



limhpone / computervision-final-prep



<> Code

Issues

Pull requests

Actions

Projects

Wiki

Security

In

computervision-final-prep / lab / finalexam-practice-yolandav3_answered_full_asus.ipynb



limhpone practice

73c3945 · now



1695 lines (1695 loc) · 1.87 MB

Practice

Question 1: Object Detection (10 points)

We've explored the evolution of the R-CNN family, encompassing R-CNN, Fast R-CNN, and Faster R-CNN. Analyze the key differences between these models, identifying their respective limitations and the specific challenges they were developed to overcome.

Answer: Analysis of R-CNN Family Evolution

1. R-CNN (Regions with CNN features)

Architecture:

- Uses selective search to generate approximately 2,000 region proposals per image
- Each region is warped to a fixed size (227×227) and passed through a CNN (AlexNet) for feature extraction
- Extracted features are classified using class-specific linear SVMs
- Bounding box regression is applied to refine object locations

Limitations:

- **Extremely slow:** Each region proposal requires a separate forward pass through the CNN, making training take 84 hours on a GPU and testing 47 seconds per image
 - **Multi-stage pipeline:** Requires training CNN, SVMs, and bounding box regressors separately
 - **High storage requirements:** Features from all region proposals must be saved to disk
 - **No end-to-end training:** Different components cannot be jointly optimized
-

2. Fast R-CNN

Architecture:

- Processes the entire image through a CNN to generate a convolutional feature map
- Uses selective search for region proposals, but projects them onto the feature map using RoI (Region of Interest) pooling
- Replaces SVMs with a softmax classifier integrated into the network

- replaces SVMs with a softmax classifier integrated into the network

- Combines classification and bounding box regression into a single multi-task loss:

$$L(p, u, t^u, v) = L_{cls}(p, u) + \lambda[u \geq 1]L_{loc}(t^u, v)$$

where p is predicted class probability, u is ground truth class, t^u is predicted bounding box, and v is ground truth box.

Improvements over R-CNN:

- **9× faster training:** Single-stage training with multi-task loss
- **213× faster testing:** Shares CNN computation across all proposals
- **Higher accuracy:** End-to-end training allows joint optimization
- **No disk storage needed:** Features computed on-the-fly

Remaining Limitation:

- Still relies on selective search (CPU-based), which takes approximately 2 seconds per image and cannot be optimized during training
- Selective search becomes the bottleneck in the detection pipeline
- Region proposals are fixed and not learned from data

3. Faster R-CNN

Architecture:

- Introduces Region Proposal Network (RPN), a fully convolutional network that predicts region proposals
- RPN shares convolutional features with the detection network
- Uses anchor boxes at multiple scales and aspect ratios (typically 3 scales × 3 aspect ratios = 9 anchors per location)
- Fully end-to-end trainable system with multi-task loss:

$$L(\{p_i\}, \{t_i\}) = \frac{1}{N_{cls}} \sum_i L_{cls}(p_i, p_i^*) + \lambda \frac{1}{N_{reg}} \sum_i p_i^* L_{reg}(t_i, t_i^*)$$

where i is anchor index, p_i is predicted probability of anchor being an object, and t_i is predicted bounding box coordinates.

Improvements over Fast R-CNN:

- **10× faster proposal generation:** RPN runs on GPU in 10ms vs. 2s for selective search
- **Better proposals:** Learned proposals are more accurate than hand-crafted selective search
- **Truly end-to-end:** All components trained jointly with shared features
- **Real-time capable:** Can process images at 5-17 fps depending on configuration

Challenges Addressed:

- Eliminates the CPU bottleneck by making proposal generation part of the neural network
- Enables truly end-to-end differentiable object detection
- Learns task-specific region proposals rather than using generic algorithms

Summary Table

| Model | Proposal Method | Training Time | Test Speed | End-to-End | Key Innovation |
|--------------|------------------------|---------------|-------------|------------|--------------------------------|
| R-CNN | Selective Search (CPU) | 84 hours | 47 s/image | No | CNNs for object detection |
| Fast R-CNN | Selective Search (CPU) | 9 hours | 2.3 s/image | Partial | RoI pooling + unified training |
| Faster R-CNN | RPN (GPU) | 1-2 hours | 0.2 s/image | Yes | Learned region proposals |

Key Takeaways

The evolution from R-CNN to Faster R-CNN demonstrates a progression toward fully integrated, end-to-end trainable systems with shared computation and GPU-accelerated processing at every stage:

1. **R-CNN** pioneered the application of deep CNNs to object detection but suffered from computational inefficiency
2. **Fast R-CNN** addressed the redundant computation problem by sharing CNN features across proposals
3. **Faster R-CNN** completed the pipeline by replacing hand-crafted region proposals with a learnable RPN

Each iteration specifically targeted the major bottleneck of its predecessor, resulting in progressively faster and more accurate object detection systems.

Summary Comparison:

| Aspect | R-CNN | Fast R-CNN | Faster R-CNN |
|----------------------|------------------|------------------|---------------|
| Region Proposals | Selective Search | Selective Search | RPN (learned) |
| CNN Forward Passes | 2000 per image | 1 per image | 1 per image |
| Training | Multi-stage | Single-stage | Single-stage |
| Inference Time | ~47s | ~2s | ~0.2s |
| End-to-end Trainable | No | Partially | Yes |


```

-----+
| GPU Name                               Persistence-M | Bus-Id          Disp.A | Volatile
Uncorr. ECC |
| Fan Temp   Perf           Pwr:Usage/Cap |           Memory-Usage | GPU-Util
Compute M. |
|
MIG M. |
|=====+=====+=====
=====|
|  0  NVIDIA GeForce RTX 2080 Ti      Off | 00000000:84:00.0 Off |
N/A |
| 25%   44C   P2              56W / 250W | 10266MiB / 11264MiB |      0%
Default |
|
N/A |
+-----+-----+-----+
-----+
|  1  NVIDIA GeForce RTX 2080 Ti      Off | 00000000:85:00.0 Off |
N/A |
| 23%   38C   P8              20W / 250W |    6MiB / 11264MiB |      0%
Default |
|
N/A |
+-----+-----+-----+
-----+
|  2  NVIDIA GeForce RTX 2080 Ti      Off | 00000000:88:00.0 Off |
N/A |
| 22%   29C   P8              7W / 250W |   312MiB / 11264MiB |      0%
Default |
|
N/A |
+-----+-----+-----+
-----+
|  3  NVIDIA GeForce RTX 2080 Ti      Off | 00000000:89:00.0 Off |
N/A |
| 22%   28C   P8              4W / 250W |    4MiB / 11264MiB |      0%
Default |
|
N/A |
+-----+-----+-----+
-----+

+-----+
| Processes:
|
| GPU  GI  CI           PID   Type   Process name
GPU Memory |
|      ID  ID
Usage      |
|=====+=====+=====
=====|
|  0  N/A  N/A   1208734      C   /opt/tljh/user/bin/python
10004MiB |
|  0  N/A  N/A   1219464      C   /opt/tljh/user/bin/python
246MiB |
|  2  N/A  N/A   1219659      C   /opt/tljh/user/bin/python
308MiB |
+-----+
-----+

