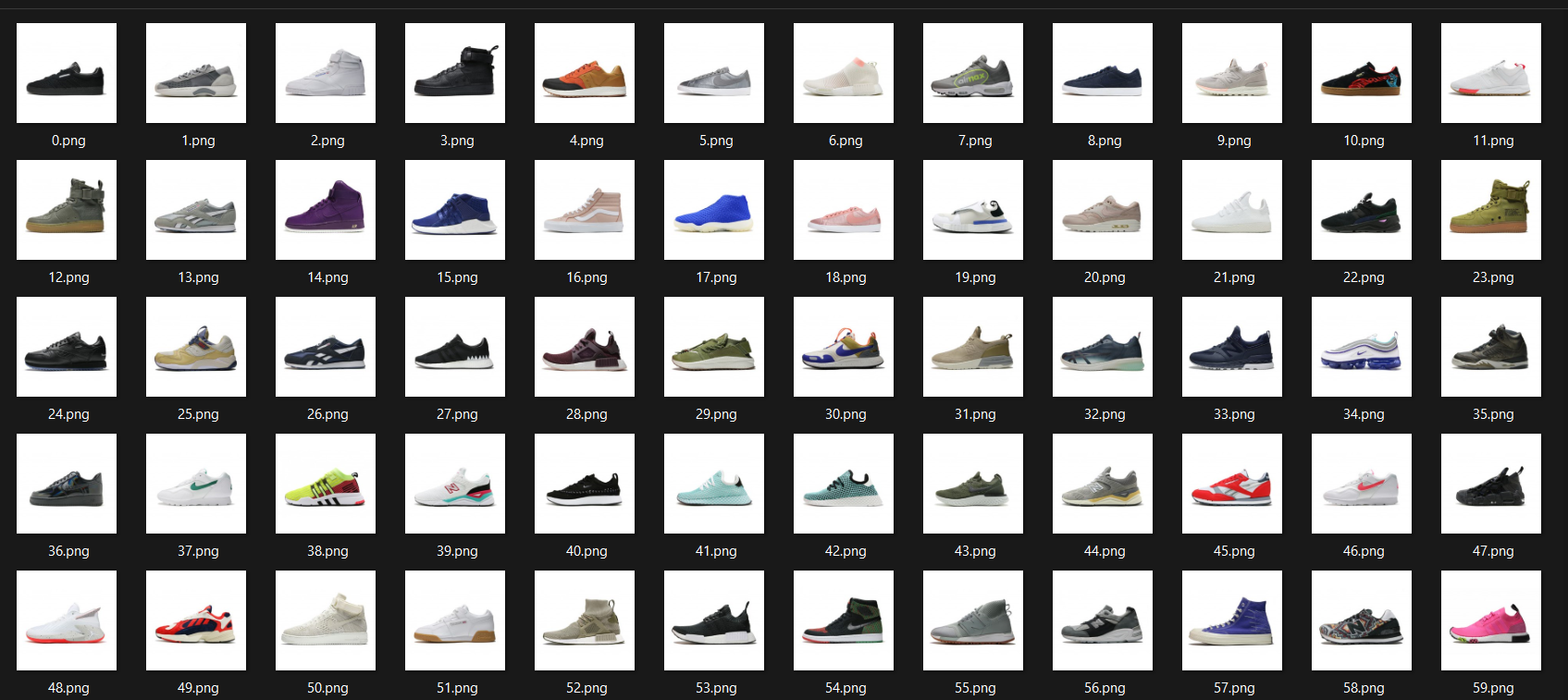
**Sneaker Design Generation using Generative Adversarial Networks**

1. Dataset
   * Link for base dataset: <https://github.com/micah5/sneaker-generator/tree/master/dataset/class1>
   * Part of dataset scraped from e-commerce websites, like amazon.
   * Total images - 2718
   * Sample –



* + Dataset annotated into black, white, blue, red and tan colors using following script:

from google.colab import drive

drive.mount('/content/drive')

!unzip "/content/drive/My Drive/CV Project/Sneaker Dataset.zip"

!mkdir "/content/dataset"

!mkdir "/content/dataset/black"

!mkdir "/content/dataset/white"

!mkdir "/content/dataset/brown"

!mkdir "/content/dataset/red"

!mkdir "/content/dataset/blue"

