OpenAI

Training GAN with Validation

```
# 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 sample and implement 80:20 test:val data and integrate it to code above

To integrate a manual random seed, along with an 80:20 split for training and validation data in your code, you need to:

1. **Set a manual seed** for reproducibility.
2. **Split the dataset** into 80% training and 20% validation.
3. Ensure that the validation is performed periodically and appropriately, perhaps at every epoch, to track performance.

Here's an updated version of your code to reflect these additions:

```python
import torch
import torch.optim as optim
import torch.nn.functional as F
from sklearn.model_selection import train_test_split

# 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
seed = 42  # Manual seed

# Set manual seed for reproducibility
torch.manual_seed(seed)
if torch.cuda.is_available():
    torch.cuda.manual_seed(seed)

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

# Split the data into training (80%) and validation (20%)
train_data, val_data = train_test_split(X_train_data, test_size=0.2, random_state=seed)

# Convert validation data to torch tensor
val_data = torch.tensor(val_data, 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 = []
val_mse_values = []  # Store validation MSE

# Training loop
```

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

            # Sample real data from training data
            real_data = train_data[torch.randint(0, train_data.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())

        # Validate the generator performance using validation data
        with torch.no_grad():
            val_real_data = val_data[torch.randint(0, val_data.size(0),
    (batch_size,))].to(device)
            val_fake_data = generator(torch.randn(batch_size, input_size).to(device))
            val_mse = F.mse_loss(val_fake_data, val_real_data)
            val_mse_values.append(val_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()}, '
                  f'Val MSE: {val_mse.item()}, Wasserstein: {wasserstein_distance.item()}')
```

Key changes:

1. **Manual Seed**: Set the seed for reproducibility using `` `torch.manual_seed(seed)` `` and `` `torch.cuda.manual_seed(seed)` `` for GPU operations.

2. **Train-test Split**: The dataset is split into an 80:20 ratio for training and validation using `train_test_split`.
3. **Validation MSE**: In each epoch, after updating the generator, the code evaluates the model using validation data and stores the results in `val_mse_values`.
4. **Epoch Printing**: The output now includes `Val MSE` in each printed statement to track validation performance.

This structure ensures that your model is validated periodically, and you can monitor both the training and validation MSE during the training process.

```
# Hyperparameters
input_size = 128  # Size of the noise vector
hidden_size = 256
output_size = 90  # Number of features in the dataset (exclude the target column)
lr = 0.00001  # 1e-5 Learning rate
batch_size = 128
n_epochs = 2500
n_critic = 5  # Critic steps per generator step
lambda_gp = 10  # Weight for gradient penalty

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

# Set device
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)

# Optimizers
# optimizer_G =   optim.Adam(generator.parameters(), lr=lr, betas=(0.5, 0.9))
# optimizer_C = optim.Adam(critic.parameters(), lr=lr, betas=(0.5, 0.9))


optimizer_G = AdamW(generator.parameters(), lr=lr, betas=(0.5, 0.9), weight_decay=1e-4)
optimizer_C = AdamW(critic.parameters(), lr=lr, betas=(0.5, 0.9), weight_decay=1e-4)




# Hyperparameters for early stopping
patience = 1000  # Number of epochs to wait for improvement
min_delta = 0.001  # Minimum change in the monitored loss to qualify as improvement
best_loss = float('inf')  # Initialize best loss with a very large number
early_stop_counter = 0  # Counter to track patience
```

```
# Training loop with early stopping
critic_losses = []
generator_losses = []

for epoch in range(n_epochs):
    for _ in range(n_critic):
        # Train the critic
        real_samples = X_train[torch.randint(0, X_train.size(0), (batch_size,))].to(device)
        latent_space_samples = torch.randn((batch_size, input_size)).to(device)
        fake_samples = generator(latent_space_samples).detach()

        critic_real = critic(real_samples)
        critic_fake = critic(fake_samples)

        gradient_penalty = compute_gradient_penalty(critic, real_samples, fake_samples)
        loss_critic = -torch.mean(critic_real) + torch.mean(critic_fake) + lambda_gp * gradient_penalty

        optimizer_C.zero_grad()
        loss_critic.backward()
        optimizer_C.step()

    # Train the generator
    latent_space_samples = torch.randn((batch_size, input_size)).to(device)
    fake_samples = generator(latent_space_samples)
    loss_generator = -torch.mean(critic(fake_samples))

    optimizer_G.zero_grad()
    loss_generator.backward()
    optimizer_G.step()

    # Store losses
    critic_losses.append(loss_critic.item())
    generator_losses.append(loss_generator.item())

    # Check for early stopping
    if loss_generator.item() < best_loss - min_delta:
        best_loss = loss_generator.item()  # Update best loss
        early_stop_counter = 0  # Reset counter if improvement
        # Optionally save the best model
        # torch.save(generator.state_dict(), "best_generator.pth")
        # torch.save(critic.state_dict(), "best_critic.pth")
    else:
        early_stop_counter += 1  # Increment the counter

    if early_stop_counter >= patience:
        print(f"Early stopping at epoch {epoch}")
        break  # Stop training if patience is exceeded

    # Print losses occasionally
    if epoch % 100 == 0:
        print(f"Epoch [{epoch}/{n_epochs}] | Critic Loss: {loss_critic.item()} | Generator Loss:
```

