```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.layers import Dense, LeakyReLU, Input
from tensorflow.keras.models import Model
import numpy as np

def build_generator(input_dim, output_dim):
    inputs = Input(shape=(input_dim,))
    x = Dense(16, activation="relu")(inputs)
    x = Dense(32, activation="relu")(x)
    x = Dense(16, activation="relu")(x)
    outputs = Dense(output_dim, activation="softmax")(x)  # Output
probabilities

    model = Model(inputs, outputs, name="Generator")
    return model

def build_discriminator(input_dim):
    inputs = Input(shape=(input_dim,))
    x = Dense(16, activation=LeakyReLU(0.2))(inputs)
    x = Dense(32, activation=LeakyReLU(0.2))(x)
    x = Dense(16, activation=LeakyReLU(0.2))(x)
    outputs = Dense(1, activation="sigmoid")(x)  # Binary
classification

    model = Model(inputs, outputs, name="Discriminator")
    return model

def build_gan(generator, discriminator):
    discriminator.trainable = False  # Freeze the discriminator while
training the generator

    inputs = Input(shape=(generator.input_shape[1],))
    generated_output = generator(inputs)
    validity = discriminator(generated_output)

    gan = Model(inputs, validity, name="GAN")
    gan.compile(loss="binary_crossentropy",
optimizer=keras.optimizers.Adam(0.0002, 0.5))
    return gan

# Define input and output dimensions
input_dim = 1  # Encoded top
```

```
output_dim = len(df["Bottom"].unique())  # Number of unique bottoms

# Build models
generator = build_generator(input_dim, output_dim)
discriminator = build_discriminator(output_dim)
discriminator.compile(loss="binary_crossentropy",
optimizer=keras.optimizers.Adam(0.0002, 0.5), metrics=["accuracy"])

# Combine into GAN
gan = build_gan(generator, discriminator)

# Training parameters
epochs = 5000
batch_size = 32

# Prepare real data (convert to one-hot vectors)
from tensorflow.keras.utils import to_categorical

X_real = np.array(df["Top_Encoded"]).reshape(-1, 1)
y_real = to_categorical(df["Bottom_Encoded"], num_classes=output_dim)

# Training loop
for epoch in range(epochs):
    # Select a random batch of real outfits
    idx = np.random.randint(0, X_real.shape[0], batch_size)
    tops_real = X_real[idx]
    bottoms_real = y_real[idx]

    # Generate fake bottoms
    noise = np.random.normal(0, 1, (batch_size, 1))  # Random noise
    bottoms_fake = generator.predict(tops_real)  # Generate fake bottom
suggestions

    # Labels for real and fake data
    real_labels = np.ones((batch_size, 1))  # Real outfits = 1
    fake_labels = np.zeros((batch_size, 1))  # Fake outfits = 0

    # Train the discriminator
    d_loss_real = discriminator.train_on_batch(bottoms_real,
real_labels)
    d_loss_fake = discriminator.train_on_batch(bottoms_fake,
fake_labels)
    d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)
```

```
    # Train the generator (trying to make fake bottoms look real)
    misleading_labels = np.ones((batch_size, 1))  # Fool the
discriminator
    g_loss = gan.train_on_batch(tops_real, misleading_labels)

    # Print progress every 500 epochs
    if epoch % 500 == 0:
        print(f"Epoch {epoch} | D Loss: {d_loss[0]} | G Loss:
{g_loss}")

def generate_outfit(top_description):
    top_encoded = top_encoder.transform([top_description]).reshape(1,
-1)  # Convert text to encoded form
    generated_bottom = generator.predict(top_encoded)
    bottom_index = np.argmax(generated_bottom)  # Get the most likely
bottom
    bottom_description =
bottom_encoder.inverse_transform([bottom_index])[0]

    return f"Suggested Bottom for '{top_description}':
{bottom_description}"

# Example usage
print(generate_outfit("Red Shirt"))

import difflib

# Sample dataset of available top descriptions
valid_tops = ["red shirt", "blue polo", "green blouse", "black
t-shirt", "white hoodie"]

def suggest_bottom(top_description):
    # Normalize input (convert to lowercase and strip spaces)
    normalized_top = top_description.lower().strip()

    # Check if the top is in the dataset
    if normalized_top in valid_tops:
        return f"Top: {top_description}  --> Suggested Bottom: Black
Leggings"

    # If not found, suggest a similar valid top
```

```
    close_matches = difflib.get_close_matches(normalized_top,
valid_tops, n=1, cutoff=0.6)

    if close_matches:
        return f"'{top_description}' not found. Did you mean
'{close_matches[0]}'?"
