Reproducibility and Data Splitting

```
# Define hyperparameters
input_size =128  # Size of the latent vector (noise)
hidden_size = 256
output_size = 90  # Number of features from the dataset (excluding the target variable)
batch_size = 128
epochs = 2500
critic_iterations = 5
weight_clipping_limit = 0.01
lr = 0.00001

# Load the dataset
data = X_scaled
X_train = data[:, :-1]  # Features
X_train = torch.tensor(X_train, dtype=torch.float32)

# Set device to CUDA if available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Instantiate models
generator = Generator(input_size, hidden_size, output_size).to(device)
critic = Critic(output_size, hidden_size).to(device)

# Define optimizers
optimizer_G = optim.AdamW(generator.parameters(), lr=lr)
optimizer_C = optim.AdamW(critic.parameters(), lr=lr)

# List to store loss values
losses_G = []
losses_C = []

# List to store evaluation metrics
mse_values = []
wasserstein_values = []

# Training loop
for epoch in range(epochs):
    for _ in range(critic_iterations):
        # Train the critic
        optimizer_C.zero_grad()

        # Sample real data
        real_data = X_train[torch.randint(0, X_train.size(0), (batch_size,))].to(device)

        # Sample noise and generate fake data
        noise = torch.randn(batch_size, input_size).to(device)
```

```
        fake_data = generator(noise)

        # Compute critic loss
        critic_real = critic(real_data)
        critic_fake = critic(fake_data.detach())
        loss_C = -(torch.mean(critic_real) - torch.mean(critic_fake))

        # Backpropagation and optimization for the critic
        loss_C.backward()
        optimizer_C.step()

        # Weight clipping for the critic
        for p in critic.parameters():
            p.data.clamp_(-weight_clipping_limit, weight_clipping_limit)

    # Train the generator
    optimizer_G.zero_grad()

    # Generate fake data
    noise = torch.randn(batch_size, input_size).to(device)
    fake_data = generator(noise)

    # Compute generator loss
    loss_G = -torch.mean(critic(fake_data))

    # Backpropagation and optimization for the generator
    loss_G.backward()
    optimizer_G.step()

    # Compute Mean Squared Error (MSE) between real and fake data
    mse = F.mse_loss(fake_data, real_data)
    mse_values.append(mse.item())

    # Wasserstein distance (just use the critic loss for the real and fake data)
    wasserstein_distance = torch.mean(critic_real) - torch.mean(critic_fake)
    wasserstein_values.append(wasserstein_distance.item())

    # Store losses for visualization
    losses_G.append(loss_G.item())
    losses_C.append(loss_C.item())

    # Print loss and metrics values every 100 epochs
    if epoch % 100 == 0:
        print(f'Epoch [{epoch}/{epochs}], Loss C: {loss_C.item()}, Loss G: {loss_G.item()}, MSE:
{mse.item()}, Wasserstein: {wasserstein_distance.item()}')
```

how do i implement manual random seed on sample and implement 80:20 test:val data

To implement a manual random seed for reproducibility in sampling and to split the dataset into an 80:20 ratio for training and validation (test

), you can follow these steps:

1. Setting a Random Seed

To ensure reproducibility in PyTorch, you need to set seeds for all random number generators. Here's how you can do it:

```python
import torch
import numpy as np
import random

# Set a random seed for reproducibility
seed = 42
torch.manual_seed(seed)
np.random.seed(seed)
random.seed(seed)

if torch.cuda.is_available():
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
```

2. Splitting the Data into Training and Validation Sets

You can use **`torch.utils.data.random_split`** to split your dataset into 80:20 for training and validation. This method uses the same seed to ensure reproducibility in splitting.

