"예로부터 나라의 인재는 성균에 모여 왔으니,
그대 머묾이 우연이겠는가"

Multi Kernel Estimation based Object Segmentation

Haim Goldfisher, Asaf Yekutiel

22 Oct 2024

Presenter: Hyunjik Kang

rkdguswlr@g.skku.edu

Contents

"예로부터 나라의 인재는 성균에 모여 왔으니,
그대 머묾이 우연이겠는가"

INTRODUCTION

Introduction

- 초해상도(Super-Resolution)
  : 저해상도 이미지를 고해상도 이미지로 복원하는 과정

  - 목표 : blur kernel & additive noise에 의해 손상된 저해상도 이미지를 원래의 고해상도 형태에 가깝게 복원

$$I_{LR} = (I_{HR} * k_{blur}) \downarrow_s + n$$

Non-Blind 방식	Blind 방식
Blur Kernel이나 노이즈와 같은 손상 모델이 알려져 있거나 고정된 경우 고해상도 이미지를 저해상도 이미지로 매핑하는데 탁월 하지만, 실제 환경에서 손상 형태가 학습 중 보였던 것과 다를 시 성능 급격히 저하	손상 모델이 알려지지 않은 복잡합 시나리오인 경우 테스트 이미지 자체에서 손상 모델 직접 추정, 이미지 내에 존재하는 통계적 패턴 활용 단일 입력 이미지 만으로 초해상도 수행 가능 (Single Image Super – Resolution)

Introduction

- ZSSR ("Zero-Shot" Super Resolution)
  - CNN을 사용하여 저해상도 이미지를 고해상도 대응 이미지로 매핑하는 접근법
    - 1. 저해상도 이미지를 다운스케일한 후, CNN을 학습하여 저해상도 이미지 복원
    - 2. 학습된 네트워크는 이후 입력 저해상도 이미지에 적용되어 고해상도 이미지 생성

  - 한계
    - 기본 다운스케일링 커널로 bicubic interpolation 사용 – 실제 손상 과정을 반영하지 못할 가능성 존재

KernelGAN		GAN(Generative Adversarial Networks)
테스트 이미지의 내부 분포로부터 손상 커널을 학습함으로써 이를 해결	+	손상 커널을 추정 추정된 커널은 이후 ZSSR 과정에서 bicubic 커널을 대체하여 초해상도(SR) 과정과 최종 복원 결과를 향상

- 단일 커널 접근
  : 다양한 텍스처와 객체를 포함하는 복잡한 이미지에서는 한계

Introduction

- Multi-KernelGAN
  - KernelGAN을 확장하여 객체 분할 마스크(Object Segmentation Masks)를 활용하여 여러 커널 추
  - 이미지를 서로 다른 영역으로 나누고 각 영역에 대해 별도의 커널을 적용하여 커널 추정의 정확도와 견고성 향상

  - 2. 학습된 네트워크는 이후 입력 저해상도 이미지에 적용되어 고해상도 이미지 생성

핵심 아이디어
“하나의 이미지 내에서도 서로 다른 영역마다 서로 다른 다운스케일링 커널이 필요” => 서로 다른 영역의 고유한 특성 포착, 보다 정밀한 초해상도 결과

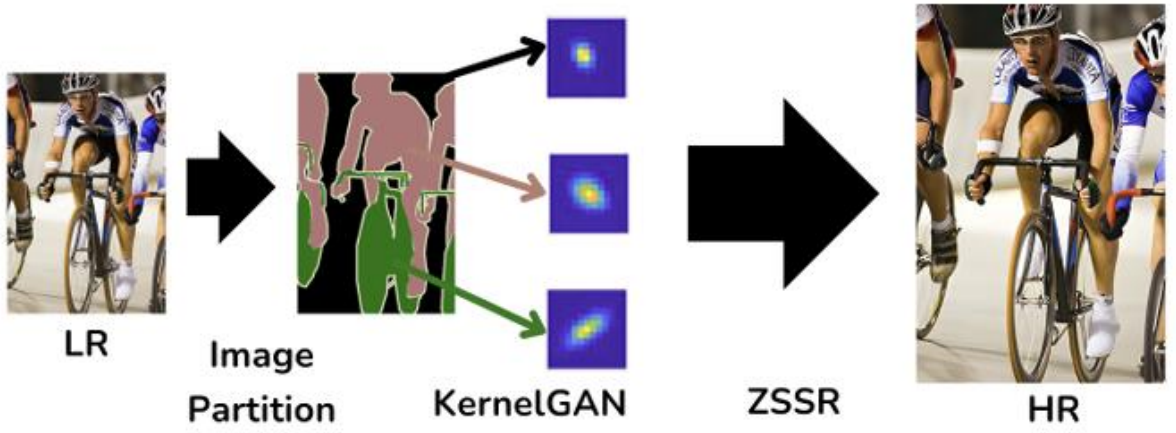
픽셀마다 별도의 커널 추정 -> 과적합 및 노이즈에 민감
: 분할 마스크 사용하여 균형

