```
# Lab 12: Deep Convolutional GAN (DCGAN) for Color Image Generation
# ================================================================

import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import torchvision.utils as vutils
import matplotlib.pyplot as plt
import numpy as np

# ----------------------------
# 1. Hyperparameters
# ----------------------------
batch_size = 128
image_size = 32
nz = 100            # latent vector size (noise)
ngf = 64            # generator feature maps
```

```
ndf = 64            # discriminator feature maps
nc = 3              # RGB
lr = 0.0002
beta1 = 0.5
epochs = 5
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# ----------------------------
# 2. Dataset
# ----------------------------
transform = transforms.Compose([
    transforms.Resize(image_size),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

dataset = datasets.CIFAR10(root="./data", download=True, transform=transform)
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

# ----------------------------
```

```
# ----------------------------
# 3. Generator
# ----------------------------
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.main = nn.Sequential(
            nn.ConvTranspose2d(nz, ngf*8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf*8),
            nn.ReLU(True),
            nn.ConvTranspose2d(ngf*8, ngf*4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf*4),
            nn.ReLU(True),
            nn.ConvTranspose2d(ngf*4, ngf*2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf*2),
            nn.ReLU(True),
            nn.ConvTranspose2d(ngf*2, ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            nn.ConvTranspose2d(ngf, nc, 3, 1, 1, bias=False),
```

```
    def forward(self, input):
        return self.main(input)

# ----------------------------
# 4. Discriminator
# ----------------------------
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.main = nn.Sequential(
            nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(ndf, ndf*2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf*2),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(ndf*2, ndf*4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf*4),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(ndf*4, 1, 4, 1, 0, bias=False),
```

```
            nn.Sigmoid()
        )

    def forward(self, input):
        return self.main(input).view(-1)

# ----------------------------
# 5. Initialize models and optimizers
# ----------------------------
netG = Generator().to(device)
netD = Discriminator().to(device)

criterion = nn.BCELoss()
optimizerD = optim.Adam(netD.parameters(), lr=lr, betas=(beta1, 0.999))
optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(beta1, 0.999))

fixed_noise = torch.randn(64, nz, 1, 1, device=device)

# ----------------------------
# 6. Training with Loss Tracking
```

https://colab.research.google.com/drive/1n1XghI5JqaDe3R2jFQdd5Sn9MUNRvhMf#scrollTo=z_4zjquYa71U

9.ipynb

File Edit View Insert Runtime Tools Help

Share

Commands + Code + Text Run all Reconnect

```
G_losses = []
D_losses = []

for epoch in range(epochs):
    for i, (data, _) in enumerate(dataloader):
        real = data.to(device)
        b_size = real.size(0)
        label_real = torch.full((b_size,), 1.0, device=device)
        label_fake = torch.full((b_size,), 0.0, device=device)

        # Train Discriminator
        netD.zero_grad()
        output_real = netD(real)
        lossD_real = criterion(output_real, label_real)

        noise = torch.randn(b_size, nz, 1, 1, device=device)
        fake = netG(noise)
        output_fake = netD(fake.detach())
        lossD_fake = criterion(output_fake, label_fake)
        lossD = lossD_real + lossD_fake
```

Variables Terminal

9:19 AM

ENG IN 14:54 03-11-2025

https://colab.research.google.com/drive/1n1XghI5JqaDe3R2jFQdd5Sn9MUNRvhMf#scrollTo=z_4zjquYa71U

9.ipynb

File Edit View Insert Runtime Tools Help

Share

Commands + Code + Text Run all Reconnect

```
            lossD = lossD_real + lossD_fake
            lossD.backward()
            optimizerD.step()

            # Train Generator
            netG.zero_grad()
            output = netD(fake)
            lossG = criterion(output, label_real)
            lossG.backward()
            optimizerG.step()

        # Save losses for graph
        D_losses.append(lossD.item())
        G_losses.append(lossG.item())

        print(f"Epoch [{epoch+1}/{epochs}]  Loss_D: {lossD.item():.4f}  Loss_G: {lossG.item():.4f}")

# ----------------------------
# 7. Plot Loss Graph
# ----------------------------
```

Variables Terminal

9:19 AM

ENG IN 14:55 03-11-2025

https://colab.research.google.com/drive/1n1XghI5JqaDe3R2jFQdd5Sn9MUNRvhMf#scrollTo=z_4zjquYa71U

9.ipynb

File Edit View Insert Runtime Tools Help

Share

Commands + Code + Text Run all Reconnect

```
plt.figure(figsize=(8,5))
plt.plot(G_losses, label="Generator Loss (G)", marker='o')
plt.plot(D_losses, label="Discriminator Loss (D)", marker='s')
plt.title("Generator vs Discriminator Loss During Training")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.grid(True)
plt.show()

# ----------------------------
# 8. Show Final Generated Images
# ----------------------------
with torch.no_grad():
    fake = netG(fixed_noise).detach().cpu()
grid = vutils.make_grid(fake, padding=2, normalize=True)
plt.figure(figsize=(8,8))
plt.axis("off")
plt.title("Generated Images (After Final Epoch)")
```

