16-820: Advanced Computer Vision - HW6 - 2023

Introduction

In this homework you will work with the state-of-the-art foundation models for segmentation, SAM / Fast. Foundation models are large deep learning neural networks that are trained on massive datasets. Until now all your deep learning models have been trained on a single dataset, but foundation models are trained on many datasets. Foundation models are very powerful and can be used for many tasks. In this homework you will use pre-trained models, and you will use prompts (points, boxes, and text) to guide the pre-trained models to perform segmentation. The steps you will implement are:

- Prompt SAM with a single point to segment an object of interest.
- Prompt the model with 10 random points.
- Run all the unseen videos, qq.
  - Convert the ground truth bounding coordinates to the image frame.
  - Automatically generate a box input to SAM and run it SAM.
  - Prompt SAM with the ground truth box and a random point within the box.
  - Prompt the point cloud using all of the short-frame segment.

Homework deadline: **Mar 31**. To submit the homework to autograder, please submit **your code and the output of your code**. Please do not submit any other files. The autograder will run your code to generate the output files. Please do not change the seed in the code.

Homework Grade: Each section (TP, Generalize, 3D, Q1, Q2, Q3) is worth 20 points. Total 100 points. Late penalty: Lose 5 points for the first 24 hours and 10 points for every 24 hours after that.

Definitions

A couple definitions that will hopefully avoid confusion:

These are the crossing. Tickets:

- Camera Frame/Image/2D: Camera Frame, Image, Tracks in the reference frame. 3D points with respect to the camera. This is the camera frame of reference.
  - Blender Camera Frame: This is the camera frame for 3D points used in Blender. So, x and y are used in square dimension to convert to pixel coordinates. z is the depth.
  - OpenCV Camera Frame: This is the camera frame for 3D points used in OpenCV. So, x and y are used in square dimension to convert to pixel coordinates. z is the depth.
  - Blender Frame: 3D points in the image. x, y, z range in [-1,1] in SAM.

If we refer to **3D points** in the image frame, x, y, z range in [-1,1] in SAM.

If we refer to **2D points** in the image frame, x, y range in [0,1] in SAM.

How to run this homework

We will use Docker and Docker Compose which requires a very minimal system, just with a Docker Engine or DDE or PDDE. The instructions sign-in to GitHub to a Google Cloud, pulls the Docker Image to Google Cloud Artifact Registry, and runs the Docker Image. Your Fork of Google Cloud Artifact Registry and Docker Image.

Fork Docker Image

Environment Set-up with Google Colab

If you're not running on Google Colab, use the `pip_requirements.txt` and `apt-packages.txt` to install the right libraries, and the model checkpoint and data. Note that the code will be developed on Google Colab, differ, contrary to the rest of your Fork. Please download and committing the files.

```
# Clone repo
git clone --depth 1 -b main --single-branch --no-tags --quiet https://github.com/cmu-16820/hw6.git
# Install apt packages
xargs -a apt-packages.txt apt -qq install
# Install pip packages
pip install -q -r pip_requirements.txt --extra-index-url https://download.pytorch.org/whl/cu118
# Download checkpoint
wget -q -O checkpoints/sam_vit_h_4b8939.pth https://dl.fbaipublicfiles.com/segment_anything/sam_vit_h_4b8939.pth
# Download data
wget -q -O data.zip "https://drive.google.com/uc?export=download&id=1Ln0Ldq_Ln0Ldq_Ln0Ldq_Ln0Ldq&confirm=t" && unzip -qq data.zip
# Download unseen data
wget -q -O data-unseen.zip "https://drive.google.com/uc?export=download&id=1Ln0Ldq_Ln0Ldq_Ln0Ldq_Ln0Ldq&confirm=t" && unzip -qq data-unseen.zip
# Log in to Weights & Biases
wandb login --relogin --key <API_KEY>
# Run the notebook
jupyter nbconvert --to notebook --execute --inplace --kernel=python3 --ExecutePreprocessor.timeout=-1 --ExecutePreprocessor.kernel_name=python3 hw6.ipynb
```

