

# Implementasi Wasserstein GAN



## total merge dataset cleaned ready

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
	NPM	IPK	Klasifikasi Profesi	ALGORITMA DA

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



python

```
import pandas as pd

# Load the uploaded dataset to inspect it
file_path = '/mnt/data/total_merge_dataset_cleaned_ready.csv'
dataset = pd.read_csv(file_path)

# Display first few rows of the dataset to understand its structure
dataset.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 memiliki beberapa fitur yang terkait dengan mata kuliah dan IPK, serta kolom target yang tampaknya adalah "**Profesi**" (berupa klasifikasi pekerjaan mahasiswa seperti "Software Engineer", "Data Engineer", dll). Selanjutnya, kita akan melakukan **Exploratory Data Analysis (EDA)** dan menyiapkan data untuk model **Wasserstein GAN (WGAN)**.

Berikut langkah-langkah EDA dan model training yang akan kita lakukan:

### 1. EDA:

- Memeriksa distribusi kolom target (Profesi).
- Melihat korelasi antara nilai mata kuliah dengan profesi.
- Memeriksa missing values atau outliers.

### 2. Preprocessing:

- Normalisasi data.
- Encoding variabel kategorikal (Profesi).

### 3. WGAN Implementation:

- Membangun generator dan discriminator.
- Training menggunakan CUDA (jika tersedia).

Mari kita mulai dengan EDA terlebih dahulu.

```
python
```


```

import matplotlib.pyplot as plt
import seaborn as sns

# Check the distribution of the target column (Profesi)
plt.figure(figsize=(10,6))
sns.countplot(y=dataset['Klasifikasi Profesi'])
plt.title('Distribusi Klasifikasi Profesi Mahasiswa')
plt.xlabel('Count')
```

```
plt.ylabel('Profesi')  
plt.show()
```

### Distribusi Klasifikasi Profesi Mahasiswa


 Output image

Distribusi klasifikasi profesi mahasiswa terlihat cukup beragam, dengan beberapa profesi memiliki lebih banyak data daripada yang lain. Langkah selanjutnya adalah mengecek korelasi antara nilai mata kuliah dan target (profesi) untuk memahami hubungan antar fitur. Mari kita lanjutkan dengan melihat korelasi antara nilai-nilai mata kuliah dan IPK.

python

```
# Remove non-numeric columns for correlation analysis  
numeric_columns = dataset.drop(columns=['NPM', 'Klasifikasi Profesi'])  
  
# Compute correlation matrix  
correlation_matrix = numeric_columns.corr()  
  
# Plot the correlation heatmap  
plt.figure(figsize=(12,10))  
sns.heatmap(correlation_matrix, cmap='coolwarm', annot=False, fmt=".2f")  
plt.title('Correlation Matrix of Student Grades and IPK')  
plt.show()
```

### Correlation Matrix of Student Grades and IPK

 Output image

Heatmap di atas menunjukkan korelasi antar fitur numeric, seperti nilai mata kuliah dan IPK. Terlihat ada beberapa hubungan yang cukup kuat antara nilai-nilai mata kuliah, yang bisa menjadi indikator bahwa beberapa mata kuliah memiliki keterkaitan dalam mempengaruhi klasifikasi profesi. Selanjutnya, kita akan melakukan preprocessing, yaitu encoding kolom target (Profesi) dan normalisasi fitur numeric, untuk mempersiapkan data ini untuk model Wasserstein GAN (WGAN).

python

```
from sklearn.preprocessing import LabelEncoder, MinMaxScaler

# Encode the target column 'Klasifikasi Profesi'
le = LabelEncoder()
dataset['Profesi_encoded'] = le.fit_transform(dataset['Klasifikasi Profesi'])

# Normalize the numeric columns
scaler = MinMaxScaler()
numeric_columns_scaled = scaler.fit_transform(numeric_columns)

# Add the encoded and normalized data to the dataset
processed_dataset = pd.DataFrame(numeric_columns_scaled, columns=numeric_columns.columns)
processed_dataset['Profesi_encoded'] = dataset['Profesi_encoded']