Variables Terminal

9:19 AM

ENG IN 14:55 03-11-2025

https://colab.research.google.com/drive/1n1XghI5JqaDe3R2jFQdd5Sn9MUNRvhMf#scrollTo=z_4zjquYa71U

9.ipynb

File Edit View Insert Runtime Tools Help

Share

Commands + Code + Text Run all Reconnect

```
# ----------------------------
# 8. Show Final Generated Images
# ----------------------------
with torch.no_grad():
    fake = netG(fixed_noise).detach().cpu()
grid = vutils.make_grid(fake, padding=2, normalize=True)
plt.figure(figsize=(8,8))
plt.axis("off")
plt.title("Generated Images (After Final Epoch)")
plt.imshow(np.transpose(grid, (1,2,0)))
plt.show()

print("Training Complete ✅")
```

```
Epoch [1/5]  Loss_D: 0.5758  Loss_G: 3.0092
Epoch [2/5]  Loss_D: 1.0516  Loss_G: 1.7795
Epoch [3/5]  Loss_D: 0.9452  Loss_G: 2.0873
Epoch [4/5]  Loss_D: 1.2850  Loss_G: 3.5633
```

Variables Terminal

9:19 AM

ENG IN 14:55 03-11-2025

https://colab.research.google.com/drive/1n1XghI5JqaDe3R2jFQdd5Sn9MUNRvhMf#scrollTo=z_4zjquYa71U

9.ipynb

File Edit View Insert Runtime Tools Help

Share

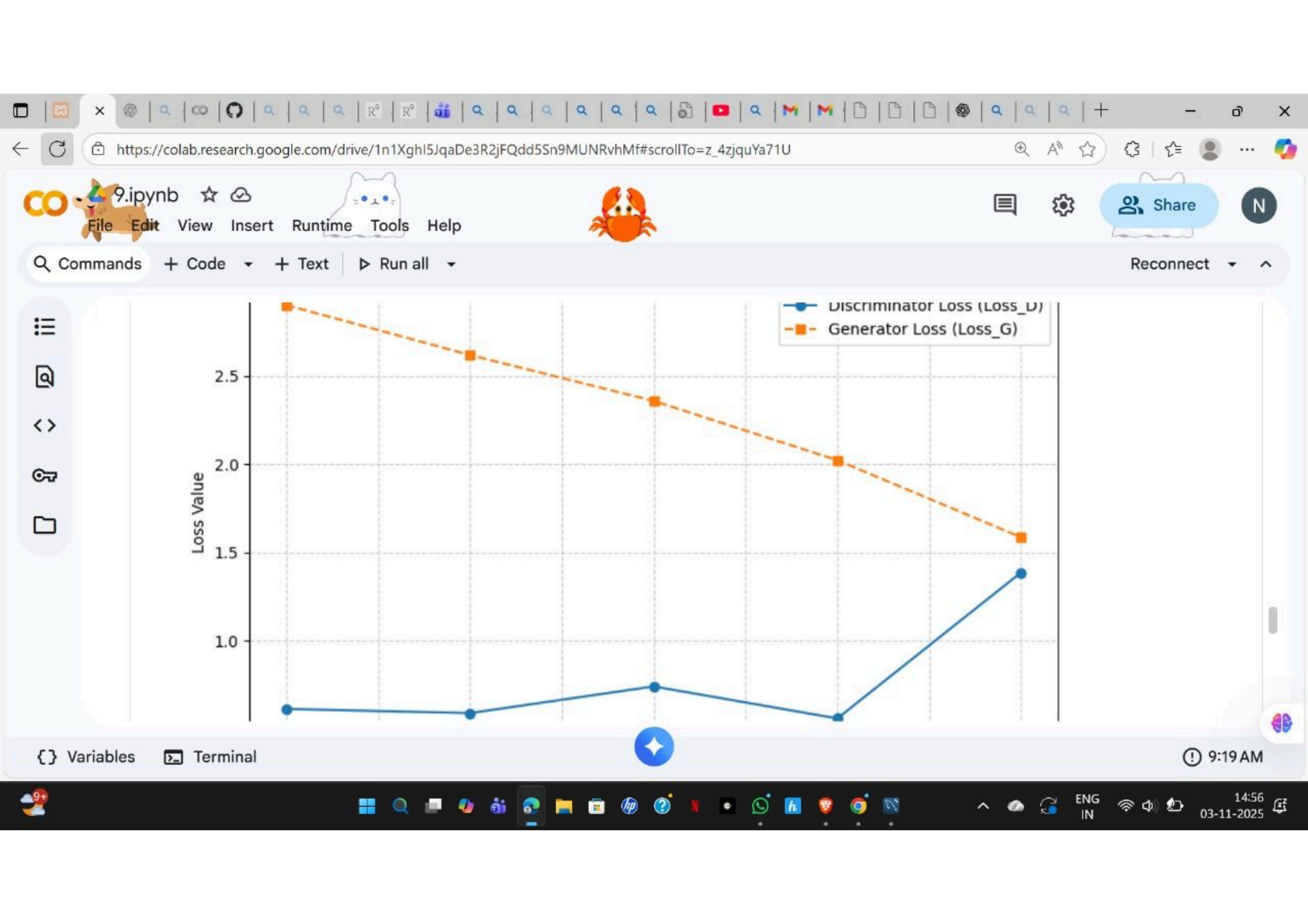
Commands + Code + Text Run all Reconnect

```
epochs = [1, 2, 3, 4, 5]
loss_d = [0.6131, 0.5894, 0.7406, 0.5590, 1.3862]
loss_g = [2.9012, 2.6182, 2.3580, 2.0222, 1.5855]

# Plot losses
plt.figure(figsize=(8,5))
plt.plot(epochs, loss_d, marker='o', linestyle='-', label='Discriminator Loss (Loss_D)')
plt.plot(epochs, loss_g, marker='s', linestyle='--', label='Generator Loss (Loss_G)')

# Labels and title
plt.title("GAN Training Progress")
plt.xlabel("Epoch")
plt.ylabel("Loss Value")
plt.legend()
plt.grid(True, linestyle='--', alpha=0.6)
plt.tight_layout()
plt.show()
```