Environment Set-up without Google Colab

If you're not running on Google Colab, use the `pip_requirements.txt` and `apt-packages.txt` to install the right libraries, and the model checkpoint and data. Note that the code will be developed on Google Colab, differ, contrary to the rest of your Fork. Please download and committing the files.

```
# Clone repo
git clone --depth 1 -b main --single-branch --no-tags --quiet https://github.com/cmu-16820/hw6.git
# Install apt packages
xargs -a apt-packages.txt apt -qq install
# Install pip packages
pip install -q -r pip_requirements.txt --extra-index-url https://download.pytorch.org/whl/cu118
# Download checkpoint
wget -q -O checkpoints/sam_vit_h_4b8939.pth https://dl.fbaipublicfiles.com/segment_anything/sam_vit_h_4b8939.pth
# Download data
wget -q -O data.zip "https://drive.google.com/uc?export=download&id=1Ln0Ldq_Ln0Ldq_Ln0Ldq_Ln0Ldq&confirm=t" && unzip -qq data.zip
# Download unseen data
wget -q -O data-unseen.zip "https://drive.google.com/uc?export=download&id=1Ln0Ldq_Ln0Ldq_Ln0Ldq_Ln0Ldq&confirm=t" && unzip -qq data-unseen.zip
# Log in to Weights & Biases
wandb login --relogin --key <API_KEY>
# Run the notebook
jupyter nbconvert --to notebook --execute --inplace --kernel=python3 --ExecutePreprocessor.timeout=-1 --ExecutePreprocessor.kernel_name=python3 hw6.ipynb
```

Set-up

Necessary imports and helper functions for displaying points, boxes, and masks.

```
import torch, torchvision
import numpy as np
import matplotlib as mp
import matplotlib.pyplot as plt
import os, glob, sys, shutil
import cv2
import wandb
import pandas as pd
import seaborn as sns
import tqdm
import IPython.display as ipd
import ffmpeg
import pickle as pkl
import scipy.io as sio
import scipy.ndimage as ndimage
import scipy.optimize as opt
import scipy.stats as stats
import sklearn.metrics as metrics
import sklearn.cluster as cluster
import sklearn.decomposition as decomposition
import sklearn.manifold as manifold
import sklearn.preprocessing as preprocessing
import sklearn.ensemble as ensemble
import sklearn.model_selection as model_selection
import sklearn.linear_model as linear_model
import sklearn.svm as svm
import sklearn.neighbors as nn
import sklearn.metrics.pairwise as pairwise
import sklearn.utils.extmath as extmath
import sklearn.cluster.hierarchical as hcluster
import sklearn.mixture as mixture
import sklearn.gaussian_process as gaussian_process
import sklearn.gaussian_process.kernels as kernels
import sklearn.calibration as calibration
import sklearn.tree as tree
import sklearn.feature_selection as feature_selection
import sklearn.pipeline as pipeline
import sklearn.metrics.cluster as cluster_metrics
import sklearn.manifold.t_sne as t_sne
import sklearn.decomposition.pca as pca
import sklearn.decomposition.kernel_pca as kernel_pca
import sklearn.decomposition.incremental_pca as incremental_pca
import sklearn.decomposition.sparse_pca as sparse_pca
import sklearn.decomposition.dictionary_learning as dictionary_learning
import sklearn.decomposition.factor_analysis as factor_analysis
import sklearn.decomposition.fastica as fastica
import sklearn.decomposition.truncated_svd as truncated_svd
import sklearn.decomposition.nmf as nmf
import sklearn.decomposition.latent_dirichlet_allocation as latent_dirichlet_allocation
import sklearn.decomposition.dictionary_learning as dictionary_learning
import sklearn.decomposition.sparse_coder as sparse_coder
import sklearn.decomposition.base as decomposition_base
import sklearn.decomposition.utils as decomposition_utils
import sklearn.decomposition._base as _base
import sklearn.decomposition._utils as _utils
import sklearn.decomposition._dictionary_learning as _dictionary_learning
import sklearn.decomposition._sparse_coder as _sparse_coder
import sklearn.decomposition._factor_analysis as _factor_analysis
import sklearn.decomposition._fastica as _fastica
import sklearn.decomposition._truncated_svd as _truncated_svd
import sklearn.decomposition._nmf as _nmf
import sklearn.decomposition._latent_dirichlet_allocation as _latent_dirichlet_allocation
import sklearn.decomposition._
```

Q4: Casting masks to new frames [10 pts]

Now we will look at how we transform an image from one frame of reference to another. In this we will cross through new viewpoints one by one, to our field of view over time. Each iteration of the loop we add the new 3D points to the previous 3D points, which are then used in the next iteration to cast the mask from the previous frame onto the current frame.

In simple approach mapping each mask using the camera parameters, we first backproject, use that to transform to a new frame, then project back to the coordinate image of the projected pixel. That is, we project all 3D points to the mask view, we wrap (0-1) pixels each mask on to the current image frame. The resulting masks could be empty (no pixels), one pixel, or a line. We would be the problem with we repeat this first.