# Show the processed dataset
processed_dataset.head()
```

Result

	IPK	ALGORITMA DAN PEMROGRAMAN 1	ALGORITMA DAN PEMROGRAMAN 2	\
0	0.413793	1.0	0.5	
1	0.241379	0.5	0.5	
2	0.370690	0.5	0.5	
3	0.482759	1.0	0.5	
4	0.775862	1.0	0.5	

	ALGORITMA DAN PEMROGRAMAN 3	ALGORITMA DAN PENGOLAHAN PARALEL	\
0	1.0	0.5	
1	0.0	0.5	
2	0.0	0.5	
3	0.0	0.5	
4	1.0	1.0	

	ALGORITMA DEEP LEARNING	ARSITEKTUR KOMPUTER	BAHASA INDONESIA	\
0	1.0	1.0	1.0	
1	0.0	0.5	0.5	
2	1.0	1.0	0.5	
3	0.5	1.0	0.5	
4	1.0	1.0	0.5	

	BAHASA INGGRIS 1	BAHASA INGGRIS 2	...	SKRIPSI / EVALUASI KOMPREHENSIF	\
0	0.5	0.666667	...	1.00	
1	0.0	0.666667	...	1.00	
2	0.5	0.666667	...	1.00	
3	0.5	1.000000	...	0.75	
4	0.5	0.666667	...	1.00	

	STATISTIKA 1	STATISTIKA 2	STRUKTUR DATA	TEKNIK KOMPILASI	\
0	0.333333	1.000000	0.0	0.5	
1	0.666667	0.666667	0.5	0.5	
2	1.000000	0.333333	0.5	1.0	
3	1.000000	1.000000	0.5	1.0	
4	1.000000	1.000000	1.0	1.0	

	TEKNOLOGI GAME	TEKNOLOGI KECERDASAN ARTIFICIAL	TEORI BAHASA DAN OTOMATA	\
0	0.666667	0.0	0.5	
1	1.000000	0.0	0.5	

```

2      1.000000      0.0      0.5
3      1.000000      0.0      0.5
4      1.000000      0.0      0.5

  TERAPAN TEORI GRAF  Profesi_encoded
0              0.0          21
1              0.0           5
2              0.0           4
3              0.0          21
4              0.0          17

[5 rows x 93 columns]

```

Data telah selesai diproses dengan encoding target kolom (Profesi) dan normalisasi fitur numerik. Sekarang, kita dapat melanjutkan ke tahap pembangunan model **Wasserstein GAN (WGAN)**. Untuk WGAN, kita akan membutuhkan dua komponen utama:

1. **Generator**: Membuat data sintetis berdasarkan data asli.
2. **Discriminator (Critic)**: Membedakan antara data asli dan data sintetis.

Selanjutnya, kita akan mulai membangun model WGAN menggunakan PyTorch dan CUDA.

python

```

import torch
import torch.nn as nn
import torch.optim as optim

# Check for CUDA
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Define the Generator model
class Generator(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(Generator, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(input_size, hidden_size),
            nn.ReLU(True),
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(True),
            nn.Linear(hidden_size, output_size),
            nn.Tanh()
        )

    def forward(self, x):
        return self.net(x)

# Define the Critic (Discriminator) model
class Critic(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(Critic, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(input_size, hidden_size),
            nn.ReLU(True),
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(True),
            nn.Linear(hidden_size, 1)
        )

    def forward(self, x):
        return self.net(x)

# Define hyperparameters
input_size = 100 # Size of the latent vector (noise)
hidden_size = 128
output_size = processed_dataset.shape[1] - 1 # Minus one for the target variable

