Practical – 7

Practical: Exploring advanced computer vision techniques.

Tasks:
Implementing a simple Generative Adversarial Network (GAN) for image generation. Building an object tracking system using techniques like correlation filters or deep learning-based methods. Developing an image captioning system using CNNs and recurrent neural networks (RNNs).

Code:

GAN for Image Generation

```
# Import libraries
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

# Define the Generator
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(100, 256),
            nn.ReLU(True),
            nn.Linear(256, 512),
            nn.ReLU(True),
            nn.Linear(512, 1024),
            nn.ReLU(True),
            nn.Linear(1024, 28*28),
            nn.Tanh()
        )

    def forward(self, x):
        return self.model(x).view(-1, 1, 28, 28)

# Define the Discriminator
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(28*28, 1024),
            nn.LeakyReLU(0.2),
```

```
            nn.Linear(1024, 512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 1),
            nn.Sigmoid()
        )

    def forward(self, x):
        return self.model(x.view(-1, 28*28))

# Instantiate models
generator = Generator()
discriminator = Discriminator()

# Define loss function and optimizers
criterion = nn.BCELoss()
optimizer_g = optim.Adam(generator.parameters(), lr=0.0002, betas=(0.5, 0.999))
optimizer_d = optim.Adam(discriminator.parameters(), lr=0.0002, betas=(0.5, 0.999))

# Load data
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])
dataloader = DataLoader(datasets.MNIST('.', download=True, transform=transform),
 batch_size=64, shuffle=True)

# Training loop
for epoch in range(10):
    for i, (images, _) in enumerate(dataloader):
        # Train Discriminator
        optimizer_d.zero_grad()
        labels = torch.ones(images.size(0), 1)
        outputs = discriminator(images)
        loss_d_real = criterion(outputs, labels)
        loss_d_real.backward()

        noise = torch.randn(images.size(0), 100)
        fake_images = generator(noise)
        labels = torch.zeros(images.size(0), 1)
        outputs = discriminator(fake_images.detach())
        loss_d_fake = criterion(outputs, labels)
        loss_d_fake.backward()
        optimizer_d.step()

        # Train Generator
        optimizer_g.zero_grad()
        labels = torch.ones(images.size(0), 1)
        outputs = discriminator(fake_images)
        loss_g = criterion(outputs, labels)
```

```
        loss_g.backward()
        optimizer_g.step()

    print(f'Epoch {epoch+1}, Loss D: {loss_d_real.item() + loss_d_fake.item()}, Loss G: {loss_g.item()}')
```

```
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.gz to ./MNIST/raw/t10k-labels-idx1-ubyte.gz
100%|██████████| 4542/4542 [00:00<00:00, 6429473.09it/s]Extracting ./MNIST/raw/t10k-labels-idx1-ubyte.gz to ./MNIST/raw


Epoch 1, Loss D: 0.7750575244426727, Loss G: 3.9478588104248047
Epoch 2, Loss D: 0.19816778786480427, Loss G: 3.1270577907562256
Epoch 3, Loss D: 0.13926053419709206, Loss G: 3.769843578338623
Epoch 4, Loss D: 0.2585000842809677, Loss G: 3.2638847827911377
Epoch 5, Loss D: 0.27333416044712067, Loss G: 2.7290661334991455
Epoch 6, Loss D: 0.7173106856644154, Loss G: 2.29699969291687
Epoch 7, Loss D: 0.6102647483348846, Loss G: 1.6247811317443848
Epoch 8, Loss D: 0.7156872749328613, Loss G: 2.2966723442077637
Epoch 9, Loss D: 0.5295298397541046, Loss G: 1.735113501548767
Epoch 10, Loss D: 0.6513427197933197, Loss G: 1.9565714597702026
```

