CV2_Lab2_LuisRuanovaLea

April 11, 2023

0.1 Computer Vision 2 - Lab. 2

Luis Ruanova Lea

Importing libraries:

[2]:
```
import sys,os
import cv2
import numpy as np
#---
import os
import cv2
import numpy as np
from sklearn.decomposition import PCA
from sklearn.decomposition import KernelPCA
from sklearn.preprocessing import StandardScaler
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

The following class is used to generate optical flow representations from an input video file. Appart from the input video file and the output directory where the generated frames will be saved, the other parameters (such as fillsquares, vec_weight and square_mult) are for the optical flow computation.

In the init method, it first checks that the output directory exists and if not it creates it, and gets the frames of the input video. Two output video writters are also created, one for the compressed optical flow (dense one, catching the movement of each pixel in the frames) and another for the sparse (*sflow.avi* , focusing only on a subset of pixels).

The *draw_flow()* method takes the computed optical flow and the previous gray frame as input and generates the visualizations by drawing "motion" vectors or rectangles. The *run()* method reads frames from the input video file, computes optical flow using Farneback's method, and calls the draw_flow() method to generate optical flow visualizations for each frame.

[3]:
```
class FBackMVFITemplates:
    def __init__(self, vidfile,  fillsquares, vec_weight, square_mult,␣
 ↪output_dir):
        self.capture = cv2.VideoCapture(vidfile)
```

```
        self.fillsquares = int(fillsquares)
        self.vec_weight = int(vec_weight)
        self.square_mult = int(square_mult)
        self.mv_step = 8
        self.mv_scale = 1.5
        self.mv_color = (0, 255, 0)
        self.cflow = None
        self.sflow = None
        self.flow = None
        self.frame_count = 0
        self.output_dir =  output_dir

        os.makedirs(self.output_dir, exist_ok=True)
        fps = self.capture.get(cv2.CAP_PROP_FPS)

        self.frame_size = (100, 100)
        ret, frame = self.capture.read()
        if ret:
            self.frame_size = (frame.shape[1], frame.shape[0])

        print(fps, self.frame_size)

        fourcc = cv2.VideoWriter_fourcc(*'XVID')
        self.cwriter = cv2.VideoWriter(self.output_dir + "/cflow.avi", fourcc,␣
↪fps, self.frame_size, True)
        self.swriter = cv2.VideoWriter(self.output_dir + "/sflow.avi", fourcc,␣
↪fps, self.frame_size, True)

        self.sflow = np.zeros((self.frame_size[1], self.frame_size[0], 3),␣
↪dtype=np.uint8)
        #cv2.namedWindow("Optical Flow", cv2.WINDOW_NORMAL)

    def draw_flow(self, flow, prevgray):

        # print flow.height, flow.width
        self.cflow = cv2.cvtColor(prevgray, cv2.COLOR_GRAY2BGR)
        self.cflow = np.zeros_like(prevgray)
        self.sflow = np.zeros((prevgray.shape[0], prevgray.shape[1], 3),␣
↪dtype=np.uint8)
        for y in range(0, flow.shape[0], self.mv_step):
            for x in range(0, flow.shape[1], self.mv_step):
                fx, fy = flow[y, x]
                if fx * fx + fy * fy > 4:
                    print(x, y, fx * fx + fy * fy)
                    cv2.line(self.cflow, (x, y), (x + int(fx), y + int(fy)),␣
↪self.mv_color, 1, cv2.LINE_AA)
                    # Circle(self.cflow, (x,y), 2, self.mv_color, -1)
```

```
                    pt = []
                    dx = int(x + self.square_mult * fx)
                    dy = int(y + self.square_mult * fy)
                    print("points:", dx, dy)
                    pt.append((x, y))
                    pt.append((dx, y))
                    pt.append((dx, dy))
                    pt.append((x, dy))
                    if self.fillsquares:
                        buf = []
                        for i in range(len(pt)):
                            buf.append(pt[i])
                        cbar = self.vec_weight * round(np.sqrt(fx * fx + fy *␣
↪fy))
                        # here is to generate a pseudocolor image...
                        gray  = cbar / 255.0
                        red   = round(255 * np.fabs(np.sin(gray * 2 + 0.0 * np.
↪pi)))
                        green = round(255 * np.fabs(np.sin(gray * 2 - 0.1 * np.
↪pi)))
                        blue  = round(255 * np.fabs(np.sin(gray * 2 - 0.3 * np.
↪pi)))
                        cv2.fillConvexPoly(self.sflow, np.array(buf), (red,␣
↪green, blue), cv2.LINE_AA, 0)
                    else:
                        cv2.rectangle(self.sflow, pt[1], pt[3], 50 * round(fx *␣
↪fx + fy * fy), -1)