!mkdir "/content/dataset/yellow"

from transformers import pipeline

from PIL import Image

import os

pipe = pipeline("zero-shot-image-classification", model="openai/clip-vit-large-patch14")

for file in os.listdir("/content/Sneaker Dataset"):

    image = Image.open(os.path.join("/content/Sneaker Dataset", file))

    label = pipe(image, candidate\_labels=["black shoe", "white shoe", "brown shoe", "red shoe", "blue shoe", "yellow shoe"])[0]['label']

    if label == "black shoe":

        os.rename(os.path.join("/content/Sneaker Dataset", file), os.path.join("/content/dataset/black", file))

    elif label == "white shoe":

        os.rename(os.path.join("/content/Sneaker Dataset", file), os.path.join("/content/dataset/white", file))

    elif label == "brown shoe":

        os.rename(os.path.join("/content/Sneaker Dataset", file), os.path.join("/content/dataset/brown", file))

    elif label == "red shoe":

        os.rename(os.path.join("/content/Sneaker Dataset", file), os.path.join("/content/dataset/red", file))

    elif label == "blue shoe":

        os.rename(os.path.join("/content/Sneaker Dataset", file), os.path.join("/content/dataset/blue", file))

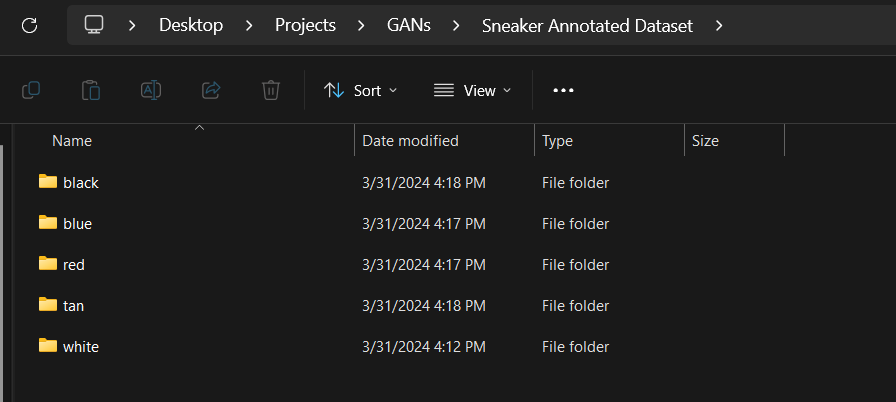
    elif label == "yellow shoe":

        os.rename(os.path.join("/content/Sneaker Dataset", file), os.path.join("/content/dataset/yellow", file))

!mv "/content/dataset" "/content/drive/My Drive/dataset"

!zip -r "/content/drive/My Drive/dataset.zip" "/content/drive/My Drive/dataset"

* + Annotated dataset file structure:



1. Training scripts

Deep Convolutional Generative Adversarial Network (DCGAN)

from google.colab import drive

drive.mount("/content/drive")

!unzip "/content/drive/My Drive/CV Project/Sneaker Dataset.zip"

import matplotlib.pyplot as plt

import numpy as np

import os

from tensorflow.keras.layers import Conv2D, Conv2DTranspose, Dense, Dropout, Flatten, BatchNormalization, Input, Reshape, LeakyReLU, ReLU

from tensorflow.keras.models import Sequential

from tensorflow.keras.optimizers import Adam

from tensorflow.keras.utils import img\_to\_array, load\_img

def discriminator(input\_shape=(128, 128, 3)):

    model = Sequential([

        Conv2D(64, (5, 5), strides=(2, 2), padding='same', input\_shape=input\_shape),

        LeakyReLU(alpha=0.2),

        Conv2D(128, (5, 5), strides=(2, 2), padding='same', use\_bias=False),

        BatchNormalization(momentum=0.5),

        LeakyReLU(alpha=0.2),

        Conv2D(256, (5, 5), strides=(2, 2), padding='same', use\_bias=False),

        BatchNormalization(momentum=0.5),

        LeakyReLU(alpha=0.2),

        Conv2D(512, (5, 5), strides=(2, 2), padding='same', use\_bias=False),

        BatchNormalization(momentum=0.5),

        LeakyReLU(alpha=0.2),

        # Output => 8 \* 8 \* 512

        Flatten(),

        Dense(1, activation='sigmoid')

    ])

    opt = Adam(learning\_rate=0.0002, beta\_1=0.5)

    model.compile(loss='binary\_crossentropy', optimizer=opt, metrics=None)

    return model

def generator():

    model = Sequential([

        Dense(4\*4\*512, input\_shape=(120,)),

        Reshape((4, 4, 512)),

        BatchNormalization(momentum=0.5),

        ReLU(),

        Conv2DTranspose(256, (5, 5), strides=(2, 2), padding='same', use\_bias=False),

        BatchNormalization(momentum=0.5),

        ReLU(),

        Conv2DTranspose(128, (5, 5), strides=(2, 2), padding='same', use\_bias=False),

        BatchNormalization(momentum=0.5),

        ReLU(),

        Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use\_bias=False),

        BatchNormalization(momentum=0.5),

        ReLU(),

        Conv2DTranspose(32, (5, 5), strides=(2, 2), padding='same', use\_bias=False),

        BatchNormalization(momentum=0.5),

        ReLU(),

        Conv2DTranspose(3, (5, 5), strides=(2, 2), padding='same', activation='tanh', use\_bias=False),