"예로부터 나라의 인재는 성균에 모여 왔으니,
그대 머묾이 우연이겠는가"

Methodology

Filtering Mechanism

- 1. Multi-KernelGAN



기존의 KernelGAN을 확장하여, 이진 마스크(binary masks)를 통해 이미지를 두 개의 영역으로 분할

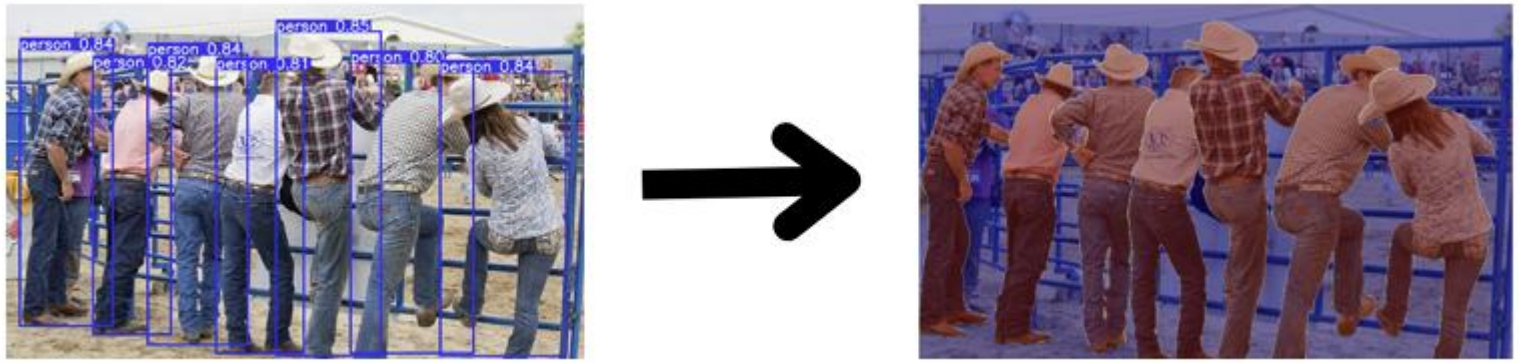
Ⅰ. **이미지 분할 (Image Segmentation)**:
입력 이미지를 마스크를 사용하여 여러 영역으로 분할

Ⅱ. **커널 추정 (Kernel Estimation)**:
각 영역에 대해 개별 커널을 GAN 프레임워크를 통해 추정

- Ⅲ. **초해상도 (Super-Resolution)**:
추정된 커널을 이용하여 ZSSR을 각 영역에 개별적으로 적용하여 초해상도를 달성

Filtering Mechanism

- 2. 영역 기반 마스크 생성



- 입력 이미지를 서로 다른 텍스처나 구조를 필요로 하는 독립적인 영역으로 나누는 데 필수적
- 각 영역마다 별도의 커널을 할당함으로써, 이미지 세부 정보의 다양한 변화를 더 잘 포착

객체 검출	분할 마스크 생성	이진 마스크 생성	다운스케일링	전경,배경 마스크
사전 학습된 **YOLOv8** 모델을 사용하여 주요 객체나 이미지 내의 영역을 식별	**SAM(Segment Anything Model)** 생성된 마스크는 이미지 내 객체들의 경계를 강조, 마스크 바깥 영역은 배경으로 처리	이미지를 전경(객체가 포함된 영역)과 배경으로 나눔	최근접 이웃 보간을 사용 낮은 해상도로 리사이즈	객체와 배경을 위한 마스크 별도 저장

