Implementasi Wasserstein GAN

total merge dataset cleaned ready

	NPM	IPK	Klasifikasi Profesi	ALGORITMA DA
1	50414234	3.3	Software Engineer	4
2	50414279	3.1	Data Engineer	3
3	50414496	3.25	Data Analyst	3
4	50414623	3.38	Software Engineer	4
5	50414925	3.72	Mobile App Developer	4

ini adalah dataset yang bersih, bantu saya mencoba untuk EDA sampai Training menggunakan Wasserstein GAN untuk implementasi prediksi profesi pekerjaan mahasiswa menggunakan Tensorflow

python

```python
import pandas as pd

# Load the dataset
file_path = '/mnt/data/total_merge_dataset_cleaned_ready.csv'
df = pd.read_csv(file_path)

# Display the first few rows to understand the data structure
df.head()
```

Result

```
        NPM   IPK   Klasifikasi Profesi  ALGORITMA DAN PEMROGRAMAN 1  \
0  50414234  3.30     Software Engineer                            4
1  50414279  3.10         Data Engineer                            3
2  50414496  3.25          Data Analyst                            3
3  50414623  3.38     Software Engineer                            4
4  50414925  3.72  Mobile App Developer                            4

   ALGORITMA DAN PEMROGRAMAN 2  ALGORITMA DAN PEMROGRAMAN 3  \
0                            3                            4
1                            3                            3
2                            3                            3
3                            3                            3
4                            3                            4

   ALGORITMA DAN PENGOLAHAN PARALEL  ALGORITMA DEEP LEARNING  \
0                                 3                        4
1                                 3                        2
```

```
2                              3                       4
3                              3                       3
4                              4                       4

   ARSITEKTUR KOMPUTER  BAHASA INDONESIA  ...  SISTEM TERDISTRIBUSI  \
0                    4                 4  ...                     3
1                    3                 3  ...                     4
2                    4                 3  ...                     4
3                    4                 3  ...                     3
4                    4                 3  ...                     0

   SKRIPSI / EVALUASI KOMPREHENSIF  STATISTIKA 1  STATISTIKA 2  STRUKTUR DATA  \
0                                4             2             4              2
1                                4             3             3              3
2                                4             4             2              3
3                                3             4             4              3
4                                4             4             4              4

   TEKNIK KOMPILASI  TEKNOLOGI GAME  TEKNOLOGI KECERDASAN ARTIFICIAL  \
0                 3               3                                0
1                 3               4                                0
2                 4               4                                0
3                 4               4                                0
4                 4               4                                0

   TEORI BAHASA DAN OTOMATA  TERAPAN TEORI GRAF
0                         3                   0
1                         3                   0
2                         3                   0
3                         3                   0
4                         3                   0

[5 rows x 94 columns]
```

Dataset ini berisi berbagai atribut yang relevan dengan prestasi akademik dan bidang studi mahasiswa. Untuk langkah awal, kita akan melakukan **Exploratory Data Analysis (EDA)** dan mempersiapkan dataset untuk model *Wasserstein GAN* dengan menggunakan TensorFlow.

Langkah-langkah EDA dan preprocessing:

1. **Cek missing values** dan distribusi variabel.
2. **Encode** fitur kategorikal, khususnya kolom *"Profesi"* (target).
3. **Feature scaling** untuk kolom numerik.
4. **Pisahkan dataset** menjadi fitur dan target.

Mari kita mulai dengan EDA dasar:

1. **Cek missing values** dan distribusi data.

```python
# Check for missing values
missing_values = df.isnull().sum()

# Check data distribution for the target column 'Profesi'
profesi_distribution = df['Klasifikasi Profesi'].value_counts()

missing_values, profesi_distribution
```

```
Result
(NPM                                0
 IPK                                0
 Klasifikasi Profesi                0
 ALGORITMA DAN PEMROGRAMAN 1        0
 ALGORITMA DAN PEMROGRAMAN 2        0
                                   ..
```

```
TEKNIK KOMPILASI                   0
TEKNOLOGI GAME                     0
TEKNOLOGI KECERDASAN ARTIFICIAL    0
TEORI BAHASA DAN OTOMATA           0
TERAPAN TEORI GRAF                 0
Length: 94, dtype: int64,
Software Engineer             42
Software Quality Assurance    32
Mobile App Developer          16
Full Stack Developer          15
Data Analyst                  12
Back End Developer            11
Data Engineer                  7
Graphic Designer               6
UI/UX Designer                 5
Web Developer                  5
IT Entrepreneur                5
Front End Developer            5
IT Project Manager             4
Network Engineer               4
Network Administrator          3
Data Scientist                 3
Business Analyst               3
Database Administrator         3
Lecturer                       3
IT Consultant                  3
IT Lecturer                    2
DevOps Engineer                2
Network Security Engineer      1
Cyber Security Manager         1
System Analyst                 1
Software Quality Engineer      1
Cyber Security Analyst         1
Name: Klasifikasi Profesi, dtype: int64)
```

EDA awal menunjukkan bahwa tidak ada **missing values** di dataset ini, sehingga kita bisa langsung melanjutkan ke tahap berikutnya. Distribusi untuk kolom target *"Klasifikasi Profesi"* juga menunjukkan variasi dalam profesi yang dipilih oleh mahasiswa, dengan beberapa profesi dominan seperti *Software Engineer* dan *Software Quality Assurance*.
Berikutnya, kita akan melakukan beberapa langkah persiapan:

1. **Encoding** kolom *"Klasifikasi Profesi"* menjadi nilai numerik.
2. **Feature scaling** untuk kolom numerik agar semua fitur memiliki skala yang sama, penting untuk training menggunakan GAN.