    ])

    opt = Adam(learning\_rate=0.00015, beta\_1=0.5)

    model.compile(loss='binary\_crossentropy', optimizer=opt, metrics=None)

    return model

def gan(gen\_model, disc\_model):

    disc\_model.trainable = False

    model = Sequential([

        gen\_model,

        disc\_model

    ])

    opt = Adam(learning\_rate=0.001, beta\_1=0.5)

    model.compile(loss='binary\_crossentropy', optimizer=opt)

    return model

def load\_dataset(directory="/content/Sneaker Dataset", target\_size=(128, 128)):

    images = []

    for filename in os.listdir(directory):

        img = load\_img(os.path.join(directory, filename), target\_size=target\_size)

        images.append(img\_to\_array(img))

    dataset = np.array(images)

    dataset = dataset.astype('float32')

    dataset /= 127.5

    dataset -= 1.0

    return dataset

def generate\_real\_samples(dataset, num\_samples):

    ix = np.random.randint(0, dataset.shape[0], num\_samples)

    X = dataset[ix]

    y = np.zeros((num\_samples, 1)) + 0.9  # One sided Label smoothing

    return X, y

def generate\_latent\_points(num\_samples): # gen model input

    x\_input = np.random.normal(0, 1, 120 \* num\_samples)

    x\_input = x\_input.reshape(num\_samples, 120)

    return x\_input

def generate\_fake\_samples(gen\_model, num\_samples): # gen model output

    x\_input = generate\_latent\_points(num\_samples)

    X = gen\_model.predict(x\_input)

    y = np.zeros((num\_samples, 1))

    return X, y

def save\_plot(examples, n):

    for i in range(n \* n):

        plt.subplot(n, n, 1 + i)

        plt.axis('off')

        examples[i] += 1

        examples[i] \*= 127.5

        plt.imshow(examples[i].astype(np.uint8), interpolation='nearest')

    plt.savefig("results.png")

    plt.show()

def train(gen\_model, disc\_model, gan\_model, dataset, epochs=500, batch\_size=64):

    num\_batches\_per\_epoch = int(dataset.shape[0] / batch\_size)

    d\_loss\_hist = []

    gan\_loss\_hist = []

    for i in range(epochs):

        for j in range(num\_batches\_per\_epoch):

            X\_real, y\_real = generate\_real\_samples(dataset, batch\_size // 2)

            X\_fake, y\_fake = generate\_fake\_samples(gen\_model, batch\_size // 2)

            disc\_model.trainable = True

            X, y = np.vstack((X\_real, X\_fake)), np.vstack((y\_real, y\_fake))

            d\_loss = disc\_model.train\_on\_batch(X, y)

            disc\_model.trainable = False

            X\_gan = generate\_latent\_points(batch\_size)

            y\_gan = np.ones((batch\_size, 1))

            gan\_loss = gan\_model.train\_on\_batch(X\_gan, y\_gan)

            print('>%d, %d/%d, d=%.3f, g=%.3f' % (i+1, j+1, num\_batches\_per\_epoch, d\_loss, gan\_loss))

        d\_loss\_hist.append(d\_loss)

        gan\_loss\_hist.append(gan\_loss)

        if (i+1) % 50 == 0:

            filename = 'generator\_model\_%03d.keras' % (i + 1)

            gen\_model.save(filename)

            latent\_points = generate\_latent\_points(25)

            y = gen\_model.predict(latent\_points)

            save\_plot(y, 5)

            plt.plot(d\_loss\_hist, label='disc\_loss')

            plt.plot(gan\_loss\_hist, label='gan\_loss')

            plt.legend()

            plt.savefig("Loss Plot.png")

disc\_model = discriminator()

gen\_model = generator()

gan\_model = gan(gen\_model, disc\_model)

dataset = load\_dataset()

train(gen\_model, disc\_model, gan\_model, dataset)

!mkdir -p "/content/drive/My Drive/CV Project/2nd train"

!mv "/content/generator\_model\_300.keras" "/content/drive/My Drive/CV Project/2nd train/generator\_model\_300.keras"

!mv "/content/generator\_model\_450.keras" "/content/drive/My Drive/CV Project/2nd train/generator\_model\_450.keras"

!mv "/content/generator\_model\_500.keras" "/content/drive/My Drive/CV Project/2nd train/generator\_model\_500.keras"

!mv "/content/results.png" "/content/drive/My Drive/CV Project/2nd train/results.png"

