the point of the first abrupt variation
in the
      LET total column.
    """
    # Define search patterns based on column type to drop columns
    if column_type == 'track':
        let_total_pattern = 'LTT'
    elif column_type == 'dose':
        let_total_pattern = 'LDT'
    else:
        raise ValueError("Invalid column_type. Use 'track' or 'dose'.")

    …
```



Apply DBSCAN to identify clusters and outliers

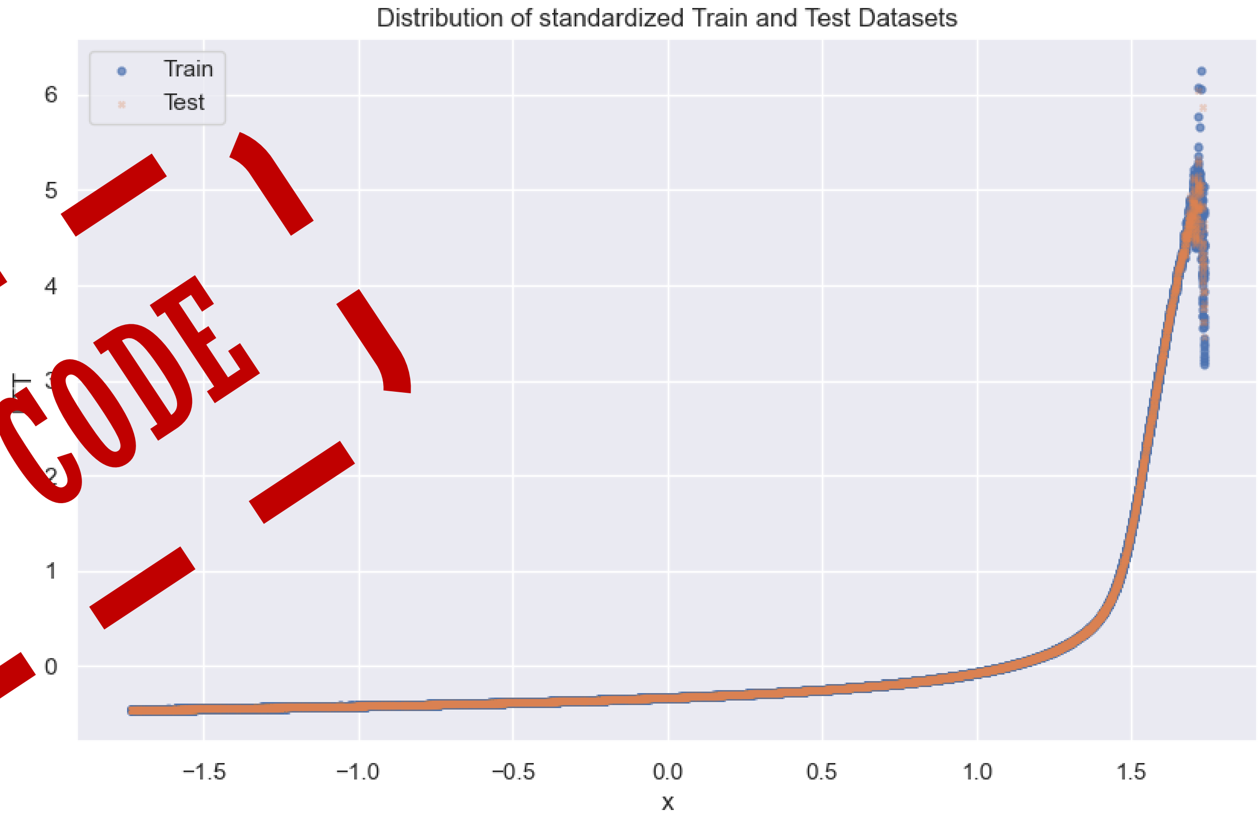
```
eps = 1 # Adjust the neighborhood radius based on your data
min_samples = 2 # Adjust the minimum number of samples in a neighborhood
dbscan = DBSCAN(eps=eps, min_samples=min_samples)
cluster_labels = dbscan.fit_predict(X)

# Create a new DataFrame with cluster labels
df_with_clusters = df_distal_non_zero.copy()
df_with_clusters['cluster'] = cluster_labels

# Identify non-outlier points (cluster != -1)
filtered_df = df_with_clusters[df_with_clusters['cluster'] != -1]
filtered_df = filtered_df.drop(columns=['cluster'])

…
```



split_and_standardize_data

```python
def split_and_standardize_data(data_df, scaler_type='standard'):
    """
    Split and standardize the input DataFrame.

    Parameters:
    - data_df (DataFrame): The input DataFrame to be split and standardized.
    - scaler_type (str, optional): Type of scaler to be used. Options are
      'standard' (default), 'minmax', or 'robust'.