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

```

```
# Define optimizers for both Generator and Critic
lr = 0.0001
optimizer_G = optim.RMSprop(generator.parameters(), lr=lr)
optimizer_C = optim.RMSprop(critic.parameters(), lr=lr)

# Set up loss for Wasserstein GAN (no specific loss function, as WGAN uses weight clipping)
# Next steps will be weight clipping, training loop, etc.
```

```
-----
ImportError                                Traceback (most recent call last)
Cell In[5], line 1
----> 1 import torch
      2 import torch.nn as nn
      3 import torch.optim as optim

File ~/local/lib/python3.11/site-packages/torch/__init__.py:229
    227     if USE_GLOBAL_DEPS:
    228         _load_global_deps()
--> 229     from torch._C import * # noqa: F403
    231 # Appease the type checker; ordinarily this binding is inserted by the
    232 # torch._C module initialization code in C
    233 if TYPE_CHECKING:
```

ImportError: libcusparse.so.11: failed to map segment from shared object

Sepertinya ada masalah dengan library CUDA saat mencoba mengimpor PyTorch di lingkungan ini. Untuk melanjutkan, kamu dapat mencoba implementasi ini di lingkungan lokal dengan memastikan bahwa PyTorch telah terinstal dengan benar dan mendukung CUDA di sistem tersebut. Jika kamu membutuhkan bantuan lebih lanjut untuk setting lingkungan lokal atau melanjutkan implementasi GAN, aku bisa memberikan panduan langkah-langkah detailnya.



Can you help me install PyTorch?



Certainly! Here's a guide to help you install PyTorch, especially with CUDA support for your local machine.

## 1. Check Your System

Ensure you know whether your machine has an NVIDIA GPU and which version of CUDA is installed (if any). If you're not sure whether you have CUDA installed, you can check by typing:

```
bash
```

```
nvcc --version
```

This will show the installed CUDA version (if any). If CUDA isn't installed, you can download it from the [official NVIDIA CUDA toolkit website](#).

## 2. Install PyTorch with CUDA Support

PyTorch provides a user-friendly installation guide. You can select your preferences using the PyTorch website: [PyTorch Get Started](#).

Here are the general steps:

### For Linux or macOS:

1. Open your terminal.
2. Use the following command to install PyTorch (change `11.8` to the version of CUDA that is supported by your machine):

- **With CUDA 11.8:**

```
bash

pip install torch torchvision torchaudio --index-url
https://download.pytorch.org/whl/cu118
```

- **With CUDA 11.7:**

```
bash

pip install torch torchvision torchaudio --index-url
https://download.pytorch.org/whl/cu117
```

- **Without GPU (CPU only):**

```
bash

pip install torch torchvision torchaudio
```

### For Windows:

1. Open a command prompt or PowerShell.
2. Use the command that fits your CUDA version (if using a GPU):

- **With CUDA 11.8:**

```
bash

pip install torch torchvision torchaudio --index-url
https://download.pytorch.org/whl/cu118
```

- **With CUDA 11.7:**

```
bash

pip install torch torchvision torchaudio --index-url
https://download.pytorch.org/whl/cu117
```

- **Without GPU (CPU only):**

```
bash
```

```
pip install torch torchvision torchaudio
```

### 3. Verify Installation

To verify that PyTorch and CUDA are installed correctly, open Python or run the following commands in a Jupyter Notebook or Python script:

```
python
```

```
import torch
print(torch.__version__) # Check PyTorch version
print(torch.cuda.is_available()) # Check if CUDA is available
```

This should return `True` if CUDA is properly installed and recognized by PyTorch.

### 4. Optional: Install with Conda

If you're using Anaconda or Miniconda, you can also install PyTorch using:

```
bash
```

```
conda install pytorch torchvision torchaudio pytorch-cuda=11.8 -c pytorch -c nvidia
```

This will install the CUDA-enabled version of PyTorch with the specified CUDA version. Let me know if you encounter any issues, and I can help troubleshoot further!



evaluasi model diatas dengan diagram yang bagus



To evaluate the Wasserstein GAN (WGAN) model, after training, we usually focus on the **loss curves** of both the generator and the critic (discriminator). These curves help us understand how the model is learning over time. Additionally, we can visualize the **generated data** to see if the generator is producing realistic data points.

Here are the steps to evaluate the model and plot diagrams:

#### 1. Plot Loss Curves for Generator and Critic

We track how the loss of both the generator and the critic evolves during training. In a successful GAN training, the generator loss should decrease while the critic loss stabilizes.

#### 2. Visualize Generated Data



Once the model is trained, we can visualize the data generated by the generator and compare it with the real data to see if it has learned the distribution well.

## Let's outline what kind of diagrams we'll generate:

1. **Loss Curve:** Plot the loss of both the generator and the critic over the epochs.
2. **Real vs Generated Data:** Visualize samples of real and generated data to compare.

Since training was not completed due to CUDA issues in this environment, I'll describe how you can plot these diagrams in your local environment.

## Code for Loss Curves Diagram

During the training loop, store the loss values for both the generator and the critic. After training, you can plot them like this:

python

```
import matplotlib.pyplot as plt

# Assuming you have stored the losses in lists
critic_losses = [] # Store critic losses after each iteration
generator_losses = [] # Store generator losses after each iteration

# Plotting the loss curves
plt.figure(figsize=(10,5))
plt.plot(critic_losses, label='Critic Loss')
plt.plot(generator_losses, label='Generator Loss')
plt.title('WGAN Loss Curves')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

## Code for Real vs Generated Data Comparison (For Numeric Data)

After training the generator, you can generate some data points and compare them with the real data using a simple scatter plot or histogram:

python

```
import numpy as np

# Generate samples from the generator
latent_space_samples = torch.randn((100, input_size)).to(device)
generated_data = generator(latent_space_samples).detach().cpu().numpy()

# Assuming you are comparing a particular feature (e.g., IPK)
real_data = processed_dataset['IPK'].values[:100] # First 100 real samples

# Plotting real vs generated data
plt.figure(figsize=(10,5))
plt.hist(real_data, alpha=0.5, label='Real Data (IPK)')
plt.hist(generated_data[:, 0], alpha=0.5, label='Generated Data (IPK)') # Assuming IPK is the first column
plt.title('Real vs Generated Data (IPK)')
plt.xlabel('IPK')
plt.ylabel('Frequency')
```

```
plt.legend()
plt.show()
```

### 3. Additional Diagrams (Optional)

- **TSNE or PCA:** Visualize the data distribution in lower dimensions (if applicable).
- **Sample Outputs:** Plot generated samples at various stages of training to see how the model improves.

Let me know if you need help with any of the training steps or further guidance!



tsne diagram



The **t-SNE** (t-Distributed Stochastic Neighbor Embedding) diagram is a useful tool for visualizing high-dimensional data in 2D or 3D by preserving the local structure of the data. It's especially useful for understanding how well the generator in a GAN has learned to generate realistic data.

Here's how you can create a t-SNE diagram to visualize the distribution of **real vs. generated data** using PyTorch and the t-SNE algorithm from Scikit-learn.

#### Steps:

1. **Generate data** from your trained generator.
2. **Combine real and generated data** for comparison.
3. **Apply t-SNE** to reduce the data dimensions.
4. **Plot the t-SNE result** to visualize the separation between real and generated data.