!mv "/content/Loss Plot.png" "/content/drive/My Drive/CV Project/2nd train/Loss Plot.png"

Wassersteins Generative Adversarial Network with Gradient Penalty (WGAN-GP)

import torch

from torch import nn

import torchvision

import torchvision.transforms as tfs

from torch.utils.data import DataLoader

from torchvision.datasets import ImageFolder

import matplotlib.pyplot as plt

from numpy import transpose, array

import torchvision.transforms.functional as TF

device = "cuda" if torch.cuda.is\_available() else "cpu"

device

class Critic(nn.Module):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        self.crit = nn.Sequential(

            nn.Conv2d(in\_channels=3, out\_channels=64, kernel\_size=4, stride=2, padding=1),

            nn.LeakyReLU(0.2),

            nn.Conv2d(in\_channels=64, out\_channels=128, kernel\_size=4, stride=2, padding=1, bias=False),

            nn.InstanceNorm2d(128, affine=True),

            nn.LeakyReLU(0.2),

            nn.Conv2d(in\_channels=128, out\_channels=256, kernel\_size=4, stride=2, padding=1, bias=False),

            nn.InstanceNorm2d(256, affine=True), # affine=True makes params learnable

            nn.LeakyReLU(0.2),

            nn.Conv2d(in\_channels=256, out\_channels=512, kernel\_size=4, stride=2, padding=1, bias=False),

            nn.InstanceNorm2d(512, affine=True),

            nn.LeakyReLU(0.2),

            # Output here - 16\*16\*512

            nn.Conv2d(in\_channels=512, out\_channels=1, kernel\_size=16, stride=1, padding=0)

        )

    def forward(self, x):

        return self.crit(x)

class Generator(nn.Module):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        self.gen = nn.Sequential(

           nn.ConvTranspose2d(in\_channels=64, out\_channels=512, kernel\_size=16, stride=1, padding=0),

           nn.BatchNorm2d(512),

           nn.ReLU(),

           nn.ConvTranspose2d(in\_channels=512, out\_channels=256, kernel\_size=4, stride=2, padding=1, bias=False),

           nn.BatchNorm2d(256),

           nn.ReLU(),

           nn.ConvTranspose2d(in\_channels=256, out\_channels=128, kernel\_size=4, stride=2, padding=1, bias=False),

           nn.BatchNorm2d(128),

           nn.ReLU(),

           nn.ConvTranspose2d(in\_channels=128, out\_channels=64, kernel\_size=4, stride=2, padding=1, bias=False),

           nn.BatchNorm2d(64),

           nn.ReLU(),

           nn.ConvTranspose2d(in\_channels=64, out\_channels=3, kernel\_size=4, stride=2, padding=1, bias=False),

           nn.Tanh()

        )

    def forward(self, x):

        return self.gen(x)

def initialize\_weights(model):

    # Initializes weights according to the DCGAN paper

    for m in model.modules():

        if isinstance(m, (nn.Conv2d, nn.ConvTranspose2d, nn.BatchNorm2d)):

            nn.init.normal\_(m.weight.data, 0.0, 0.02) # normal distribution with mean 0 and std dev 0.02

def gradient\_penalty(critic, real, fake, device="cpu"):

    BATCH\_SIZE, C, H, W = real.shape

    alpha = torch.rand((BATCH\_SIZE, 1, 1, 1)).repeat(1, C, H, W).to(device)

    interpolated\_images = real \* alpha + fake \* (1 - alpha)

    # Calculate critic scores

    mixed\_scores = critic(interpolated\_images)

    # Take the gradient of the scores with respect to the images

    gradient = torch.autograd.grad(

        inputs=interpolated\_images,

        outputs=mixed\_scores,

        grad\_outputs=torch.ones\_like(mixed\_scores),

        create\_graph=True,

        retain\_graph=True,

    )[0]

    gradient = gradient.view(gradient.shape[0], -1)

    gradient\_norm = gradient.norm(2, dim=1) # L2 norm

    gradient\_penalty = torch.mean((gradient\_norm - 1) \*\* 2)

    return gradient\_penalty

def plot\_generated\_images():

    noise = torch.randn(25, 64, 1, 1).to(device)

    fake = gen(noise)

    fake\_images = [(TF.to\_pil\_image((img \* 0.5) + 0.5)) for img in fake]

    print(array(fake\_images[5]))

    # Plot the images

    for i in range(25):

      plt.subplot(5, 5, 1 + i)

      plt.axis('off')

      plt.imshow(fake\_images[i])

    plt.savefig(f"/kaggle/working/results\_{epoch+1}.png")

    plt.show()

# gen = torch.load("../input/model-100-pt/model\_100.pt").to(device) # resume training

transform = tfs.Compose(

    [

        tfs.Resize(256),

        tfs.ToTensor(),

        tfs.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])