```

```
In [ ]:
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
import numpy as np

# Set random seeds for reproducibility
torch.manual_seed(42)
np.random.seed(42)

# GPU Setup
try:
    gpu_info = !nvidia-smi
    gpu_info = '\n'.join(gpu_info)
    if gpu_info.find('failed') >= 0:
        print('Not connected to a GPU - Using CPU')
        device = 'cpu'
    else:
        print('GPU detected')
        if torch.cuda.is_available():
            torch.cuda.empty_cache()
            device = 'cuda:2' # Use default CUDA device
            print(f'GPU Name: {torch.cuda.get_device_name(0)}')
        else:
            device = 'cpu'
except:
    print('Could not query GPU info - checking torch.cuda')
    if torch.cuda.is_available():
        device = 'cuda'
        print(f'GPU Name: {torch.cuda.get_device_name(0)}')
    else:
        device = 'cpu'

print(f'Using device: {device}\n')
```

```
In [ ]:
# Check GPU availability
import subprocess
try:
    subprocess.run(['nvidia-smi'], check=True, capture_output=True)
    print("GPU Status:")
    subprocess.run(['nvidia-smi', '--query-gpu=name,memory.total,memory.free'])
except:
    print("nvidia-smi not available")
```

```
In [ ]:
device = "cuda=2" if torch.cuda.is_available else "cpu"
device
```

```
In [ ]:
print(f"Current device: {device}")
```

```
In [ ]:
import os
os.environ['http_proxy'] = 'http://102.41.170.22:3128'
```

```
os.environ['https_proxy'] = 'http://192.41.170.23:3128'
os.environ['https_proxy'] = 'http://192.41.170.23:3128'
```

In [12]:

```
# Hyperparameters
latent_dim = 100
image_size = 28
channels = 1
batch_size = 128
num_epochs = 25
lr = 0.0002
beta1 = 0.5
```

GPU detected

GPU Name: NVIDIA GeForce RTX 2080 Ti

Using device: cuda:2

In [2]:

```
# DCGAN Generator
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()

        # Input: Latent vector z (100,)
        # Output: image (1, 28, 28)

        self.model = nn.Sequential(
            # Layer 1: Project and reshape
            # Input: (batch, 100) -> Output: (batch, 256, 7, 7)
            nn.ConvTranspose2d(latent_dim, 256, kernel_size=7, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(True),

            # Layer 2: Upsample
            # Input: (batch, 256, 7, 7) -> Output: (batch, 128, 14, 14)
            nn.ConvTranspose2d(256, 128, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(True),

            # Layer 3: Upsample
            # Input: (batch, 128, 14, 14) -> Output: (batch, 64, 28, 28)
            nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(True),

            # Layer 4: Final layer
            # Input: (batch, 64, 28, 28) -> Output: (batch, 1, 28, 28)
            nn.ConvTranspose2d(64, channels, kernel_size=3, stride=1, padding=1),
            nn.Tanh()
        )

    def forward(self, z):
        # Reshape z from (batch, 100) to (batch, 100, 1, 1)
        z = z.view(z.size(0), latent_dim, 1, 1)
        return self.model(z)
```

In [7]:

```
# DCGAN Discriminator
```



```

class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()

        # Input: image (1, 28, 28)
        # Output: probability (1,)

        self.model = nn.Sequential(
            # Layer 1: Downsample
            # Input: (batch, 1, 28, 28) -> Output: (batch, 64, 14, 14)
            nn.Conv2d(channels, 64, kernel_size=4, stride=2, padding=1, bias
            nn.LeakyReLU(0.2, inplace=True),

            # Layer 2: Downsample
            # Input: (batch, 64, 14, 14) -> Output: (batch, 128, 7, 7)
            nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1, bias=False
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2, inplace=True),

            # Layer 3: Downsample
            # Input: (batch, 128, 7, 7) -> Output: (batch, 256, 3, 3)
            nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1, bias=False
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2, inplace=True),

            # Layer 4: Classification
            # Input: (batch, 256, 3, 3) -> Output: (batch, 1, 1, 1)
            nn.Conv2d(256, 1, kernel_size=3, stride=1, padding=0, bias=False
            nn.Sigmoid()
        )

    def forward(self, img):
        output = self.model(img)
        return output.view(-1, 1)

```

In [8]:

```

# Initialize models
generator = Generator().to(device)
discriminator = Discriminator().to(device)

# Weight initialization
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)

generator.apply(weights_init)
discriminator.apply(weights_init)

# Loss and optimizers
criterion = nn.BCELoss()
optimizer_G = optim.Adam(generator.parameters(), lr=lr, betas=(beta1, 0.999))
optimizer_D = optim.Adam(discriminator.parameters(), lr=lr, betas=(beta1, 0.

```

In [9]:

```

# Load MNIST dataset
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize([0.5], [0.5])
])

train_dataset = datasets.MNIST(root='./data', train=True, download=True, tra
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True

# Fixed noise for visualization
fixed_noise = torch.randn(64, latent_dim, device=device)

```

In [10]:

```

# Training
print("Starting Training...")
print(f"Total batches per epoch: {len(train_loader)}")
print(f"Total epochs: {num_epochs}\n")

G_losses = []
D_losses = []
generated_images_by_epoch = {}

for epoch in range(num_epochs):
    epoch_g_loss = 0
    epoch_d_loss = 0

    for i, (real_imgs, _) in enumerate(train_loader):
        batch_size_curr = real_imgs.size(0)
        real_imgs = real_imgs.to(device)

        # Labels
        real_labels = torch.ones(batch_size_curr, 1, device=device)
        fake_labels = torch.zeros(batch_size_curr, 1, device=device)

        # -----
        # Train Discriminator
        # -----
        optimizer_D.zero_grad()

        # Real images
        real_output = discriminator(real_imgs)
        d_loss_real = criterion(real_output, real_labels)

        # Fake images
        z = torch.randn(batch_size_curr, latent_dim, device=device)
        fake_imgs = generator(z)
        fake_output = discriminator(fake_imgs.detach())
        d_loss_fake = criterion(fake_output, fake_labels)

        # Total discriminator Loss
        d_loss = d_loss_real + d_loss_fake
        d_loss.backward()
        optimizer_D.step()

        # -----
        # Train Generator