Mari kita mulai dengan **encoding** target dan melakukan **feature scaling**.

```python
from sklearn.preprocessing import LabelEncoder, StandardScaler

# Encode the target variable 'Klasifikasi Profesi'
label_encoder = LabelEncoder()
df['Klasifikasi Profesi Encoded'] = label_encoder.fit_transform(df['Klasifikasi Profesi'])

# Select numeric columns (excluding NPM and Profesi related columns)
numeric_cols = df.columns[2:-1]

# Scale the numeric features
scaler = StandardScaler()
df_scaled = df.copy()
df_scaled[numeric_cols] = scaler.fit_transform(df[numeric_cols])

# Separate the features and target
X = df_scaled[numeric_cols]
y = df_scaled['Klasifikasi Profesi Encoded']
```

```
# Display the first few rows after scaling and encoding
df_scaled.head()
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipykernel_12/1203448472.py in ?()
      9 
     10 # Scale the numeric features
     11 scaler = StandardScaler()
     12 df_scaled = df.copy()
---> 13 df_scaled[numeric_cols] = scaler.fit_transform(df[numeric_cols])
     14 
     15 # Separate the features and target
     16 X = df_scaled[numeric_cols]

~/.local/lib/python3.11/site-packages/sklearn/base.py in ?(self, X, y, **fit_params)
    863         # non-optimized default implementation; override when a better
    864         # method is possible for a given clustering algorithm
    865         if y is None:
    866             # fit method of arity 1 (unsupervised transformation)
--> 867             return self.fit(X, **fit_params).transform(X)
    868         else:
    869             # fit method of arity 2 (supervised transformation)
    870             return self.fit(X, y, **fit_params).transform(X)

~/.local/lib/python3.11/site-packages/sklearn/preprocessing/_data.py in ?(self, X, y, sample_weight)
    805             Fitted scaler.
    806         """
    807         # Reset internal state before fitting
    808         self._reset()
--> 809         return self.partial_fit(X, y, sample_weight)

~/.local/lib/python3.11/site-packages/sklearn/preprocessing/_data.py in ?(self, X, y, sample_weight)
    840         self : object
    841             Fitted scaler.
    842         """
    843         first_call = not hasattr(self, "n_samples_seen_")
--> 844         X = self._validate_data(
    845             X,
    846             accept_sparse=("csr", "csc"),
    847             dtype=FLOAT_DTYPES,

~/.local/lib/python3.11/site-packages/sklearn/base.py in ?(self, X, y, reset, validate_separately, **check_p
    573 
    574         if no_val_X and no_val_y:
    575             raise ValueError("Validation should be done on X, y or both.")
    576         elif not no_val_X and no_val_y:
--> 577             X = check_array(X, input_name="X", **check_params)
    578             out = X
    579         elif no_val_X and not no_val_y:
    580             y = _check_y(y, **check_params)

~/.local/lib/python3.11/site-packages/sklearn/utils/validation.py in ?(array, accept_sparse, accept_large_sp
    853                     )
    854                     array = array.astype(dtype, casting="unsafe", copy=False)
    855                 else:
    856                     array = np.asarray(array, order=order, dtype=dtype)
--> 857             except ComplexWarning as complex_warning:
    858                 raise ValueError(
    859                     "Complex data not supported\n{}\n".format(array)
    860                 ) from complex_warning

~/.local/lib/python3.11/site-packages/pandas/core/generic.py in ?(self, dtype)
   2069     def __array__(self, dtype: npt.DTypeLike | None = None) -> np.ndarray:
-> 2070         return np.asarray(self._values, dtype=dtype)

ValueError: could not convert string to float: 'Software Engineer'
```

Sepertinya terjadi kesalahan saat melakukan **feature scaling**. Kesalahan ini muncul karena kita mencoba melakukan scaling pada kolom target yang masih berupa teks.
Saya akan memperbaiki dengan memastikan hanya kolom numerik yang di-*scale*. Mari kita lanjutkan dengan benar.

```python
# Exclude non-numeric columns and scale only the numeric features
numeric_cols = df.columns[3:-1]  # Skipping NPM, Profesi, and the new encoded column

# Scale the numeric features
scaler = StandardScaler()
df_scaled[numeric_cols] = scaler.fit_transform(df[numeric_cols])

# Separate the features (X) and target (y)
X = df_scaled[numeric_cols]
y = df_scaled['Klasifikasi Profesi Encoded']

# Display the first few rows after scaling and encoding
df_scaled.head()
```

Result

```
        NPM   IPK   Klasifikasi Profesi  ALGORITMA DAN PEMROGRAMAN 1  \
0  50414234  3.30     Software Engineer                     0.917753
1  50414279  3.10         Data Engineer                    -0.936676
2  50414496  3.25          Data Analyst                    -0.936676
3  50414623  3.38     Software Engineer                     0.917753
4  50414925  3.72  Mobile App Developer                     0.917753

   ALGORITMA DAN PEMROGRAMAN 2  ALGORITMA DAN PEMROGRAMAN 3  \
0                     -0.70618                     1.010257
1                     -0.70618                    -0.989847
2                     -0.70618                    -0.989847
3                     -0.70618                    -0.989847
4                     -0.70618                     1.010257

   ALGORITMA DAN PENGOLAHAN PARALEL  ALGORITMA DEEP LEARNING  \
0                         -0.540102                 0.506370
1                         -0.540102                -2.801912
2                         -0.540102                 0.506370
3                         -0.540102                -1.147771
4                          1.285070                 0.506370

   ARSITEKTUR KOMPUTER  BAHASA INDONESIA  ...  \
0             0.745804          1.024985  ...
1            -1.284440         -0.706886  ...
2             0.745804         -0.706886  ...
3             0.745804         -0.706886  ...
4             0.745804         -0.706886  ...

   SKRIPSI / EVALUASI KOMPREHENSIF  STATISTIKA 1  STATISTIKA 2  STRUKTUR DATA  \
0                         0.375315     -2.310088      0.985622      -2.501851
1                         0.375315     -0.732467     -0.535495      -0.787296
2                         0.375315      0.845154     -2.056613      -0.787296
3                        -1.853829      0.845154      0.985622      -0.787296
4                         0.375315      0.845154      0.985622       0.927260

   TEKNIK KOMPILASI  TEKNOLOGI GAME  TEKNOLOGI KECERDASAN ARTIFICIAL  \
0         -1.064678       -1.106000                        -0.534522
1         -1.064678        0.499748                        -0.534522
2          0.749904        0.499748                        -0.534522
3          0.749904        0.499748                        -0.534522
4          0.749904        0.499748                        -0.534522

   TEORI BAHASA DAN OTOMATA  TERAPAN TEORI GRAF  Klasifikasi Profesi Encoded
0                      -0.5            -0.63847                           21
1                      -0.5            -0.63847                            5
2                      -0.5            -0.63847                            4
3                      -0.5            -0.63847                           21
4                      -0.5            -0.63847                           17

[5 rows x 95 columns]
```

Dataset telah berhasil di-*scale* dan *target* sudah di-*encode*. Berikutnya, kita akan melanjutkan ke langkah pembuatan **Wasserstein GAN** untuk prediksi profesi pekerjaan mahasiswa menggunakan

TensorFlow.
Langkah-langkah yang akan dilakukan untuk model ini:

1. **Membangun arsitektur GAN** dengan generator dan discriminator.
2. **Menerapkan Wasserstein loss** untuk memastikan pelatihan stabil.
3. **Training model GAN** untuk menghasilkan prediksi berdasarkan data input.