- 3. Multi-KernelGAN

  - **3.1 각 영역별 이상적인 커널 학습**
    - 각 영역의 손상(degradation)을 가장 잘 설명하는 이상적인 다운샘플링 커널(Downsampling Kernel)을 학습하는 역할
    - 외부 학습 데이터 없이, 저해상도(region patches)만을 사용하여 **자가 지도(Self-Supervised)** 방식으로 커널을 학습
      => 각 영역의 복원 품질을 최적화

  - **3.2 ZSSR을 이용한 영역 기반 초해상도**
    - 내부 이미지 정보를 활용하여, 학습된 커널을 적용해 각 세그먼트를 업스케일링
    - 초해상도 과정은 각 영역마다 개별적으로 수행
    - 이미지의 각 부분 특성을 보존

- 3. Multi-KernelGAN

  - **3.3 전체 이미지 재구성**
    - 초해상도(SR)된 영역들을 결합하여 최종 **초해상도 이미지(SR image)**를 생성
    - 고해상도 패치들이 자연스럽게 통합되어, 각 영역에서 학습된 디테일과 특성을 유지할 수 있도록 함

  - **3.4 Multi-KernelGAN의 장점**
    - 서로 다른 커널과 업스케일링 전략을 적용하여, 영역마다 최적화된 초해상도를 수행
    - 더 정확하고 세밀한 결과
    - 복합적인 이질적(heterogeneous) 영역을 가진 이미지에서도 높은 품질의 복원이 가능

- 4. 기존 시도들
  - **4.1 글로벌 주파수 영역 텍스처 분할**
    - 이미지 패치들의 주파수 표현을 계산하여 객체와 배경을 텍스처 특성에 따라 분할
    - 고주파 텍스처와 저주파 영역을 구별

Ⅰ. **주파수 표현(FFT) 생성:**
각 이미지 패치에 대해 FFT를 계산, 공간 도메인을 주파수 도메인으로 변환.

Ⅱ. **스펙트럼 평균(Magnitude Spectrum Averaging):**
각 변환된 패치의 평균 스펙트럼 크기를 계산하여, 주요 주파수 성분을 식별

Ⅲ. **이진 마스크 생성(Binary Mask Creation):**
각 패치의 평균 주파수에 기반하여 이진 마스크를 생성

- 4. 기존 시도들

  - **4.2 로컬 주파수 기반 텍스처 분할**
    - 이미지 내의 정보(information content)를 분석하기 위해 두 가지 방법 활용
      - 1. 에지 및 윤곽선 검출 : **중요한 텍스처 변화가 발생하는 영역**을 강조
        - 에지 검출 기법을 적용하고, 결과로 생성된 윤곽선을 정제
        - 이미지 내 객체들의 세밀한 경계를 강조하는 마스크를 생성
        - 에지 강도(edge intensity)와 텍스처 경계(texture boundaries)를 기반으로 이미지를 분할

뚜렷한 영역은 확장되고
작은 불필요한 특징들은 부드럽게 처리되어 제거

- 4. 기존 시도들
  - **4.2 로컬 주파수 기반 텍스처 분할**
    - 이미지 내의 정보(information content)를 분석하기 위해 두 가지 방법 활용
      - 2. 앵커 픽셀 식별
        - 작은 이미지 패치들의 그래디언트 크기(gradient magnitude)를 계산
        - 강한 지역 그래디언트(local gradients)를 가진 영역은 앵커 포인트로 선택
        - 에지 강도(edge intensity)와 텍스처 경계(texture boundaries)를 기반으로 이미지를 분할

세밀한 이미지 디테일을 표현하는데 필수적
결과 마스크는 정보가 집중된 영역을 강조

- 4. 기존 시도들

  - **4.2.1 한계점**
    - **너무 작고 잡음이 많은 마스크**가 생성
    - 많은 마스크 영역에서 **이상한 색상의 경계선**

      -> 검출된 특징들의 크기가 너무 작아 발생한 문제

보다 강건한 기법 필요
잡음이나 artifact없이 정교한 텍스처의 정확한 포착

- 4. 기존 시도들
  - **4.3 딥러닝 기반 객체 분할**
    - **Region-based CNN** (R-CNN 및 Faster R-CNN)
    - 이후 더 발전된 **Detectron2**와 **SAM(Segment Anything Model)**