Variables Terminal

9:19 AM

ENG IN 14:55 03-11-2025


https://colab.research.google.com/drive/1n1XghI5JqaDe3R2jFQdd5Sn9MUNRvhMf#scrollTo=z_4zjquYa71U
9.ipynb
File Edit View Insert Runtime Tools Help
Share
N
Commands
+ Code
+ Text
Run all
Reconnect
Discriminator Loss (Loss_D)
Generator Loss (Loss_G)
Loss Value
2.5
2.0
1.5
1.0
Variables
Terminal
9:19 AM
ENG
IN
14:56
03-11-2025

27/10/25 12. Implement a Deep Convolutional GAN to generate Complex Color Images

Aim:

To implement deep Convolutional Generative Adversial Network (DCGAN) that generates Complex Color images from random noise.

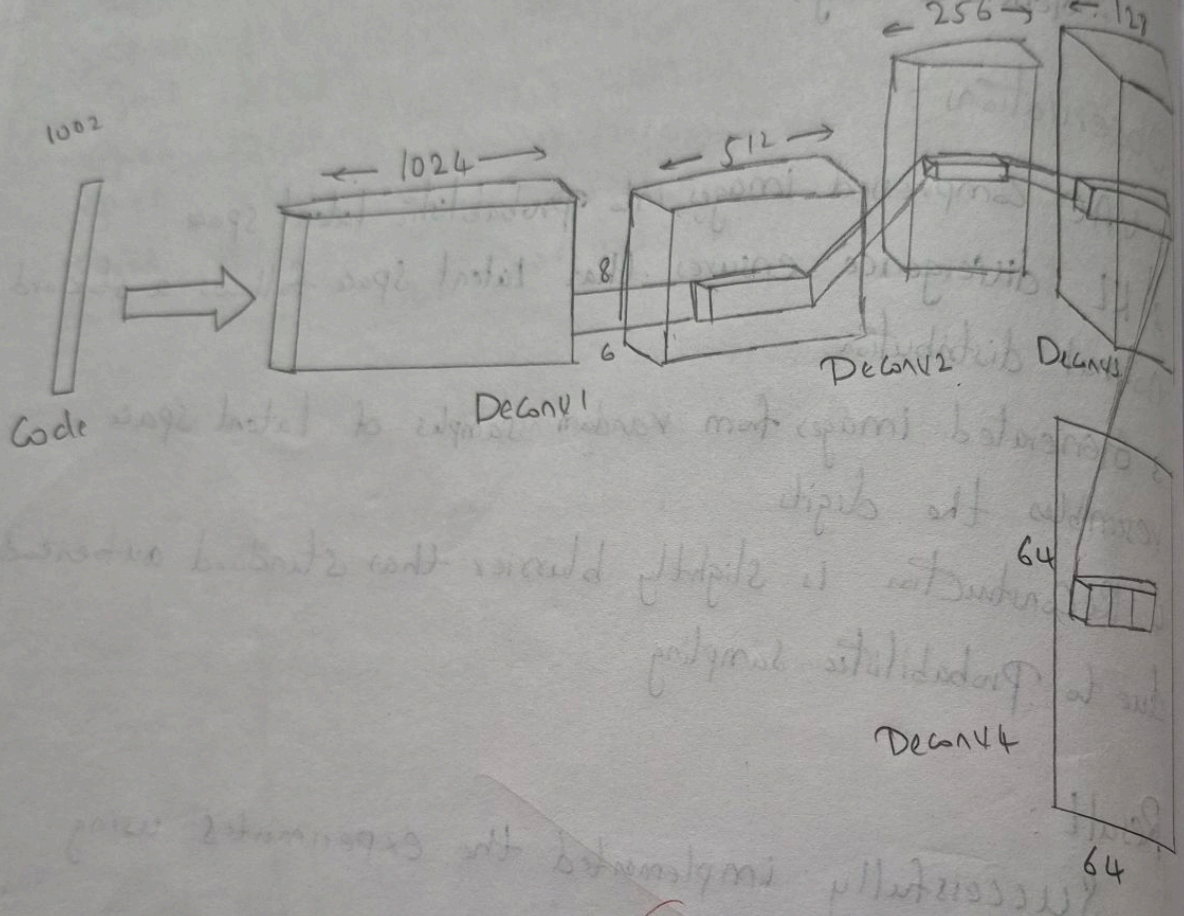
Objective:

- To understand the working of GANs
- To implement Convolutional and transposed Convolutional layers for image generation
- To train generator & discriminator networks to produce realistic color images.
- To visualize & evaluate the quality of generated images during & after training.