Mari kita mulai dengan membuat arsitektur GAN.

python

```python
import tensorflow as tf
from tensorflow.keras import layers

# Set the dimension of the latent space
latent_dim = 100
input_dim = X.shape[1]  # Number of features in the dataset

# Build the generator
def build_generator():
    model = tf.keras.Sequential()
    model.add(layers.Dense(128, input_dim=latent_dim, activation='relu'))
    model.add(layers.Dense(256, activation='relu'))
    model.add(layers.Dense(input_dim, activation='tanh'))
    return model

# Build the discriminator
def build_discriminator():
    model = tf.keras.Sequential()
    model.add(layers.Dense(256, input_dim=input_dim, activation='relu'))
    model.add(layers.Dense(128, activation='relu'))
    model.add(layers.Dense(1))
    return model

# Create the generator and discriminator
generator = build_generator()
discriminator = build_discriminator()

# Define the Wasserstein loss function
def wasserstein_loss(y_true, y_pred):
    return tf.keras.backend.mean(y_true * y_pred)

# Compile the discriminator
discriminator.compile(optimizer=tf.keras.optimizers.RMSprop(learning_rate=0.00005),
loss=wasserstein_loss)

# Build and compile the combined GAN model
discriminator.trainable = False
gan_input = layers.Input(shape=(latent_dim,))
generated_data = generator(gan_input)
gan_output = discriminator(generated_data)
gan_model = tf.keras.Model(gan_input, gan_output)
gan_model.compile(optimizer=tf.keras.optimizers.RMSprop(learning_rate=0.00005),
loss=wasserstein_loss)

gan_model.summary()
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[5], line 1
----> 1 import tensorflow as tf
      2 from tensorflow.keras import layers
      4 # Set the dimension of the latent space

ModuleNotFoundError: No module named 'tensorflow'
```

Sepertinya TensorFlow tidak tersedia dalam lingkungan ini, sehingga saya tidak bisa melanjutkan pembuatan dan pelatihan model **Wasserstein GAN** secara langsung.
Namun, saya dapat memberikan langkah-langkah dan kode yang dapat kamu jalankan di lingkungan lokal dengan TensorFlow terpasang:

1. Install TensorFlow:

```bash
pip install tensorflow
```

2. Langkah-Langkah Implementasi:

Berikut adalah rangkaian kode yang bisa kamu gunakan untuk membangun dan melatih **Wasserstein GAN**:

```python
import tensorflow as tf
from tensorflow.keras import layers
import numpy as np

# Define dimensions
latent_dim = 100
input_dim = X.shape[1]  # Number of features in the dataset

# Build generator
def build_generator():
    model = tf.keras.Sequential()
    model.add(layers.Dense(128, input_dim=latent_dim, activation='relu'))
    model.add(layers.Dense(256, activation='relu'))
    model.add(layers.Dense(input_dim, activation='tanh'))
    return model

# Build discriminator
def build_discriminator():
    model = tf.keras.Sequential()
    model.add(layers.Dense(256, input_dim=input_dim, activation='relu'))
    model.add(layers.Dense(128, activation='relu'))
    model.add(layers.Dense(1))
    return model

# Create models
generator = build_generator()
discriminator = build_discriminator()

# Define Wasserstein loss
def wasserstein_loss(y_true, y_pred):
    return tf.keras.backend.mean(y_true * y_pred)

# Compile discriminator
discriminator.compile(optimizer=tf.keras.optimizers.RMSprop(learning_rate=0.00005),
loss=wasserstein_loss)

# Create GAN model
discriminator.trainable = False
gan_input = layers.Input(shape=(latent_dim,))
generated_data = generator(gan_input)
gan_output = discriminator(generated_data)
gan_model = tf.keras.Model(gan_input, gan_output)
gan_model.compile(optimizer=tf.keras.optimizers.RMSprop(learning_rate=0.00005),
loss=wasserstein_loss)
```

```
# GAN training loop
def train_gan(epochs, batch_size):
    for epoch in range(epochs):
        for _ in range(5):  # Critic updates
            random_latent_vectors = np.random.normal(size=(batch_size, latent_dim))
            generated_samples = generator.predict(random_latent_vectors)
            real_samples = X.sample(batch_size).values
            combined_samples = np.concatenate([generated_samples, real_samples])
            labels = np.concatenate([np.ones((batch_size, 1)), -np.ones((batch_size, 1))])
            d_loss = discriminator.train_on_batch(combined_samples, labels)

        random_latent_vectors = np.random.normal(size=(batch_size, latent_dim))
        misleading_targets = -np.ones((batch_size, 1))
        a_loss = gan_model.train_on_batch(random_latent_vectors, misleading_targets)

        if epoch % 100 == 0:
            print(f"Epoch: {epoch}, Discriminator Loss: {d_loss}, Adversarial Loss:
{a_loss}")

# Train the GAN
train_gan(epochs=10000, batch_size=32)
```

Kamu bisa menggunakan kode di atas untuk melatih **Wasserstein GAN** di lingkungan lokal kamu. Semoga berhasil!

how do i evaluate model above?

To evaluate the **Wasserstein GAN (WGAN)** model, you can approach it in two main ways:

1. **Discriminator Loss (Wasserstein Loss):**
    - Evaluate how well the discriminator distinguishes between real and generated samples. Ideally, the discriminator's loss should stabilize near 0, meaning it can't easily distinguish real from generated data.
2. **Generator Evaluation (Visualizing Generated Data or Performance on Tasks):**
    - For GANs, especially in tasks like predicting professions or generating features, the performance of the generator is key. In your case, you'll want to assess how well the generator produces data that match the true data distribution (features corresponding to specific professions).

Here’s a step-by-step outline to evaluate your model:

1. Track Losses During Training

- The first way to evaluate the WGAN model is by monitoring the discriminator and generator losses during training.
- The discriminator loss represents how well it can distinguish between real and fake samples.
- The generator loss represents how well it can fool the discriminator.

2. Evaluate Generated Data

- Periodically, after training a certain number of epochs, generate new data using the generator and compare it with the real data.

- Since your goal is to predict professions, you can:
  - **Generate data samples** using the generator.
  - **Visualize** the generated features or check statistics (such as distribution, means) against the real data.
  - **Check for mode collapse** (where the generator always outputs the same or very similar data, regardless of the input noise).

3. Quality of Generated Data

- **Sample Data**: Generate a batch of new samples using the trained generator and inspect them to see if they resemble real data.
- **Statistical Tests**: Compute statistical measures like mean, variance, and compare these between generated and real data.
- **Train a Classifier**: Another effective method is to train a classifier (like a simple neural network or logistic regression) on the real data and test its performance on both the real and generated data. This will give you an idea of whether the generator is producing meaningful data.