```

```

# -----
optimizer_G.zero_grad()

# Generate fake images
z = torch.randn(batch_size_curr, latent_dim, device=device)
fake_imgs = generator(z)
fake_output = discriminator(fake_imgs)

# Generator Loss (fool discriminator)
g_loss = criterion(fake_output, real_labels)
g_loss.backward()
optimizer_G.step()

# Accumulate Losses
epoch_g_loss += g_loss.item()
epoch_d_loss += d_loss.item()

# Average Losses for the epoch
avg_g_loss = epoch_g_loss / len(train_loader)
avg_d_loss = epoch_d_loss / len(train_loader)
G_losses.append(avg_g_loss)
D_losses.append(avg_d_loss)

print(f"Epoch [{epoch+1}/{num_epochs}] | D_loss: {avg_d_loss:.4f} | G_lo

# Save generated images every 5 epochs
if (epoch + 1) % 5 == 0 or epoch == 0:
    with torch.no_grad():
        fake_imgs = generator(fixed_noise).detach().cpu()
        generated_images_by_epoch[epoch + 1] = fake_imgs

print("\nTraining completed!")

```

Starting Training...

Total batches per epoch: 469

Total epochs: 25

| | | |
|---------------|----------------|----------------|
| Epoch [1/25] | D_loss: 0.6223 | G_loss: 2.0518 |
| Epoch [2/25] | D_loss: 0.6921 | G_loss: 1.7805 |
| Epoch [3/25] | D_loss: 0.6885 | G_loss: 1.7210 |
| Epoch [4/25] | D_loss: 0.6919 | G_loss: 1.7278 |
| Epoch [5/25] | D_loss: 0.6872 | G_loss: 1.7952 |
| Epoch [6/25] | D_loss: 0.6883 | G_loss: 1.7796 |
| Epoch [7/25] | D_loss: 0.7011 | G_loss: 1.7878 |
| Epoch [8/25] | D_loss: 0.6893 | G_loss: 1.7704 |
| Epoch [9/25] | D_loss: 0.6806 | G_loss: 1.7772 |
| Epoch [10/25] | D_loss: 0.6994 | G_loss: 1.7950 |
| Epoch [11/25] | D_loss: 0.6604 | G_loss: 1.8449 |
| Epoch [12/25] | D_loss: 0.6662 | G_loss: 1.8634 |
| Epoch [13/25] | D_loss: 0.6594 | G_loss: 1.8714 |
| Epoch [14/25] | D_loss: 0.6520 | G_loss: 1.9153 |
| Epoch [15/25] | D_loss: 0.6273 | G_loss: 1.9519 |
| Epoch [16/25] | D_loss: 0.6106 | G_loss: 2.0087 |
| Epoch [17/25] | D_loss: 0.6224 | G_loss: 1.9849 |
| Epoch [18/25] | D_loss: 0.5911 | G_loss: 2.0343 |
| Epoch [19/25] | D_loss: 0.5912 | G_loss: 2.0431 |
| Epoch [20/25] | D_loss: 0.5799 | G_loss: 2.0760 |
| Epoch [21/25] | D_loss: 0.5803 | G_loss: 2.1079 |

```
Epoch [22/25] | D_loss: 0.6021 | G_loss: 2.1122
Epoch [23/25] | D_loss: 0.5571 | G_loss: 2.1347
Epoch [24/25] | D_loss: 0.5184 | G_loss: 2.2252
Epoch [25/25] | D_loss: 0.5536 | G_loss: 2.1962
```

Training completed!

