

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: libcusparsesparse.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.



kalau gitu batu saya untuk memberikan saya full kode nya saja, biar saya coba di lokal



Berikut adalah kode lengkap untuk mengimplementasikan **Wasserstein GAN (WGAN)** menggunakan PyTorch, yang bisa kamu coba di lingkungan lokal dengan CUDA.

```
python

import torch
import torch.nn as nn
import torch.optim as optim
from torch.autograd import Variable
import numpy as np

# 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 = 92 # Number of features from the dataset (excluding the target variable)
batch_size = 64
epochs = 10000
critic_iterations = 5
weight_clipping_limit = 0.01
lr = 0.0001

# Load the dataset
data = np.loadtxt('your_dataset_path.csv', delimiter=',', skiprows=1)
X_train = data[:, :-1] # Features
X_train = torch.tensor(X_train, dtype=torch.float32)

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

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

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

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

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

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

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

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

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

    # Train the generator
    optimizer_G.zero_grad()

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

    # Compute generator loss
```

```
loss_G = -torch.mean(critic(fake_data))

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

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

Penjelasan Kode:

1. **Generator:** Model ini menghasilkan data sintetis berdasarkan noise yang diinput.
2. **Critic (Discriminator):** Model ini mengevaluasi seberapa nyata data yang dihasilkan oleh generator.
3. **Training Loop:** Pada setiap epoch, model critic dilatih beberapa kali lebih banyak daripada generator (sesuai iterasi critic). Critic mengoptimalkan loss Wasserstein, dan generator mencoba meminimalkan feedback dari critic.
4. **Weight Clipping:** Digunakan untuk menjaga nilai parameter critic agar tidak menjadi terlalu besar atau kecil, yang penting untuk kestabilan WGAN.

Langkah-Langkah Menjalankan:

1. Pastikan kamu telah menginstal PyTorch dan mendukung CUDA pada sistem lokal.
2. Ganti ``your_dataset_path.csv`` dengan path ke file dataset kamu.
3. Jalankan kode ini di IDE atau notebook lokal yang mendukung CUDA.

Jika kamu membutuhkan bantuan lebih lanjut dalam menjalankan kode ini di lokal atau memodifikasinya, aku siap membantu!



bagaimana saya tune parameter WGAN ?



Tuning parameter WGAN (Wasserstein GAN) adalah proses iteratif yang melibatkan beberapa eksperimen untuk mendapatkan kombinasi parameter yang optimal. Berikut adalah langkah-langkah dan tips penting yang bisa kamu lakukan untuk tuning WGAN:

1. Latent Vector Size (Input Size)

- **Default:** Biasanya digunakan ukuran 100.
- **Tuning:** Cobalah berbagai ukuran seperti 50, 100, atau 200. Latent vector yang lebih besar dapat memungkinkan generator menghasilkan data yang lebih variatif, tetapi juga bisa membuat training menjadi lebih lambat.

2. Hidden Layer Size

- **Default:** Hidden size seringkali diatur di antara 128 hingga 256.
- **Tuning:** Eksperimen dengan ukuran hidden layer yang lebih besar atau lebih kecil (misalnya, 64, 128, 256, 512). Hidden layer yang lebih besar dapat membantu model menangkap lebih banyak

detail, tetapi dapat menambah kompleksitas dan waktu training.

3. Learning Rate (LR)

- **Default:** Biasanya sekitar 0.00005 hingga 0.0001 untuk WGAN.
- **Tuning:** Coba nilai yang lebih rendah atau lebih tinggi dari nilai default, misalnya 0.00001 hingga 0.0005. Learning rate yang lebih tinggi dapat mempercepat training tetapi berisiko menghasilkan model yang tidak stabil.

4. Critic Iterations

- **Default:** Biasanya antara 1 hingga 5 kali lebih banyak daripada generator.
- **Tuning:** Critic lebih sering di-train dibandingkan generator. Cobalah 1 hingga 10 iterasi critic per satu iterasi generator. Lebih banyak iterasi untuk critic memungkinkan model belajar lebih baik dalam membedakan data nyata dan palsu.

5. Weight Clipping Limit

- **Default:** Biasanya nilai weight clipping berada di kisaran [-0.01, 0.01].
- **Tuning:** Cobalah berbagai batas kliping seperti [-0.005, 0.005] atau [-0.02, 0.02]. Kliping yang terlalu kecil bisa menyebabkan critic tidak belajar dengan baik, sedangkan yang terlalu besar bisa menyebabkan overfitting.

6. Batch Size

- **Default:** Biasanya antara 64 hingga 128.
- **Tuning:** Coba nilai batch size yang berbeda seperti 32, 64, 128, atau bahkan 256. Batch size yang lebih besar membantu stabilitas training, tetapi memerlukan lebih banyak memori.

7. Number of Epochs

- **Default:** Mulai dengan 10,000 iterasi.
- **Tuning:** Tentukan jumlah epoch berdasarkan seberapa stabil loss yang kamu amati. Lakukan plot loss dari generator dan critic selama training untuk menentukan apakah model sudah mulai stabil atau perlu lebih banyak epoch.

8. Gradient Penalty (WGAN-GP)

Jika WGAN dengan weight clipping memberikan hasil yang tidak memuaskan, pertimbangkan untuk menggunakan **WGAN-GP (Wasserstein GAN with Gradient Penalty)**. Gradient Penalty menggantikan weight clipping dan membantu menjaga stabilitas training lebih baik.

Contoh penerapan WGAN-GP:

python