    Returns:
    - tuple: A tuple containing three elements:
        - X_train (numpy.ndarray): Standardized training data.
        - X_test (numpy.ndarray): Standardized testing data.
        - scaler (StandardScaler, MinMaxScaler, or RobustScaler): The selecte
          scaler used for standardization.
    """

    # Convert the DataFrame into a 2D NumPy array
    data_np = data_df.values.reshape(-1, data_df.shape[1]).astype('float32')

    # Randomly split the data into training and testing sets
    X_train, X_test = train_test_split(data_np, test_size=0.3, random_state=42)

    # Convert scaler_type to lowercase for case-insensitivity
    scaler_type_lower = scaler_type.lower()

    # Choose the scaler based on the user's selection
    if scaler_type_lower == 'standard':
        scaler = preprocessing.StandardScaler()
    elif scaler_type_lower == 'minmax':
        scaler = preprocessing.MinMaxScaler()
    elif scaler_type_lower == 'robust':
        scaler = preprocessing.RobustScaler()

    …
```

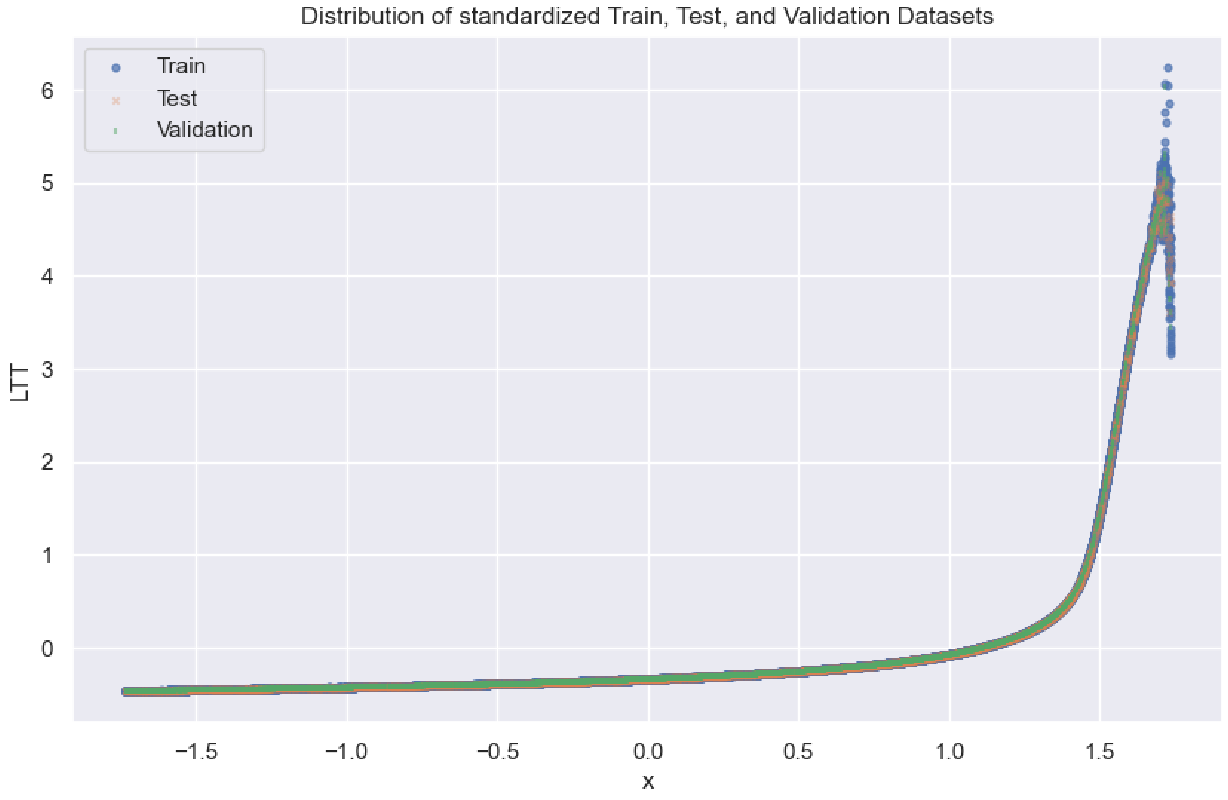
OLD CODE



split_and_standardize_data

NEW CODE

```python
def split_and_standardize_data(data_df, scaler_type='standard', random_state=42,
                               create_validation=True):
    """
    Split and standardize the input DataFrame.

    Parameters:
    - data_df (DataFrame): The input DataFrame to be split and standardized.
    - scaler_type (str, optional): Type of scaler to be used. Options are
      'standard' (default), 'minmax', or 'robust'.
    - random_state (int, optional): Random seed for reproducibility.
    - create_validation (bool, optional): Whether to create a validation dataset
      or not. Default is True.