PseudoCode:

1. Import necessary libraries
2. Load & preprocess dataset (CIFAR-10)
   - Normalize image pixels to range [-1, 1]
3. Define Generator network:
   - Input: random noise vector (z)
   - layers: Dense → Reshape → Conv2DTranspose2D → BatchNorm → ReLU
   - output: 3-channel Color image with tanh activation
4. Define discriminator network:
   - Input: image (real / fake)
   - layers: Conv2D → LeakyReLU → Dropout
   - output: Probability (real / fake) using sigmoid activation

DCGAN architecture



27/10/25 12. Implement

Aim:

To ir

Adversial

images fro

Objective:

- To unders
- To implem layers for
- To train g realistic co
- To visualiz & after tr

PseudoCod

1. Import n
2. Load & p
   - No
3. Define G
   - Inpu
   - laye
   - outp
4. Define d
   - Inpu
   - lay
   - Ou

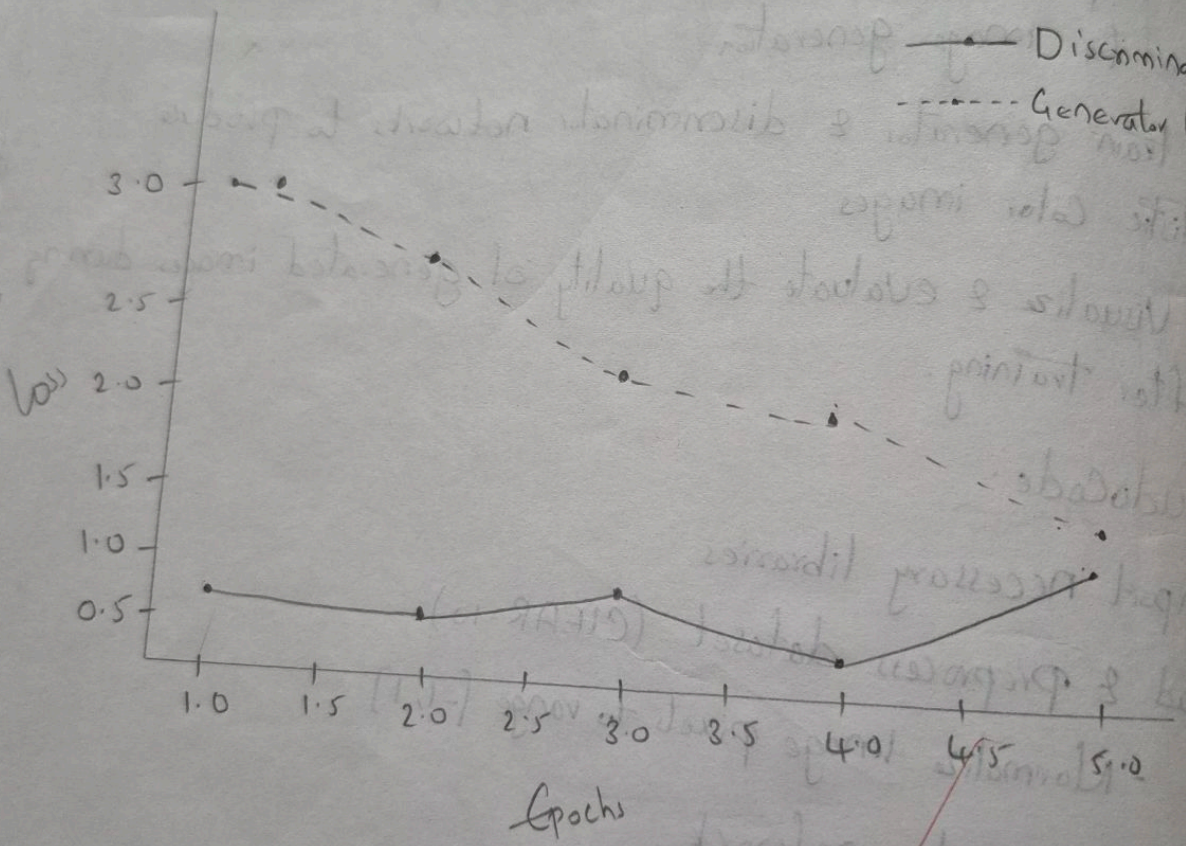
Epoch [1/5] Loss-D: 0.6131 Loss-G: 2.9012