Occlusions and mask merge frames are the best some pixels that weren't segmented correctly, occlusions can cause entirely erroneous masks. Occlusions are a problem because we are projecting points from a different viewpoint, so we can't know what is in front of what. Filter out points that are too far away from anything else and cast up to. 100 Carefully reduce the boundingbox dimensions to avoid noise to have a big effect on the result

```
def cast_mask(mask, d, K, T):
    # Cast a mask from one frame to image coordinates [0,1]
    # Input: mask [H,W], depth [H,W], intrinsics K [3,3], extrinsics T [4,4]
    # Output: cast_mask [H,W], cast_depth [H,W]
    H, W = mask.shape
    cast_mask = np.zeros(H, W)
    cast_depth = np.zeros(H, W)

    return cast_mask, cast_depth
```

In [1]:

```
# [1] filter out masks that have a confidence that's too low [0,pts]
def filter_masks(masks, c, pts):
    # Input: masks [N,H,W], confidence [N], points [N,H,W,3]
    # Output: filtered_masks [N,H,W], filtered_points [N,H,W,3]
    N = masks.shape[0]
    if N == 0:
        return masks, pts
    keep = c > 0.5
    return masks[keep], pts[keep]
```

In [2]:

```
# [2] filter out points that are too far away from anything else [0,pts]
def filter_points(pts, masks, radius=0.5):
    # Input: pts [N,H,W,3], masks [N,H,W], radius (default 0.5)
    # Output: filtered_pts [N,H,W,3]
    N, H, W = masks.shape
    if N == 0:
        return pts
    valid = masks > 0.5
    pts_flat = pts[valid]
    if len(pts_flat) < 2:
        return pts
    from scipy.spatial import cKDTree
    tree = cKDTree(pts_flat)
    dists, _ = tree.query(pts_flat, k=2)
    keep = dists[:, 1] < radius
    return pts[valid][keep]
```

In [3]:

```
# [3] filter out masks that have a confidence that's too low [0,pts]
def filter_masks(masks, c, pts):
    # Input: masks [N,H,W], confidence [N], points [N,H,W,3]
    # Output: filtered_masks [N,H,W], filtered_points [N,H,W,3]
    N = masks.shape[0]
    if N == 0:
        return masks, pts
    keep = c > 0.5
    return masks[keep], pts[keep]
```

In [4]:

```
# [4] filter out points that are too far away from anything else [0,pts]
def filter_points(pts, masks, radius=0.5):
    # Input: pts [N,H,W,3], masks [N,H,W], radius (default 0.5)
    # Output: filtered_pts [N,H,W,3]
    N, H, W = masks.shape
    if N == 0:
        return pts
    valid = masks > 0.5
    pts_flat = pts[valid]
    if len(pts_flat) < 2:
        return pts
    from scipy.spatial import cKDTree
    tree = cKDTree(pts_flat)
    dists, _ = tree.query(pts_flat, k=2)
    keep = dists[:, 1] < radius
    return pts[valid][keep]
```

In [5]:

```
# [5] Cast masks from one frame to image coordinates [0,pts]
def cast_masks(masks, d, K, T):
    # Input: masks [N,H,W], depth [H,W], intrinsics K [3,3], extrinsics T [4,4]
    # Output: cast_masks [N,H,W], cast_depth [H,W]
    N, H, W = masks.shape
    cast_masks = np.zeros(N, H, W)
    cast_depth = np.zeros(H, W)

    return cast_masks, cast_depth
```

In [6]:

```
# [6] Merge cast masks into existing masks, resolving overlaps [0,pts]
def merge_masks(existing, cast, cast_depth):
    # Input: existing [N,H,W], cast [N,H,W], cast_depth [H,W]
    # Output: merged [N+1,H,W]
    N, H, W = existing.shape
    if N == 0:
        return cast[:1]
    return np.concatenate([existing, cast[:1]], axis=0)
```

In [7]:

```
# [7] Project 3D points back into 2D mask [0,1]
def project_points(pts, K, T, H, W):
    # Input: pts [N,3], intrinsics K [3,3], extrinsics T [4,4], height H, width W
    # Output: mask [H,W] with 1 where points land, 0 elsewhere
    N = pts.shape[0]
    if N == 0:
        return np.zeros(H, W)
    ones = np.ones(N, 1)
    pts_h = np.concatenate([pts, ones], axis=1)
    pts_cam = (T @ pts_h.T).T[:, :3]
    z = pts_cam[:, 2]
    valid = z > 0
    if not valid.any():
        return np.zeros(H, W)
    pts_cam = pts_cam[valid]
    z = z[valid]
    u = (K[0, 0] * pts_cam[:, 0] / z + K[0, 2]).astype(int)
    v = (K[1, 1] * pts_cam[:, 1] / z + K[1, 2]).astype(int)
    in_bounds = (u >= 0) & (u < W) & (v >= 0) & (v < H)
    u = u[in_bounds]
    v = v[in_bounds]
    mask = np.zeros(H, W)
    mask[v, u] = 1
    return mask
```