Example: Generator Data Evaluation

```python
# Generate a batch of new data samples
latent_vectors = np.random.normal(size=(32, latent_dim))
generated_samples = generator.predict(latent_vectors)

# Compare with real data
real_samples = X.sample(32).values

# For example, check the mean and standard deviation of real vs generated samples
real_mean = np.mean(real_samples, axis=0)
generated_mean = np.mean(generated_samples, axis=0)

real_std = np.std(real_samples, axis=0)
generated_std = np.std(generated_samples, axis=0)

print("Mean difference between real and generated data:", np.mean(np.abs(real_mean - generated_mean)))
print("Std difference between real and generated data:", np.mean(np.abs(real_std - generated_std)))
```

4. Use Inception Score or FID

- If you're working with images, **Inception Score (IS)** or **Fréchet Inception Distance (FID)** is commonly used to measure how well the generated samples match the true distribution.
- However, for your case (structured/tabular data), you might consider training a classifier on real data and then testing it on both real and generated data to see how well it generalizes.

5. Mode Collapse Detection

- **Visual Check:** After training for several epochs, you can check for mode collapse by inspecting the diversity of the generated samples. If the generator outputs similar results regardless of input noise, it suggests mode collapse.
- **Plotting Latent Space:** Visualizing the latent space (if possible) using techniques like t-SNE could show how much variation the generator is producing.

Example: Evaluate by Classifying Generated Data

```python
# Train a simple classifier on real data
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

clf = RandomForestClassifier()
clf.fit(X_train, y_train)

# Now evaluate classifier on real test data and generated data
y_pred_real = clf.predict(X_test)
print("Performance on real data:")
print(classification_report(y_test, y_pred_real))

# Generate fake data and evaluate
generated_samples = generator.predict(np.random.normal(size=(X_test.shape[0],
latent_dim)))
y_pred_generated = clf.predict(generated_samples)
print("Performance on generated data:")
print(classification_report(y_test, y_pred_generated))
```

6. Human Review for Qualitative Data

- In some cases, it's helpful to manually inspect generated samples to see if they make sense in the context of the domain (e.g., professions).

By following these approaches, you can assess how well the WGAN model is learning and generating realistic predictions based on the dataset you have.

evaluate with beautiful diagrams

To visually evaluate the **Wasserstein GAN** model's performance, we can plot a series of diagrams that provide insights into the model's training process and the quality of the generated data. Here are some useful diagrams:

1. **Loss Curves for Generator and Discriminator**: Plot the loss values over the training epochs to monitor convergence.
2. **Comparison of Data Distribution**: Plot the distribution of features in real vs. generated data to evaluate the model's ability to replicate the real data distribution.
3. **t-SNE or PCA Plot of Latent Space**: Visualize the generator's ability to create diverse samples by reducing the dimensionality of the generated data and comparing it to real data.

Below is a general outline of the code to generate these diagrams:

1. Loss Curves Plot

- Track and plot the generator and discriminator loss values over training epochs.

2. Real vs Generated Data Distribution

- Compare the distribution of key features between real and generated data using histograms or density plots.

3. t-SNE Plot for Real and Generated Data

- Use t-SNE to reduce the dimensionality and visualize the diversity in real and generated data.

Example Code to Plot the Evaluation Diagrams

First, let's outline the evaluation step with diagrams:

```python
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
import seaborn as sns
import numpy as np

# Assuming you have stored generator and discriminator losses during training
def plot_loss(generator_loss, discriminator_loss):
    plt.figure(figsize=(10, 5))
    plt.plot(generator_loss, label='Generator Loss')
    plt.plot(discriminator_loss, label='Discriminator Loss')
    plt.title('Generator and Discriminator Loss During Training')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()

# 1. Plot loss curves
# Example lists for generator and discriminator loss, these would come from training
generator_loss = np.random.randn(1000)  # replace with actual generator loss values
discriminator_loss = np.random.randn(1000)  # replace with actual discriminator loss
values
plot_loss(generator_loss, discriminator_loss)

# 2. Real vs Generated Data Distribution
def plot_distribution(real_data, generated_data, feature_name):
    plt.figure(figsize=(10, 5))
    sns.kdeplot(real_data, label='Real', shade=True)
    sns.kdeplot(generated_data, label='Generated', shade=True)
    plt.title(f'Real vs Generated Data Distribution for {feature_name}')
    plt.xlabel('Feature Value')
    plt.ylabel('Density')
    plt.legend()
    plt.show()

# Example real and generated data, use your real dataset and generated samples
real_samples = X.sample(1000).values  # real data
latent_vectors = np.random.normal(size=(1000, latent_dim))
generated_samples = generator.predict(latent_vectors)  # generated data

# Plot for a selected feature (e.g., the first feature)
plot_distribution(real_samples[:, 0], generated_samples[:, 0], 'Feature 1')

# 3. t-SNE Visualization
def plot_tsne(real_data, generated_data):
    tsne = TSNE(n_components=2, perplexity=30, n_iter=300)
    real_tsne = tsne.fit_transform(real_data)
    generated_tsne = tsne.fit_transform(generated_data)

    plt.figure(figsize=(10, 5))
```

```
    plt.scatter(real_tsne[:, 0], real_tsne[:, 1], label='Real Data', alpha=0.6)
    plt.scatter(generated_tsne[:, 0], generated_tsne[:, 1], label='Generated Data',
alpha=0.6)
    plt.title('t-SNE Visualization of Real and Generated Data')
    plt.xlabel('Dimension 1')
    plt.ylabel('Dimension 2')
    plt.legend()
    plt.show()

# Apply t-SNE on the first 1000 real and generated samples
plot_tsne(real_samples[:1000], generated_samples[:1000])
```

Explanation of Each Diagram:

1. **Loss Curves**:
    - **Generator Loss**: A stable loss curve indicates that the generator is improving and not oscillating.
    - **Discriminator Loss**: The discriminator loss should stabilize near zero as the model learns to distinguish between real and generated data.
2. **Real vs. Generated Data Distribution**:
    - This plot compares the distribution of specific features in real vs. generated data. The closer the two distributions, the better the generator replicates the real data distribution.
3. **t-SNE Plot**:
    - t-SNE reduces the high-dimensional data into 2D space to visualize how well the generator captures the diversity of real data. A good GAN should generate data points that overlap or are close to real data points in the t-SNE plot.

These visualizations will give you a deeper understanding of how well the GAN is performing and whether further tuning or adjustments are required.
Let me know if you'd like further assistance with generating these visualizations!

how do i improve model above?