Below is the code to implement this:

python

```
from sklearn.manifold import TSNE
import numpy as np
import matplotlib.pyplot as plt

# Step 1: Generate synthetic data using the generator
latent_space_samples = torch.randn((1000, input_size)).to(device) # Generate 1000 samples
generated_data = generator(latent_space_samples).detach().cpu().numpy()

# Step 2: Get real data for comparison (you can take a subset)
real_data = processed_dataset.drop(columns='Profesi_encoded').values[:1000] # First 1000
real_samples

# Step 3: Combine real and generated data for visualization
combined_data = np.vstack([real_data, generated_data])
labels = np.array([0]*1000 + [1]*1000) # 0 for real, 1 for generated

# Step 4: Apply t-SNE
tsne = TSNE(n_components=2, perplexity=30, n_iter=300)
tsne_results = tsne.fit_transform(combined_data)

# Step 5: Plot the results
plt.figure(figsize=(10,7))
plt.scatter(tsne_results[:1000, 0], tsne_results[:1000, 1], label='Real Data', alpha=0.6)
```

```
plt.scatter(tsne_results[1000:, 0], tsne_results[1000:, 1], label='Generated Data',
alpha=0.6)
plt.title('t-SNE Visualization of Real vs Generated Data')
plt.xlabel('t-SNE 1')
plt.ylabel('t-SNE 2')
plt.legend()
plt.show()
```

## Explanation:

1. **Generated Data:** We generate 1000 samples using the trained generator model.
2. **Real Data:** We take a subset of the real data (1000 samples) for comparison.
3. **t-SNE:** We apply t-SNE to reduce the data from high-dimensional space to 2D, allowing us to visualize it.
4. **Plot:** The resulting plot shows how well the generator is performing. Ideally, the real and generated data points should overlap if the generator is performing well.

You can experiment with different ``perplexity`` and ``n_iter`` values for better visualization. Let me know if you encounter any issues!



sekarang, bagaimana caranya nge improve atau nge finetune model diatas



To improve or fine-tune the Wasserstein GAN (WGAN) model, you can consider a variety of strategies that target both the **training process** and the **model architecture**. Below are several key areas where you can enhance the WGAN:

## 1. Hyperparameter Tuning

Adjusting hyperparameters can have a significant impact on the performance of the model. Some common hyperparameters to tune include:

- **Learning Rate:** Lowering the learning rate (e.g., from 0.0001 to 0.00005) can stabilize the training of GANs.
- **Batch Size:** Try increasing or decreasing the batch size to see how it affects convergence.
- **Latent Vector Size:** Experiment with different sizes of the noise vector (``input_size``) that feeds into the generator. A larger latent space might help the generator produce more diverse samples.
- **Critic Iterations (``n_critic``):** Increase the number of times the critic is updated before the generator is updated (e.g., ``n_critic` = 5`). This ensures the critic learns faster than the generator, which is beneficial for WGAN.

## 2. Weight Clipping vs. Gradient Penalty

In standard WGANs, weight clipping is used to enforce the Lipschitz constraint. However, this can sometimes lead to suboptimal performance due to overly restricting the capacity of the critic. A common improvement is to replace weight clipping with a **gradient penalty**, as seen in the WGAN-GP variant.

## Implementing WGAN-GP (Gradient Penalty)

Gradient penalty helps enforce the Lipschitz constraint more effectively. Below is how you can add gradient penalty to your WGAN:

python

```
# Gradient Penalty function
def compute_gradient_penalty(critic, real_samples, fake_samples):
    alpha = torch.rand((real_samples.size(0), 1), device=device)
    alpha = alpha.expand_as(real_samples)

    interpolates = alpha * real_samples + (1 - alpha) * fake_samples
    interpolates = interpolates.requires_grad_(True)

    critic_interpolates = critic(interpolates)

    gradients = torch.autograd.grad(
        outputs=critic_interpolates,
        inputs=interpolates,
        grad_outputs=torch.ones_like(critic_interpolates),
        create_graph=True,
        retain_graph=True,
        only_inputs=True
    )[0]

    gradients = gradients.view(gradients.size(0), -1)
    gradient_penalty = ((gradients.norm(2, dim=1) - 1) ** 2).mean()
    return gradient_penalty

# Update critic with gradient penalty in the training loop
lambda_gp = 10 # Gradient penalty weight

# Inside your training loop (Critic Update Step)
real_samples = real_data_batch.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()
```

## 3. Improving Generator and Critic Architectures

You can experiment with deeper or more complex architectures for both the generator and the critic to improve the model's ability to capture data distributions.