Object Tracking System with DeepSORT

```
!pip install deep-sort-realtime

import cv2
import numpy as np

# Load YOLO model (paths should be set correctly)
net = cv2.dnn.readNet('yolov3.weights', 'yolov3.cfg')
layer_names = net.getLayerNames()
output_layers = [layer_names[i - 1] for i in net.getUnconnectedOutLayers()]

# Open video file
cap = cv2.VideoCapture('video.mp4')

while True:
    ret, frame = cap.read()
    if not ret:
        break

    # Object detection
    blob = cv2.dnn.blobFromImage(frame, 0.00392, (416, 416), (0, 0, 0), True, crop=False)
    net.setInput(blob)
    outs = net.forward(output_layers)

    # Initialize an empty list for detections
    detections = []

    # Process each output from the YOLO network
    for out in outs:
        for detection in out:
```

```
            scores = detection[5:]
            class_id = np.argmax(scores)  # Get the class ID
            confidence = scores[class_id]  # Get the confidence

            if confidence > 0.5:  # Filter out weak detections
                center_x = int(detection[0] * frame.shape[1])
                center_y = int(detection[1] * frame.shape[0])
                w = int(detection[2] * frame.shape[1])
                h = int(detection[3] * frame.shape[0])
                x = int(center_x - w / 2)
                y = int(center_y - h / 2)
                detections.append((x, y, w, h, confidence))

    # Draw bounding boxes around detections
    for (x, y, w, h, confidence) in detections:
        cv2.rectangle(frame, (x, y), (x+w, y+h), (0, 255, 0), 2)
        cv2.putText(frame, f'{confidence:.2f}', (x, y - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0,
255, 0), 2)

    cv2.imshow('Frame', frame)
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

cap.release()
cv2.destroyAllWindows()
```

Image Captioning System

```
import torch
import torch.nn as nn
from torchvision import models, transforms
from PIL import Image

# Define Encoder (CNN) and Decoder (RNN) models
class Encoder(nn.Module):
    def __init__(self):
        super(Encoder, self).__init__()
        self.model = models.resnet50(pretrained=True)
        self.model = nn.Sequential(*list(self.model.children())[:-1])

    def forward(self, x):
        with torch.no_grad():
            x = self.model(x)
        return x.view(x.size(0), -1)  # Shape should be [batch_size, 2048]

class Decoder(nn.Module):
    def __init__(self, vocab_size, embed_size, hidden_size):
        super(Decoder, self).__init__()
```

```
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.lstm = nn.LSTM(embed_size + 2048, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, vocab_size)

    def forward(self, features, captions):
        embeddings = self.embedding(captions)  # [batch_size, caption_length, embed_size]

        # Debugging: Check the shape of features and embeddings
        print(f'features shape before unsqueeze: {features.shape}')
        print(f'captions shape: {captions.shape}')
        print(f'embeddings shape: {embeddings.shape}')

        # Ensure features has correct shape
        if features.dim() == 2:  # [batch_size, 2048]
            features = features.unsqueeze(1)  # [batch_size, 1, 2048]
            print(f'features shape after unsqueeze: {features.shape}')

        # Repeat features to match the caption length
        features = features.repeat(1, captions.size(1), 1)  # [batch_size, caption_length, 2048]
        print(f'features shape after repeat: {features.shape}')

        # Concatenate features and embeddings
        inputs = torch.cat((features, embeddings), dim=2)  # [batch_size, caption_length, 2048 +
embed_size]
        print(f'inputs shape after concatenation: {inputs.shape}')

        outputs, _ = self.lstm(inputs)
        return self.fc(outputs)

# Load pretrained model and set up data transforms
encoder = Encoder()
decoder = Decoder(vocab_size=1000, embed_size=256, hidden_size=512)

preprocess = 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]),
])

# Load and preprocess an image
image = Image.open('apple.jpeg')
image = preprocess(image).unsqueeze(0)  # [1, 3, 224, 224]
features = encoder(image)  # [1, 2048]

# Debugging: Check the shape of the encoded features
print(f'Encoded features shape: {features.shape}')
```

```
# Generate captions
captions = torch.LongTensor([[1, 2, 3, 4]])  # Example caption indices with batch size 1
outputs = decoder(features, captions)

print(f'Final outputs shape: {outputs.shape}')
```

```
Encoded features shape: torch.Size([1, 2048])
features shape before unsqueeze: torch.Size([1, 2048])
captions shape: torch.Size([1, 4])
embeddings shape: torch.Size([1, 4, 256])
features shape after unsqueeze: torch.Size([1, 1, 2048])
features shape after repeat: torch.Size([1, 4, 2048])
inputs shape after concatenation: torch.Size([1, 4, 2304])
Final outputs shape: torch.Size([1, 4, 1000])
```