        # mean and std of 0.5 normalizes between [-1, 1]. applied to all 3 channels

    ]

)

# Load dataset

dataset = ImageFolder(root="../input/sneaker-dataset", transform=transform)

loader = DataLoader(dataset, batch\_size=64, shuffle=True)

print("Folder loaded")

# Initialize gen and crit

gen = Generator().to(device)

crit = Critic().to(device)

initialize\_weights(gen)

initialize\_weights(crit)

# Initialize optimizer

gen\_opt = torch.optim.Adam(gen.parameters(), lr=0.0001, betas=(0, 0.9))

crit\_opt = torch.optim.Adam(crit.parameters(), lr=0.0001, betas=(0, 0.9))

gen.train()

crit.train()

print("Training started")

for epoch in range(0, 300):

    for i, (images, \_) in enumerate(loader):

        real = images.to(device)

        batch\_size = real.size(0)

        # Train Critic: max E[critic(real)] - E[critic(fake)]

        for \_ in range(5):

            noise = torch.randn(batch\_size, 64, 1, 1).to(device)

            fake = gen(noise)

            critic\_real = crit(real).reshape(-1)

            critic\_fake = crit(fake).reshape(-1) # reshape(-1) will flatten the 1x1 conv2d output

            gp = gradient\_penalty(crit, real, fake, device)

            critic\_loss = (

                -(torch.mean(critic\_real) - torch.mean(critic\_fake)) + 10 \* gp

            )

            crit.zero\_grad()

            critic\_loss.backward(retain\_graph=True)

            crit\_opt.step()

        # Train Generator: max E[critic(gen\_fake)] <-> min -E[critic(gen\_fake)]

        gen\_fake = crit(fake).reshape(-1)

        gen\_loss = -torch.mean(gen\_fake)

        gen.zero\_grad()

        gen\_loss.backward()

        gen\_opt.step()

    print(f"Epoch [{epoch}/{300}] Loss D: {critic\_loss:.4f}, loss G: {gen\_loss:.4f}")

    if (epoch + 1) % 25 == 0:

      plot\_generated\_images()

      torch.save(gen, f"/kaggle/working/model\_{epoch+1}.pt")

Conditional Generative Adversarial Network (cGAN)

from google.colab import drive

drive.mount('/content/drive')

!unzip "/content/drive/My Drive/Sneaker Annotated Dataset.zip"

import torch

from torch import nn

import torchvision

import torchvision.transforms as tfs

from torch.utils.data import DataLoader

from torchvision.datasets import ImageFolder

import matplotlib.pyplot as plt

from numpy import transpose, array

import torchvision.transforms.functional as TF

z\_dim = 100

img\_dim = 256

num\_classes = 5

lr = 0.0002

beta1 = 0.5

beta2 = 0.999

weight\_decay = 1e-5

num\_epochs = 300

device = "cuda" if torch.cuda.is\_available() else "cpu"

device

class Discriminator(nn.Module):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        self.label\_embedding = nn.Embedding(num\_classes, img\_dim\*img\_dim)

        self.disc = nn.Sequential(

            nn.Conv2d(in\_channels=3+1, out\_channels=64, kernel\_size=4, stride=2, padding=1),

            nn.LeakyReLU(0.2),

            nn.Conv2d(in\_channels=64, out\_channels=128, kernel\_size=4, stride=2, padding=1, bias=False),

            nn.BatchNorm2d(128),

            nn.LeakyReLU(0.2),

            nn.Conv2d(in\_channels=128, out\_channels=256, kernel\_size=4, stride=2, padding=1, bias=False),

            nn.InstanceNorm2d(256),

            nn.LeakyReLU(0.2),

            nn.Conv2d(in\_channels=256, out\_channels=512, kernel\_size=4, stride=2, padding=1, bias=False),

            nn.InstanceNorm2d(512),

            nn.LeakyReLU(0.2),

            # Shape here - 16\*16\*512

            nn.Conv2d(in\_channels=512, out\_channels=1, kernel\_size=16, stride=1, padding=0),