R-CNN 및 Faster R-CNN	Detectron2	SAM (Segment Anything Model)
속도(Speed): 특히 R-CNN은 연산량이 많아 real-time이나 대규모 응용에서 너무 느림 **정확성(Precision)**:생성된 바운딩 박스는 너무 조잡하여, 세밀한 객체 디테일이나 복잡한 경계를 포착하는 데 실패	**경계 정밀도:** 객체가 겹치거나, 크기가 비슷한 객체들이 다수 존재할 때, 경계가 정확하지 않거나 객체 분할이 불완전 **복잡한 장면**: 복잡하거나 혼잡한 환경에서 마스크가 종종 부정확하거나 불완전하게 생성	명시적인 객체 검출 없이 복잡한 객체를 효과적으로 분할 사전에 장면에 대한 지식 없이 정확한 객체 수를 결정하거나, 올바른 객체를 선택하는 것이 어려움

- 4. 기존 시도들
  - **4.3 딥러닝 기반 객체 분할**

SAM (Segment Anything Model)
가장 큰 세그먼트(large segment)를 우선시

명확한 경계나 객체 수에 대한 정보 없이 모든 객체를 일관되게 캡처하는 데 어려움
따라서, SAM 단독으로는 우리의 분할 작업에 충분하지 않음이 명백

YOLO + SAM
YOLO : 객체 수와 경계를 명확히 정의 SAM : 세밀한 픽셀 수준의 분할 수행

"예로부터 나라의 인재는 성균에 모여 왔으니,
그대 머묾이 우연이겠는가"

Experiments

Experiments

오픈소스 코드 실행 결과 및 분석 보고서

(Multi-KernelGAN의 핵심)

- **YOLOv8으로 객체를 찾고**
  → SAM으로 그 객체를 고해상도 수준으로 분할(segmentation)하여 마스크 생성

1. YOLO 써서 bounding box 잡고
2. SAM으로 박스 내부를 픽셀 단위로 Segmentation
3. 배경/객체 마스크 따로 생성
4. 이 마스크를 저장해서 이후 KernelGAN, ZSSR에 넘길 준비

=> 논문에서 실험한 데이터셋과 mask 분리 구조를 재현하는 데 매우 중요한 역할을 수행하는 코드

Experiments

오픈소스 코드 실행 결과 및 분석 보고서

```
!git clone https://github.com/kuty007/KernelGAN.git
```

```
Cloning into 'KernelGAN'...
remote: Enumerating objects: 357, done.
remote: Counting objects: 100% (105/105), done.
remote: Compressing objects: 100% (48/48), done.
remote: Total 357 (delta 62), reused 97 (delta 55), pack-reused 252 (from 1)
Receiving objects: 100% (357/357), 1.05 MiB | 19.57 MiB/s, done.
Resolving deltas: 100% (221/221), done.
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
코딩을 시작하거나 AI로 코드를 생성하세요.
```

```
# Install SAM and necessary dependencies
!pip install -q 'git+https://github.com/facebookresearch/segment-anything.git'
!pip install -q jupyter_bbox_widget roboflow dataclasses-json supervision
# Install YOLO and necessary dependencies
!pip install ultralytics
```

```
Downloading nvidia_cuda_runtime_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl (883 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 883.7/883.7 kB 64.3 MB/s eta 0:00:00
Downloading nvidia_cudnn_cu12-9.1.0.70-py3-none-manylinux2014_x86_64.whl (664.8 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 664.8/664.8 MB 2.7 MB/s eta 0:00:00
Downloading nvidia_cufft_cu12-11.2.1.3-py3-none-manylinux2014_x86_64.whl (211.5 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 211.5/211.5 MB 6.7 MB/s eta 0:00:00
Downloading nvidia_curand_cu12-10.3.5.147-py3-none-manylinux2014_x86_64.whl (56.3 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 56.3/56.3 MB 13.3 MB/s eta 0:00:00
Downloading nvidia_cusolver_cu12-11.6.1.9-py3-none-manylinux2014_x86_64.whl (127.9 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 127.9/127.9 MB 7.4 MB/s eta 0:00:00
Downloading nvidia_cusparse_cu12-12.3.1.170-py3-none-manylinux2014_x86_64.whl (207.5 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 207.5/207.5 MB 5.4 MB/s eta 0:00:00
Downloading nvidia_nvjitlink_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl (21.1 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 21.1/21.1 MB 105.0 MB/s eta 0:00:00
```