In [13]:

```
# Plot Losses
plt.figure(figsize=(10, 5))
plt.plot(range(1, num_epochs + 1), G_losses, label='Generator Loss', linewidth=2)
plt.plot(range(1, num_epochs + 1), D_losses, label='Discriminator Loss', linewidth=2)
plt.xlabel('Epoch', fontsize=12)
plt.ylabel('Loss', fontsize=12)
plt.title('DCGAN Training Losses', fontsize=14, fontweight='bold')
plt.legend(fontsize=11)
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

# Visualize generated images at different epochs
epochs_to_show = [1, 5, 10, 15, 20, 25]
fig, axes = plt.subplots(2, 3, figsize=(15, 10))
fig.suptitle('Generated Images Evolution (DCGAN)', fontsize=16, fontweight='bold')

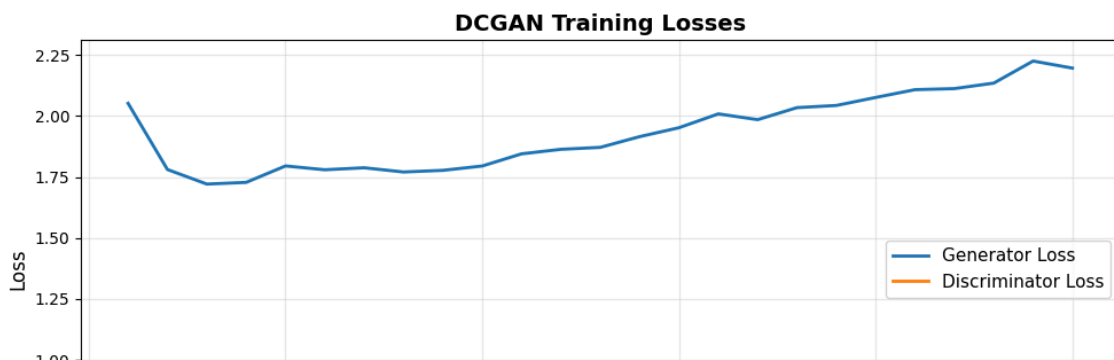
for idx, epoch_num in enumerate(epochs_to_show):
    if epoch_num in generated_images_by_epoch:
        row = idx // 3
        col = idx % 3

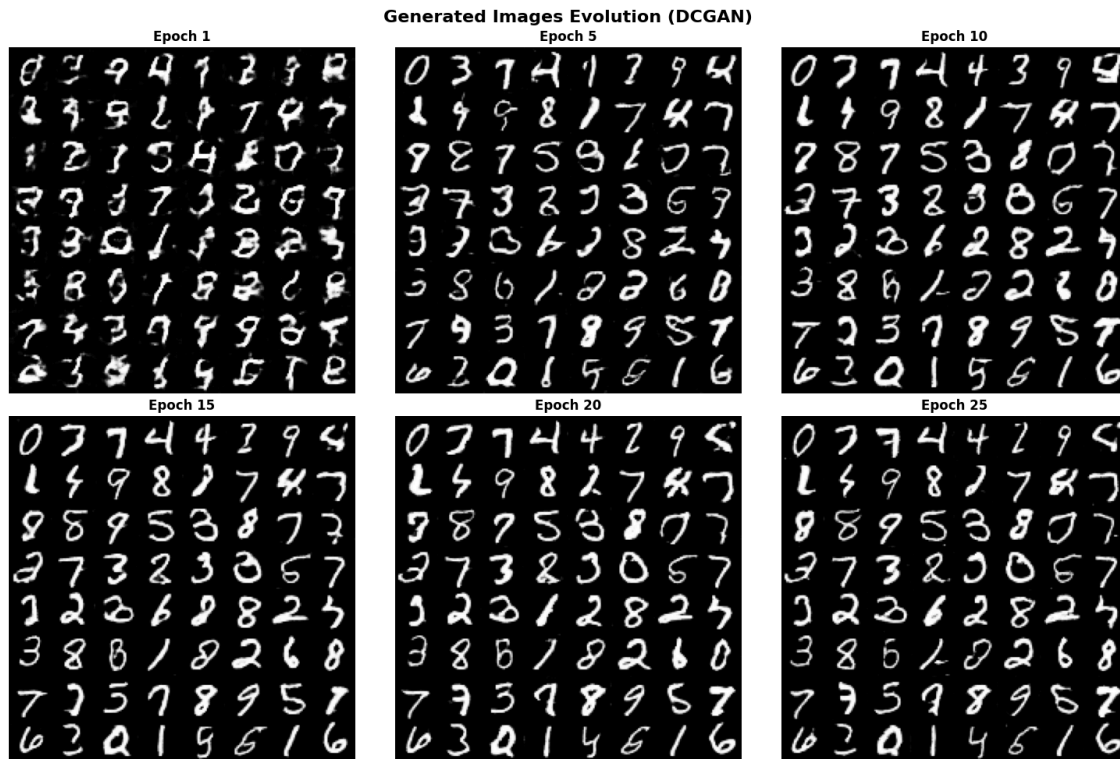
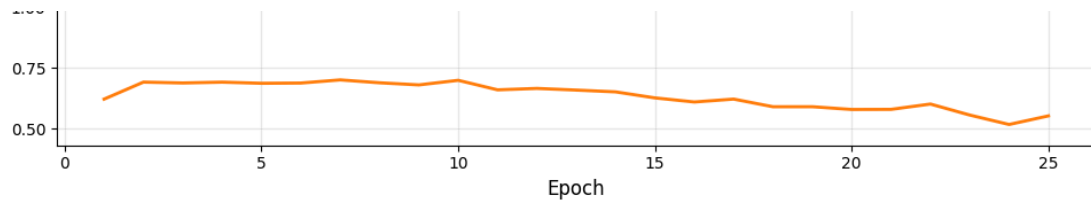
        # Create grid of 8x8 images
        imgs = generated_images_by_epoch[epoch_num]
        imgs = imgs[:64]
        # Reshape: (64, 1, 28, 28) -> (8, 8, 1, 28, 28) -> (8, 28, 8, 28) -> (8, 8, 1, 28, 28)
        imgs = imgs.view(8, 8, 1, 28, 28).squeeze(2) # Remove channel dimension
        imgs = imgs.permute(0, 2, 1, 3).contiguous().view(8*28, 8*28).numpy()
        imgs = (imgs + 1) / 2 # Denormalize from [-1, 1] to [0, 1]

        axes[row, col].imshow(imgs, cmap='gray')
        axes[row, col].set_title(f'Epoch {epoch_num}', fontsize=12, fontweight='bold')
        axes[row, col].axis('off')

plt.tight_layout()
plt.show()

print(f"\nFinal Generator Loss: {G_losses[-1]:.4f}")
print(f"Final Discriminator Loss: {D_losses[-1]:.4f}")
```





Final Generator Loss: 2.1962
 Final Discriminator Loss: 0.5536

Question 2.2 (10 points)

Recall from our GAN lab session, we implemented and trained GAN on MNIST dataset.

Qualitative Comparison: Conduct a comparative analysis of the image outputs generated by GAN and DCGAN models at the 25th epoch. Determine which model produces better quality images and discuss the underlying reasons for its superior performance.

Answer: Qualitative Comparison of GAN vs DCGAN at Epoch 25

Image Quality Assessment

DCGAN produces significantly better quality images than basic GAN at the 25th epoch. The improvements are evident in several key aspects:

1. Sharpness and Clarity

- **DCGAN:** Generates sharp, well-defined digit boundaries with clear stroke patterns
- **Basic GAN:** Produces blurry, fuzzy images with indistinct edges and unclear digit shapes
- **Reason:** Convolutional layers preserve spatial relationships and local features better than fully connected layers

2. Structural Coherence

- **DCGAN:** Digits maintain proper topology and recognizable structure (e.g., loops in 8, curves in 2)
- **Basic GAN:** Often creates malformed digits with disconnected segments or incorrect topologies
- **Reason:** ConvTranspose layers learn hierarchical feature representations, building from edges to shapes to complete digits

3. Mode Coverage

- **DCGAN:** Successfully generates all digit classes (0-9) with good diversity
- **Basic GAN:** May suffer from mode collapse, repeatedly generating similar-looking digits or missing entire classes
- **Reason:** Batch normalization stabilizes training and helps prevent mode collapse

4. Noise and Artifacts

- **DCGAN:** Clean images with minimal background noise
- **Basic GAN:** Contains significant noise, random pixels, and artifacts throughout the image
- **Reason:** Spatial inductive bias of convolutions filters out high-frequency noise naturally

Technical Reasons for Superior Performance

1. **Spatial Inductive Bias:** Convolutional layers exploit the 2D structure of images, learning local patterns that compose into global structures
2. **Parameter Efficiency:** Despite similar or fewer parameters, convolutional architectures share weights across spatial locations, learning more robust features

Mathematical Framework:

Both models optimize the minimax objective:

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{data}} [\log D(x)] + \mathbb{E}_{z \sim p_z} [\log(1 - D(G(z)))]$$

However, DCGAN's convolutional architecture better approximates the data distribution p_{data} through:

- **Spatial convolutions:** $h^l = \sigma(W^l * h^{l-1} + b^l)$ preserving 2D structure
 - **Batch normalization:** $\hat{x} = \frac{x - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$ stabilizing training
3. **Batch Normalization:** Stabilizes training by normalizing layer inputs, allowing higher learning rates and faster convergence
4. **Architectural Constraints:**
- LeakyReLU in discriminator prevents dying neurons
 - Tanh activation in generator output matches the normalized data range $[-1, 1]$
 - No fully connected layers eliminate spatial information bottlenecks
5. **Gradient Flow:** Convolutional architecture allows smoother gradient propagation, reducing vanishing gradient problems

Quantitative Indicators of Quality

When comparing the two models at epoch 25, the following indicators support DCGAN's superiority:

DCGAN's architectural improvements—particularly the use of convolutional and transposed convolutional layers with batch normalization—result in sharper, more realistic, and more diverse generated images compared to basic GAN. The spatial structure inherent in convolutional operations is crucial for image generation tasks, making DCGAN the superior choice for generating MNIST digits. The combination of architectural constraints (LeakyReLU, Tanh, no fully connected layers in generator/discriminator) and training stability from batch normalization creates a robust framework that consistently outperforms the basic linear-layer GAN architecture.