Epoch [2/5] Loss-D: 0.5894 Loss-G: 2.6182

Epoch [3/5] Loss-D: 0.7406 Loss-G: 2.3580

Epoch [4/5] Loss-D: 0.5890 Loss-G: 2.0222

Epoch [5/5] Loss-D: 1.3862 Loss-G: 1.5855



5. Define loss
- Gene
- Dis
6. Initialize
7. For each
.
.
.
8. Display
9. End tra

Observa
. The dis
. The ge
Colorful
. After
& gener

Result
S
Complex

5. Define loss functions:
   - Generator loss: encourage $D(G(z)) \to 1$
   - Discriminator loss: distinguish real vs fake correctly
6. Initialize optimizers
7. For each training epoch:
   - Train Discriminator on real image (label = 1), fake image (label = 0)
   - Train Generator to fool the discriminator (label = 1)
   - Save sample generated image every few epochs
8. Display loss curves & generated image samples
9. End training when generated images become visually realistic

Observation:

- The discriminator initially learns faster & easily detects fake images.
- The generator improves gradually, producing sharper & more colorful images over epochs.
- After several epochs, both networks reach a balanced state, & generated images closely resemble real samples.

Result:

Successfully implemented a deep Convolutional GAN to generate Complex Color images.

```
# Lab 13: Understanding the Architecture of Pre-trained Model

import os
import urllib.request
import torch
import torchvision.models as models
import torchvision.transforms as transforms
from PIL import Image
import matplotlib.pyplot as plt

# ---------------------------
# 1. Download sample image (with fallback)
# ---------------------------
img_path = "sample.jpg"

def download_image():
    try:
        # Free-to-download image (always works)
```

```
# Count parameters
total_params = sum(p.numel() for p in model.parameters())
trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"\nTotal Parameters: {total_params:,}")
print(f"Trainable Parameters: {trainable_params:,}")

# --------------------------
# 3. Load and Preprocess Image
# --------------------------
transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])

img = Image.open(img_path).convert('RGB')
input_tensor = transform(img).unsqueeze(0)
```

```
# Display input image
plt.imshow(img)
plt.title("Input Image")
plt.axis("off")
plt.show()

# ---------------------------
# 4. Forward Pass & Extract Features
# ---------------------------
activation = {}

def get_activation(name):
    def hook(model, input, output):
        activation[name] = output.detach()
    return hook

# Register hooks
model.layer1.register_forward_hook(get_activation('layer1'))
model.layer2.register_forward_hook(get_activation('layer2'))
```

Variables Terminal 3:15 PM Python 3

```
model.layer3.register_forward_hook(get_activation('layer3'))
model.layer4.register_forward_hook(get_activation('layer4'))

# Forward pass
with torch.no_grad():
    output = model(input_tensor)

# Print feature map shapes
print("\n--- Intermediate Feature Map Shapes ---")
for name, act in activation.items():
    print(f"{name}: {tuple(act.shape)}")

# --------------------------
# 5. Visualize Feature Maps
# --------------------------
layer_name = 'layer1'
feature_map = activation[layer_name].squeeze(0)

plt.figure(figsize=(12, 6))
for i in range(6):
```

```
        plt.subplot(2, 3, i + 1)
        fm = feature_map[i].cpu()
        fm = (fm - fm.min()) / (fm.max() - fm.min())
        plt.imshow(fm, cmap='viridis')
        plt.axis('off')
        plt.title(f'{layer_name} - Channel {i}')
plt.tight_layout()
plt.show()

# --------------------------
# 6. Show Top-5 Predictions
# --------------------------
probabilities = torch.nn.functional.softmax(output[0], dim=0)
top5_prob, top5_catid = torch.topk(probabilities, 5)

# Load ImageNet labels if available
try:
    with open("imagenet_classes.txt") as f:
        categories = [line.strip() for line in f.readlines()]
```

Variables Terminal 3:15 PM Python 3

```
    categories = [f"Class {i}" for i in range(1000)]

print("\n--- Top 5 Predictions ---")
for i in range(5):
    print(f"{categories[top5_catid[i]]}: {top5_prob[i].item()*100:.2f}%")
```