1. Segment Anything Model (SAM) 을 설치
2. 추가 패키지 설치로 Segmentation
   시각화와 데이터 처리 지원
3. YOLOv8 모델을 설치해 객체 탐지를 가능하게 함

오픈소스 코드 실행 결과 및 분석 보고서

```
import os
import re
import cv2
import numpy as np
import matplotlib.pyplot as plt
import torch
import supervision as sv
from segment_anything import sam_model_registry, SamAutomaticMaskGenerator, SamPredictor
import ultralytics
from ultralytics import YOLO

torch.cuda.is_available()
ultralytics.checks()
```

```
Ultralytics 8.3.119 🚀 Python-3.11.12 torch-2.6.0+cu124 CUDA:0 (Tesla T4, 15095MiB)
Setup complete ✅ (2 CPUs, 12.7 GB RAM, 41.3/112.6 GB disk)
```

```
# CHECKPOINT downloading
!wget https://dl.fbaipublicfiles.com/segment_anything/sam_vit_h_4b8939.pth
```

```
--2025-04-28 05:55:44--  https://dl.fbaipublicfiles.com/segment_anything/sam_vit_h_4b8939.pth
Resolving dl.fbaipublicfiles.com (dl.fbaipublicfiles.com)... 13.226.210.111, 13.226.210.25, 13.226.210.15, ...
Connecting to dl.fbaipublicfiles.com (dl.fbaipublicfiles.com)|13.226.210.111|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 2564550879 (2.4G) [binary/octet-stream]
Saving to: 'sam_vit_h_4b8939.pth'

sam_vit_h_4b8939.pt 100%[===================>]   2.39G   187MB/s    in 15s

2025-04-28 05:55:59 (163 MB/s) - 'sam_vit_h_4b8939.pth' saved [2564550879/2564550879]
```

1. 이미지 처리 및 딥러닝을 위한 필수 라이브러리(cv2, torch, YOLO, SAM)를 import
2. torch.cuda.is_available()로 GPU 사용 가능 여부를 확인
3. ultralytics.checks()로 YOLO 환경 구성이 정상인지 검사
4. wget 명령어로 SAM 모델 가중치(sam_vit_h_4b8939.pth) 다운로드

```
# Define model type and checkpoint path
MODEL_TYPE = "vit_h"
CHECKPOINT_PATH = "/content/sam_vit_h_4b8939.pth"

# Load the model
sam = sam_model_registry[MODEL_TYPE](checkpoint=CHECKPOINT_PATH).to(device = torch.device('cuda' if torch.cuda.is_available() else 'cpu'))

# Init sam predictor
predictor = SamPredictor(sam)

# Init YOLO
yolo = YOLO('/content/yolov8n.pt')
```

```
Downloading https://github.com/ultralytics/assets/releases/download/v8.3.0/yolov8n.pt to '/content/yolov8n.pt'...
100%|██████████| 6.25M/6.25M [00:00<00:00, 115MB/s]
```

1. SAM 기반으로 Predictor 객체 초기화
2. YOLOv8 모델(yolov8n.pt)을 로드해 객체 탐지 준비
3. 이후 segmentation(SAM)과 detection(YOLO)을 모두 사용할 수 있게 세팅