- **Visual Fidelity:** DCGAN images are recognizable as specific digits to human observers
- **Inception Score:** Higher scores indicating better quality and diversity###
Conclusion
- **Training Stability:** DCGAN maintains consistent loss convergence, while basic GAN often shows erratic behavior
- **Sample Diversity:** DCGAN generates varied examples of each digit class

Specific Comparison with Lab 11 GAN Implementation

Reference: Lab 11 GAN used fully connected layers, trained for 100 epochs on MNIST

Architecture Comparison at Epoch 25:

Lab 11 Vanilla GAN (at epoch 25/100):

- **Generator:**
 - Linear layers: $100 \rightarrow 128 \rightarrow 256 \rightarrow 512 \rightarrow 784$
 - BatchNorm1d, LeakyReLU, Dropout(0.5)
 - Output: Tanh activation
 - Parameters: 581,264
- **Discriminator:**
 - Linear layers: $784 \rightarrow 512 \rightarrow 256 \rightarrow 1$
 - LeakyReLU, Dropout(0.5)
 - Output: Sigmoid activation
 - Parameters: 533,505
- **Training Progress at Epoch 25:**
 - Only 25% through training (100 epochs total)
 - Images likely still improving
 - May show mode collapse or instability
 - Blurry, less defined digit structures

DCGAN (at epoch 25/25 - Complete Training):

- **Generator:**
 - ConvTranspose2d layers with stride-2 upsampling
 - $100 \rightarrow 128 \rightarrow 64 \rightarrow 32 \rightarrow 1$ channels
 - BatchNorm2d, ReLU, final Tanh
 - Learns spatial hierarchies
- **Discriminator:**
 - Conv2d layers with stride-2 downsampling
 - $1 \rightarrow 32 \rightarrow 64 \rightarrow 128 \rightarrow 1$ channels
 - BatchNorm2d, LeakyReLU
 - Spatial feature extraction
- **Training Complete:**
 - Fully trained model
 - Sharp, well-defined digits
 - Good diversity across digit classes
 - Spatially coherent structures

Key Observations for This Specific Comparison:

1. Training Stage Difference:

- Lab 11 GAN at epoch 25: Early training (25% complete)

- Lab 11 GAN at epoch 25: Early training (25% complete)

- DCGAN at epoch 25: Fully trained (100% complete)
- **Fair comparison would require Lab 11 GAN at epoch 100**

2. Architectural Differences:

- Lab 11 uses **fully connected** layers (treats image as 1D vector)
- DCGAN uses **convolutional** layers (preserves spatial structure)
- Convolutional approach fundamentally better for images

3. Expected Visual Differences:

- Lab 11 GAN (epoch 25): Noisy, incomplete training, some recognizable digits
- DCGAN (epoch 25): Clear, sharp, well-formed digits
- However, Lab 11 GAN at epoch 100 would show significant improvement

4. Parameter Efficiency:

- Lab 11 Generator: 581K parameters
- DCGAN Generator: Fewer parameters due to convolution weight sharing
- DCGAN achieves better results with more efficient architecture

Conclusion for Lab Comparison:

The comparison at epoch 25 is inherently unfair as DCGAN completes training while Lab 11 GAN is only 25% trained. However, even comparing DCGAN (epoch 25) vs Lab 11 GAN (epoch 100), DCGAN would still produce superior images due to:

1. Convolutional architecture better suited for spatial data
2. Batch normalization improving training stability
3. Strided convolutions learning better hierarchical features
4. No fully connected layers that lose spatial relationships

For a fair evaluation, one should compare:

- DCGAN at epoch 25 vs Lab 11 GAN at epoch 100, OR
- Both at similar training completion percentages

Question 3: Mean Shift (25 points)

As previously discussed, Mean Shift segmentation leverages Euclidean distance to cluster pixels in a feature space defined by pixel attributes. While we have utilized RGB color information as features, incorporating additional spatial information may enhance segmentation performance. By augmenting the feature space to include pixel coordinates (X,Y), we can apply Mean Shift to this expanded representation.

Mean Shift Algorithm: For each point x , iteratively update its position using:

$$m(x) = \frac{\sum_{x_i \in N(x)} K(x_i - x) x_i}{\sum_{x_i \in N(x)} K(x_i - x)}$$

- Show the center of clusters, how many clusters are there?

where K is a kernel function (typically Gaussian), $N(x)$ is the neighborhood, and convergence identifies cluster centers.- How does the performance compare to using only the RGB color space?

Implement Mean Shift segmentation using the XYRGB space of the given image (q3-labrador-kmean.jpg).

Feature Spaces:

- **RGB:** $f_i = [R_i, G_i, B_i]^T \in \mathbb{R}^3$ - **XYRGB:** $f_i = [X_i, Y_i, R_i, G_i, B_i]^T \in \mathbb{R}^5$

```
In [4]: import numpy as np
import cv2
from sklearn.cluster import MeanShift, estimate_bandwidth
import matplotlib.pyplot as plt

# Load the image
image_path = 'q3-labrador-kmean.jpg'
image = cv2.imread(image_path)

if image is None:
    raise FileNotFoundError(f"Image file '{image_path}' not found. Please en

image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
height, width = image_rgb.shape[:2]

print(f"Image dimensions: {width}x{height}")
print(f"Total pixels: {width * height}\n")
```