In [8]:

```
# [8] Cast masks from one frame to image coordinates [0,pts]
def cast_masks(masks, d, K, T):
    # Input: masks [N,H,W], depth [H,W], intrinsics K [3,3], extrinsics T [4,4]
    # Output: cast_masks [N,H,W], cast_depth [H,W]
    N, H, W = masks.shape
    cast_masks = np.zeros(N, H, W)
    cast_depth = np.zeros(H, W)
    for i in range(N):
        pts = backproject(masks[i], d, K)
        if len(pts) > 0:
            mask = project_points(pts, K, T, H, W)
            cast_masks[i] = mask
            if mask.sum() > 0:
                cast_depth = np.maximum(cast_depth, mask * d)
    return cast_masks, cast_depth
```

In [9]:

```
# [9] Merge cast masks into existing masks, resolving overlaps [0,pts]
def merge_masks(existing, cast, cast_depth):
    # Input: existing [N,H,W], cast [N,H,W], cast_depth [H,W]
    # Output: merged [N+1,H,W]
    N, H, W = existing.shape
    if N == 0:
        return cast[:1]
    return np.concatenate([existing, cast[:1]], axis=0)
```

In [10]:

```
# [10] Project 3D points back into 2D mask [0,1]
def project_points(pts, K, T, H, W):
    # Input: pts [N,3], intrinsics K [3,3], extrinsics T [4,4], height H, width W
    # Output: mask [H,W] with 1 where points land, 0 elsewhere
    N = pts.shape[0]
    if N == 0:
        return np.zeros(H, W)
    ones = np.ones(N, 1)
    pts_h = np.concatenate([pts, ones], axis=1)
    pts_cam = (T @ pts_h.T).T[:, :3]
    z = pts_cam[:, 2]
    valid = z > 0
    if not valid.any():
        return np.zeros(H, W)
    pts_cam = pts_cam[valid]
    z = z[valid]
    u = (K[0, 0] * pts_cam[:, 0] / z + K[0, 2]).astype(int)
    v = (K[1, 1] * pts_cam[:, 1] / z + K[1, 2]).astype(int)
    in_bounds = (u >= 0) & (u < W) & (v >= 0) & (v < H)
    u = u[in_bounds]
    v = v[in_bounds]
    mask = np.zeros(H, W)
    mask[v, u] = 1
    return mask
```

In [11]:

```
# [11] Cast masks from one frame to image coordinates [0,pts]
def cast_masks(masks, d, K, T):
    # Input: masks [N,H,W], depth [H,W], intrinsics K [3,3], extrinsics T [4,4]
    # Output: cast_masks [N,H,W], cast_depth [H,W]
    N, H, W = masks.shape
    cast_masks = np.zeros(N, H, W)
    cast_depth = np.zeros(H, W)
    for i in range(N):
        pts = backproject(masks[i], d, K)
        if len(pts) > 0:
            mask = project_points(pts, K, T, H, W)
            cast_masks[i] = mask
            if mask.sum() > 0:
                cast_depth = np.maximum(cast_depth, mask * d)
    return cast_masks, cast_depth
```

In [12]:

```
# [12] Merge cast masks into existing masks, resolving overlaps [0,pts]
def merge_masks(existing, cast, cast_depth):
    # Input: existing [N,H,W], cast [N,H,W], cast_depth [H,W]
    # Output: merged [N+1,H,W]
    N, H, W = existing.shape
    if N == 0:
        return cast[:1]
    return np.concatenate([existing, cast[:1]], axis=0)
```

In [13]:

```
# [13] Project 3D points back into 2D mask [0,1]
def project_points(pts, K, T, H, W):
    # Input: pts [N,3], intrinsics K [3,3], extrinsics T [4,4], height H, width W
    # Output: mask [H,W] with 1 where points land, 0 elsewhere
    N = pts.shape[0]
    if N == 0:
        return np.zeros(H, W)
    ones = np.ones(N, 1)
    pts_h = np.concatenate([pts, ones], axis=1)
    pts_cam = (T @ pts_h.T).T[:, :3]
    z = pts_cam[:, 2]
    valid = z > 0
    if not valid.any():
        return np.zeros(H, W)
    pts_cam = pts_cam[valid]
    z = z[valid]
    u = (K[0, 0] * pts_cam[:, 0] / z + K[0, 2]).astype(int)
    v = (K[1, 1] * pts_cam[:, 1] / z + K[1, 2]).astype(int)
    in_bounds = (u >= 0) & (u < W) & (v >= 0) & (v < H)
    u = u[in_bounds]
    v = v[in_bounds]
    mask = np.zeros(H, W)
    mask[v, u] = 1
    return mask
```