- **Larger Networks:** Add more layers or increase the number of hidden units in each layer to capture more complex relationships in the data.
- **Batch Normalization:** Use batch normalization in the generator to help stabilize training and improve sample quality.

Example of expanding the Generator with Batch Normalization:

python

```
class Generator(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(Generator, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(input_size, hidden_size),
            nn.BatchNorm1d(hidden_size), # Adding batch normalization
            nn.ReLU(True),
            nn.Linear(hidden_size, hidden_size),
            nn.BatchNorm1d(hidden_size), # Adding batch normalization
            nn.ReLU(True),
            nn.Linear(hidden_size, output_size),
            nn.Tanh()
        )

    def forward(self, x):
        return self.net(x)
```

## 4. Label Smoothing

Label smoothing can help regularize the discriminator (critic) and prevent it from becoming too confident in distinguishing real from fake samples, which can destabilize GAN training.

python

```
real_labels = torch.ones(batch_size, 1) * 0.9 # Use label smoothing for real labels
fake_labels = torch.zeros(batch_size, 1)
```

## 5. Data Augmentation

If you don't have enough training data, you can augment the real data to make the generator and critic learn more effectively. Data augmentation strategies could involve adding noise, scaling, or rotating real data samples.

## 6. Learning Rate Scheduling

Instead of using a fixed learning rate, applying a **learning rate scheduler** can help the model converge more smoothly over time. You can decrease the learning rate as training progresses:

python

```
scheduler_G = torch.optim.lr_scheduler.StepLR(optimizer_G, step_size=1000, gamma=0.99)
scheduler_C = torch.optim.lr_scheduler.StepLR(optimizer_C, step_size=1000, gamma=0.99)

# Inside the training loop
scheduler_G.step()
scheduler_C.step()
```

## 7. Early Stopping and Model Checkpoints

Monitor the model's performance and apply early stopping or save the model's checkpoints based on improvements in critic or generator loss. This will prevent overfitting or divergence.

## 8. Increase Dataset Size

If possible, adding more data to the training set is one of the best ways to improve the model's generalization and performance.

### Summary of Next Steps:

- **Implement gradient penalty** (WGAN-GP) for better training stability.
- **Tune hyperparameters** such as learning rate, batch size, and latent space size.
- **Improve model architecture** by adding layers, batch normalization, and larger networks.
- Use **label smoothing** to regularize the critic.
- Experiment with **learning rate scheduling** and **early stopping**.

By experimenting with these strategies, you can improve the performance and stability of the Wasserstein GAN. Let me know which direction you'd like to explore first!



How can I implement WGAN-GP in full?



Here's a full implementation of the **Wasserstein GAN with Gradient Penalty (WGAN-GP)** in PyTorch. This implementation includes the **generator**, **critic** (discriminator), **gradient penalty**, and the **training loop**. The WGAN-GP improves upon the standard WGAN by adding the gradient penalty to ensure the Lipschitz continuity constraint is satisfied more effectively.

## Full Code for WGAN-GP Implementation

### 1. Import Required Libraries

python

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
from torch.autograd import grad
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.manifold import TSNE
```

### 2. Define Generator and Critic (Discriminator) Models

python

```
class Generator(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(Generator, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(input_size, hidden_size),
            nn.ReLU(True),
```

```

        nn.Linear(hidden_size, hidden_size),
        nn.ReLU(True),
        nn.Linear(hidden_size, output_size),
        nn.Tanh()
    )

    def forward(self, x):
        return self.net(x)

class Critic(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(Critic, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(input_size, hidden_size),
            nn.ReLU(True),
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(True),
            nn.Linear(hidden_size, 1)
        )

    def forward(self, x):
        return self.net(x)

```

### 3. Gradient Penalty Function

The gradient penalty is used to enforce the Lipschitz constraint.

python