    else:
        return f"'{top_description}' is not in the dataset. Please
provide a valid top description."

# Example usage
print(suggest_bottom("Black T-shirt"))  # Now this should work!
print(suggest_bottom("Black Tee"))  # Should suggest "Black T-shirt"
print(suggest_bottom("Yellow Hoodie"))  # Should return not found
message

def get_best_match(query, choices):
    """
    Finds the closest match for the given query in the dataset.
    """
    best_match, score = process.extractOne(query, choices)
    return best_match if score > 80 else None  # Only return if
confidence > 80%

def test_model_fuzzy(top_list):
    """
    Tests the model with fuzzy matching for top descriptions.
    """
    for top in top_list:
        # Find the closest match if the top isn't in the dataset
        if top not in top_encoder.classes_:
            best_match = get_best_match(top, top_encoder.classes_)
            if best_match:
                print(f"'{top}' not found. Did you mean
'{best_match}'?")
                top = best_match  # Use the corrected name
            else:
                print(f"'{top}' not found in the dataset.")
                continue  # Skip this top if no good match is found

        # Convert the top description into its encoded form
        top_encoded = top_encoder.transform([top]).reshape(1, -1)
```

```
        # Generate a bottom suggestion
        generated_bottom = generator.predict(top_encoded)
        bottom_index = np.argmax(generated_bottom)  # Get the most
likely bottom
        bottom_description =
bottom_encoder.inverse_transform([bottom_index])[0]

        print(f"Top: {top}  -->  Suggested Bottom:
{bottom_description}")

# Example test cases
test_tops = ["Black Tee", "Yellow Hoodie", "Red Shirt", "Blue Polo"]
test_model_fuzzy(test_tops)

from fuzzywuzzy import process

def get_best_match(query, choices, threshold=60):
    """
    Finds the closest match for the given query in the dataset.
    """
    best_match, score = process.extractOne(query, choices)
    return best_match if score > threshold else None  # Lowered
threshold to 60%

def test_model_fuzzy_debug(top_list):
    """
    Tests the model with improved fuzzy matching and debugging.
    """
    for top in top_list:
        # Find the closest match if the top isn't in the dataset
        if top not in top_encoder.classes_:
            best_match = get_best_match(top, top_encoder.classes_,
threshold=60)

            # Debugging: Show the closest match found
            if best_match:
                print(f"'{top}' not found. Did you mean '{best_match}'?
(Confidence Score: {process.extractOne(top,
top_encoder.classes_)[1]}%)")
                top = best_match  # Use the corrected name
            else:
                print(f"'{top}' not found in the dataset. Skipping.")