```
class Critic_GP(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(Critic_GP, 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)

# Gradient Penalty
def gradient_penalty(critic, real_data, fake_data, device):
    batch_size = real_data.size(0)
    alpha = torch.rand(batch_size, 1).to(device)
    alpha = alpha.expand_as(real_data)

    interpolated = alpha * real_data + (1 - alpha) * fake_data
    interpolated = Variable(interpolated, requires_grad=True).to(device)

    prob_interp = critic(interpolated)

    gradients = torch.autograd.grad(outputs=prob_interp, inputs=interpolated,
    grad_outputs=torch.ones(prob_interp.size()).to(device),
    create_graph=True, retain_graph=True)[0]
    gradients = gradients.view(batch_size, -1)
    gradient_norm = gradients.norm(2, dim=1)

    return ((gradient_norm - 1) ** 2).mean()

```

9. Logging and Visualization

- **Track Losses:** Plot loss dari generator dan critic di setiap epoch. Kamu ingin melihat critic loss stabil di sekitar nol, yang menandakan bahwa generator menghasilkan data realistis, dan critic tidak mampu membedakan dengan jelas antara data asli dan palsu.
- **Check Generated Data:** Setiap beberapa epoch, cek data sintesis yang dihasilkan oleh generator. Hal ini memungkinkan kamu untuk mengevaluasi apakah generator belajar dengan benar atau tidak.

10. Evaluation Metrics

- Gunakan metrik seperti FID (Fréchet Inception Distance) atau IS (Inception Score) untuk mengevaluasi seberapa baik data yang dihasilkan oleh GAN.

Contoh Tuning:

```

python

# Adjusting hyperparameters for tuning
input_size = 200 # Latent vector size
hidden_size = 256 # Larger hidden size
batch_size = 128 # Larger batch size
lr = 0.00005 # Lower learning rate
critic_iterations = 5 # More critic iterations