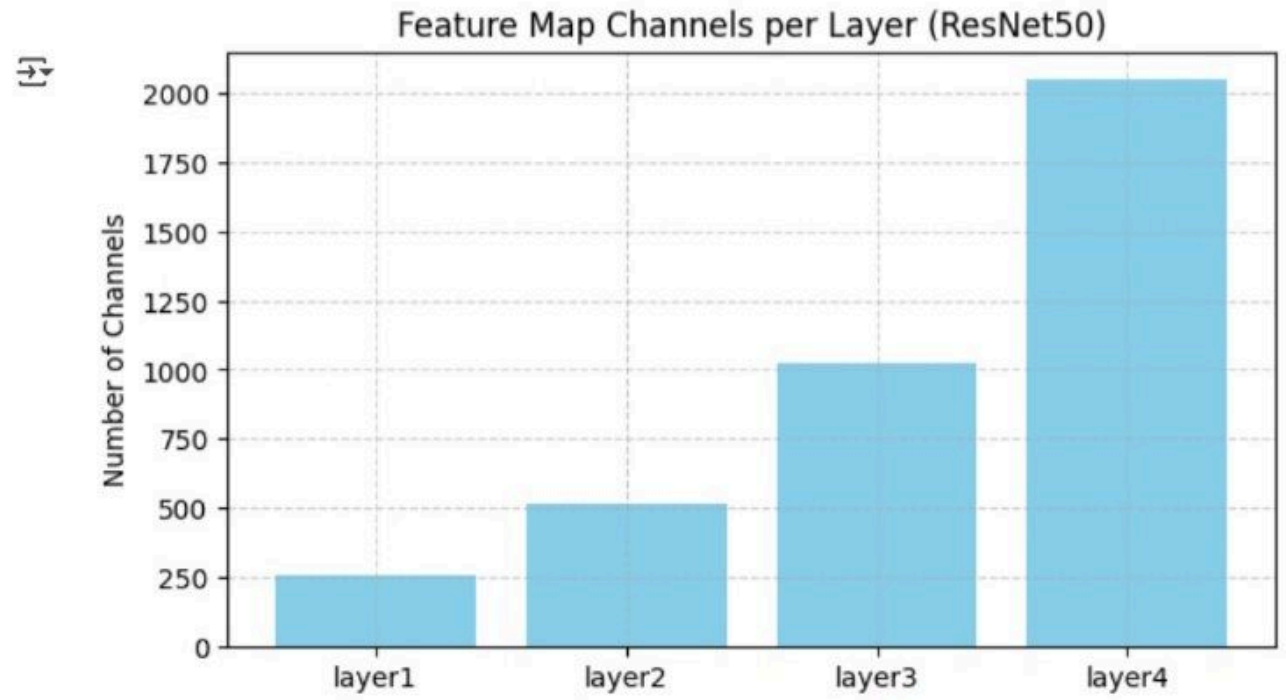
```
✅ Downloaded sample.jpg successfully!

/usr/local/lib/python3.12/dist-packages/torchvision/models/_utils.py:208: UserWarning: The parameter 'pretrained' is deprec
  warnings.warn(
/usr/local/lib/python3.12/dist-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum o
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/resnet50-0676ba61.pth" to /root/.cache/torch/hub/checkpoints/resnet50-067
100%|██████████| 97.8M/97.8M [00:00<00:00, 157MB/s]

--- Model Architecture ---

ResNet(
```

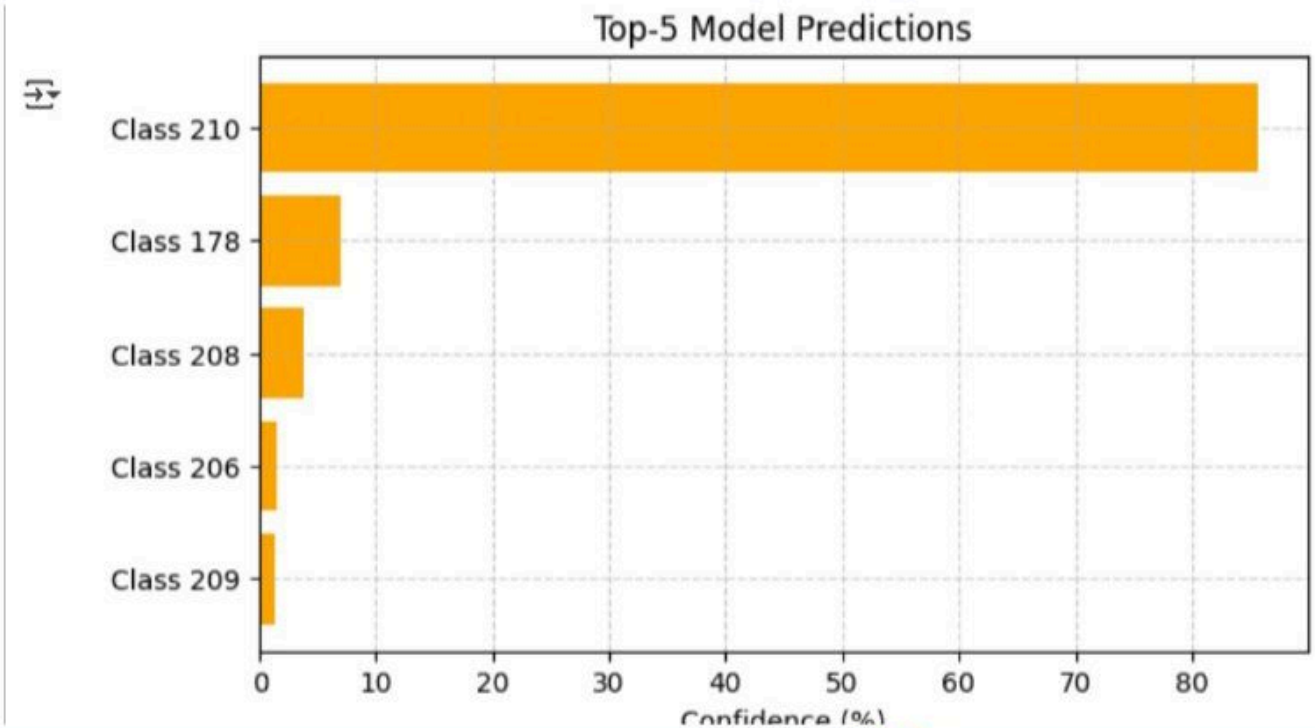
```
--- Model Architecture ---

ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
```

```
    (0): Bottleneck(
      (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