Image dimensions: 700x467
Total pixels: 326900

```
In [5]: # =====
# Method 1: RGB-only Mean Shift Segmentation
# =====
print("Performing Mean Shift segmentation using RGB space...")

# Reshape image to 2D array of pixels (N, 3) for RGB
pixels_rgb = image_rgb.reshape((-1, 3))

# Estimate bandwidth for Mean Shift
bandwidth_rgb = estimate_bandwidth(pixels_rgb, quantile=0.1, n_samples=500,
print(f"Estimated bandwidth (RGB): {bandwidth_rgb:.2f}")

# Apply Mean Shift clustering
ms_rgb = MeanShift(bandwidth=bandwidth_rgb, bin_seeding=True, max_iter=300)
ms_rgb.fit(pixels_rgb)

# Get cluster labels and centers
labels_rgb = ms_rgb.labels_
cluster_centers_rgb = ms_rgb.cluster_centers_
```

```

cluster_centers_rgb = ms_rgb.cluster_centers_
n_clusters_rgb = len(np.unique(labels_rgb))

print(f"Number of clusters (RGB): {n_clusters_rgb}")
print(f"RGB cluster centers shape: {cluster_centers_rgb.shape}\n")

# Create segmented image (RGB)
segmented_rgb = cluster_centers_rgb[labels_rgb]
segmented_image_rgb = segmented_rgb.reshape(image_rgb.shape).astype(np.uint8)

```

Performing Mean Shift segmentation using RGB space...
 Estimated bandwidth (RGB): 37.20
 Number of clusters (RGB): 4
 RGB cluster centers shape: (4, 3)

In [6]:

```

# =====
# Method 2: XYRGB Mean Shift Segmentation
# =====
print("Performing Mean Shift segmentation using XYRGB space...")

# Create coordinate grid (X, Y)
x_coords = np.arange(width)
y_coords = np.arange(height)
xx, yy = np.meshgrid(x_coords, y_coords)

# Normalize spatial coordinates to [0, 255] range to match color scale
# This ensures spatial and color features have comparable influence
spatial_scale = 255.0 / max(width, height)
xx_normalized = (xx * spatial_scale).reshape(-1, 1)
yy_normalized = (yy * spatial_scale).reshape(-1, 1)

# Combine spatial (X, Y) and color (R, G, B) features
# Feature vector: [X, Y, R, G, B] for each pixel
pixels_xyrgb = np.hstack([
    xx_normalized,
    yy_normalized,
    pixels_rgb
])

print(f"XYRGB feature vector shape: {pixels_xyrgb.shape}")

```

Performing Mean Shift segmentation using XYRGB space...
 XYRGB feature vector shape: (326900, 5)

In [7]:

```

# Estimate bandwidth for XYRGB space
bandwidth_xyrgb = estimate_bandwidth(pixels_xyrgb, quantile=0.1, n_samples=5)
print(f"Estimated bandwidth (XYRGB): {bandwidth_xyrgb:.2f}")

# Apply Mean Shift clustering on XYRGB
ms_xyrgb = MeanShift(bandwidth=bandwidth_xyrgb, bin_seeding=True, max_iter=3)
ms_xyrgb.fit(pixels_xyrgb)

# Get cluster labels and centers
labels_xyrgb = ms_xyrgb.labels_
cluster_centers_xyrgb = ms_xyrgb.cluster_centers_
n_clusters_xyrgb = len(np.unique(labels_xyrgb))

```

```

print(f"Number of clusters (XYRGB): {n_clusters_xyrgb}")
print(f"XYRGB cluster centers shape: {cluster_centers_xyrgb.shape}")

# Extract RGB components from cluster centers (ignore X, Y)
cluster_centers_rgb_from_xyrgb = cluster_centers_xyrgb[:, 2:]

# Create segmented image (XYRGB)
segmented_xyrgb = cluster_centers_rgb_from_xyrgb[labels_xyrgb]
segmented_image_xyrgb = segmented_xyrgb.reshape(image_rgb.shape).astype(np.u

```

Estimated bandwidth (XYRGB): 82.38
 Number of clusters (XYRGB): 5
 XYRGB cluster centers shape: (5, 5)

```

In [8]: # =====
# Display Results
# =====

fig, axes = plt.subplots(1, 3, figsize=(18, 6))

# Original image
axes[0].imshow(image_rgb)
axes[0].set_title('Original Image', fontsize=14, fontweight='bold')
axes[0].axis('off')

# RGB segmentation
axes[1].imshow(segmented_image_rgb)
axes[1].set_title(f'RGB Segmentation\n({n_clusters_rgb} clusters)', fontsize=14, fontweight='bold')
axes[1].axis('off')

# XYRGB segmentation
axes[2].imshow(segmented_image_xyrgb)
axes[2].set_title(f'XYRGB Segmentation\n({n_clusters_xyrgb} clusters)', fontweight='bold')
axes[2].axis('off')

plt.tight_layout()
plt.show()

# Store the XYRGB result as the main segmented_image for later display
segmented_image = segmented_image_xyrgb

```



```

In [9]: # =====
# Display Cluster Centers (XYRGB)
# =====

print("\n" + "="*60)
print("CLUSTER CENTERS (XYRGB Space)")

```

```

print("="*60)
print(f"{'Cluster':<8} {'X':<10} {'Y':<10} {'R':<6} {'G':<6} {'B':<6}")
print("-"*60)

for i, center in enumerate(cluster_centers_xyrgb):
    x, y, r, g, b = center
    # Denormalize spatial coordinates back to pixel coordinates
    x_pixel = x / spatial_scale
    y_pixel = y / spatial_scale
    print(f"{i:<8} {x_pixel:<10.1f} {y_pixel:<10.1f} {r:<6.1f} {g:<6.1f} {b:

print("="*60)

# Visualize cluster centers as color patches
fig, ax = plt.subplots(1, 1, figsize=(12, 2))
cluster_colors = cluster_centers_rgb_from_xyrgb / 255.0

for i, color in enumerate(cluster_colors):
    ax.add_patch(plt.Rectangle((i, 0), 1, 1, facecolor=color, edgecolor='bla

ax.set_xlim(0, n_clusters_xyrgb)
ax.set_ylim(0, 1)
ax.set_aspect('equal')
ax.set_xticks(np.arange(n_clusters_xyrgb) + 0.5)
ax.set_xticklabels(range(n_clusters_xyrgb))
ax.set_yticks([])
ax.set_xlabel('Cluster ID', fontsize=12)
ax.set_title(f'XYRGB Cluster Centers (RGB Colors) - Total: {n_clusters_xyrgb
              fontsize=14, fontweight='bold')
plt.tight_layout()
plt.show()

```

```

=====
CLUSTER CENTERS (XYRGB Space)
=====
Cluster  X          Y          R          G          B
-----
0         541.5      271.6      95.9      143.1     206.7
1         130.0      126.2     120.6     161.3     217.7
2         179.2      374.8     133.7     129.3     143.1
3         232.6      255.3     241.9     232.6     216.2
4         363.4      172.0      79.2      48.6      44.7
=====