    Returns:
    - tuple: A tuple containing three elements:
        - X_train (numpy.ndarray): Standardized training data.
        - X_val (numpy.ndarray, optional): Standardized validation data if
          create_validation is True.
        - X_test (numpy.ndarray): Standardized testing data.
        - scaler (StandardScaler, MinMaxScaler, or RobustScaler): The selected
          scaler used for standardization.
    """

    # Convert the DataFrame into a 2D NumPy array
    data_np = data_df.values.reshape(-1, data_df.shape[1]).astype('float32')

    # Split the data into train, validation, and testing sets
    X_train, X_test = train_test_split(data_np, test_size=0.2, shuffle=True,
                                       random_state=random_state)
    if create_validation:
        X_val, X_test = train_test_split(X_test, test_size=0.5, shuffle=True,
                                         random_state=random_state)

    …
```



DataLoader enhancement

OLD CODE

```python
class DataBuilder(Dataset):
    def __init__(self, df, train=True, scaler='standard', random_state=42):
        self.X_train, self.X_test, self.standardizer = \
            split_and_standardize_data(df, scaler_type=scaler,
                                       random_state=random_state)
        if train:
            self.x = torch.from_numpy(self.X_train)
        else:
            self.x = torch.from_numpy(self.X_test)
        self.len = self.x.shape[0]
        del self.X_train
        del self.X_test

    def __getitem__(self, index):
        return self.x[index]

    def __len__(self):
        return self.len

# %%
traindata_set = DataBuilder(filtered_df, train=True, random_state=random_seed)
testdata_set = DataBuilder(filtered_df, train=False, random_state=random_seed)

batch_size = 1024
trainloader = DataLoader(dataset=traindata_set, batch_size=batch_size)
testloader = DataLoader(dataset=testdata_set, batch_size=batch_size)
```

```python
class DataBuilder(Dataset):
    """
    [ ... ]
    """
    def __init__(self, df, train=True, scaler='standard', random_state=42,
                 create_validation=False):
        if create_validation:
            X_train, X_val, X_test, self.standardizer = \
                split_and_standardize_data(df, scaler_type=scaler,
                                           random_state=random_state,
                                           create_validation=create_validation)
        else:
            X_train, X_test, self.standardizer = \
                split_and_standardize_data(df, scaler_type=scaler,
                                           random_state=random_state,
                                           create_validation=create_validation)

        self.data = torch.from_numpy(X_train if train else
                                     (X_test if not create_validation else X_val))

        self.length = len(self.data)

    def __getitem__(self, index):
        return self.data[index]

    def __len__(self):
        return self.length
```

Added support for creating validation datasets in `DataBuilder` class.

DataLoader enhancement

OLD CODE

```
class DataBuilder(Dataset):
[ ... ]

# %%
traindata_set = DataBuilder(filtered_df, train=True, random_state=random_seed)
testdata_set = DataBuilder(filtered_df, train=False, random_state=random_seed)

batch_size = 1024
trainloader = DataLoader(dataset=traindata_set, batch_size=batch_size)
testloader = DataLoader(dataset=testdata_set, batch_size=batch_size)
```

The `create_data_loaders` method constructs data loaders for training and, optionally, validation datasets.

- It takes input data and splits it into training and validation sets,
- Standardizes the data,
- Creates PyTorch DataLoader objects for efficient batch-wise processing during model training.

This method encapsulates the data preparation steps required before training a machine learning model, providing a convenient interface for generating data loaders tailored to the specific needs of the model.

```
def create_data_loaders(df, batch_size, random_state=42, create_validation=False):
    """
    Create DataLoader objects for the train, test, and optionally validation
datasets.

    Args:
        - df (DataFrame): The input DataFrame containing the data.
        - batch_size (int): The batch size for DataLoader objects.
        - random_state (int, optional): Random seed for reproducibility (default
          is 42).
        - create_val_dataset (bool, optional): Whether to create a validation
          dataset (default is True).

    Returns:
       -  tuple: A tuple containing three DataLoader objects (train_loader,
          test_loader, val_loader). The val_loader is None if create_val_dataset
          is False.

    """
    train_data_set = DataBuilder(df, train=True, random_state=random_state,
                                 create_validation=create_validation)
    test_data_set = DataBuilder(df, train=False, random_state=random_state,
                                create_validation=create_validation)

    val_data_set = None
    if create_validation:
        val_data_set = DataBuilder(df, train=False, random_state=random_state,
                                   create_validation=create_validation)

    train_loader = DataLoader(dataset=train_data_set, batch_size=batch_size)
    test_loader = DataLoader(dataset=test_data_set, batch_size=batch_size)
    val_loader = (DataLoader(dataset=val_data_set, batch_size=batch_size) if
                  val_data_set else None)

    return train_loader, test_loader, val_loader
```

class VAE(nn.Module)

```python
class Vae(nn.Module):
    def __init__(self, D_in, H=50, H2=12, latent_dim=3):