```
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (layer3): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
```

```
[ ]
import matplotlib.pyplot as plt

layer_names = ['layer1', 'layer2', 'layer3', 'layer4']
channels = [256, 512, 1024, 2048]

plt.figure(figsize=(7,4))
plt.bar(layer_names, channels, color='skyblue')
plt.title('Feature Map Channels per Layer (ResNet50)')
plt.xlabel('Layer')
plt.ylabel('Number of Channels')
plt.grid(True, linestyle='--', alpha=0.6)
plt.show()
classes = ['Class 210', 'Class 178', 'Class 208', 'Class 206', 'Class 209']
confidence = [85.62, 6.93, 3.82, 1.46, 1.38]

plt.figure(figsize=(7,4))
plt.barh(classes, confidence, color='orange')
plt.xlabel('Confidence (%)')
plt.title('Top-5 Model Predictions')
```


Feature Map Channels per Layer (ResNet50)
Number of Channels
2000
1750
1500
1250
1000
750
500
250
0
layer1
layer2
layer3
layer4


Top-5 Model Predictions
Class 210
Class 178
Class 208
Class 206
Class 209
0
10
20
30
40
50
60
70
80
Confidence (%)

27/10/25

13. Understanding the architecture of Pre-trained Model

Aim:

To explore & understand the architecture of Pre-trained deep learning model used for image classificato.

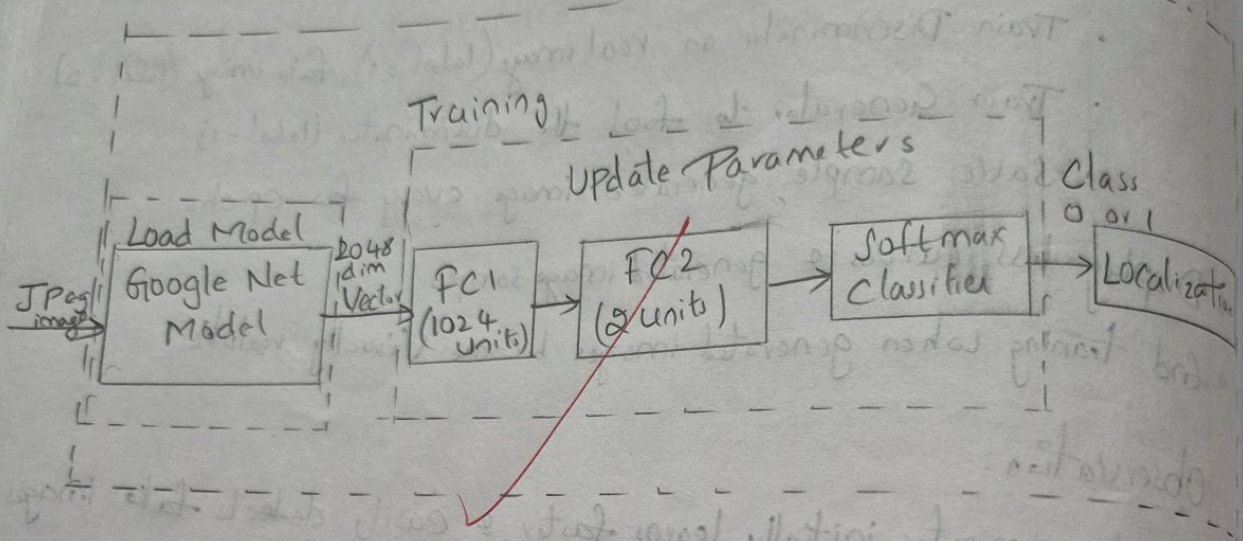
Objective:

- To load & Visualize a pre-trained model from a deep learning library.
- To analyze the layer wise structure, Parameters and activation functions.
- To understand the Concept of transfer learning using pre-trained weights.
- To visualize feature extraction from intermediate layers.

PseudoCode:

1. Import necessary libraries
2. Load Pretrained model with imagenet weights
3. Display model summary