                continue  # Skip this top if no good match is found
```

```
        # Convert the top description into its encoded form
        top_encoded = top_encoder.transform([top]).reshape(1, -1)

        # Generate a bottom suggestion
        generated_bottom = generator.predict(top_encoded)
        bottom_index = np.argmax(generated_bottom)  # Get the most
likely bottom
        bottom_description =
bottom_encoder.inverse_transform([bottom_index])[0]

        print(f"Top: {top}  -->  Suggested Bottom:
{bottom_description}")

# Example test cases
test_tops = ["Black Tee", "Yellow Hoodie", "Red Shirt", "Blue Polo"]
test_model_fuzzy_debug(test_tops)

def build_gan(generator, discriminator):
    discriminator.trainable = False  # Freeze discriminator for GAN
training

    inputs = Input(shape=(generator.input_shape[1],))
    generated_output = generator(inputs)
    validity = discriminator(generated_output)

    gan = Model(inputs, validity, name="GAN")
    gan.compile(loss="binary_crossentropy",
optimizer=keras.optimizers.Adam(0.0002, 0.5))

    # After compiling GAN, set discriminator back to trainable
    discriminator.trainable = True
    return gan

from tensorflow.keras.optimizers import Adam

# Define the learning rate (same as before)
learning_rate = 0.0001

# Compile Discriminator
discriminator.compile(loss='binary_crossentropy',
optimizer=Adam(learning_rate), metrics=['accuracy'])
```

```
# GAN (Generator + Discriminator)
discriminator.trainable = False  # Freeze discriminator for GAN
training
gan_input = tf.keras.Input(shape=(100,))  # Latent space size
generated_outfit = generator(gan_input)
gan_output = discriminator(generated_outfit)

# Create a new GAN model
gan = tf.keras.Model(gan_input, gan_output)
gan.compile(loss='binary_crossentropy', optimizer=Adam(learning_rate))

print("✅ Models successfully recompiled and ready for training!")

import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow.keras.layers import Dense, LeakyReLU,
BatchNormalization, Dropout, Flatten
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import Adam
from sklearn.preprocessing import LabelEncoder

# Load dataset
df = pd.read_csv("matching_outfits.csv")  # Update with your actual
dataset path

# Balance dataset to ensure equal number of bottoms
min_count = df['Bottom'].value_counts().min()
df_balanced = df.groupby('Bottom').apply(lambda x:
x.sample(min_count)).reset_index(drop=True)

# Encode clothing names as categorical numbers
label_encoder_top = LabelEncoder()
label_encoder_bottom = LabelEncoder()
df_balanced['Top'] =
label_encoder_top.fit_transform(df_balanced['Top'])
df_balanced['Bottom'] =
label_encoder_bottom.fit_transform(df_balanced['Bottom'])

# Get the number of unique tops and bottoms
num_tops = len(label_encoder_top.classes_)
num_bottoms = len(label_encoder_bottom.classes_)
```

```
latent_dim = 100  # Size of noise vector

def build_generator():
    model = Sequential([
        Dense(128, input_dim=latent_dim),
        LeakyReLU(alpha=0.2),
        BatchNormalization(),
        Dense(256),
        LeakyReLU(alpha=0.2),
        BatchNormalization(),
        Dense(num_tops + num_bottoms, activation='softmax')  # Output
probabilities
    ])
    return model

generator = build_generator()

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LeakyReLU,
BatchNormalization

# Fix Generator Model
generator = Sequential([
    Dense(128, input_dim=latent_dim, activation='relu'),
    Dense(64, activation='relu'),
    Dense(2, activation='linear')  # Output must be (Top, Bottom)
])

# Recompile Generator
generator.compile(optimizer=Adam(learning_rate),
loss='binary_crossentropy')

print("✅ Generator model fixed! Now outputs (Top, Bottom).")

epochs = 5000
batch_size = 64

for epoch in range(epochs):
    # Generate random noise
    noise = np.random.normal(0, 1, (batch_size, latent_dim))

    # Generate fake outfits
    fake_outfits_raw = generator.predict(noise)
```

```
    # Convert raw softmax outputs to category indices
    fake_tops = np.argmax(fake_outfits_raw[:, :num_tops], axis=1)
    fake_bottoms = np.argmax(fake_outfits_raw[:, num_tops:], axis=1)
    fake_outfits = np.vstack((fake_tops, fake_bottoms)).T  # Shape:
(batch_size, 2)

    # Select a batch of real outfits
    real_outfits = df_balanced[['Top',
'Bottom']].sample(batch_size).values

    # Combine real and fake outfits for training the discriminator
    X_train = np.vstack((real_outfits, fake_outfits))
    y_train = np.array([1] * batch_size + [0] * batch_size)  # 1 =
real, 0 = fake

    # Shuffle the training data
    indices = np.arange(len(X_train))
    np.random.shuffle(indices)
    X_train, y_train = X_train[indices], y_train[indices]

    # Train the discriminator
    d_loss = discriminator.train_on_batch(X_train, y_train)

    # Train the generator (fooling the discriminator)
    y_fake = np.ones(batch_size)  # Labels flipped to "real"
    g_loss = gan.train_on_batch(noise, y_fake)

    # Print progress every 500 epochs
    if epoch % 500 == 0:
        print(f"Epoch {epoch} | D Loss: {d_loss[0]:.4f} | G Loss:
{g_loss:.4f}")

print("🎉 Training Completed!")

