Summer School Exercise Sheet - Computer Vision

Sebastian Ruiz, Shijia Li, Cheng Minghao, Henrik Trommer, Matthias Nuske

October 2023

1 Introduction

In the context of robotics and automation, demonstration learning is the concept of a robot learning a skill by imitating an expert. One way to do this is to observe how a human moves and imitate it. The most natural way to do this is using computer vision.

The exercise is split into the following tasks: 2D hand pose detection, stereo vision to world coordinates, hand pose classification, and tracking.

Task outline Utilise a stereo vision setup to track hand movements and replicate them in a robot simulation. Various methods can be employed to solve such problems. Our preferred approach is as follows:

1. Detect the hand pose based on a model trained on labelled 2D key points.
2. Construct 3D keypoints from the matching 2D key points in both images of the stereo setup.
3. Use the 2D or 3D key points for hand gesture classification. A small dataset of 18 different gestures is provided. Decide if you want to use template matching or a shallow neural network for classification.
4. Use the 3D key points for tracking and translate this into the robot's coordinate system. The simulated robot then has to replicate your hand movements.

2 Setup

St up a Python environment for solving the task. Instructions are provided for using Conda.

1. Install conda if you don't have it already from https://docs.conda.io/projects/conda/en/latest/user-guide/install/index.html#regular-installation
2. Download our git repository by doing
   `git clone git@github.com:sebastian-ruiz/summer_2023_exercises_cv.git`
3. Install the conda environment by running `sh environment.sh`

Before you can view the images from the camera, you need to run:

```
export ROS_MASTER_URI=http://192.168.0.102:11311
```

With the environment set up you are ready to go.

2.1 Try it Out

Run `example.py`.

To load images from a folder use

```
1 ImgLoader(use_ros=False, folder=dataset_dir)
```

To use the stereo cameras use

```
ImgLoader(use_ros=True)
```

The `folder` parameter is ignored in this case.

3 2D Hand keypoint Detection

We utilize the MMPose toolbox for hand keypoint estimation, as documented by its contributors [MMPose Contributors 2020] which is implemented in PyTorch. The annotations for the hand

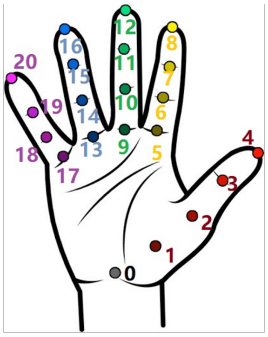

Figure 3.1: hand keypoint annotations

keypoints results are depicted in Figure 3.1.

To detect hand keypoints in a natural environment, the process typically involves two steps. First, a hand detector is employed to identify the bounding box (bbox) of the hands within the image. Subsequently, the image is cropped based on the bounding box coordinates, and this cropped region is then fed into the keypoint detector. We provided the pre-trained models that enable direct utilization. The usage could see below:

```
CLASS HandKPDetBatch(**kwargs)
    """detect the key points for hand. input could be one image(ndarray/str) or list
    ↪  of images
    Args:
        device(str) - cpu or cuda
        bbox_thr(float) - the threshold for bboxe in hand detector use in nms.
↪   default: 0.3
        kpt_thr(float) - the threshold for confidence scores in kerpoint detector.
↪   default: 0.1
        min_valid_kps(int) - the minimum number of keypoints confirmed to be valid
↪   hand. default: 10
        draw_bbox(bool) - draw bbox in the visualize results. default: True
        show_kpt_idx(bool) - show key points id in the visualize results. default:
↪   False
    Example:
        from handkpdet import HandKPDetBatch
        handkpdet = HandKPDetBatch(device='cpu')
        img_path = 'models/demo.jpg'  # replace this with your own image path
        img = cv2.imread(img_path)  # bgr order
        result_list, vis_list = handkpdet([img, img], show=False, draw_bbox=True,
↪   show_kpt_idx=True, kpt_thr=0.1)
        keypoints_list = [result.get('keypoints') if result else None for result in
↪   result_list]
        keypoint_scores_list = [result.get('keypoint_scores') if result else None
↪   for result in result_list]
    """
```

Exercise 1 We could obtain the keypoints and keypoint confidence scores from the detector(HandKPDetBatch). In situations where there are multiple individuals with their hands visible in the image, additional processing becomes necessary to select a specific hand for tracking. A

straightforward approach involves retaining only the hand with the highest average confidence score. See the Example of HandKPDetBatch to figure out the basic usage.

Implement this in the file `handkpdet.py`.

```
def filter_onehand(keypoints_list, keypoint_scores_list):
    pass # to be completed
    return onehand_keypoints_list, onehand_keypoint_scores_list

handkpdet = HandKPDetBatch(device='cpu')
img_path = 'models/demo.jpg'
img = cv2.imread(img_path)
img_list = [img, img]
result_list, vis_list = handkpdet(img_list, show=False, draw_bbox=True,
↪  show_kpt_idx=True, kpt_thr=0.1)
keypoints_list = [result.get('keypoints') if result else None for result in
↪  result_list]
keypoint_scores_list = [result.get('keypoint_scores') if result else None for result
↪  in result_list]
onehand_keypoints_list, onehand_keypoint_scores_list =
↪  filter_onehand(keypoints_list, keypoint_scores_list)
handkpdet.show_img(vis_list[0])
handkpdet.show_img_kp(img_list[0], onehand_keypoints_list[0] , input_color_order =
↪  'bgr')
```

4 Stereo Vision to World Coordinates

We provide you with a stereo camera setup. We provide also the calibration data. The calibration data was obtained using the `cv2.stereoCalibrate` function. See `camera_calibrate.py`.