            # nn.Sigmoid()

        )

    def forward(self, x, label):

        label\_embed = self.label\_embedding(label)

        label\_embed = label\_embed.view(-1, 1, img\_dim, img\_dim)

        x = torch.cat((x, label\_embed), dim=1)

        return self.disc(x)

class Generator(nn.Module):

    def \_\_init\_\_(self, label\_embed\_size=10):

        super().\_\_init\_\_()

        self.label\_embedding = nn.Embedding(num\_classes, label\_embed\_size)

        self.gen = nn.Sequential(

            nn.ConvTranspose2d(in\_channels=z\_dim+label\_embed\_size, out\_channels=512, kernel\_size=16, stride=1, padding=0),

            nn.BatchNorm2d(512),

            nn.ReLU(),

            nn.ConvTranspose2d(in\_channels=512, out\_channels=256, kernel\_size=4, stride=2, padding=1, bias=False),

            nn.BatchNorm2d(256),

            nn.ReLU(),

            nn.ConvTranspose2d(in\_channels=256, out\_channels=128, kernel\_size=4, stride=2, padding=1, bias=False),

            nn.BatchNorm2d(128),

            nn.ReLU(),

            nn.ConvTranspose2d(in\_channels=128, out\_channels=64, kernel\_size=4, stride=2, padding=1, bias=False),

            nn.BatchNorm2d(64),

            nn.ReLU(),

            nn.ConvTranspose2d(in\_channels=64, out\_channels=3, kernel\_size=4, stride=2, padding=1, bias=False),

            nn.Tanh()

        )

    def forward(self, x, label):

        label\_embed = self.label\_embedding(label)

        label\_embed= label\_embed.view(label\_embed.shape[0], -1, 1, 1) # (batch\_size, z\_dim, 1, 1), concat along dim=1

        x = torch.cat((x, label\_embed), dim=1)

        return self.gen(x)

def initialize\_weights(model):

    # Initializes weights according to the DCGAN paper

    for m in model.modules():

        if isinstance(m, (nn.Conv2d, nn.ConvTranspose2d, nn.BatchNorm2d)):

            nn.init.normal\_(m.weight.data, 0.0, 0.02) # normal distribution with mean 0 and std dev 0.02

def plot\_generated\_images():

    noise = torch.randn(25, z\_dim, 1, 1).to(device)

    labels = torch.randint(0, num\_classes, (25,), device=device)

    print(labels)

    fake = gen(noise, labels)

    # Convert tensor to PIL images and renormalize

    fake\_images = [(TF.to\_pil\_image((img \* 0.5) + 0.5)) for img in fake]

    # Plot the images

    for i in range(25):

      plt.subplot(5, 5, 1 + i)

      plt.axis('off')

      plt.imshow(fake\_images[i])

    plt.savefig(f"/content/drive/My Drive/Sneaker cGAN/results\_{epoch+1}.png")

    plt.show()

transform = tfs.Compose(

    [

        tfs.Resize(img\_dim),

        tfs.ToTensor(),

        tfs.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])

        # mean and std of 0.5 normalizes between [-1, 1]. applied to all 3 channels

    ]

)

# Load dataset

dataset = ImageFolder(root="/content/Sneaker Annotated Dataset", transform=transform)

loader = DataLoader(dataset, batch\_size=256, shuffle=True)

print("Folder loaded")

# Initialize gen and disc

gen = Generator(label\_embed\_size=10).to(device)

disc = Discriminator().to(device)

initialize\_weights(gen)

initialize\_weights(disc)

# Initialize optimizer

gen\_opt = torch.optim.Adam(gen.parameters(), lr=lr, betas=(beta1, beta2), weight\_decay=weight\_decay)

disc\_opt = torch.optim.Adam(disc.parameters(), lr=lr, betas=(beta1, beta2), weight\_decay=weight\_decay)

# Loss functions

loss\_fn = nn.BCEWithLogitsLoss() # combination of sigmoid and BCE loss, more numerically stable

gen.train()

disc.train()

print("Training started")

for epoch in range(num\_epochs):

    for i, (images, labels) in enumerate(loader):

        real = images.to(device)

        batch\_size = real.size(0)

        noise = torch.randn(batch\_size, z\_dim, 1, 1).to(device)

        fake = gen(noise, labels.to(device))

        disc\_real = disc(real.detach(), labels.to(device))

        disc\_fake = disc(fake.detach(), labels.to(device))

        disc\_loss = (loss\_fn(disc\_real, torch.ones\_like(disc\_real)) + loss\_fn(disc\_fake, torch.zeros\_like(disc\_fake))) / 2

        disc\_opt.zero\_grad()

        disc\_loss.backward()

        disc\_opt.step()

        disc\_fake = disc(fake, labels.to(device))

        gen\_loss = loss\_fn(disc\_fake, torch.ones\_like(disc\_fake))

        gen\_opt.zero\_grad()

        gen\_loss.backward()

        gen\_opt.step()

    print(f"Epoch [{epoch+1}/{num\_epochs}] Loss D: {disc\_loss:.4f} Loss G: {gen\_loss:.4f}")

    if (epoch + 1) % 50 == 0:

        plot\_generated\_images()

        torch.save(gen, f"/content/drive/My Drive/Sneaker cGAN/model\_{epoch+1}.pt")

Streamlit Deployment Script

import streamlit as st

st.title("Generative Adversarial Networks")

model\_option = st.selectbox(

    label="Select a model",

    options=["Deep Convolutional GAN", "Wasserstein GAN with GP", "Conditional GAN"],

    index=None,

    placeholder="Select a model..."