temperature = 0.7  # Lower values make selections more random

def sample_with_temperature(probs, temperature=1.0):
    probs = np.log(probs + 1e-8) / temperature  # Apply temperature
scaling
    exp_probs = np.exp(probs)
    return np.argmax(exp_probs / np.sum(exp_probs, axis=-1,
keepdims=True), axis=-1)
```

```
def generate_outfits(num_samples=10):
    noise = np.random.normal(0, 1, (num_samples, latent_dim))
    fake_outfits_raw = generator.predict(noise)

    fake_tops = sample_with_temperature(fake_outfits_raw[:, :num_tops],
temperature)
    fake_bottoms = sample_with_temperature(fake_outfits_raw[:,
num_tops:], temperature)

    decoded_outfits = [(label_encoder_top.inverse_transform([top])[0],

label_encoder_bottom.inverse_transform([bottom])[0])
                       for top, bottom in zip(fake_tops,
fake_bottoms)]

    print("\n🔷 Generated Outfit Combinations:")
    for i, outfit in enumerate(decoded_outfits):
        print(f"{i+1}. {outfit[0]} + {outfit[1]}")

generate_outfits(10)

import numpy as np
import pandas as pd
import tensorflow as tf
from sklearn.preprocessing import LabelEncoder
from google.colab import drive

# ✅ Step 1: Mount Google Drive
drive.mount('/content/drive')

# ✅ Step 2: Load Input Dataset
input_file_path = "/content/drive/MyDrive/10.csv"
input_df = pd.read_csv(input_file_path)

# Extract unique top and bottom names
top_classes = input_df["Top"].unique().tolist()
bottom_classes = input_df["Bottom"].unique().tolist()

# ✅ Step 3: Load the Trained Generator Model
generator =
tf.keras.models.load_model("/content/drive/MyDrive/gan_generator.keras"
)
```

```
print(generator.input_shape)  # Verify expected input shape

# ✅ Step 4: Adjust Latent Dimension Based on Model Input Shape
latent_dim = generator.input_shape[1]  # Dynamically set correct input
shape

# ✅ Step 5: Encode Categories
label_encoder_top = LabelEncoder()
label_encoder_bottom = LabelEncoder()
label_encoder_top.fit(top_classes)
label_encoder_bottom.fit(bottom_classes)

# ✅ Step 6: Generate Unique Outfit Combinations
def generate_unique_outfits(num_samples=5):
    unique_outfits = set()  # Store unique combinations

    while len(unique_outfits) < num_samples:
        noise = np.random.normal(0, 1, (1, latent_dim))  # Generate one
at a time
        generated_outfits_raw = generator.predict(noise)

        # Convert softmax probabilities to actual categories
        num_tops = len(top_classes)
        fake_top = np.argmax(generated_outfits_raw[:, :num_tops],
axis=1)[0]
        fake_bottom = np.argmax(generated_outfits_raw[:, num_tops:],
axis=1)[0]

        # Decode category names
        outfit = (label_encoder_top.inverse_transform([fake_top])[0],

label_encoder_bottom.inverse_transform([fake_bottom])[0])

        unique_outfits.add(outfit)  # Ensure uniqueness

    # Convert to list and display
    unique_outfits = list(unique_outfits)
    print("\n🎉 **Generated Unique Outfit Combinations:**")
    for i, (top, bottom) in enumerate(unique_outfits, start=1):
        print(f"{i}. {top} + {bottom}")

# ✅ Generate and Display Outfits
generate_unique_outfits(5)
```