# Adjust the training loop accordingly

```

Catatan

- **WGAN-GP** sering memberikan performa yang lebih baik dibandingkan WGAN biasa dengan weight clipping karena menghilangkan masalah yang disebabkan oleh clipping berat.
- **Tuning hyperparameters** sebaiknya dilakukan secara bertahap. Ubah satu parameter pada satu waktu agar lebih mudah mengevaluasi pengaruh setiap perubahan.

Dengan tuning dan eksperimen yang cukup, kamu dapat menemukan kombinasi parameter terbaik untuk modelmu.



bisakah kah anda menjelaskan dari apa itu WGAN-GP dan arsitektur nya



WGAN-GP (Wasserstein GAN with Gradient Penalty) adalah versi modifikasi dari **Wasserstein GAN (WGAN)** yang memperbaiki beberapa kelemahan pada metode WGAN asli, khususnya terkait dengan stabilitas training dan penggunaan weight clipping pada critic (discriminator).

Berikut penjelasan lebih detail mengenai WGAN-GP dan arsitekturnya:

1. Wasserstein GAN (WGAN) Recap

WGAN menggunakan **distance Wasserstein (Earth Mover's Distance)** sebagai metrik untuk mengukur perbedaan antara distribusi data nyata dan data palsu yang dihasilkan oleh generator. Metrik ini lebih stabil dibandingkan **Jensen-Shannon Divergence** yang digunakan pada GAN konvensional. Namun, dalam WGAN asli, critic dibatasi oleh **weight clipping** untuk menjaga fungsi critic agar tetap Lipschitz continuous, yaitu tidak memiliki gradien yang tidak terkendali.

2. Masalah dengan WGAN Original

Meskipun WGAN secara signifikan meningkatkan stabilitas training GAN, weight clipping yang digunakan pada WGAN dapat menyebabkan beberapa masalah:

- **Training critic tidak optimal:** Clipping gradien secara kasar dapat menyebabkan critic underfitting atau gagal belajar hubungan yang lebih baik antara data nyata dan data palsu.
- **Degradasi performa critic:** Dalam jangka panjang, clipping yang ketat pada weight critic menyebabkan model kehilangan detail penting dalam pembelajaran dan membuat training menjadi tidak stabil.

3. WGAN-GP: Perbaikan dengan Gradient Penalty

WGAN-GP memperbaiki kelemahan WGAN dengan mengganti weight clipping menggunakan **gradient penalty**. Alih-alih melakukan clipping pada critic, WGAN-GP memastikan bahwa critic tetap Lipschitz continuous melalui penalti yang diberikan pada gradien critic. Dengan demikian, training lebih stabil dan critic dapat belajar lebih baik.

Penjelasan Gradient Penalty

Gradient penalty diterapkan pada interpolasi antara real data dan fake data (hasil dari generator), dengan menghitung gradien critic pada titik-titik ini. Gradient penalty dihitung sebagai:

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

di mana:

- λ adalah koefisien regularisasi (biasanya 10),

- $D(\hat{x})$ adalah critic yang dievaluasi pada interpolasi \hat{x} ,
- $\nabla_{\hat{x}} D(\hat{x})$ adalah gradien dari critic D pada interpolasi \hat{x} .