)

if model\_option == "Conditional GAN":

    from test.cgan import Generator

    from test.cgan import plot\_generated\_images

    label\_option = st.selectbox(

        label="Select a color",

        options=["Black", "Blue", "Red", "Yellow", "White"],

        index=None,

        placeholder="Select a color.."

    )

    label\_mapping = {

        "Black": 0,

        "Blue": 1,

        "Red": 2,

        "Yellow": 3,

        "White": 4

    }

    container = st.container()

    cols = container.columns(5)

    if label\_option:

        for i in range(10):

            img = plot\_generated\_images(label\_mapping[label\_option])

            cols[i % 5].image(img, caption=f"Generated Image {i + 1}", width=100)

elif model\_option == "Wasserstein GAN with GP":

    from test.wgangp import Generator

    from test.wgangp import plot\_generated\_images

    container = st.container()

    cols = container.columns(5)

    for i in range(10):

        img = plot\_generated\_images()

        cols[i % 5].image(img, caption=f"Generated Image {i + 1}", width=100)

elif model\_option == "Deep Convolutional GAN":

    from test.dcgan import generate\_single\_image

    container = st.container()

    cols = container.columns(5)

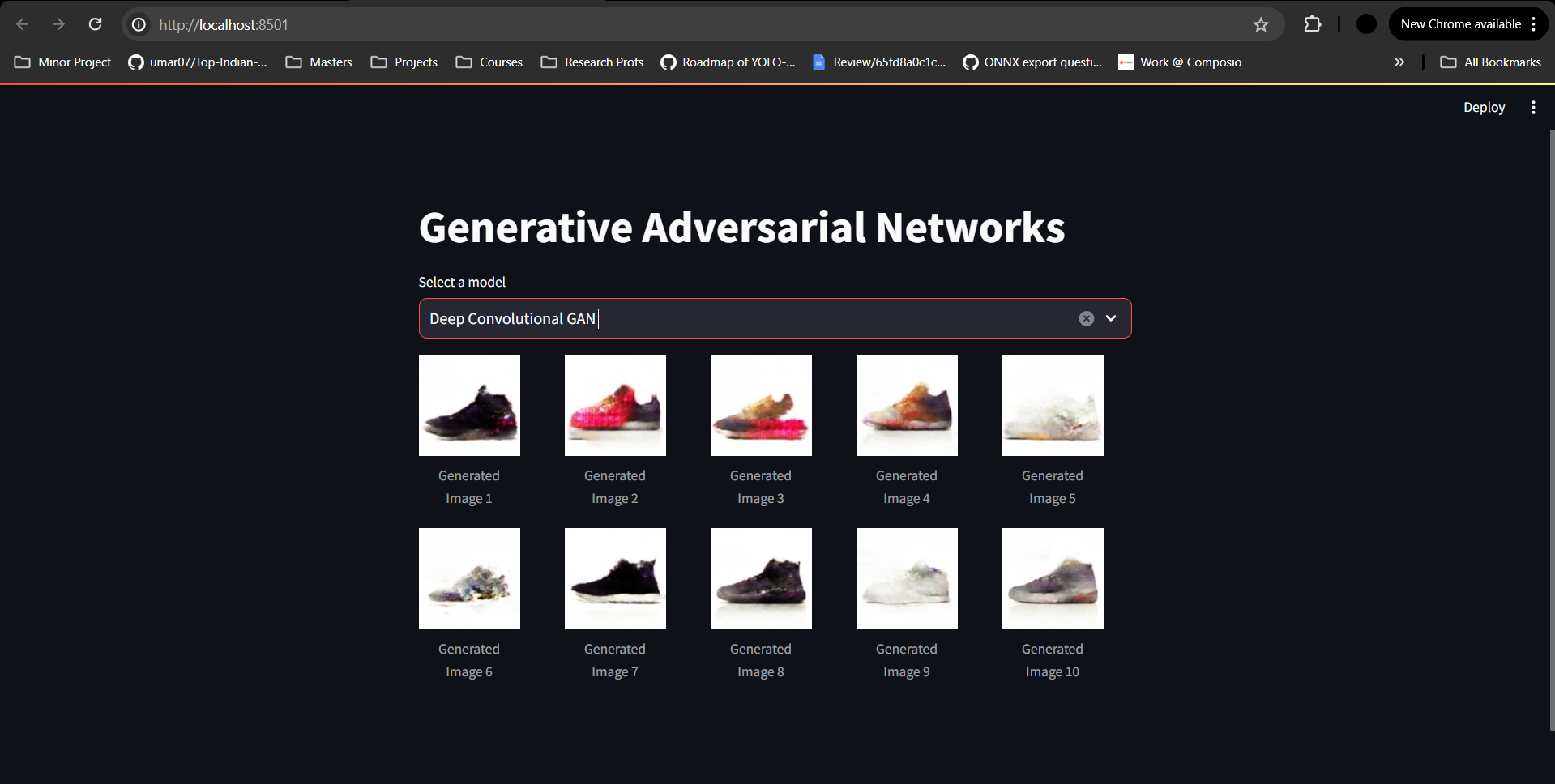
    for i in range(10):

        img = generate\_single\_image()