{loss_generator.item()}")

how do i implement manual random seed sample and implement 80:20 test:val data and integrate it to code above too

To implement a manual random seed for reproducibility and an 80:20 train-validation split in your code, follow these steps:

1. **Manual Random Seed**: You can set a random seed using `torch.manual_seed()` for PyTorch operations and `random.seed()` for Python's built-in random operations.
2. **Train-Validation Split**: Use `train_test_split` from `sklearn` to split your dataset into 80% training data and 20% validation data.

Here is your updated code with these modifications:

python

```
import torch
import torch.optim as optim
from torch.optim import AdamW
from sklearn.model_selection import train_test_split
import random

# Hyperparameters
input_size = 128  # Size of the noise vector
hidden_size = 256
output_size = 90  # Number of features in the dataset (exclude the target column)
lr = 0.00001  # Learning rate
batch_size = 128
n_epochs = 2500
n_critic = 5  # Critic steps per generator step
lambda_gp = 10  # Weight for gradient penalty
seed = 42  # Manual seed for reproducibility

# Set manual random seed for reproducibility
random.seed(seed)
torch.manual_seed(seed)
if torch.cuda.is_available():
    torch.cuda.manual_seed(seed)

# Load and split data (80% train, 20% validation)
data = X_scaled
X_train_data = data[:, :-1]  # Features
X_train_data = torch.tensor(X_train_data, dtype=torch.float32)

# Train-validation split
train_data, val_data = train_test_split(X_train_data, test_size=0.2, random_state=seed)

# Convert validation data to PyTorch tensor
val_data = torch.tensor(val_data, dtype=torch.float32)

# Set device
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)

# Optimizers
optimizer_G = AdamW(generator.parameters(), lr=lr, betas=(0.5, 0.9), weight_decay=1e-4)
optimizer_C = AdamW(critic.parameters(), lr=lr, betas=(0.5, 0.9), weight_decay=1e-4)
```

```python
# Hyperparameters for early stopping
patience = 1000  # Number of epochs to wait for improvement
min_delta = 0.001  # Minimum change in the monitored loss to qualify as improvement
best_loss = float('inf')  # Initialize best loss with a very large number
early_stop_counter = 0  # Counter to track patience

# Training loop with early stopping
critic_losses = []
generator_losses = []
val_mse_values = []  # To store validation MSE for tracking performance

for epoch in range(n_epochs):
    for _ in range(n_critic):
        # Train the critic
        real_samples = train_data[torch.randint(0, train_data.size(0),
(batch_size,))].to(device)
        latent_space_samples = torch.randn((batch_size, input_size)).to(device)
        fake_samples = generator(latent_space_samples).detach()

        critic_real = critic(real_samples)
        critic_fake = critic(fake_samples)

        gradient_penalty = compute_gradient_penalty(critic, real_samples, fake_samples)
        loss_critic = -torch.mean(critic_real) + torch.mean(critic_fake) + lambda_gp *
gradient_penalty

        optimizer_C.zero_grad()
        loss_critic.backward()
        optimizer_C.step()

    # Train the generator
    latent_space_samples = torch.randn((batch_size, input_size)).to(device)
    fake_samples = generator(latent_space_samples)
    loss_generator = -torch.mean(critic(fake_samples))

    optimizer_G.zero_grad()
    loss_generator.backward()
    optimizer_G.step()

    # Store losses
    critic_losses.append(loss_critic.item())
    generator_losses.append(loss_generator.item())

    # Validation step: Use validation data to calculate MSE
    with torch.no_grad():
        val_real_samples = val_data[torch.randint(0, val_data.size(0),
(batch_size,))].to(device)
        val_latent_space_samples = torch.randn((batch_size, input_size)).to(device)
        val_fake_samples = generator(val_latent_space_samples)
        val_mse = torch.mean((val_fake_samples - val_real_samples) ** 2).item()
        val_mse_values.append(val_mse)

    # Check for early stopping
    if loss_generator.item() < best_loss - min_delta:
        best_loss = loss_generator.item()  # Update best loss
        early_stop_counter = 0  # Reset counter if improvement
        # Optionally save the best model
        # torch.save(generator.state_dict(), "best_generator.pth")
        # torch.save(critic.state_dict(), "best_critic.pth")
    else:
        early_stop_counter += 1  # Increment the counter

    if early_stop_counter >= patience:
        print(f"Early stopping at epoch {epoch}")
        break  # Stop training if patience is exceeded

    # Print losses occasionally
    if epoch % 100 == 0:
        print(f"Epoch [{epoch}/{n_epochs}] | Critic Loss: {loss_critic.item()} | Generator
Loss: {loss_generator.item()} | Validation MSE: {val_mse}")
```