Improving a **Wasserstein GAN (WGAN)** model can be approached from several angles, including architecture tuning, optimization strategies, and training techniques. Here are some key strategies to enhance your WGAN model's performance:

1. Network Architecture Optimization

- **Increase Model Capacity**: You can add more layers or increase the number of neurons in both the generator and discriminator networks. This allows the model to capture more complex patterns in the data.
    - Increase layers:

```python
def build_generator():
    model = tf.keras.Sequential()
    model.add(layers.Dense(256, input_dim=latent_dim, activation='relu'))
```

```
        model.add(layers.Dense(512, activation='relu'))
        model.add(layers.Dense(1024, activation='relu'))
        model.add(layers.Dense(input_dim, activation='tanh'))
        return model

    def build_discriminator():
        model = tf.keras.Sequential()
        model.add(layers.Dense(1024, input_dim=input_dim, activation='relu'))
        model.add(layers.Dense(512, activation='relu'))
        model.add(layers.Dense(256, activation='relu'))
        model.add(layers.Dense(1))
        return model
```

- **Batch Normalization**: Add batch normalization layers to the generator to stabilize training and improve convergence.

```python
def build_generator():
    model = tf.keras.Sequential()
    model.add(layers.Dense(256, input_dim=latent_dim, activation='relu'))
    model.add(layers.BatchNormalization())
    model.add(layers.Dense(512, activation='relu'))
    model.add(layers.BatchNormalization())
    model.add(layers.Dense(1024, activation='relu'))
    model.add(layers.BatchNormalization())
    model.add(layers.Dense(input_dim, activation='tanh'))
    return model
```

- **LeakyReLU in Discriminator**: Use LeakyReLU instead of ReLU in the discriminator to help with gradient flow during training.

```python
def build_discriminator():
    model = tf.keras.Sequential()
    model.add(layers.Dense(1024, input_dim=input_dim))
    model.add(layers.LeakyReLU(0.2))
    model.add(layers.Dense(512))
    model.add(layers.LeakyReLU(0.2))
    model.add(layers.Dense(256))
    model.add(layers.LeakyReLU(0.2))
    model.add(layers.Dense(1))
    return model
```

2. Optimization Strategies

- **Gradient Penalty**: Implement gradient penalty instead of weight clipping to better enforce the Lipschitz constraint and stabilize training. This improves the stability of WGAN and avoids the issues caused by weight clipping.

```python
def gradient_penalty_loss(real_data, generated_data, discriminator, batch_size):
    # Calculate the gradient penalty
    alpha = tf.random.normal([batch_size, 1], 0.0, 1.0)
    interpolated_data = alpha * real_data + (1 - alpha) * generated_data

    with tf.GradientTape() as tape:
```

```
        tape.watch(interpolated_data)
        prediction = discriminator(interpolated_data)
    gradients = tape.gradient(prediction, [interpolated_data])[0]
    gradient_norm = tf.sqrt(tf.reduce_sum(tf.square(gradients), axis=1))
    penalty = tf.reduce_mean((gradient_norm - 1.0) ** 2)
    return penalty
```

- **Learning Rate Scheduling**: Use a learning rate scheduler to decrease the learning rate during training. This can help the model converge to a better solution as training progresses.

```python
learning_rate_schedule = tf.keras.optimizers.schedules.ExponentialDecay(
    initial_learning_rate=0.0002, decay_steps=10000, decay_rate=0.96)
optimizer = tf.keras.optimizers.RMSprop(learning_rate=learning_rate_schedule)
```

- **Adam Optimizer**: You can experiment with using the Adam optimizer (with parameters $\beta_1 = 0.5$ and $\beta_2 = 0.9$) instead of RMSProp for both the generator and discriminator, which is widely used in GAN training.

```python
generator_optimizer = tf.keras.optimizers.Adam(learning_rate=0.0001, beta_1=0.5)
discriminator_optimizer = tf.keras.optimizers.Adam(learning_rate=0.0001, beta_1=0.5)
```

3. Training Techniques

- **Increase Critic Updates**: The discriminator (critic) in WGAN is updated multiple times for each generator update. You can experiment with increasing the number of critic updates (e.g., 5 updates for every generator update).

```python
n_critic_updates = 5
```

- **Add Noise to Labels**: Introduce noise to the real and generated labels during training to prevent the discriminator from becoming too strong and overpowering the generator. This technique, known as **label smoothing**, can make training more stable.

```python
real_labels = np.ones((batch_size, 1)) * 0.9  # real labels are smoothed
fake_labels = np.zeros((batch_size, 1))  # fake labels remain zero
```

- **Training with Noisy Data**: Add small noise to the input data to make the generator and discriminator more robust.

```python
real_samples = X.sample(batch_size).values + np.random.normal(0, 0.05, size=X.shape)
# Add noise
```

- **Model Checkpoints**: Use model checkpoints to save the model at its best performing state during training.

```python
checkpoint = tf.keras.callbacks.ModelCheckpoint('best_model.h5', monitor='loss',
save_best_only=True)
```

4. Monitoring and Evaluation

- **Evaluate Diversity and Mode Collapse**: Use visualization techniques like t-SNE, PCA, and histogram comparison to monitor whether the model is generating diverse data and avoiding mode collapse.
- **Use Metrics**:
  - **Fréchet Inception Distance (FID)**: Although typically used for images, FID can be adapted for non-image data to measure how similar generated data is to real data.
  - **Inception Score**: Similar to FID, you can adapt this score to non-image data to measure the quality and diversity of the generated data.

Example: Improved Generator and Discriminator with Adam and Gradient Penalty

Here's an updated example of the generator and discriminator with these improvements:

```python
# Improved Generator
def build_generator():
    model = tf.keras.Sequential()
    model.add(layers.Dense(256, input_dim=latent_dim, activation='relu'))
    model.add(layers.BatchNormalization())
    model.add(layers.Dense(512, activation='relu'))
    model.add(layers.BatchNormalization())
    model.add(layers.Dense(1024, activation='relu'))
    model.add(layers.BatchNormalization())
    model.add(layers.Dense(input_dim, activation='tanh'))
    return model

# Improved Discriminator with LeakyReLU
def build_discriminator():
    model = tf.keras.Sequential()
    model.add(layers.Dense(1024, input_dim=input_dim))
    model.add(layers.LeakyReLU(0.2))
    model.add(layers.Dense(512))
    model.add(layers.LeakyReLU(0.2))
    model.add(layers.Dense(256))
    model.add(layers.LeakyReLU(0.2))
    model.add(layers.Dense(1))
    return model

# Wasserstein Loss and Gradient Penalty
def wasserstein_loss(y_true, y_pred):
    return tf.keras.backend.mean(y_true * y_pred)
```

```python
def gradient_penalty(real_data, generated_data, discriminator):
    epsilon = tf.random.uniform([real_data.shape[0], 1], 0.0, 1.0)
    interpolated = epsilon * real_data + (1 - epsilon) * generated_data
    with tf.GradientTape() as tape:
        tape.watch(interpolated)
        prediction = discriminator(interpolated)
    gradients = tape.gradient(prediction, interpolated)
    gradient_l2_norm = tf.sqrt(tf.reduce_sum(tf.square(gradients), axis=[1]))
    gradient_penalty = tf.reduce_mean((gradient_l2_norm - 1.0) ** 2)
    return gradient_penalty
```

5. Hyperparameter Tuning

- **Batch Size**: Experiment with different batch sizes (e.g., 32, 64, 128).
- **Latent Dim**: Adjust the latent dimension (e.g., 50, 100, 200) to see its effect on the quality of generated samples.
- **Learning Rate**: Tune the learning rate for both the generator and discriminator (0.00005 to 0.0002 typically works well).