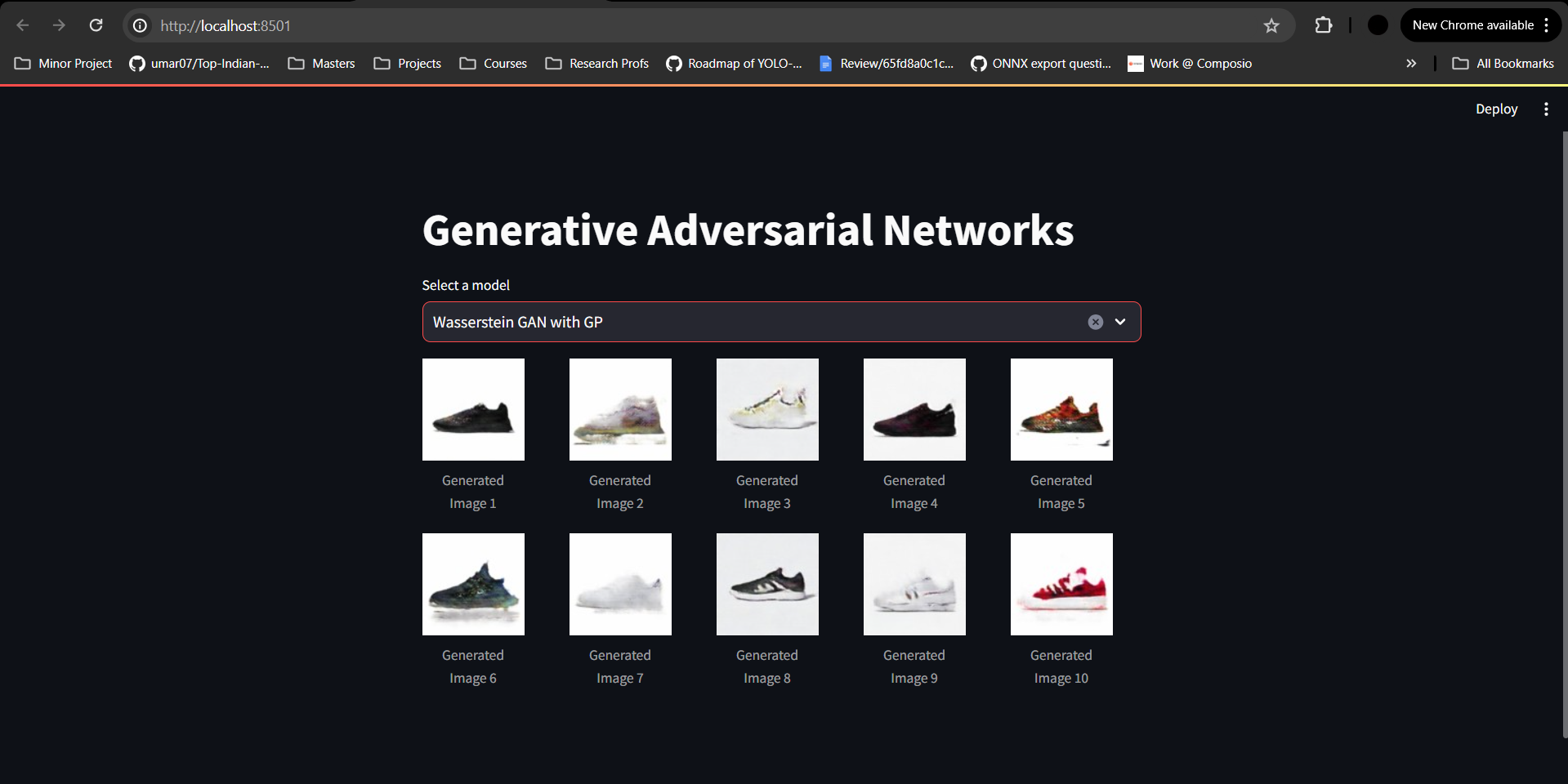
        cols[i % 5].image(img, caption=f"Generated Image {i + 1}", width=100)

1. Output

DCGAN



WGAN-GP



cGAN