```

def compute_gradient_penalty(critic, real_samples, fake_samples):
    alpha = torch.rand((real_samples.size(0), 1), device=device)
    alpha = alpha.expand_as(real_samples)

    # Interpolate between real and fake data
    interpolates = alpha * real_samples + (1 - alpha) * fake_samples
    interpolates = interpolates.requires_grad_(True)

    critic_interpolates = critic(interpolates)

    gradients = grad(
        outputs=critic_interpolates,
        inputs=interpolates,
        grad_outputs=torch.ones_like(critic_interpolates),
        create_graph=True,
        retain_graph=True,
        only_inputs=True
    )[0]

    gradients = gradients.view(gradients.size(0), -1)
    gradient_penalty = ((gradients.norm(2, dim=1) - 1) ** 2).mean()
    return gradient_penalty

```

### 4. Hyperparameters and Initial Setup

python

```

# Hyperparameters
input_size = 100 # Size of the noise vector
hidden_size = 128
output_size = processed_dataset.shape[1] - 1 # Number of features in the dataset (exclude
the target column)
lr = 0.0001 # Learning rate
batch_size = 64

```

```
n_epochs = 5000
n_critic = 5 # Critic steps per generator step
lambda_gp = 10 # Weight for gradient penalty

# 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))
```

## 5. Training Loop

python

```
# Training loop
critic_losses = []
generator_losses = []

for epoch in range(n_epochs):
    for _ in range(n_critic):
        # Train the critic more often than the generator

        # Sample real data
        real_samples =
torch.tensor(processed_dataset.drop(columns='Profesi_encoded').sample(batch_size).values,
dtype=torch.float).to(device)

        # Generate fake data
        latent_space_samples = torch.randn((batch_size, input_size)).to(device)
        fake_samples = generator(latent_space_samples).detach()

        # Critic forward passes
        critic_real = critic(real_samples)
        critic_fake = critic(fake_samples)

        # Compute gradient penalty
        gradient_penalty = compute_gradient_penalty(critic, real_samples, fake_samples)

        # Critic loss
        loss_critic = -torch.mean(critic_real) + torch.mean(critic_fake) + lambda_gp *
gradient_penalty

        # Backpropagation for critic
        optimizer_C.zero_grad()
        loss_critic.backward()
        optimizer_C.step()

        # Generator forward pass
        latent_space_samples = torch.randn((batch_size, input_size)).to(device)
        fake_samples = generator(latent_space_samples)

        # Generator loss
        loss_generator = -torch.mean(critic(fake_samples))

        # Backpropagation for generator
        optimizer_G.zero_grad()
        loss_generator.backward()
        optimizer_G.step()

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

    # Print losses occasionally
```



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

## 6. Visualize Losses

After the training, you can visualize the losses of the generator and critic to see if the training is stable.

python

```
# Plot the loss curves
plt.figure(figsize=(10,5))
plt.plot(critic_losses, label="Critic Loss")
plt.plot(generator_losses, label="Generator Loss")
plt.title("WGAN-GP Loss Curves")
plt.xlabel("Iterations")
plt.ylabel("Loss")
plt.legend()
plt.show()
```

## 7. Visualizing Real vs Generated Data with t-SNE

Once the model is trained, you can visualize how well the generator is doing by plotting a t-SNE of real vs generated data.

python

```
# Generate data for t-SNE visualization
latent_space_samples = torch.randn((1000, input_size)).to(device)
generated_data = generator(latent_space_samples).detach().cpu().numpy()

# Get real data for comparison
real_data = processed_dataset.drop(columns='Profesi_encoded').values[:1000]

# Combine real and generated data
combined_data = np.vstack([real_data, generated_data])
labels = np.array([0]*1000 + [1]*1000) # 0 for real, 1 for generated