4. Observe:
   - Input & output shapes
   - layer names, types and number of Parameters
5. Visualise the architecture using:
   Plot_model(model, show_shapes = True)
6. Load a sample input image & preprocess it for the model
7. Pass the image through the model & get predictions

Architecture of Pretrained Model



27/10/25 13. Und

Aim:

To

Pre-trained

Objective

- To load
learning li
- To analy
activation
- To under
weights.
- To visual

PseudoCo

1. Import
2. Load Pre
3. Display
4. Observe:
-
-
5. Visualise
6. Load a
7. Pass th

(Linear (in_features = 2048, out_features = 10, bias = True))

Output

Total Parameters : 25,557,032
Trainable Parameters : 25,557,032

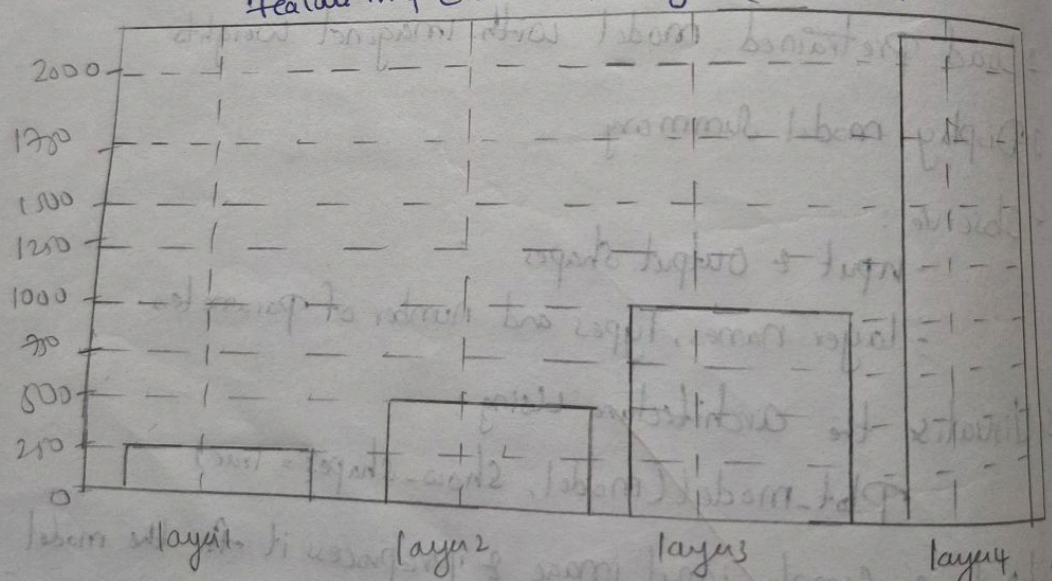
Intermediate feature map shapes

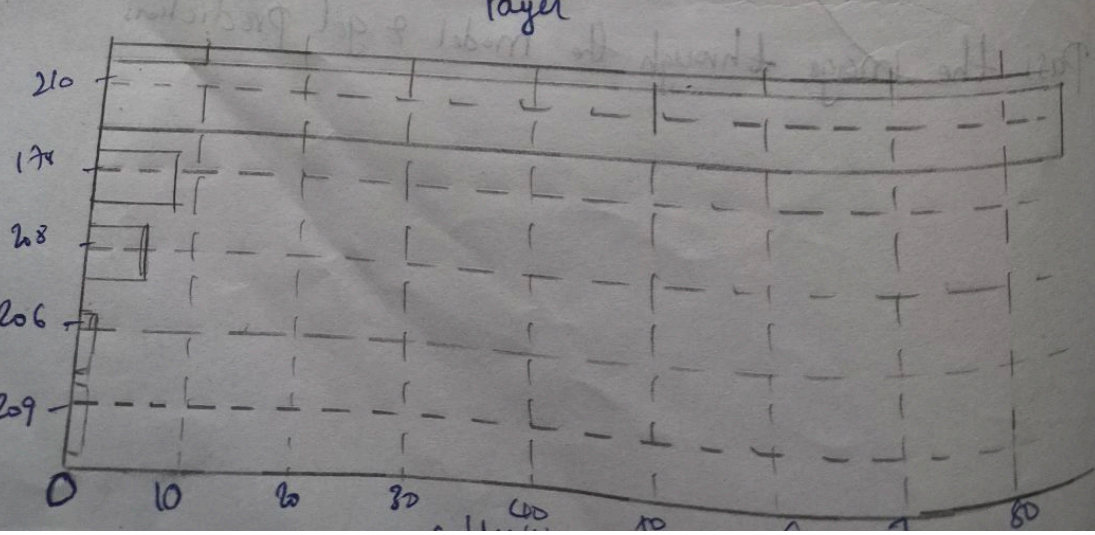
layer1 : (1, 256, 56, 56)
layer2 : (1, 512, 28, 28)
layer3 : (1, 1024, 14, 14)
layer4 : (1, 2048, 7, 7)

Top 5 Predictions

Class 210 : 85.62%
Class 178 : 6.93%
Class 208 : 3.82%
Class 206 : 1.46%
Class 209 : 1.38%

Feature map channels per layer (ResNet-50)





Observation:

- The pre-trained
across convolution
- Early layers extra
high-level featur
- The use of Pr
improve model

Result:

Succ

Pre-trained mo

Observation:

- The pre-trained model contains millions of parameters spread across convolutional & dense layers
- Early layers extract edges & textures, while deeper layers capture high-level features such as object parts.
- The use of pre-trained weights drastically reduces training time & improves model accuracy for new tasks.

Result:

Successfully explored and understood the architecture of pre-trained model.