A camera matrix P is a 3×4 matrix which describes the mapping of a pinhole camera from 3D points in the world to 2D points in an image.

$$P = K \cdot [R|t]$$

where P is the camera matrix, K the intrinsic matrix and $[R|t]$ the extrinsic matrix. R is a 3×3 rotation matrix and t is a 3×1 translation vector.

Given a point in world coordinates $(X, Y, Z, 1)^T$, we get its point in the image $(U, V, 1)^T$ by

$$s \begin{pmatrix} U \\ V \\ 1 \end{pmatrix} = P \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}, \quad \text{where} \quad s \in \mathbf{R}.$$

The intrinsic matrix K depends on the characteristics of the camera. The extrinsic matrix defines a rigid-body transform. This provides a transformation that relates points from the world reference system to the camera reference system.

The `cv2.stereoCalibrate` function returns the matrices K_1, K_2, R and, t. Here K_1 and K_2 are the intrinsic matrices of cameras 1 and 2 respectively. R and t are the rotation matrix and translation vector from camera 1's coordinate frame to camera 2's reference frame.

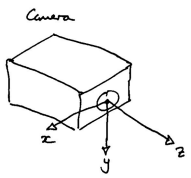

Figure 4.1: Camera Coordinate Frame

Exercise 2 We get the 2D hand poses from section 3. Get the 2D hand pose from the left and right camera. Construct the 3D hand pose from this. Hint: you can use `cv2.triangulatePoints`.

Implement this in the file `hand_pose3d.py`.

5 Hand Gesture Classification

The cameras provide 2D hand positions for both perspectives, and from section section 4, the 3D hand position is also available. Use this to categorise hand gestures.

A small dataset of different gestures is available in the GitHub. The gestures are: *fist, one, two , three, four, five, metal, fingers-crossed, thumbs down, pointing, pointing2, pinch, thumbs up, c, flat, perfect, startrek*.

Exercise 3 Use the dataset to classify the hand gestures. Start by classifying *pointing* and *fist*, if these two work improve your approach to include the others as well.

Choose **one of the following** approaches to implement this.

a. Template Matching For each class in the dataset take some of the 3D hand poses as the templates. The 3D keypoints are the template. Given a query 3D hand pose find the best matching template.

One method is to fit an affine transform between template and the query keypoints. In the 3D case, we want to find rotation matrix A and translation vector b, where A is a 3×3 matrix and b is a vector of length 3, such that

$$\mathbf{y} = f(\mathbf{x}) = A\mathbf{x} + b.$$

This can be written as

$$\begin{bmatrix} \mathbf{y} \\ 1 \end{bmatrix} = \left[\begin{array}{ccc|c} & A & & \mathbf{b} \\ 0 & \cdots & 0 & 1 \end{array}\right] \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix}$$

and this can be solved using `np.linalg.lstsq`. Compute the error in the transform. The template with the smallest error is the best fitting template. Implement this in `template_matching.py`.

b. Neural Network Classifier Another method is to train a neural network. This can be trained on the 2D or the 3D key points.

For the neural network approach the following steps are necessary:

1. design a basic network architecture from a few linear layers that takes the key points as inputs and predicts the hand gesture label
2. select a reasonable loss term
3. set reasonable training parameters & train the model
4. use the trained model for inference on the camera feed (keypoint_classifier.py)

For the neural network method, implement this in the files contained within classification_model.

6 Filtering

The data should be filtered with a low-pass FIR filter so the trajectory of the hand would be smooth, and it would not move too fast (too big acceleration) to cause the failure of the robot.

Exercise 4 Design a FIR low-pass filter, then apply the filter to the detected key points of the hand. It should smooth the trajectory and prevent the saturation of the acceleration in hand movement.

Implement this in `traj_filter.py` using numpy, scipy or other packages that support FIR filtering and convolution.

7 Plotting trajectory

After obtaining the trajectory of the hand or the specified finger's movement, you can visualize it using the provided functions. Compare the results to ascertain if they align with your expectations. See class PlotTraj in the file `plot_traj.py`.

```
# Usage:
pt = PlotTraj()
#traj_list should be a list of trajectory points(1*3 ndarray for 3d coordinates).
animation = pt.animate(traj_list)
```

8 Sending the Coordinates to the Simulation

From section 6 we are given a filtered movement in real-time. We replicate this on the (simulated) robot in real-time. For this, we publish the finger joint coordinates to a rostopic called

```
"/human_finger_points"
```

The message type of this rostopic is *PointCloud* from sensor_msgs.msg, meaning that all the finger-points have to be converted into *geometry_msgs/Point* objects first. Pass the classified gesture via *PointCloud.header.frame_id*. The topic is getting subscribed by a rosnode, which takes all the points and transforms them into the robots reference frame. If the hand gesture is equal to ..., the point of the index finger will be published to a Cartesian impedance controller which controls the robots end effector. This way you can make the robot follow your index finger if your hand is in a pointing position.

References

MMPose Contributors (Aug. 2020). *OpenMMLab Pose Estimation Toolbox and Benchmark*. original-date: 2020-07-08T06:02:55Z. URL: https://github.com/open-mmlab/mmpose (visited on 10/12/2023).