        # Encoder
        super(Vae, self).__init__()
        self.linear1 = nn.Linear(D_in, H)
        self.lin_bn1 = nn.BatchNorm1d(num_features=H)
        self.linear2 = nn.Linear(H, H2)
        self.lin_bn2 = nn.BatchNorm1d(num_features=H2)
        self.linear3 = nn.Linear(H2, H2)
        self.lin_bn3 = nn.BatchNorm1d(num_features=H2)

        # Latent vectors mu and sigma
        self.fc1 = nn.Linear(H2, latent_dim)
        self.bn1 = nn.BatchNorm1d(num_features=latent_dim)
        self.fc21 = nn.Linear(latent_dim, latent_dim)
        self.fc22 = nn.Linear(latent_dim, latent_dim)

        # Sampling vector
        self.fc3 = nn.Linear(latent_dim, latent_dim)
        self.fc_bn3 = nn.BatchNorm1d(num_features=latent_dim)
        self.fc4 = nn.Linear(latent_dim, H2)
        self.fc_bn4 = nn.BatchNorm1d(num_features=H2)

        # Decoder
        self.linear4 = nn.Linear(H2, H2)
        self.lin_bn4 = nn.BatchNorm1d(num_features=H2)
        self.linear5 = nn.Linear(H2, H)
        self.lin_bn5 = nn.BatchNorm1d(num_features=H)
        self.linear6 = nn.Linear(H, D_in)
        self.lin_bn6 = nn.BatchNorm1d(num_features=D_in)

        self.relu = nn.ReLU()

    …
```

```python
    def encode(self, x):
        lin1 = self.relu(self.lin_bn1(self.linear1(x)))
        lin2 = self.relu(self.lin_bn2(self.linear2(lin1)))
        lin3 = self.relu(self.lin_bn3(self.linear3(lin2)))

        fc1 = F.relu(self.bn1(self.fc1(lin3)))

        r1 = self.fc21(fc1)
        r2 = self.fc22(fc1)

        return r1, r2

    # define the reparameterization method, which is used during training to
    # sample from the learned distributions.
    def reparameterize(self, mu, logvar):
        if self.training:
            std = logvar.mul(0.5).exp_()
            # eps = Variable(std.data.new(std.size()).normal_()) # PyTorch < 0.4.0
            eps = torch.randn_like(std)
            return eps.mul(std).add_(mu)
        else:
            return mu

    def decode(self, z):
        fc3 = self.relu(self.fc_bn3(self.fc3(z)))
        fc4 = self.relu(self.fc_bn4(self.fc4(fc3)))

        lin4 = self.relu(self.lin_bn4(self.linear4(fc4)))
        lin5 = self.relu(self.lin_bn5(self.linear5(lin4)))
        return self.lin_bn6(self.linear6(lin5))

    def forward(self, x):
        mu, logvar = self.encode(x)
        z = self.reparameterize(mu, logvar)
        # self.decode(z) is later recon_batch, mu is mu, logvar is logvar
        return self.decode(z), mu, logvar
```

class customLoss(nn.Module)

```python
class customLoss(nn.Module):
    """
    Custom loss function for the Variational Autoencoder.

    The Variational Autoencoder (VAE) requires a loss function that combines a
    reconstruction loss (typically Mean Squared Error) and a Kullback-Leibler
    Divergence (KLD) loss to ensure that the latent space distribution closely
    matches a prior distribution.

    Args:
        beta (float, optional): Weight parameter for the KLD loss. Default is
        1.0.

    Attributes:
        mse_loss (torch.nn.MSELoss): Mean Squared Error loss function.
        beta (float): Weight parameter for the KLD loss.

    Methods:
        forward(x_recon, x, mu, logvar): Computes the total loss, Mean Squared
        Error (MSE) loss, and Kullback-Leibler Divergence (KLD) loss.
    """

    def __init__(self, beta=1.0):
        super(customLoss, self).__init__()
        self.mse_loss = nn.MSELoss(reduction="sum")
        self.beta = beta
```

```python
    def forward(self, x_recon, x, mu, logvar):
        """
        Calculate the total loss, Mean Squared Error (MSE) loss, and
        Kullback-Leibler Divergence (KLD) loss.

        Args:
            - x_recon (torch.Tensor): Reconstructed data from the VAE.
            - x (torch.Tensor): Original input data batch.
            - mu (torch.Tensor): Mean of the latent space distribution.
            - logvar (torch.Tensor): Log variance of the latent space
              distribution.

        Returns:
            tuple: A tuple containing the total loss, MSE loss, and KLD loss.