By implementing these improvements and monitoring the training process, you can greatly enhance the performance of your WGAN model.

Can you explain the gradient penalty?

Sure! Let's dive into the **gradient penalty** in the context of **Wasserstein GAN (WGAN)** and why it's important.

Background: Why Do We Need Gradient Penalty?

The original WGAN introduced the **Wasserstein distance** (or Earth Mover's Distance) as a way to measure how similar two distributions are (real vs. generated). In the WGAN paper, the authors showed that minimizing this distance leads to better training stability compared to traditional GANs that use Jensen-Shannon divergence.
However, to ensure that the Wasserstein distance is valid, the discriminator (called a "critic" in WGAN) must be constrained to be **Lipschitz continuous**. In simpler terms, this means that the critic must have a bounded gradient, ensuring it behaves "smoothly."
In the original WGAN, this constraint was enforced by **clipping the weights** of the critic (forcing all weights to stay within a certain range, e.g., [-0.01, 0.01]). While weight clipping worked to enforce Lipschitz continuity, it caused several problems:

- It led to **vanishing or exploding gradients**.
- The critic often became too simple (capacity-limited), preventing it from learning well.

Gradient Penalty: A Better Solution

The **gradient penalty** (introduced in **WGAN-GP**) is a more robust and effective way of enforcing Lipschitz continuity on the critic. Instead of clipping the weights, WGAN-GP adds a **penalty** to the loss function, ensuring that the **gradient norm** of the critic with respect to its input is close to 1.

In other words, it penalizes the critic if the magnitude (norm) of the gradient is too far from 1, which helps in maintaining the 1-Lipschitz constraint.

How Gradient Penalty Works

Let's break it down step by step:

1. **Interpolation**: For each real sample x_{real} and a generated sample $x_{\text{generated}}$, we create a new "interpolated" sample. This interpolated sample lies between the real and fake samples and is defined as:

$$\hat{x} = \epsilon \cdot x_{\text{real}} + (1 - \epsilon) \cdot x_{\text{generated}}$$

   where ϵ is a random number between 0 and 1.
2. **Compute Gradient**: We pass this interpolated sample $\hat{x}$ through the critic and compute the gradient of the critic's output with respect to $\hat{x}$. This measures how much the critic's output changes for a small change in the input:

$$\nabla_{\hat{x}} D(\hat{x})$$

   where $D(\cdot)$ is the critic.
3. **Calculate Gradient Norm**: We calculate the **norm** (or magnitude) of this gradient, which tells us how steep the critic's output is around $\hat{x}$:

$$||\nabla_{\hat{x}} D(\hat{x})||_2$$

   This should ideally be close to 1 for the Lipschitz constraint to hold.
4. **Penalty Term**: We add a penalty term to the loss function based on how much the gradient norm deviates from 1:

$$\lambda \cdot \mathbb{E}\left((||\nabla_{\hat{x}} D(\hat{x})||_2 - 1)^2\right)$$

   where λ is a constant (typically set to 10) that controls the strength of the gradient penalty.
5. **Final Loss**: The critic's loss function now becomes:

$$L_{\text{critic}} = \mathbb{E}[D(x_{\text{generated}})] - \mathbb{E}[D(x_{\text{real}})] + \lambda \cdot \text{Gradient Penalty}$$

   This ensures that while the critic is trained to distinguish real from generated samples, it also maintains the Lipschitz constraint via the gradient penalty.

Why Does This Help?

- **No Weight Clipping**: Unlike the original WGAN that used weight clipping (which can degrade the model's performance), gradient penalty ensures that the critic maintains a smooth behavior without having to restrict its weights.
- **Better Gradient Flow**: By enforcing the Lipschitz constraint via the gradient norm, you prevent the gradients from vanishing or exploding, which results in more stable training.
- **Stabilized GAN Training**: Gradient penalty provides a smoother optimization landscape for the critic and the generator, leading to better convergence and more realistic outputs.

Code Example: Implementing Gradient Penalty in WGAN-GP

Here's how you can implement gradient penalty in the critic's loss function:

```python
import tensorflow as tf

# Function to compute the gradient penalty
def gradient_penalty(real_samples, generated_samples, batch_size, critic):
    # Interpolation between real and generated samples
    alpha = tf.random.uniform([batch_size, 1], 0.0, 1.0)
    interpolated = alpha * real_samples + (1 - alpha) * generated_samples

    # Calculate gradients with respect to the interpolated samples
    with tf.GradientTape() as tape:
        tape.watch(interpolated)
        prediction = critic(interpolated)

    gradients = tape.gradient(prediction, interpolated)

    # Calculate the L2 norm of the gradients
    gradient_l2_norm = tf.sqrt(tf.reduce_sum(tf.square(gradients), axis=[1]))

    # Compute the gradient penalty (the deviation from the norm 1)
    gradient_penalty = tf.reduce_mean((gradient_l2_norm - 1.0) ** 2)

    return gradient_penalty
```

In this example:

1. We randomly interpolate between real and generated samples using a mix factor α.
2. We compute the gradients of the critic's predictions with respect to these interpolated samples.
3. We calculate the L2 norm of these gradients and penalize any deviations from 1.

Conclusion

The gradient penalty method significantly improves the stability of GAN training by enforcing the Lipschitz constraint in a more natural and effective way than weight clipping. It prevents problems like vanishing/exploding gradients and allows the critic to learn more complex decision boundaries, leading to better performance for both the critic and the generator.
By using gradient penalty, your WGAN model can generate more realistic data with improved stability during training.

implement gpu acceleration in model trainin

To implement GPU acceleration in your model training, we need to leverage **TensorFlow's automatic GPU support**. TensorFlow will automatically use a GPU if it's available, provided that you have the necessary **GPU drivers** and **CUDA/cuDNN** installed. Below are steps and tips to ensure you're using the GPU for your **WGAN-GP** model training.