Tujuan dari penalti ini adalah memastikan bahwa norm gradien critic mendekati 1, yang diperlukan untuk menjaga fungsi Lipschitz continuity.

4. Arsitektur WGAN-GP

WGAN-GP memiliki dua komponen utama yang mirip dengan GAN konvensional:

1. **Generator:** Bertanggung jawab untuk menghasilkan data sintetik dari noise.
2. **Critic:** Mengganti peran **discriminator** dalam GAN konvensional. Critic memprediksi seberapa realistis data yang dihasilkan oleh generator dibandingkan dengan data nyata.

Arsitektur Critic (Discriminator) dalam WGAN-GP

- Tidak menggunakan sigmoid di layer output, karena critic menghasilkan skor yang bukan probabilitas.
- Pada WGAN-GP, critic tidak harus dibatasi melalui weight clipping, tetapi melalui penalti gradien.
- Sama seperti WGAN, output critic adalah skalar yang mencerminkan "jarak Wasserstein" antara distribusi data nyata dan distribusi data palsu.

Arsitektur Generator dalam WGAN-GP

- Generator tetap menerima noise sebagai input (biasanya berupa vector random dengan distribusi Gaussian) dan menghasilkan data yang menyerupai data nyata.
- Mirip dengan WGAN, tetapi dioptimalkan terhadap feedback dari critic.

5. Loss Function dalam WGAN-GP

Critic Loss: Critic dilatih untuk memaksimalkan perbedaan antara nilai prediksi untuk data nyata dan data palsu:

$$\text{Loss Critic} = \mathbb{E}[D(x_{\text{real}})] - \mathbb{E}[D(x_{\text{fake}})] + \lambda \cdot \text{Gradient Penalty}$$

Generator Loss: Generator dilatih untuk meminimalkan nilai critic pada data palsu yang dihasilkannya:

$$\text{Loss Generator} = -\mathbb{E}[D(x_{\text{fake}})]$$

6. Cara Menghitung Gradient Penalty

Di WGAN-GP, penalti gradien diterapkan pada interpolasi antara data nyata dan data palsu. Berikut langkah-langkahnya:

1. **Interpolasi:** Interpolasi dilakukan antara sampel nyata dan palsu. Interpolasi ini dihitung sebagai:

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

di mana α adalah scalar acak yang mengikuti distribusi seragam.

2. **Gradien Critic:** Critic dievaluasi pada titik interpolasi \hat{x} dan gradien dari critic dihitung dengan backpropagation.
3. **Penalti Gradien:** Gradient penalty dihitung sebagai:

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

7. Keunggulan WGAN-GP

1. **Stabilitas Training:** Gradient penalty membantu menjaga critic dalam kondisi Lipschitz tanpa perlu menggunakan weight clipping yang dapat merusak performa model.
2. **Generalisasi Lebih Baik:** Critic bisa belajar distribusi lebih luas tanpa kehilangan detail penting.
3. **Lebih Mudah Di-tuning:** Gradient penalty lebih mudah di-tuning dibandingkan weight clipping, yang bisa terlalu membatasi critic.

8. Contoh Arsitektur WGAN-GP (PyTorch)

Berikut contoh kode untuk critic dengan gradient penalty:

python