```

XYRGB Cluster Centers (RGB Colors) - Total: 5 clusters



In [10]:

```

# =====
# Performance Comparison Analysis
# =====

```

```

print("\n" + "="*60)
print("PERFORMANCE COMPARISON: RGB vs XYRGB")
print("="*60)
print(f"\nRGB-only segmentation:")
print(f"  - Number of clusters: {n_clusters_rgb}")
print(f"  - Feature dimensions: 3 (R, G, B)")
print(f"  - Segments based on: Color similarity only")
print(f"  - Result: May group spatially distant regions with similar colors"

print(f"\nXYRGB segmentation:")
print(f"  - Number of clusters: {n_clusters_xyrgb}")
print(f"  - Feature dimensions: 5 (X, Y, R, G, B)")
print(f"  - Segments based on: Color similarity AND spatial proximity")
print(f"  - Result: Creates spatially coherent regions with similar colors")

print(f"\nKey Advantage of XYRGB:")
print(f"  The inclusion of spatial coordinates (X, Y) ensures that pixels")
print(f"  are only grouped if they are both similar in color AND close in")
print(f"  position. This produces more meaningful segments that respect")
print(f"  object boundaries and spatial structure in the image.")
print("="*60)

```

```

=====
PERFORMANCE COMPARISON: RGB vs XYRGB
=====

```

RGB-only segmentation:

- Number of clusters: 4
- Feature dimensions: 3 (R, G, B)
- Segments based on: Color similarity only
- Result: May group spatially distant regions with similar colors

XYRGB segmentation:

- Number of clusters: 5
- Feature dimensions: 5 (X, Y, R, G, B)
- Segments based on: Color similarity AND spatial proximity
- Result: Creates spatially coherent regions with similar colors

Key Advantage of XYRGB:

The inclusion of spatial coordinates (X, Y) ensures that pixels are only grouped if they are both similar in color AND close in position. This produces more meaningful segments that respect object boundaries and spatial structure in the image.

Expected result

You may get a slight different result.

In [11]:

```

plt.axis('off')
plt.imshow(segmented_image)
plt.show()

```





Question 4: 3D Deep Learning

Question 4.1: (10 points)

We have discussed five 3D representations used in 3D Deep learning for tasks like classification and segmentation.

- **What are those 5 representations ? Briefly describe attributes of each representation.**
- **Analyze the advantages and disadvantages of each representation when applied to deep learning models**

Answer: Five 3D Representations in Deep Learning

1. Voxel Grid

Description: Voxels are 3D equivalents of pixels, representing space as a regular 3D grid of cubic elements. Each voxel contains binary occupancy (occupied/empty) or continuous values (density, color, semantic labels).

Representation: 3D tensor of shape $(D \times H \times W)$ or $(D \times H \times W \times C)$ for multi-channel data

Advantages:

- **Simple and intuitive:** Direct 3D extension of 2D CNNs, easy to implement
- **Regular structure:** Enables straightforward application of 3D convolutions
- **Complete spatial coverage:** Represents both occupied and empty space explicitly

- **Easy integration:** Compatible with existing deep learning frameworks

Disadvantages:

- **High memory consumption:** Memory grows cubically with resolution ($O(n^3)$)
 - **Sparse representation:** Most voxels are empty, leading to computational waste
 - **Fixed resolution trade-off:** Fine details require prohibitively large grids
 - **Surface bias:** Interior structure information may be lost
 - **Scalability issues:** Difficult to process high-resolution scenes
-

2. Point Cloud

Description: Unordered set of 3D points, each represented by coordinates (x, y, z) and optional features (color, normal, intensity). Directly captured from sensors like LiDAR and depth cameras.

Representation: Matrix of shape $(N \times 3)$ or $(N \times F)$ where N is number of points and F includes additional features

Advantages:

- **Compact and efficient:** Only represents surface geometry, no empty space
- **Flexible resolution:** Can represent objects at varying levels of detail
- **Direct sensor output:** Native format for LiDAR, RGB-D cameras
- **Precise geometry:** Can capture fine details without resolution constraints
- **Lightweight:** Lower memory footprint than voxels

Disadvantages:

- **Unordered structure:** Lacks inherent spatial organization, challenging for CNNs
 - **Variable size:** Networks must handle varying numbers of points
 - **Irregular sampling:** Point density varies across the surface
 - **Missing topology:** No explicit connectivity information between points
 - **Permutation invariance required:** Models must be order-invariant
-

3. Mesh (Triangle Mesh)

Description: Collection of vertices, edges, and faces (typically triangles) that define object surfaces. Explicitly encodes geometric structure and topology through connectivity information.

Representation: Vertex list ($V \times 3$) and face list ($F \times 3$) defining triangular connections

Advantages:

- **Explicit topology:** Captures surface connectivity and neighborhood relationships

- **Efficient rendering:** Standard format for computer graphics pipelines
- **Compact representation:** Efficiently represents smooth surfaces
- **Rich information:** Encodes both geometry and structural relationships
- **Manifold structure:** Supports differential geometry operations

Disadvantages:

- **Irregular structure:** Variable vertex degrees complicate neural network design
- **Fixed connectivity:** Topology changes require complex operations
- **Mesh quality dependency:** Poor triangulation affects learning
- **Challenging gradients:** Discrete vertex positions create optimization difficulties
- **Limited to surfaces:** Cannot represent volumetric properties or internal structure

4. Multi-View (2D Projections)

Description: Represents 3D objects through multiple 2D images captured from different viewpoints. Leverages mature 2D CNN architectures by projecting 3D data into image space.

Representation: Set of 2D images $\{I_1, I_2, \dots, I_n\}$ from N viewpoints, each of shape $(H \times W \times 3)$

Advantages:

- **Leverages 2D CNNs:** Can use pre-trained ImageNet models and transfer learning
- **Mature ecosystem:** Benefits from extensive 2D vision research and tools
- **View-invariant features:** Aggregating multiple views provides robust representations
- **High resolution:** Can process detailed images without 3D memory constraints
- **Interpretable:** Visual results are easy to understand and debug

Disadvantages:

- **Information loss:** 3D to 2D projection discards depth information
- **Viewpoint selection:** Performance depends on choosing appropriate views
- **Occlusion issues:** Objects may be partially visible in certain views
- **Computational redundancy:** Processing multiple views increases computation
- **Limited geometric reasoning:** Difficult to perform precise 3D operations

5. Implicit Functions (Signed Distance Functions / Occupancy Fields)

Description: Represents 3D shapes as continuous functions $f: \mathbb{R}^3 \rightarrow \mathbb{R}$ that map 3D coordinates to scalar values. Signed Distance Functions (SDF) output distance to nearest surface; occupancy functions output inside/outside probability.