        """
        loss_MSE = self.mse_loss(x_recon, x)
        loss_KLD = (-0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp(),
                                     dim=-1)).sum()

        total_loss = loss_MSE + self.beta * loss_KLD

        return total_loss, loss_MSE, loss_KLD
```

Define train & test step

```
def train(epoch):
    model.train()
    train_loss = 0
    num_samples = 0  # Total number of samples processed

    # Record the start time of the epoch
    start_time = time.time()

    for batch_idx, data in enumerate(trainloader):
        data = data.to(device)
        optimizer.zero_grad()
        recon_batch, mu, logvar = model(data)

        # Call the loss function
        total_loss, loss_MSE, loss_KLD = loss_mse(recon_batch, data, mu, logvar)

        # Perform backward pass on the total_loss component
        total_loss.backward()    # total_loss = loss_MSE + self.beta * loss_KLD

        train_loss += total_loss.item()
        num_samples += len(data)
        optimizer.step()

    # Calculate and append the average training loss per sample for the epoch
    average_train_loss_per_sample = train_loss / num_samples
    train_losses.append(average_train_loss_per_sample)

    # Record and append the the train time of the epoch
    end_time = time.time()
    train_times.append(end_time - start_time)

    # Print or log the average training loss per sample for the epoch
    if epoch % show_every == 0:
        print('====> Epoch: {} Average training loss per sample: {:.4f}'.format(
            epoch, average_train_loss_per_sample))
        print('====> Epoch: {} Training duration: {:.2f} seconds'.format(
            epoch, end_time - start_time))
```

The `train_step` method serves as the workhorse for training the Variational Autoencoder (VAE) model by performing a single iteration of the training process. Key features of the `train_step` method include:

1. Training Loop: It iterates over the training data in batches, computing the reconstruction loss and the Kullback-Leibler Divergence (KLD) loss for each batch.

2. Gradient Calculation: It computes the gradients of the total loss with respect to the model parameters using backpropagation, allowing for parameter updates through optimization.

3. Loss Aggregation: It aggregates reconstruction loss and KLD loss across all batches to obtain the total training loss for the epoch.

4. Performance Metrics: It tracks performance metrics such as the average training loss per sample, the average MSE loss per sample, the average KLD loss per sample, and the training time for each epoch.

5. Device Agnosticism: It supports training on both CPU and GPU devices by dynamically moving data and model parameters to the appropriate device based on user configuration.

Define train & test step

```python
def train(epoch):
    model.train()
    train_loss = 0
    num_samples = 0  # Total number of samples processed

    # Record the start time of the epoch
    start_time = time.time()

    for batch_idx, data in enumerate(trainloader):
        data = data.to(device)
        optimizer.zero_grad()
        recon_batch, mu, logvar = model(data)

        # Call the loss function
        total_loss, loss_MSE, loss_KLD = loss_mse(recon_batch, data, mu, logvar)

        # Perform backward pass on the total_loss component
        total_loss.backward()

        train_loss += total_loss.item()
        num_samples += len(data)
        optimizer.step()

    # Calculate and append the average training loss per sample for the epoch
    average_train_loss_per_sample = train_loss / num_samples
    train_losses.append(average_train_loss_per_sample)

    # Record and append the the train time of the epoch
    end_time = time.time()
    train_times.append(end_time - start_time)

    # Print or log the average training loss per sample for the epoch
    if epoch % show_every == 0:
        print('====> Epoch: {} Average training loss per sample: {:.4f}'.format(
            epoch, average_train_loss_per_sample))
        print('====> Epoch: {} Training duration: {:.2f} seconds'.format(
            epoch, end_time - start_time))
```

```python
def test(epoch):
    model.eval()
    test_loss = 0
    num_samples = 0 # Total number of samples processed

    with torch.no_grad():
        for batch_idx, data in enumerate(testloader):
            data = data.to(device)
            optimizer.zero_grad()
            recon_batch, mu, logvar = model(data)

            # Call the loss function
            total_loss, loss_MSE, loss_KLD = loss_mse(recon_batch, data, mu,
                                                      logvar)

            test_loss += total_loss.item()
            num_samples += len(data)

    # Calculate and append the average testing loss per sample for the epoch
    average_test_loss_per_sample = test_loss / num_samples
    test_losses.append(average_test_loss_per_sample)

    # Print or log the average testing loss per sample for the epoch
    if epoch % show_every == 0:
        print('====> Epoch: {} Average test loss per sample: {:.4f}'.format(
            epoch, average_test_loss_per_sample))
```

Model training

Encoder nodes per layer: [38, 50, 12, 12, 3]
Decoder nodes per layer: [3, 12, 12, 50, 38]