```
class Critic_GP(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(Critic_GP, 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)

def gradient_penalty(critic, real_data, fake_data, device):
    batch_size = real_data.size(0)
    alpha = torch.rand(batch_size, 1).to(device)
    alpha = alpha.expand_as(real_data)

    interpolated = alpha * real_data + (1 - alpha) * fake_data
    interpolated = Variable(interpolated, requires_grad=True).to(device)

    prob_interp = critic(interpolated)

    gradients = torch.autograd.grad(outputs=prob_interp, inputs=interpolated,
    grad_outputs=torch.ones(prob_interp.size()).to(device),
    create_graph=True, retain_graph=True)[0]
    gradients = gradients.view(batch_size, -1)
    gradient_norm = gradients.norm(2, dim=1)

    return ((gradient_norm - 1) ** 2).mean()

# Loss Critic with gradient penalty
def critic_loss(critic, real_data, fake_data, gp_weight, device):
    critic_real = critic(real_data)
    critic_fake = critic(fake_data)
    gp = gradient_penalty(critic, real_data, fake_data, device)
    return -(torch.mean(critic_real) - torch.mean(critic_fake)) + gp_weight * gp
```

9. Hyperparameter untuk WGAN-GP

- **Lambda (Gradient Penalty Coefficient):** Biasanya diatur ke 10.
- **Learning Rate:** Coba mulai dari 0.0001 hingga 0.00005.
- **Critic Iterations:** Antara 5 hingga 10 kali per iterasi generator.

Kesimpulan

WGAN-GP adalah perbaikan dari WGAN yang mengatasi kelemahan weight clipping dengan menggunakan gradient penalty. Ini memberikan hasil yang lebih stabil dan memungkinkan critic untuk belajar lebih baik tanpa batasan keras dari clipping.



Apa perbedaan utama WGAN dan GAN?



Perbedaan utama antara **Wasserstein GAN (WGAN)** dan **GAN** konvensional terletak pada cara mereka mengukur perbedaan antara distribusi data nyata dan distribusi data yang dihasilkan oleh generator, serta beberapa aspek teknik yang mempengaruhi stabilitas training.

1. Loss Function

- **GAN Konvensional:** Menggunakan **Jensen-Shannon Divergence (JS Divergence)** untuk mengukur perbedaan antara distribusi data nyata dan distribusi data palsu. Discriminator mencoba memaksimalkan JS Divergence antara distribusi nyata dan palsu, sementara generator mencoba meminimalkan nilai tersebut.

- **Discriminator Loss:**

$$\mathbb{E}[\log(D(x_{\text{real}}))] + \mathbb{E}[\log(1 - D(x_{\text{fake}}))]$$

- **Generator Loss:**

$$\mathbb{E}[\log(1 - D(x_{\text{fake}}))]$$

- **Masalah:** JS Divergence dapat menjadi 0 ketika dua distribusi tidak memiliki overlap, yang sering terjadi selama training GAN. Ini menyebabkan **vanishing gradients** (gradien mendekati nol), sehingga generator tidak dapat belajar dengan baik.
- **WGAN:** Menggunakan **Wasserstein Distance (Earth Mover's Distance)** untuk mengukur perbedaan antara distribusi. Wasserstein Distance adalah ukuran berapa banyak "usaha" yang diperlukan untuk mengubah satu distribusi menjadi distribusi lain. WGAN menggunakan loss ini karena memberikan gradien yang lebih stabil dan tidak menyebabkan vanishing gradients, bahkan ketika dua distribusi tidak tumpang tindih.

- **Critic (Discriminator) Loss:**

$$\mathbb{E}[D(x_{\text{real}})] - \mathbb{E}[D(x_{\text{fake}})]$$

- **Generator Loss:**

$$-\mathbb{E}[D(x_{\text{fake}})]$$

- **Keuntungan:** Wasserstein Distance memberikan gradien yang lebih bermakna dan stabil selama training, sehingga generator tetap mendapatkan sinyal yang kuat untuk

memperbaiki kinerjanya, bahkan pada kondisi di mana JS Divergence dalam GAN menyebabkan tidak ada gradien yang bisa digunakan.

2. Discriminator vs Critic

- **GAN Konvensional:** Discriminator adalah model klasifikasi biner yang dilatih untuk memprediksi apakah data yang diberikan adalah asli atau palsu. Outputnya berupa probabilitas (antara 0 dan 1) menggunakan fungsi aktivasi sigmoid pada layer terakhir.
- **WGAN:** Mengganti discriminator dengan **critic**. Critic tidak melakukan klasifikasi, tetapi mengevaluasi "jarak" antara distribusi data nyata dan palsu. Tidak ada fungsi aktivasi sigmoid pada output layer terakhir critic, sehingga output bisa berupa nilai real (bukan probabilitas). Critic memberikan skor untuk menunjukkan seberapa jauh distribusi data palsu dari distribusi data nyata.