오픈소스 코드 실행 결과 및 분석 보고서

MyDrive 안 KernelGAN-Masks 폴더를 생성하고, 그 안에 imgs, masks 폴더 생성

img136 파일 안에 저해상도 사진 1개, 고해상도 사진 1개 업로드

- ex) 고해상도 _ 새 _ lr.png & 저해상도_ 새 _lr.png

```
import os
import re
import cv2
import numpy as np
import matplotlib.pyplot as plt

def create_blocky_mask(img):
    scale_factor = 0.15
    _, binary_mask = cv2.threshold(img, 127, 255, cv2.THRESH_BINARY)
    small_lr_back = cv2.resize(binary_mask, (0, 0), fx=scale_factor, fy=scale_factor, interpolation=cv2.INTER_NEAREST)
    blocky_lr_back = cv2.resize(small_lr_back, img.shape[::-1], interpolation=cv2.INTER_NEAREST)
    return blocky_lr_back

def extract_numbers_from_filename(filename):
    match = re.search(r'₩d+', filename)
    if match:
        return int(match.group())
    else:
        return None

def extract_largest_component(mask):
    true_map_binary = (mask * 255).astype(np.uint8)
    kernel = np.ones((10, 10), np.uint8)
    true_map_binary = cv2.morphologyEx(true_map_binary, cv2.MORPH_CLOSE, kernel)
    num_labels, labels, stats, centroids = cv2.connectedComponentsWithStats(true_map_binary)
    largest_component = 1 + np.argmax(stats[1:, cv2.CC_STAT_AREA])
    largest_component_mask = (labels == largest_component).astype(np.uint8) * 255
    return largest_component_mask
```

1. 함수 정의(create_blocky_mask) : 입력된 마스크를 리사이즈해 블록처럼 보이게 만든다.
2. Extract_numbers_from_filename : 파일명에서 숫자 추출
3. Extract_largest_component : 마스크에서 가장 큰 연결 영역만 추출

Experiments

오픈소스 코드 실행 결과 및 분석 보고서

```python
def create_mask(image_path):
    results = yolo.predict(source=image_path, conf=0.5)

    # Load and preprocess the image
    image = cv2.cvtColor(cv2.imread(image_path), cv2.COLOR_BGR2RGB)
    predictor.set_image(image)

    # Initialize an empty mask for combining all individual masks
    combined_mask = np.zeros(image.shape[:2], dtype=np.uint8)

    # Loop through each detected object
    for result in results:
        boxes = result.boxes

        for box in boxes.xyxy.tolist():
            input_box = np.array(box)

            masks, _, _ = predictor.predict(
                point_coords=None,
                point_labels=None,
                box=input_box[None, :],
                multimask_output=False,
            )

            segmentation_mask = masks[0]
            binary_mask = np.where(segmentation_mask > 0.5, 1, 0)

            # Combine the individual mask with the combined mask
            combined_mask = np.maximum(combined_mask, binary_mask)
    return combined_mask

# do the code above for all the imgs in the src folder
base = "/content/drive/MyDrive/KernelGAN-Masks"
src = base + "/imgs"
dst = base + "/masks"

files = os.listdir(src)
```

1. YOLO로 객체를 탐지하고, SAM을 이용해 각 객체 영역을 정밀 segmentation해서 binary mask를 만든다.

2. 모든 객체 마스크를 하나로 합쳐 최종 combined mask를 생성

3. 이후 src 폴더(/imgs) 안에 있는 이미지들을 순회할 준비.

Experiments

오픈소스 코드 실행 결과 및 분석 보고서

```
for file_name in files:
    image_path = os.path.join(src, file_name)
    img_num = extract_numbers_from_filename(file_name)
    dir_path = os.path.join(dst, 'img' + str(img_num))
    if not os.path.exists(dir_path):
        os.mkdir(dir_path)
    print(f"Processing image: {image_path}")

    image_path_lr = None
    for img in os.listdir(image_path):
        if "lr" in img:
            image_path_lr = os.path.join(image_path, img)
            break

    if image_path_lr is None:
        print(f"No low-resolution image found in {image_path}")
        continue

    lr_back = create_mask(image_path_lr)

    # Ensure lr_back is valid
    if lr_back is None or lr_back.size == 0:
        print(f"Invalid mask generated for {image_path_lr}")
        continue

    print(f"lr_back shape: {lr_back.shape}")

    # Convert lr_back to uint8 format if not already
    if lr_back.dtype != np.uint8:
        lr_back = (lr_back * 255).astype(np.uint8)
    print(f"lr_back dtype: {lr_back.dtype}")

    new_path_back_hr = os.path.join(dir_path, "back_hr_mask.png")
    new_path_back_lr = os.path.join(dir_path, "back_lr_mask.png")
    new_path_obj_hr = os.path.join(dir_path, "obj_hr_mask.png")
    new_path_obj_lr = os.path.join(dir_path, "obj_lr_mask.png")
```