# Apply t-SNE
tsne = TSNE(n_components=2, perplexity=30, n_iter=300)
tsne_results = tsne.fit_transform(combined_data)

# Plot the t-SNE results
plt.figure(figsize=(10,7))
plt.scatter(tsne_results[:1000, 0], tsne_results[:1000, 1], label="Real Data", alpha=0.6)
plt.scatter(tsne_results[1000:, 0], tsne_results[1000:, 1], label="Generated Data",
alpha=0.6)
plt.title("t-SNE Visualization of Real vs Generated Data")
plt.legend()
plt.show()
```

## Key Points

- **Gradient Penalty:** The key improvement over WGAN is the addition of the gradient penalty, which ensures that the Lipschitz condition is satisfied without weight clipping.
- **Critic Steps:** The critic is trained more frequently than the generator (`n\_critic = 5`), allowing it to become stronger before updating the generator.

- **Loss Curves:** Both the critic and generator losses are tracked to ensure training stability.

You can experiment with different hyperparameters, architectures, or even additional regularization techniques to further improve performance.

Let me know if you need more clarification or help in fine-tuning!



How can I use early stopping?



**Early stopping** is a technique to prevent overfitting and ensure that your model stops training when it reaches the point where further training would degrade its performance on validation data. This is particularly useful in training GANs, where balancing the generator and critic is crucial.

To implement **early stopping** in the **WGAN-GP** training loop, we'll monitor either the **generator loss** or the **critic loss**, or both, and if there is no improvement after a certain number of epochs (known as the **patience** parameter), we stop the training.

Here's how you can implement early stopping:

## Steps to Implement Early Stopping

1. **Choose a metric to monitor:** For GANs, you can monitor:
  - **Generator loss:** You want this to decrease over time.
  - **Critic loss:** This should stabilize after some training.
2. **Set a patience level:** This defines how many epochs to wait for an improvement before stopping.
3. **Monitor the loss values:** Track the best loss, and if the loss does not improve over the defined number of patience epochs, stop the training.
4. **Restore the best model:** Optionally, restore the model with the best weights (you can save the model whenever it improves).

## Code Implementation

Here's how to integrate early stopping into your WGAN-GP training loop:

### 1. Modify the Training Loop to Track Loss and Use Early Stopping

python

```
# Hyperparameters for early stopping
patience = 20 # 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 =
torch.tensor(processed_dataset.drop(columns='Profesi_encoded').sample(batch_size).values,
dtype=torch.float).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()}")

```

## Explanation of Code:

1. **Patience:** We set a patience value (e.g., 20 epochs), which means if the loss doesn't improve for 20 consecutive epochs, we stop the training.
2. **Min Delta:** This defines the minimum threshold for an improvement. If the change in loss is less than `min\_delta` (e.g., 0.001), it's not considered an improvement.
3. **Early Stop Counter:** This tracks how many epochs have passed without improvement. Once it exceeds the patience, the training stops.
4. **Saving the Best Model:** Whenever the generator loss improves, we save the best generator and critic models. This way, after stopping, we can restore the best model.

## 2. Restoring the Best Model

If you stopped early and want to continue training later, or just want to evaluate the best model saved during training, you can load the saved weights like this:

python

```
# Load the best saved model
generator.load_state_dict(torch.load("best_generator.pth"))
critic.load_state_dict(torch.load("best_critic.pth"))
```

### 3. Early Stopping Criteria

You could also choose to monitor the **critic loss**, or use a more sophisticated stopping criterion, like calculating the **inception score** (in image GANs) or a custom evaluation metric suited for your domain.

### Fine-Tuning Early Stopping

You can fine-tune early stopping by:

- **Adjusting `patience`:** A smaller patience means the model will stop earlier.
- **Adjusting `min\_delta`:** Increasing `min\_delta` means the model requires a more significant improvement to reset the patience counter.

Let me know if you need further clarification or any additional modifications!

ChatGPT can make mistakes. Check important info.