```
Vae(
  (linear1): Linear(in_features=38, out_features=50, bias=True)
  (lin_bn1): BatchNorm1d(50, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (linear2): Linear(in_features=50, out_features=12, bias=True)
  (lin_bn2): BatchNorm1d(12, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (linear3): Linear(in_features=12, out_features=12, bias=True)
  (lin_bn3): BatchNorm1d(12, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (fc1): Linear(in_features=12, out_features=3, bias=True)
  (bn1): BatchNorm1d(3, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (fc21): Linear(in_features=3, out_features=3, bias=True)
  (fc22): Linear(in_features=3, out_features=3, bias=True)
  (fc3): Linear(in_features=3, out_features=3, bias=True)
  (fc_bn3): BatchNorm1d(3, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (fc4): Linear(in_features=3, out_features=12, bias=True)
  (fc_bn4): BatchNorm1d(12, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (linear4): Linear(in_features=12, out_features=12, bias=True)
  (lin_bn4): BatchNorm1d(12, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (linear5): Linear(in_features=12, out_features=50, bias=True)
  (lin_bn5): BatchNorm1d(50, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (linear6): Linear(in_features=50, out_features=38, bias=True)
  (lin_bn6): BatchNorm1d(38, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu): ReLU()
)
```

```python
start_time = time.time()
for epoch in range(1, epochs + 1):
    train(epoch)
    test(epoch)

end_time = time.time()
print('Total training time: {:.0f} minutes {:.2f} seconds'.format(
    (end_time - start_time) // 60, (end_time - start_time) % 60))
```

When two curves have similar shapes but the testing loss is consistently higher than the training loss, it usually indicates that the model is overfitting to the training data. Overfitting happens when the model learns to perform well on the training data but fails to generalize to unseen data, as represented by the testing set.

class VaeModular(nn.Module)

CODE UPDATE

Key features of the `VaeModular` class include:

1. Modularity: The class is designed to separate the encoder, reparameterization, and decoder components into distinct modules. This modular structure allows for easy customization and experimentation with different architectures.

2. Flexibility: Users can easily modify the architecture of each module, such as changing the number of layers, hidden units, or activation functions, to suit their specific needs or domain requirements.

3. Reproducibility: The class ensures reproducibility by encapsulating the entire VAE model within a single object. This makes it easier to train, evaluate, and reproduce results across different experiments or datasets.

Overall, the `VaeModular` class provides a versatile and efficient framework for building and training Variational Autoencoder models, empowering users to explore and experiment with different architectures and hyperparameters to achieve optimal performance for their specific tasks.

```
class VaeModular(nn.Module):
    def __init__(
        self, D_in:int,
        hidden_nodes:list,
        latent_dim:int=3,
        batch_norm=True
    ):
        """
        Variational Autoencoder (VAE) with a modular architecture.

        Args:
            D_in (int): Input dimension.
            hidden_nodes (list): List of hidden layer sizes.
            latent_dim (int, optional): Size of the latent space. Defaults to 3.
            batch_norm (bool, optional): Whether to apply batch normalization.
            Defaults to True.
        """
        super(VaeModular, self).__init__()

        # Use the * operator to unpack the list of layers into individual
        # arguments for nn.Sequential

        # Encoder
        self.encoder = nn.Sequential(
            *concat_lin_layers(D_in, hidden_nodes, batch_norm)
        )
        self.out_features_ = self.encoder[-1][0].out_features

        # Latent vectors mu and logvar
        self.fc_latent = nn.Linear(self.out_features_, latent_dim)
        self.bn_latent = nn.BatchNorm1d(num_features=latent_dim)
        # Layer `fc_latent` is responsible for transforming the output of the
        # encoder layers into the latent space
        self.fc_mu = nn.Linear(latent_dim, latent_dim)
        self.fc_logvar = nn.Linear(latent_dim, latent_dim)

[ ... ]
```

CODE UPDATE

Refactoring Train Step

```
def train_step(
    model,
    optimizer,
    trainloader,
    device,
    loss_fn
):
    """
    Perform one training step for the VAE model.

    Args:
        - model (nn.Module): VAE model to be trained.
        - optimizer (torch.optim.Optimizer): Optimizer for model parameter
          updates.
        - trainloader (DataLoader): DataLoader for the training dataset.
        - device (torch.device): Device on which to perform computations ('cpu'
          or 'cuda').
        - loss_fn (callable): Loss function for calculating the training loss.

    Returns:
        tuple: A tuple containing the following elements:
            - average_train_loss_per_sample (float): Average training loss per
              sample for the epoch.
            - train_loss_mse_per_sample (float): Average MSE loss per sample for
              the epoch.
            - train_loss_kld_per_sample (float): Average KLD loss per sample for
              the epoch.
            - epoch_time (float): Time taken for the epoch in seconds.
    """
    model.train()

[ ... ]
```

Validation Step (prev. ‘test’)

CODE UPDATE

The `validation_step` method is responsible for evaluating the performance of the Variational Autoencoder (VAE) model on a validation dataset. Key features of the `validation_step` method include:

1. Validation Loop: It iterates over the validation data in batches, computing the reconstruction loss and the Kullback-Leibler Divergence (KLD) loss for each batch.

2. Loss Calculation: Similar to the training step, it calculates the reconstruction loss and the KLD loss for each batch of validation data.

3. Loss Aggregation: It aggregates the reconstruction loss and the KLD loss across all batches to obtain the total validation loss for the epoch.

4. Performance Evaluation: It evaluates the performance of the model on the validation dataset by computing metrics such as the average validation loss per sample.

5. Device Agnosticism: Similar to the training step, it supports evaluation on both CPU and GPU devices by dynamically moving data to the appropriate device based on user configuration.