3. Vanishing Gradients

- **GAN Konvensional:** Masalah utama pada GAN adalah **vanishing gradients**, di mana generator berhenti mendapatkan sinyal gradien yang kuat dari discriminator ketika distribusi data palsu sangat berbeda dari distribusi data nyata. Hal ini sering menyebabkan **mode collapse**, di mana generator hanya menghasilkan sebagian kecil dari variasi data yang diharapkan.
- **WGAN:** Dengan menggunakan Wasserstein Distance, WGAN memberikan sinyal gradien yang lebih kuat dan stabil, bahkan ketika data palsu sangat berbeda dari data nyata. Ini membantu mengatasi masalah vanishing gradients dan mengurangi kemungkinan mode collapse.

4. Training Stabilization

- **GAN Konvensional:** GAN konvensional seringkali tidak stabil selama training. Dalam banyak kasus, model tidak konvergen, dan training bisa sangat sensitif terhadap pilihan hyperparameter, seperti learning rate.
- **WGAN:** WGAN lebih stabil dalam training karena penggunaan Wasserstein Distance yang menghasilkan gradien yang lebih bermakna. Namun, WGAN awal masih memiliki beberapa masalah stabilitas, yang akhirnya disempurnakan lebih lanjut dengan **WGAN-GP (Wasserstein GAN with Gradient Penalty)**.

5. Weight Clipping vs Gradient Penalty

- **WGAN (original):** WGAN memastikan bahwa critic tetap **Lipschitz continuous** dengan menerapkan **weight clipping** pada parameter critic. Parameter critic dibatasi ke dalam range tertentu (misalnya antara -0.01 hingga 0.01) untuk mencegah gradien yang tidak terkendali.
 - **Masalah:** Weight clipping terlalu kasar dan dapat menyebabkan critic underfitting atau gagal belajar representasi yang akurat tentang distribusi data.
- **WGAN-GP:** **WGAN-GP** memperbaiki masalah weight clipping dengan menggantinya menggunakan **gradient penalty**. Gradient penalty menghukum critic jika gradiennya terlalu jauh dari nilai 1, sehingga menjaga Lipschitz continuity tanpa harus membatasi parameter secara langsung. Gradient penalty menghasilkan training yang lebih stabil dan critic yang lebih kuat.

6. Mode Collapse

- **GAN Konvensional:** Sering kali GAN konvensional mengalami **mode collapse**, di mana generator hanya menghasilkan satu atau beberapa pola output yang terbatas dan mengabaikan variasi yang lebih luas dalam distribusi data.

- **WGAN:** Karena critic memberikan sinyal yang lebih bermakna dan konsisten, WGAN lebih tahan terhadap mode collapse dibandingkan dengan GAN konvensional. Ini terjadi karena Wasserstein Distance terus memberikan gradien meskipun distribusi nyata dan palsu sangat berbeda.

7. Evaluasi

- **GAN Konvensional:** Menggunakan probabilitas diskriminator untuk mengevaluasi seberapa baik generator bekerja.
- **WGAN:** Menggunakan output critic untuk mengukur Wasserstein Distance antara distribusi nyata dan palsu. Semakin kecil nilai dari distance ini, semakin baik generator bekerja.

Rangkuman Perbedaan Utama:

Aspek	GAN Konvensional	WGAN
Loss Function	JS Divergence, menggunakan log-loss	Wasserstein Distance (Earth Mover's Distance)
Discriminator vs Critic	Discriminator, klasifikasi biner dengan sigmoid	Critic, tidak menggunakan sigmoid, memberikan skor
Vanishing Gradients	Bisa terjadi, menyebabkan mode collapse	Tidak ada vanishing gradients, gradien selalu ada
Training Stability	Sering tidak stabil, sensitif terhadap hyperparameter	Lebih stabil selama training
Weight Clipping	Tidak diperlukan	Diperlukan untuk menjaga Lipschitz continuity
Mode Collapse	Sering terjadi	Lebih tahan terhadap mode collapse
Critic Training	Tidak ada	Critic harus dilatih beberapa kali lebih banyak dari generator
Weight Clipping/Gradient Penalty	Tidak diterapkan	WGAN: weight clipping, WGAN-GP: gradient penalty

Kesimpulan

WGAN adalah versi yang lebih stabil dari GAN yang mengatasi banyak masalah utama GAN konvensional seperti vanishing gradients dan mode collapse dengan menggunakan **Wasserstein Distance** dan **critic**. Jika stabilitas training menjadi masalah pada GAN, WGAN atau lebih lanjut **WGAN-GP** sering kali menjadi pilihan yang lebih baik, karena memberikan sinyal gradien yang lebih kuat dan lebih bermakna kepada generator, sehingga membantu menghasilkan model yang lebih stabil dan akurat.

ChatGPT can make mistakes. Check important info.