1. Ensure GPU Support is Installed

Before starting, make sure the following software components are installed:

- **NVIDIA CUDA Toolkit**: A parallel computing platform and programming model for NVIDIA GPUs.
- **cuDNN**: NVIDIA's deep neural network library.
- **TensorFlow with GPU support**: You can install the GPU-enabled version of TensorFlow by running:

```bash
pip install tensorflow-gpu
```

You can check whether TensorFlow recognizes your GPU using the following code:

```python
import tensorflow as tf

print("Num GPUs Available: ", len(tf.config.list_physical_devices('GPU')))
```

This should return a number greater than zero if TensorFlow is able to detect the GPU.

2. Using GPU for Model Training

If the environment is correctly set up and the GPU is available, TensorFlow will automatically use it for both model training and inference. However, you can explicitly place operations on the GPU using `` `tf.device('/GPU:0')` ``.
Here's how you can modify the training loop for WGAN-GP to utilize the GPU:

Full Example of WGAN-GP Training with GPU

```python
import tensorflow as tf
from tensorflow.keras import layers
import numpy as np

# Set up your generator and discriminator models
def build_generator():
    model = tf.keras.Sequential()
    model.add(layers.Dense(256, input_dim=latent_dim, activation='relu'))
    model.add(layers.BatchNormalization())
    model.add(layers.Dense(512, activation='relu'))
    model.add(layers.BatchNormalization())
    model.add(layers.Dense(1024, activation='relu'))
    model.add(layers.BatchNormalization())
    model.add(layers.Dense(input_dim, activation='tanh'))
    return model

def build_discriminator():
    model = tf.keras.Sequential()
    model.add(layers.Dense(1024, input_dim=input_dim))
    model.add(layers.LeakyReLU(0.2))
    model.add(layers.Dense(512))
    model.add(layers.LeakyReLU(0.2))
    model.add(layers.Dense(256))
    model.add(layers.LeakyReLU(0.2))
    model.add(layers.Dense(1))
```

```
    return model

# Wasserstein Loss
def wasserstein_loss(y_true, y_pred):
    return tf.keras.backend.mean(y_true * y_pred)

# Gradient Penalty
def gradient_penalty(real_samples, generated_samples, batch_size, critic):
    alpha = tf.random.uniform([batch_size, 1], 0.0, 1.0)
    interpolated = alpha * real_samples + (1 - alpha) * generated_samples

    with tf.GradientTape() as tape:
        tape.watch(interpolated)
        prediction = critic(interpolated)

    gradients = tape.gradient(prediction, interpolated)
    gradient_l2_norm = tf.sqrt(tf.reduce_sum(tf.square(gradients), axis=[1]))
    gradient_penalty = tf.reduce_mean((gradient_l2_norm - 1.0) ** 2)

    return gradient_penalty

# Model Training Loop (with GPU acceleration)
def train_wgan_gp(generator, discriminator, epochs, batch_size, latent_dim, X_train,
lambda_gp=10):
    # Create optimizers
    generator_optimizer = tf.keras.optimizers.Adam(learning_rate=0.0001, beta_1=0.5)
    discriminator_optimizer = tf.keras.optimizers.Adam(learning_rate=0.0001, beta_1=0.5)

    # Begin training loop
    for epoch in range(epochs):
        for _ in range(5):  # Critic (Discriminator) updates more frequently than
Generator
            real_samples = X_train.sample(batch_size).values
            latent_vectors = np.random.normal(size=(batch_size, latent_dim))
            generated_samples = generator.predict(latent_vectors)

            with tf.device('/GPU:0'):  # Ensure operations run on GPU
                # Train the critic
                with tf.GradientTape() as tape:
                    real_pred = discriminator(real_samples)
                    generated_pred = discriminator(generated_samples)

                    gp = gradient_penalty(real_samples, generated_samples, batch_size,
discriminator)
                    critic_loss = tf.reduce_mean(generated_pred) -
tf.reduce_mean(real_pred) + lambda_gp * gp

                grads = tape.gradient(critic_loss, discriminator.trainable_weights)
                discriminator_optimizer.apply_gradients(zip(grads,
discriminator.trainable_weights))

        # Train the generator
        latent_vectors = np.random.normal(size=(batch_size, latent_dim))
        with tf.device('/GPU:0'):
            with tf.GradientTape() as tape:
                generated_samples = generator(latent_vectors)
                generated_pred = discriminator(generated_samples)
                generator_loss = -tf.reduce_mean(generated_pred)

            grads = tape.gradient(generator_loss, generator.trainable_weights)
            generator_optimizer.apply_gradients(zip(grads, generator.trainable_weights))

        # Logging the progress
        if epoch % 100 == 0:
            print(f"Epoch {epoch}/{epochs}, Critic Loss: {critic_loss.numpy()}, Generator
Loss: {generator_loss.numpy()}")

# Set parameters
latent_dim = 100  # Dimension of the latent space
input_dim = X_train.shape[1]  # Number of input features
epochs = 10000
batch_size = 64

# Build generator and discriminator
generator = build_generator()
```

```
discriminator = build_discriminator()

# Start training WGAN with GPU acceleration
train_wgan_gp(generator, discriminator, epochs, batch_size, latent_dim, X_train)
```

Key Notes for GPU Acceleration:

1. **Using `tf.device('/GPU:0')`:** This ensures that all model operations (training loops, loss calculations, gradient calculations, etc.) are performed on the GPU. The argument `'GPU:0'` refers to the first GPU available. If you have multiple GPUs, you can specify them with `'GPU:1'`, `'GPU:2'`, etc.
2. **Model and Optimizer on GPU:** Ensure that the **generator** and **discriminator** models, along with the **optimizers**, are all running on the GPU. This is automatically handled by TensorFlow when you place operations within `tf.device('/GPU:0')`.
3. **Data Transfer**: Make sure the data is correctly transferred between the CPU and GPU. TensorFlow automatically handles this for you when using `tf.device()`.
4. **Mixed Precision Training**: If you want to further speed up your model, you can use **mixed precision training**, which allows models to use both 16-bit and 32-bit floating point types during training. This can improve performance, especially on NVIDIA GPUs with Tensor Cores (e.g., Volta, Turing, or Ampere architectures). You can enable this using:

```python
from tensorflow.keras.mixed_precision import experimental as mixed_precision
policy = mixed_precision.Policy('mixed_float16')
mixed_precision.set_policy(policy)
```

3. Check GPU Utilization

You can use the following command to monitor GPU utilization during training:

```bash
nvidia-smi
```

This command will show the GPU's memory and processing usage, and you can confirm whether the GPU is being used effectively for training.