```
def validate_step(
    model,
    valloader,
    loss_fn,
    device
):
    """
    Perform one validation step for the VAE model.

    Args:
        - model (nn.Module): VAE model to be evaluated.
        - valloader (DataLoader): DataLoader for the validation dataset.
        - device (torch.device): Device on which to perform computations ('cpu'
          or 'cuda').
        - loss_fn (callable): Loss function for calculating the validation loss.

    Returns:
        float: Average validation loss per sample for the epoch.
    """
    model.eval()
    val_loss = 0
    num_samples = 0

[ ... ]
```

CODE UPDATE

Fit method

The `fit` method, now part of the `VaeModular` class, is the core training function of the Variational Autoencoder (VAE) model. It orchestrates the training process over a specified number of epochs, during which the model learns to reconstruct input data and optimize its latent space representation. Key features of the `fit` method include:

1. Epoch-based Training Loop: It iterates over a specified number of epochs, during which the model undergoes training on the entire training dataset.

2. Training Progress Tracking: It tracks and records various training metrics such as training loss, reconstruction loss, Kullback-Leibler Divergence (KLD) loss, and training times at each epoch.

3. Batch-wise Training: The "Batch-wise Training" is performed within the `train_step` function and, if applicable, within the `validation_step` function. The `fit` method coordinates the batch-wise training process by calling these individual step functions iteratively over the specified number of epochs.

4. Validation (Optional): If provided, it optionally evaluates the model's performance on a separate validation dataset after each epoch, allowing for monitoring of model generalization and early stopping based on validation loss.

5. Device Agnosticism: It supports training on both CPU and GPU devices, seamlessly handling data movement and computation on the appropriate device

```
def fit(
        self,
        trainloader,
        optimizer,
        loss_fn,
        device,
        num_epochs=50,
        valloader=None,
        verbose=False,
        show_every=50
    ):
        """
        Train the VAE model using the provided data.

        Args:
            - trainloader (DataLoader): DataLoader for the training data.
            - optimizer (torch.optim.Optimizer): Optimizer for updating model
              parameters.
            - loss_fn: Loss function for computing the training loss.
            - device (torch.device): Device to be used for training (e.g.,
              'cuda' or 'cpu').
            - num_epochs (int, optional): Number of epochs for training. Default
              is 50.
            - valloader (DataLoader, optional): DataLoader for the validation
              data. Default is None.
            - verbose (bool, optional): Whether to print training progress.
              Default is False.
            - show_every (int, optional): Frequency of printing training
              progress. Default is 50.

        Returns:
            None

        Note:
            The training progress can be printed to the console if 'verbose' is
            set to True.
            The 'history' property of the model will be updated with training
            metrics.

        """