In [14]:

```
# [14] Cast masks from one frame to image coordinates [0,pts]
def cast_masks(masks, d, K, T):
    # Input: masks [N,H,W], depth [H,W], intrinsics K [3,3], extrinsics T [4,4]
    # Output: cast_masks [N,H,W], cast_depth [H,W]
    N, H, W = masks.shape
    cast_masks = np.zeros(N, H, W)
    cast_depth = np.zeros(H, W)
    for i in range(N):
        pts = backproject(masks[i], d, K)
        if len(pts) > 0:
            mask = project_points(pts, K, T, H, W)
            cast_masks[i] = mask
            if mask.sum() > 0:
                cast_depth = np.maximum(cast_depth, mask * d)
    return cast_masks, cast_depth
```

In [15]:

```
# [15] Merge cast masks into existing masks, resolving overlaps [0,pts]
def merge_masks(existing, cast, cast_depth):
    # Input: existing [N,H,W], cast [N,H,W], cast_depth [H,W]
    # Output: merged [N+1,H,W]
    N, H, W = existing.shape
    if N == 0:
        return cast[:1]
    return np.concatenate([existing, cast[:1]], axis=0)
```

In [16]:

```
# [16] Project 3D points back into 2D mask [0,1]
def project_points(pts, K, T, H, W):
    # Input: pts [N,3], intrinsics K [3,3], extrinsics T [4,4], height H, width W
    # Output: mask [H,W] with 1 where points land, 0 elsewhere
    N = pts.shape[0]
    if N == 0:
        return np.zeros(H, W)
    ones = np.ones(N, 1)
    pts_h = np.concatenate([pts, ones], axis=1)
    pts_cam = (T @ pts_h.T).T[:, :3]
    z = pts_cam[:, 2]
    valid = z > 0
    if not valid.any():
        return np.zeros(H, W)
    pts_cam = pts_cam[valid]
    z = z[valid]
    u = (K[0, 0] * pts_cam[:, 0] / z + K[0, 2]).astype(int)
    v = (K[1, 1] * pts_cam[:, 1] / z + K[1, 2]).astype(int)
    in_bounds = (u >= 0) & (u < W) & (v >= 0) & (v < H)
    u = u[in_bounds]
    v = v[in_bounds]
    mask = np.zeros(H, W)
    mask[v, u] = 1
    return mask
```

In [17]:

```
# [17] Cast masks from one frame to image coordinates [0,pts]
def cast_masks(masks, d, K, T):
    # Input: masks [N,H,W], depth [H,W], intrinsics K [3,3], extrinsics T [4,4]
    # Output: cast_masks [N,H,W], cast_depth [H,W]
    N, H, W = masks.shape
    cast_masks = np.zeros(N, H, W)
    cast_depth = np.zeros(H, W)
    for i in range(N):
        pts = backproject(masks[i], d, K)
        if len(pts) > 0:
            mask = project_points(pts, K, T, H, W)
            cast_masks[i] = mask
            if mask.sum() > 0:
                cast_depth = np.maximum(cast_depth, mask * d)
    return cast_masks, cast_depth
```

In [18]:

```
# [18] Merge cast masks into existing masks, resolving overlaps [0,pts]
def merge_masks(existing, cast, cast_depth):
    # Input: existing [N,H,W], cast [N,H,W], cast_depth [H,W]
    # Output: merged [N+1,H,W]
    N, H, W = existing.shape
    if N == 0:
        return cast[:1]
    return np.concatenate([existing, cast[:1]], axis=0)
```

0_1_original_points

0_1_mask

0_1_mask

0_1_all points found so far.

0_4_original_points

0_4_mask

0_4_mask

0_4_all points found so far.

0_10_original_points

0_10_mask

0_10_mask

0_10_all points found so far.

0_14_original_points

0_14_mask

0_14_mask

0_14_all points found so far.

0_71_original_points

0_71_mask

0_71_mask

0_71_all points found so far.

```
0.000 0.000 0.000
0.000 0.000 0.000
0.000 0.000 0.000
```