Key Changes:

1. **Manual Random Seed**: The manual random seed is set using `random.seed()` for Python operations and `torch.manual_seed()` for PyTorch, ensuring reproducibility.
2. **Train-Validation Split**: The data is split into 80% training and 20% validation using `train_test_split` from `sklearn`, with the split controlled by `random_state=seed` for consistency.
3. **Validation Step**: During each epoch, the validation data is used to compute the mean squared error (MSE) between the generated fake samples and the real validation samples. This is stored in the `val_mse_values` list for tracking.
4. **Early Stopping**: Early stopping is based on the generator loss (`loss_generator`) and is controlled by `min_delta` and `patience`. Training stops if the loss doesn't improve over a specified number of epochs.

This setup now allows you to track both training and validation performance, as well as apply early stopping to prevent overfitting.

can you implement this

mse_values = []
wasserstein_values = []

to code above?

To integrate the calculation and storage of **MSE values** and **Wasserstein distance values** into the training loop of the code, you can follow these steps:

1. **MSE Values**: These will be computed between the real and generated (fake) data for both training and validation data.
2. **Wasserstein Values**: These are simply the difference between the mean critic scores for real and fake data, which you're already calculating during the training process.

Here's the updated version of your code with the addition of **MSE values** and **Wasserstein values** tracking:

python

```
import torch
import torch.optim as optim
from torch.optim import AdamW
from sklearn.model_selection import train_test_split
import random

# Hyperparameters
input_size = 128  # Size of the noise vector
hidden_size = 256
output_size = 90  # Number of features in the dataset (exclude the target column)
lr = 0.00001  # Learning rate
batch_size = 128
```

```python
n_epochs = 2500
n_critic = 5  # Critic steps per generator step
lambda_gp = 10  # Weight for gradient penalty
seed = 42  # Manual seed for reproducibility

# Set manual random seed for reproducibility
random.seed(seed)
torch.manual_seed(seed)
if torch.cuda.is_available():
    torch.cuda.manual_seed(seed)

# Load and split data (80% train, 20% validation)
data = X_scaled
X_train_data = data[:, :-1]  # Features
X_train_data = torch.tensor(X_train_data, dtype=torch.float32)

# Train-validation split
train_data, val_data = train_test_split(X_train_data, test_size=0.2, random_state=seed)

# Convert validation data to PyTorch tensor
val_data = torch.tensor(val_data, dtype=torch.float32)

# Set device
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)

# Optimizers
optimizer_G = AdamW(generator.parameters(), lr=lr, betas=(0.5, 0.9), weight_decay=1e-4)
optimizer_C = AdamW(critic.parameters(), lr=lr, betas=(0.5, 0.9), weight_decay=1e-4)

# Hyperparameters for early stopping
patience = 1000  # Number of epochs to wait for improvement
min_delta = 0.001  # Minimum change in the monitored loss to qualify as improvement
best_loss = float('inf')  # Initialize best loss with a very large number
early_stop_counter = 0  # Counter to track patience

# Training loop with early stopping
critic_losses = []
generator_losses = []

# Lists to store evaluation metrics
mse_values = []
wasserstein_values = []
val_mse_values = []  # To store validation MSE for tracking performance

for epoch in range(n_epochs):
    for _ in range(n_critic):
        # Train the critic
        real_samples = train_data[torch.randint(0, train_data.size(0),
(batch_size,))].to(device)
        latent_space_samples = torch.randn((batch_size, input_size)).to(device)
        fake_samples = generator(latent_space_samples).detach()

        critic_real = critic(real_samples)
        critic_fake = critic(fake_samples)

        # Compute Wasserstein distance
        wasserstein_distance = torch.mean(critic_real) - torch.mean(critic_fake)

        # Compute gradient penalty
        gradient_penalty = compute_gradient_penalty(critic, real_samples, fake_samples)
        loss_critic = -wasserstein_distance + lambda_gp * gradient_penalty

        optimizer_C.zero_grad()
        loss_critic.backward()
        optimizer_C.step()

    # Train the generator
    latent_space_samples = torch.randn((batch_size, input_size)).to(device)
    fake_samples = generator(latent_space_samples)
    loss_generator = -torch.mean(critic(fake_samples))
```

```
    optimizer_G.zero_grad()
    loss_generator.backward()
    optimizer_G.step()

    # Store losses
    critic_losses.append(loss_critic.item())
    generator_losses.append(loss_generator.item())

    # Compute MSE between real and fake data (training set)
    mse = torch.mean((fake_samples - real_samples) ** 2).item()
    mse_values.append(mse)

    # Compute Wasserstein distance
    wasserstein_values.append(wasserstein_distance.item())

    # Validation step: Use validation data to calculate MSE
    with torch.no_grad():
        val_real_samples = val_data[torch.randint(0, val_data.size(0),
(batch_size,))].to(device)
        val_latent_space_samples = torch.randn((batch_size, input_size)).to(device)
        val_fake_samples = generator(val_latent_space_samples)
        val_mse = torch.mean((val_fake_samples - val_real_samples) ** 2).item()
        val_mse_values.append(val_mse)

    # Check for early stopping
    if loss_generator.item() < best_loss - min_delta:
        best_loss = loss_generator.item()  # Update best loss
        early_stop_counter = 0  # Reset counter if improvement
        # Optionally save the best model
        # torch.save(generator.state_dict(), "best_generator.pth")
        # torch.save(critic.state_dict(), "best_critic.pth")
    else:
        early_stop_counter += 1  # Increment the counter

    if early_stop_counter >= patience:
        print(f"Early stopping at epoch {epoch}")
        break  # Stop training if patience is exceeded

    # Print losses occasionally
    if epoch % 100 == 0:
        print(f"Epoch [{epoch}/{n_epochs}] | Critic Loss: {loss_critic.item()} | Generator
Loss: {loss_generator.item()} | "
              f"Train MSE: {mse} | Val MSE: {val_mse} | Wasserstein Distance:
{wasserstein_distance.item()}")
```

Breakdown of New Code Segments:

1. **MSE Values Calculation (Training and Validation)**:
    - MSE between the generated fake data and the real data is computed during both the training and validation steps.
    - For training MSE:

      ```python
      mse = torch.mean((fake_samples - real_samples) ** 2).item()
      mse_values.append(mse)
      ```

    - For validation MSE:

```python
val_mse = torch.mean((val_fake_samples - val_real_samples) ** 2).item()
val_mse_values.append(val_mse)
```

2. **Wasserstein Distance Calculation**:
   - Wasserstein distance is already part of the critic's loss calculation, which is simply the difference between the real and fake critic scores:

```python
wasserstein_distance = torch.mean(critic_real) - torch.mean(critic_fake)
wasserstein_values.append(wasserstein_distance.item())
```

3. **Print Statements**:
   - The losses and evaluation metrics, including **training MSE**, **validation MSE**, and **Wasserstein distance**, are printed every 100 epochs:

```python
print(f"Epoch [{epoch}/{n_epochs}] | Critic Loss: {loss_critic.item()} | Generator Loss: {loss_generator.item()} | "
      f"Train MSE: {mse} | Val MSE: {val_mse} | Wasserstein Distance: {wasserstein_distance.item()}")
```

This will allow you to track the **MSE** for both training and validation datasets, as well as the **Wasserstein distance** over epochs.