[ ... ]
```

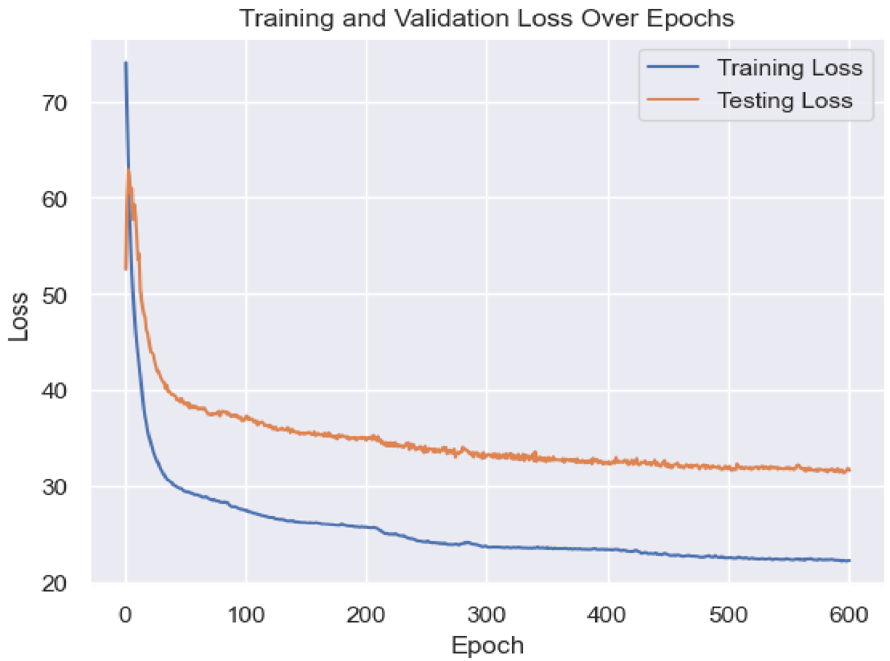
Model training

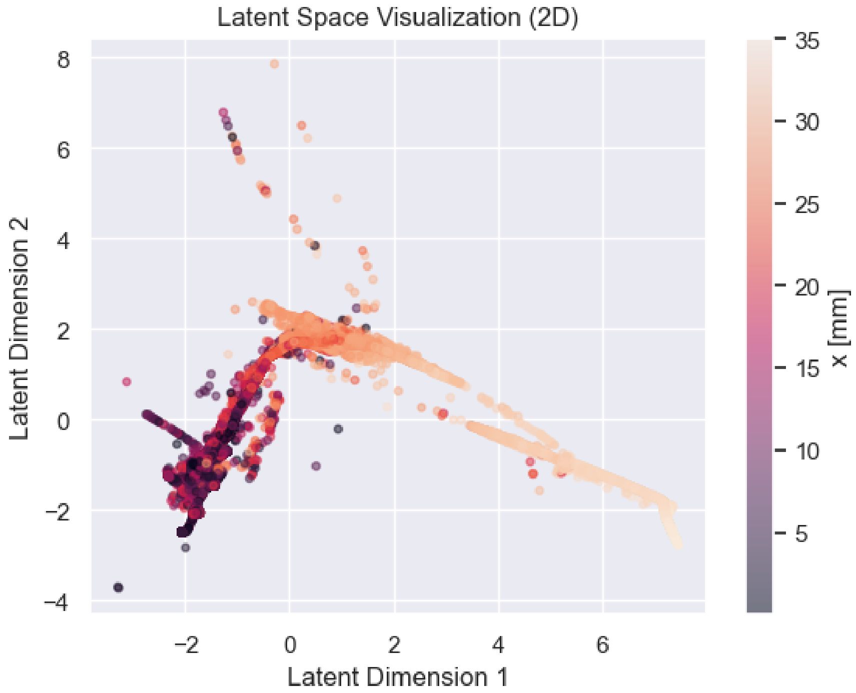
```
VaeModular(
  (encoder): Sequential(
    (0): Sequential(
      (0): Linear(in_features=38, out_features=50, bias=True)
      (1): BatchNorm1d(50, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU()
    )
    (1): Sequential(
      (0): Linear(in_features=50, out_features=12, bias=True)
      (1): BatchNorm1d(12, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU()
    )
    (2): Sequential(
      (0): Linear(in_features=12, out_features=12, bias=True)
      (1): BatchNorm1d(12, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU()
    )
  )

  (fc_latent): Linear(in_features=12, out_features=3, bias=True)
  (bn_latent): BatchNorm1d(3, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)

  (fc_mu): Linear(in_features=3, out_features=3, bias=True)
  (fc_logvar): Linear(in_features=3, out_features=3, bias=True)
  (fc_z): Linear(in_features=3, out_features=3, bias=True)
  (bn_z): BatchNorm1d(3, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

```
  (latent_to_output): Linear(in_features=3, out_features=12, bias=True)
  (bn_lto): BatchNorm1d(12, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)

  (decoder): Sequential(
    (0): Sequential(
      (0): Linear(in_features=12, out_features=12, bias=True)
      (1): BatchNorm1d(12, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU()
    )
    (1): Sequential(
      (0): Linear(in_features=12, out_features=50, bias=True)
      (1): BatchNorm1d(50, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU()
    )
    (2): Sequential(
      (0): Linear(in_features=50, out_features=38, bias=True)
      (1): BatchNorm1d(38, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
)
```

Model training

Encoder nodes per layer: [38, 50, 12, 12, 3]
Decoder nodes per layer: [3, 12, 12, 50, 38]





Next Steps

Possible Solutions to Overfitting:

1. **Hyperparameter Tuning:**
   Perform systematic tuning of hyperparameters using methods like grid search or optimization libraries like Optuna to find configurations that generalize well.
2. **Test Simpler Model Architecture:**
   Experiment with reducing the complexity of the model to avoid overfitting.
3. **Include a Warm-Up Phase:**
   Gradually increase the learning rate or anneal other hyperparameters at the beginning of training to stabilize the optimization process.
4. **Introduce Regularization Techniques:**
   Utilize techniques such as dropout regularization to prevent overfitting by randomly dropping units during training.

Other Enhancements:

1. **Refactor the EDA and Data Preparation:**
   Separate and streamline the code for Exploratory Data Analysis (EDA) and data preparation to optimize workflow efficiency and avoid unnecessary computations during VAE testing.

Try SDV (The Synthetic Data Vault)