Conclusion

By placing the operations on the GPU with `tf.device('/GPU:0')` and ensuring that TensorFlow's GPU support is properly installed, you can significantly speed up your **WGAN-GP** training process. Additionally, you can explore techniques like **mixed precision** for even better performance on modern GPUs.

how do i evaluate with beautiful visual after tuning

After tuning your **Wasserstein GAN with Gradient Penalty (WGAN-GP)** model, evaluating the performance through **visualizations** is essential. Here are some effective techniques to visualize and evaluate the model after tuning:

Key Visualizations:

1. **Loss Curves**: Plot both generator and discriminator (critic) losses over the epochs to observe training stability and convergence.
2. **Feature Distributions**: Compare real vs generated feature distributions using histograms or KDE (Kernel Density Estimation) plots.
3. **t-SNE / PCA Plot**: Visualize the latent space to check the diversity and structure of generated samples compared to real data.
4. **Generated Samples Over Time**: Track and visualize how generated samples improve over training.
5. **Mode Collapse Detection**: Visualize the generated samples across epochs to detect mode collapse.

Let's implement these visualizations in Python, assuming the tuning has been done and the models are ready.

1. Plot Loss Curves

The **generator** and **discriminator** (critic) loss curves can reveal how the model is learning and whether training has stabilized.

```python
import matplotlib.pyplot as plt

def plot_loss(generator_loss, critic_loss):
    plt.figure(figsize=(10, 6))
    plt.plot(generator_loss, label='Generator Loss')
    plt.plot(critic_loss, label='Critic Loss')
    plt.title('Generator and Critic Loss During Training')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)
    plt.show()

# Example lists of losses over epochs
generator_loss = [0.5, 0.3, 0.2, 0.15, 0.12]  # replace with actual loss values
critic_loss = [-0.7, -0.5, -0.35, -0.3, -0.25]  # replace with actual loss values

plot_loss(generator_loss, critic_loss)
```

2. Compare Real vs Generated Data Distribution

Using **histograms** or **KDE plots** to compare real vs. generated data for important features will show how well the model captures the distribution of real data.

```python
import seaborn as sns
import numpy as np

# Example function to plot real vs generated distributions
def plot_feature_distribution(real_data, generated_data, feature_name):
    plt.figure(figsize=(10, 6))
    sns.kdeplot(real_data, label='Real Data', shade=True)
    sns.kdeplot(generated_data, label='Generated Data', shade=True)
    plt.title(f'Distribution of {feature_name}: Real vs Generated')
    plt.xlabel('Feature Value')
    plt.ylabel('Density')
    plt.legend()
    plt.show()

# Replace with actual feature columns from real and generated samples
real_samples = X_train.sample(1000).values[:, 0]  # First feature of real samples
latent_vectors = np.random.normal(size=(1000, latent_dim))
generated_samples = generator.predict(latent_vectors)
generated_samples_feature = generated_samples[:, 0]  # First feature of generated samples

plot_feature_distribution(real_samples, generated_samples_feature, "Feature 1")
```

3. t-SNE or PCA Visualization of Latent Space

t-SNE (t-distributed Stochastic Neighbor Embedding) or **PCA** (Principal Component Analysis) is useful for reducing high-dimensional data to 2D or 3D for visualization. This helps us compare real and generated samples to assess diversity and overlap.

```python
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt

def plot_tsne(real_data, generated_data, title='t-SNE Visualization'):
    tsne = TSNE(n_components=2, perplexity=30, n_iter=300)
    real_tsne = tsne.fit_transform(real_data)
    generated_tsne = tsne.fit_transform(generated_data)

    plt.figure(figsize=(10, 6))
    plt.scatter(real_tsne[:, 0], real_tsne[:, 1], label='Real Data', alpha=0.5)
    plt.scatter(generated_tsne[:, 0], generated_tsne[:, 1], label='Generated Data',
alpha=0.5)
    plt.title(title)
    plt.legend()
    plt.show()

# Run t-SNE on real and generated samples
real_samples_tsne = X_train.sample(1000).values  # Sample real data
generated_samples_tsne = generator.predict(np.random.normal(size=(1000, latent_dim)))

plot_tsne(real_samples_tsne, generated_samples_tsne, title="t-SNE: Real vs Generated
Data")
```

4. Tracking Generated Samples Over Time

You can visualize how well the generated samples improve as training progresses. This is useful to track the quality of generated data over different epochs.

```python
def plot_generated_samples(generator, latent_dim, n_samples=1000, title="Generated
Samples"):
    latent_vectors = np.random.normal(size=(n_samples, latent_dim))
    generated_samples = generator.predict(latent_vectors)

    # For visualization, we could use the first 2 dimensions if possible
    plt.figure(figsize=(10, 6))
    plt.scatter(generated_samples[:, 0], generated_samples[:, 1], alpha=0.6, color='blue')
    plt.title(title)
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.grid(True)
    plt.show()

# Call this function after training for several epochs
plot_generated_samples(generator, latent_dim, title="Generated Samples After Tuning")
```

5. Mode Collapse Detection

Mode collapse happens when the generator produces very similar samples for different input noise vectors. You can visualize whether the generator is outputting diverse data by plotting generated samples at different points in training.

```python
def plot_generated_samples_over_epochs(generator, latent_dim, n_epochs, n_samples=1000):
    for epoch in range(0, n_epochs, 100):  # Plot every 100 epochs
        latent_vectors = np.random.normal(size=(n_samples, latent_dim))
        generated_samples = generator.predict(latent_vectors)

        plt.figure(figsize=(10, 6))
        plt.scatter(generated_samples[:, 0], generated_samples[:, 1], alpha=0.6,
label=f'Epoch {epoch}')
        plt.title(f'Generated Samples at Epoch {epoch}')
        plt.xlabel('Feature 1')
        plt.ylabel('Feature 2')
        plt.legend()
        plt.show()

# Track the diversity of generated samples over epochs
plot_generated_samples_over_epochs(generator, latent_dim, n_epochs=1000)
```

Evaluation Summary

- **Loss Curves** help you understand how well the generator and discriminator are learning and whether training has stabilized.
- **Real vs. Generated Distributions** allow you to compare how well the model captures the true data distribution for key features.
- **t-SNE or PCA Visualizations** help you evaluate the diversity of generated data and its similarity to real data.
- **Generated Samples Over Time** let you track the improvement in the generator's output quality as training progresses.
- **Mode Collapse Detection** helps identify if the generator is producing diverse samples or collapsing to similar outputs.

By combining these visualizations, you can assess the performance of your tuned **WGAN-GP** model and identify areas where further improvements may be needed.