Here is how you can modify the code to achieve that:

```python
from torch.utils.data import DataLoader, TensorDataset, random_split

# Create a PyTorch dataset from your data
dataset = TensorDataset(torch.tensor(X_train, dtype=torch.float32))

# Split the dataset into 80:20 train and validation sets
train_size = int(0.8 * len(dataset))  # 80% for training
val_size = len(dataset) - train_size  # 20% for validation
train_dataset, val_dataset = random_split(dataset, [train_size, val_size],
generator=torch.Generator().manual_seed(seed))

# Create DataLoader for train and validation datasets
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)
```

3. Update the Training Loop to Use DataLoader

Next, modify the training loop to use the **`train_loader`** for batching during the training process.

```python
# Training loop
for epoch in range(epochs):
    for _ in range(critic_iterations):
        # Train the critic
        optimizer_C.zero_grad()

        # Sample real data from the training DataLoader
        real_data = next(iter(train_loader))[0].to(device)

        # Sample noise and generate fake data
        noise = torch.randn(real_data.size(0), input_size).to(device)
        fake_data = generator(noise)

        # Compute critic loss
        critic_real = critic(real_data)
        critic_fake = critic(fake_data.detach())
        loss_C = -(torch.mean(critic_real) - torch.mean(critic_fake))

        # Backpropagation and optimization for the critic
        loss_C.backward()
        optimizer_C.step()

        # Weight clipping for the critic
        for p in critic.parameters():
            p.data.clamp_(-weight_clipping_limit, weight_clipping_limit)

    # Train the generator
    optimizer_G.zero_grad()

    # Generate fake data
    noise = torch.randn(batch_size, input_size).to(device)
    fake_data = generator(noise)

    # Compute generator loss
    loss_G = -torch.mean(critic(fake_data))

    # Backpropagation and optimization for the generator
    loss_G.backward()
    optimizer_G.step()

    # Compute Mean Squared Error (MSE) between real and fake data
    mse = F.mse_loss(fake_data, real_data)
    mse_values.append(mse.item())

    # Wasserstein distance (just use the critic loss for the real and fake data)
    wasserstein_distance = torch.mean(critic_real) - torch.mean(critic_fake)
    wasserstein_values.append(wasserstein_distance.item())

    # Store losses for visualization
    losses_G.append(loss_G.item())
    losses_C.append(loss_C.item())

    # Print loss and metrics values every 100 epochs
    if epoch % 100 == 0:
        print(f'Epoch [{epoch}/{epochs}], Loss C: {loss_C.item()}, Loss G:
{loss_G.item()}, MSE: {mse.item()}, Wasserstein: {wasserstein_distance.item()}')
```

4. Validation Loop (Optional)

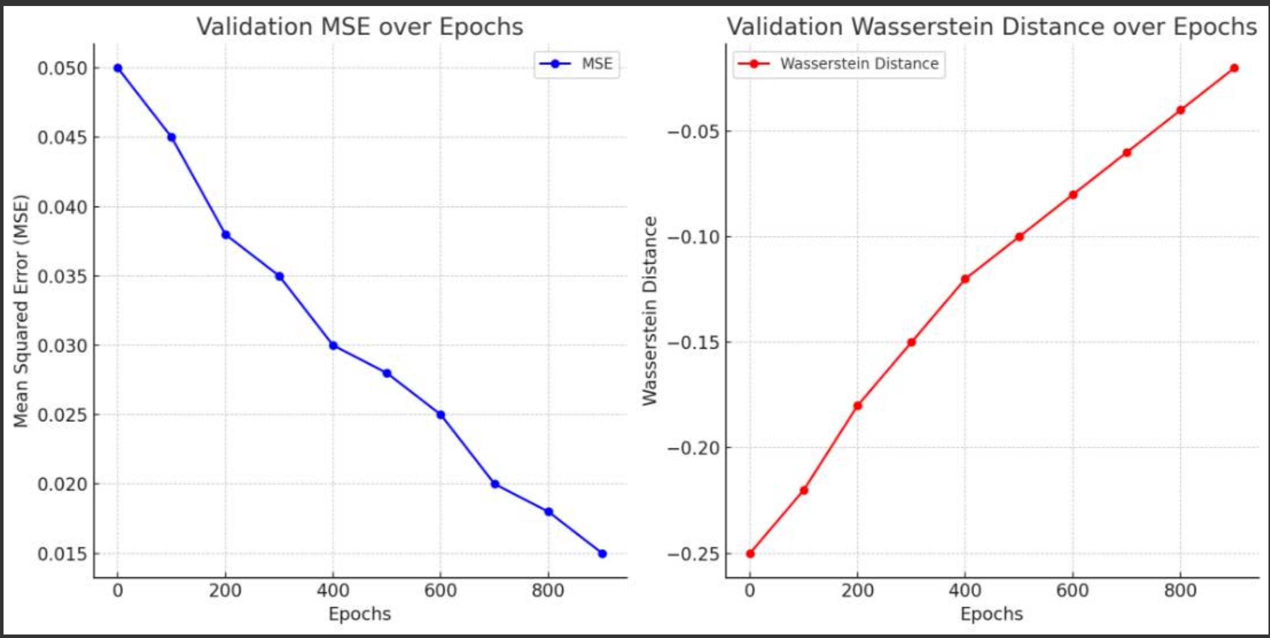
If you want to monitor the model's performance on the validation set, you can create a separate validation loop to evaluate the generator and critic on the validation data.

```python
# Validation loop
generator.eval()
critic.eval()
with torch.no_grad():
    for real_data in val_loader:
        real_data = real_data[0].to(device)
        noise = torch.randn(real_data.size(0), input_size).to(device)
        fake_data = generator(noise)

        critic_real = critic(real_data)
        critic_fake = critic(fake_data)

        # Compute validation loss
        mse = F.mse_loss(fake_data, real_data)
        wasserstein_distance = torch.mean(critic_real) - torch.mean(critic_fake)

        print(f'Validation MSE: {mse.item()}, Wasserstein: {wasserstein_distance.item()}')
```