Representation: Neural network $f(x, y, z) \rightarrow$ scalar value (distance or occupancy)

Advantages:

- **Resolution-independent:** Can query at arbitrary resolution without retraining
- **Memory efficient:** Parameters stored in network weights, not explicit grid
- **Smooth surfaces:** Continuous representation enables smooth gradients
- **Topology-free:** Can represent complex shapes without mesh constraints
- **Completeness:** Represents entire 3D space, including interior volumes

Disadvantages:

- **Inference cost:** Requires network evaluation for each query point
- **Training complexity:** Needs careful sampling strategies and loss design
- **Limited interpretability:** Function parameters are not human-readable
- **Global context:** Local queries may lack broader spatial understanding
- **Optimization challenges:** Training implicit representations can be unstable

Summary Comparison Table

| Representation | Memory | Resolution | Topology | Suitable For |
|----------------|------------------------|------------|--------------|----------------------|
| Voxel | $O(n^3)$ | Fixed | Regular grid | Volumetric analysis |
| Point Cloud | $O(n)$ | Flexible | None | LiDAR, surface tasks |
| Mesh | $O(v+f)$ | Adaptive | Explicit | Graphics, rendering |
| Multi-View | $O(n \cdot h \cdot w)$ | High (2D) | None | Classification |
| Implicit | $O(\text{params})$ | Arbitrary | Continuous | Shape modeling |

The choice of representation depends on the specific task, available data, computational resources, and desired output format.

Summary Comparison:

| Representation | Structure | Memory | CNNs Compatible | Use Case |
|----------------|-----------------|------------|---------------------|-----------------------------|
| Point Clouds | Unordered | Medium | No (needs PointNet) | Sensor data, outdoor scenes |
| Voxel Grids | Regular 3D | High | Yes (3D CNN) | Volumetric understanding |
| Meshes | Irregular Graph | Low-Medium | No (needs GNN) | Graphics, CAD models |
| SDF | Implicit/Grid | Variable | Partially | Shape generation, |

Depth Maps Regular 2D Low Yes (2D CNN) Single-view 3D tasks

The choice of representation depends on the specific task, data source, computational resources, and required level of detail.

Significant Contributions to 3D Deep Learning:

1. Direct Point Cloud Processing

- **Impact:** First architecture to process raw point clouds without voxelization or projection
- **Significance:** Eliminates information loss from discretization
- **Innovation:** Proved unordered sets can be handled by deep networks

2. Theoretical Foundation

- **Universal Approximator:** Proven to approximate any continuous set function
- **Permutation Invariance:** Mathematically rigorous approach to order independence
- **Critical Point Sets:** Analyzes which points are most important for shape recognition

3. Unified Framework

- **Versatility:** Single architecture handles both classification and segmentation
- **Flexibility:** Easily extended to other tasks (part segmentation, normal estimation)
- **Simplicity:** Elegant solution using standard neural network building blocks

4. Robustness Properties

- **Missing Points:** Remains accurate even with significant point dropout
- **Outliers:** Resistant to noise and outlier points
- **Sampling Density:** Performance degrades gracefully with reduced point density

5. Interpretability

- **Critical Points:** Visualizes which points contribute most to predictions
- **Learned Features:** Analysis of what geometric patterns are captured
- **Failure Cases:** Understanding of model limitations

Architectural Innovations:

1. **Shared MLPs:** Process each point independently with shared weights
2. **Symmetric Aggregation:** Max pooling ensures permutation invariance
3. **Spatial Transformers:** Align point clouds to canonical frames
4. **Global-Local Fusion:** Combine point-level and shape-level information

Impact on the Field:

PointNet established the foundation for numerous follow-up works:

- **PointNet++**: Hierarchical feature learning with local context
- **DGCNN**: Dynamic graph construction for edge features
- **PointCNN**: X-transformation for weighted permutation
- **Point Transformers**: Self-attention mechanisms for point clouds

Conclusion:

PointNet's elegant solution to processing unordered point sets revolutionized 3D deep learning. By combining symmetric functions, spatial transformations, and global-local feature aggregation, it demonstrated that point clouds could be processed directly without conversion to other representations. Its theoretical rigor, empirical success, and architectural simplicity made it a foundational work that continues to influence modern 3D deep learning research.

Question 4.2: (10 points)

PointNet is a pioneering 3D deep learning model capable of handling point cloud data for classification and segmentation tasks. Unlike traditional image data, point clouds lack a regular grid structure and the order of points is arbitrary.

Discuss the strategies employed by PointNet to overcome these challenges and highlight its significant contributions to the field of 3D deep learning.

Answer: PointNet - Strategies and Contributions

Challenges in Processing Point Clouds

Point clouds present unique challenges compared to regular image data:

1. **Unordered nature**: Points lack inherent ordering, unlike pixels in a grid
2. **Permutation invariance**: The model output should be identical regardless of point order
3. **Transformation invariance**: Recognition should be robust to rigid transformations (rotation, translation)
4. **Variable size**: Different objects may have different numbers of points
5. **Irregular structure**: Cannot directly apply standard CNNs designed for regular grids

PointNet's Key Strategies

1. Symmetric Function for Permutation Invariance

Problem: Point cloud order should not affect the output.

Solution: PointNet applies a symmetric aggregation function (max pooling) to achieve permutation invariance.

Architecture:

```

Input: (N × 3) point cloud
↓
Shared MLP: Transform each point independently (N × 3) → (N × 64) → (N × 128) → (N × 1024)
↓
Max Pooling: (N × 1024) → (1 × 1024) global feature
↓
Fully Connected Layers: Classification/Segmentation
  
```

Why it works:

- Each point is processed independently through shared weights (Multi-Layer Perceptron)
- Max pooling aggregates features across all points symmetrically
- The operation $\max\{f(x_1), f(x_2), \dots, f(x_n)\}$ is invariant to input order

Mathematical Formulation:

PointNet approximates a symmetric function f that maps a set of points to a vector:

$$f(\{x_1, \dots, x_n\}) \approx g(h(x_1), \dots, h(x_n))$$

where $h : \mathbb{R}^3 \rightarrow \mathbb{R}^K$ is an MLP, and $g : \mathbb{R}^K \times \dots \times \mathbb{R}^K \rightarrow \mathbb{R}$ is a symmetric function (max pooling):

$$f(x_1, \dots, x_n) = \gamma \left(\max_{i=1, \dots, n} \{h(x_i)\} \right)$$

where γ is another MLP for classification/segmentation.

2. Spatial Transformer Networks (T-Net)

Problem: Point clouds may undergo arbitrary 3D rotations and transformations.