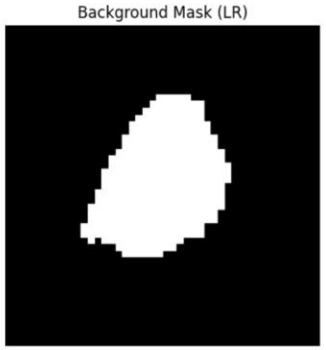
```
hr_back = cv2.resize(lr_back, dsize=(lr_back.shape[1]*2, lr_back.shape[0]*2), interpolation=cv2.INTER_CUBIC)
hr_obj = cv2.bitwise_not(hr_back)
lr_obj = cv2.bitwise_not(lr_back)

cv2.imwrite(new_path_back_hr, create_blocky_mask(hr_back))
cv2.imwrite(new_path_back_lr, create_blocky_mask(lr_back))
cv2.imwrite(new_path_obj_hr, create_blocky_mask(hr_obj))
cv2.imwrite(new_path_obj_lr, create_blocky_mask(lr_obj))

print(hr_back.shape)
print(lr_back.shape)
print(hr_obj.shape)
print(lr_obj.shape)

print(new_path_back_hr)
print(new_path_back_lr)
print(new_path_obj_hr)
print(new_path_obj_lr)

# Plotting the images and masks
original_image = cv2.imread(image_path_lr)
back_lr_mask = cv2.imread(new_path_back_lr, cv2.IMREAD_GRAYSCALE)
obj_lr_mask = cv2.imread(new_path_obj_lr, cv2.IMREAD_GRAYSCALE)

# Creating overlays
back_overlay = cv2.addWeighted(original_image, 0.5, cv2.cvtColor(back_lr_mask, cv2.COLOR_GRAY2BGR), 0.5, 0)
obj_overlay = cv2.addWeighted(original_image, 0.5, cv2.cvtColor(obj_lr_mask, cv2.COLOR_GRAY2BGR), 0.5, 0)
```

1. 각 이미지 폴더에서 저해상도(LR) 이미지를 찾아 create_mask로 배경/객체 마스크 생성
2. 생성된 마스크를 블록화(blocky mask)하고 HR/LR 버전으로 저장
3. 원본 이미지와 마스크를 오버레이해 시각화 준비 수행

Experiments

오픈소스 코드 실행 결과 및 분석 보고서

```
plt.figure(figsize=(20, 10))

plt.subplot(2, 3, 1)
plt.title('Original Image')
plt.imshow(cv2.cvtColor(original_image, cv2.COLOR_BGR2RGB))
plt.axis('off')

plt.subplot(2, 3, 2)
plt.title('Background Mask (LR)')
plt.imshow(back_lr_mask, cmap='gray')
plt.axis('off')

plt.subplot(2, 3, 3)
plt.title('Object Mask (LR)')
plt.imshow(obj_lr_mask, cmap='gray')
plt.axis('off')

plt.subplot(2, 3, 5)
plt.title('Image on Object Mask')
plt.imshow(cv2.cvtColor(obj_overlay, cv2.COLOR_BGR2RGB))
plt.axis('off')

plt.subplot(2, 3, 6)
plt.title('Image on Background Mask')
plt.imshow(cv2.cvtColor(back_overlay, cv2.COLOR_BGR2RGB))
plt.axis('off')

plt.show()
```



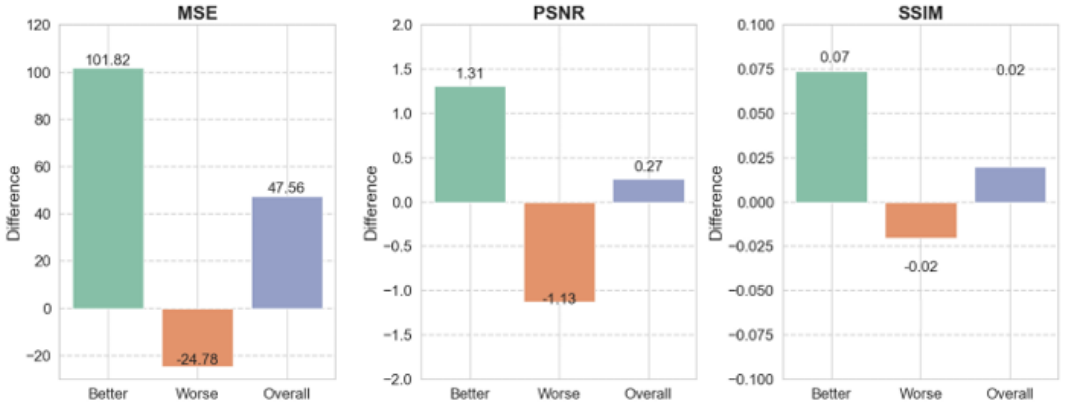
1. SAM + YOLO를 이용해 새(객체)와 배경을 분리하는 데 성공
2. 마스크를 blocky(조악한 격자 형태)로 만들어, 이후 KernelGAN + ZSSR 단계에서 학습을 더 안정적이고 단순하게 하기 위함
3. 이렇게 나눈 배경/객체를 각각 따로 초해상화(SR)할 수 있도록 준비