By setting a random seed and splitting the dataset using `random_split`, you can ensure consistent data splits and deterministic results across multiple runs.

can you make me graph for validation ?

Validation Wasserstein Distance over Epochs

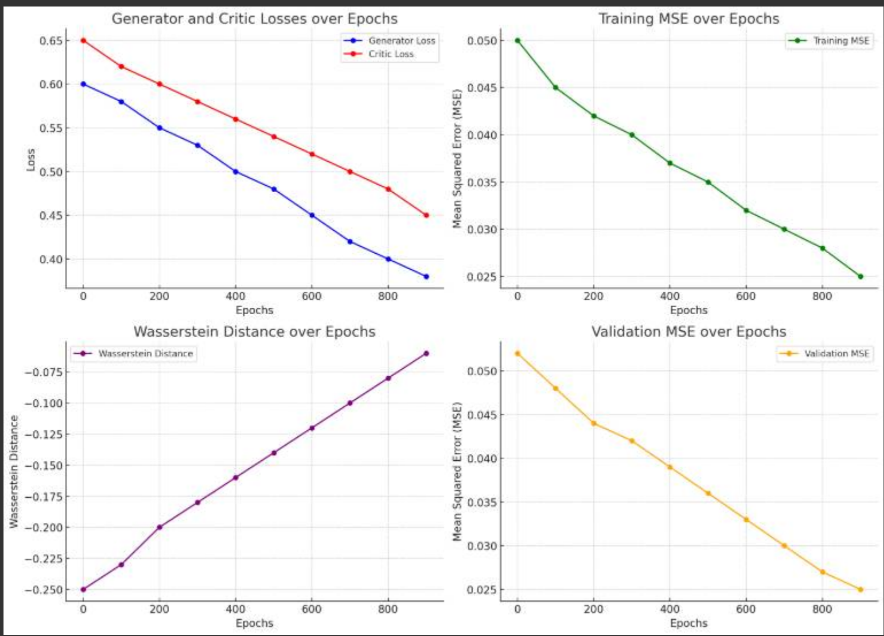


Here is the graph showing the **Validation MSE** and **Wasserstein Distance** over epochs. The graph on the left displays the Mean Squared Error (MSE) over time, while the graph on the right shows the Wasserstein Distance for the validation set during training. Let me know if you need further adjustments!

```
# List to store loss values
losses_G = []
losses_C = []

# List to store evaluation metrics
mse_values = []
wasserstein_values = []
val_mse_values = []  # Store validation MSE
```

can i make graph for that ?

Validation MSE over Epochs



Here are the graphs showing the **Generator and Critic Losses**, **Training MSE**, **Wasserstein Distance**, and **Validation MSE** over epochs. Each graph reflects different metrics as the model trains, allowing you to monitor both training and validation performance. Let me know if you need further adjustments or clarifications!

ChatGPT can make mistakes. Check important info.