=> Multi-KernelGAN의 파이프라인의 핵심 인프라 구축

Experiments

Settings, 결과

- **평가지표**
  - PSNR (Peak Signal-to-Noise Ratio)
  - SSIM (Structural Similarity Index Measure)
  - MSE (Mean Squared Error)



- **성능 비교 대상**
  - Multi-KernelGAN+ZSSRPSNR
  - KernelGAN+ZSSR

- **결과**
  - MSE : Multi-KernelGAN이 **MSE를 더 낮춤** (오차 감소).
  - PSNR : Multi-KernelGAN이 **PSNR을 증가시킴** (신호 대 잡음비 향상).
  - SSIM: Multi-KernelGAN이 **SSIM을 소폭 개선** (구조적 유사도 향상).

"예로부터 나라의 인재는 성균에 모여 왔으니,
그대 머묾이 우연이겠는가"

Conclusion

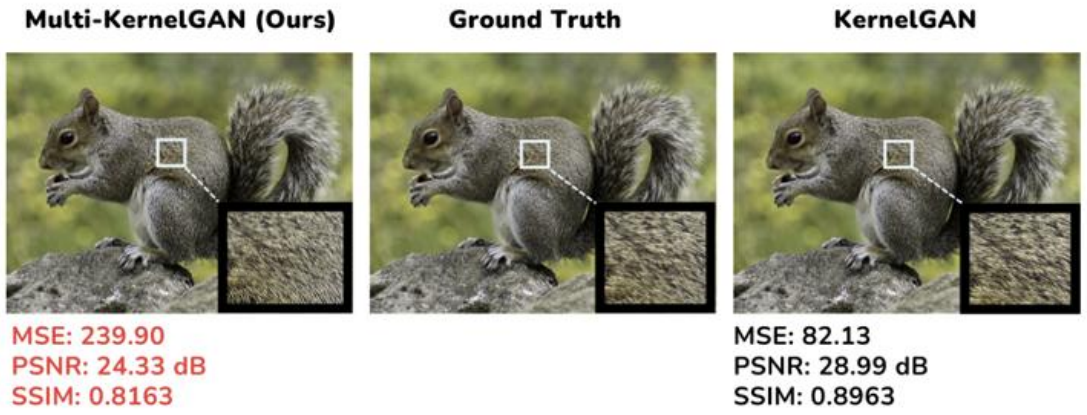
Conclusion

- **성능 비교**

Method	Average PSNR	Average SSIM	Average MSE
Multi-KernelGAN+ZSSR	26.4559	0.8214	180.5488
KernelGAN+ZSSR	26.1906	0.8013	228.1127

-> **Multi-KernelGAN+ZSSR**이 복원 품질 및 정밀성에서 전반적으로 우수함을 확인

- 결론 1 : 객체 텍스처 제약
  - 높은 텍스쳐 복잡성, 빠른 색상 변화를 가진 객체들의 경우 KernelGAN은 불안정하거나 부정확한 커널 생성
  - 여러 커널이 동시에 존재함으로써 객체를 더욱 불안정하게 형성



Conclusion

- 결론 2 : 분할 크기 제한
  - 각 분할된 영역이 일정 크기 이상이어야 한다
  - KernelGAN과 ZSSR이 효과적으로 학습하기 위해서는,
    각 세그먼트가 충분히 커서 의미있는 학습할 영역을 제공해야 한다.

- 결론 3 : 이상적인 사용 사례
  - 적절 : 명확한 경계와 배경을 가진, 구별되는(distinct) 객체가 있는 이미지.
  - 부적절 : 객체 경계가 명확하지 않거나, 텍스처가 너무 복잡하거나 불규칙한 경우