                    # so that it comes out in gray:
        cv2.imshow("Flow", self.sflow)
        cv2.imshow("Optical Flow", self.cflow)


        # Check the length of self.frame_count and pad with zeros accordingly
        if self.frame_count < 10:
            padded_frame_count = f"000{self.frame_count}"
        elif self.frame_count < 100:
            padded_frame_count = f"00{self.frame_count}"
        elif self.frame_count < 1000:
            padded_frame_count = f"0{self.frame_count}"
        else:
            padded_frame_count = f"{self.frame_count}"

        filename = f"{self.output_dir}/frame_{padded_frame_count}.png"

        cv2.imwrite(filename, self.sflow)
```

```
        self.cwriter.write(self.cflow)
        self.swriter.write(self.sflow)
        self.frame_count += 1

    def run(self):
        first_frame = True
        ix = 0
        while True:
            ret, frame = self.capture.read()
            print(ix, "-----")
            if ret:
                if first_frame:
                    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
                    prev_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
                    flow = np.zeros_like(frame)
                    self.cflow = np.zeros_like(frame)

                gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
                if not first_frame:
                    flow = cv2.calcOpticalFlowFarneback(prev_gray, gray, flow,
                                                        pyr_scale=0.5,␣
↪levels=3, winsize=15,
                                                        iterations=3, poly_n=5,␣
↪poly_sigma=1.2, flags=0)

                    self.draw_flow(flow, prev_gray)
                    c = cv2.waitKey(7)
                    if c in [27, ord('q'), ord('Q')]:
                        break

                prev_gray = gray
                ix = ix + 1
                first_frame = False
            else:
                break

            k = cv2.waitKey(30) & 0xff
            if k == 27:
                break
        # Release video capture and close windows
        self.capture.release()
        cv2.destroyAllWindows()
```

Here different useful functions are defined: *load_images_from_directory()* : loads images from a directory recursively, and returns the loaded images as well as corresponding labels. *flatten_images()* : having a list of images as input, returns another list of flattened ones. *plot_pc_lda()* : creates

a scatter plot of PCA (Principal Component Analysis) and LDA (Linear Discriminant Analysis) *plot_3d_pca()* : creates a 3D scatter plot of PCA data points. *analysis_pca()*, *analysis_pca_lda()* and *analysis_kernelpca()* : performs PCA, PCA and LDA or kernel PCA analysis, respectively, over the training data.

[4]:
```
def load_images_from_directory(directory_path, target_size=(100, 100)):
    images = []
    labels = []
    for action_dir in os.listdir(directory_path):
        action_path = os.path.join(directory_path, action_dir)
        if os.path.isdir(action_path):
            for sequence_dir in os.listdir(action_path):
                sequence_path = os.path.join(action_path, sequence_dir)
                if os.path.isdir(sequence_path):
                    for file in os.listdir(sequence_path):
                        if file.endswith((".jpg", ".jpeg", ".png")):
                            img = cv2.imread(os.path.join(sequence_path, file),␣
 ↪cv2.IMREAD_GRAYSCALE)
                            img = cv2.resize(img, target_size)
                            images.append(img)
                            labels.append(action_dir)
    return images, labels

def flatten_images(images):
    return [img.flatten() for img in images]


def plot_pca_lda(pca_data, lda_data, labels):
    fig, axes = plt.subplots(1, 2, figsize=(12, 6))
    axes.scatter(pca_data[:, 0], pca_data[:, 1], c=labels, cmap='viridis')
    axes[0].set_title('PCA')
    axes[1].scatter(lda_data[:, 0], lda_data[:, 1], c=labels, cmap='viridis')
    axes[1].set_title('LDA')
    plt.show()

def plot_3d_pca(pca_data, labels):
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    unique_labels = list(set(labels))
    colors = plt.cm.jet(np.linspace(0, 1, len(unique_labels)))

    for label, color in zip(unique_labels, colors):
        indices = [i for i, lbl in enumerate(labels) if lbl == label]
        ax.scatter(pca_data[indices, 0], pca_data[indices, 1],␣
 ↪pca_data[indices, 2], c=[color], label=label)
    ax.set_xlabel('PC1')
    ax.set_ylabel('PC2')
```

```
    ax.set_zlabel('PC3')
    ax.legend()
    plt.show()



def analysis_pca(X_train, y_train):
    pca = PCA(n_components=3)
    pca_data = pca.fit_transform(X_train)
    print(pca_data.shape)
    pca = PCA(n_components=3)
    pca_data = pca.fit_transform(X_train)
    plot_3d_pca(pca_data, y_train)

def analysis_pca_lda(X_train, y_train):
    pca = PCA(n_components=3)
    pca_data = pca.fit_transform(X_train)
    print(pca_data.shape)
    pca = PCA(n_components=3)
    pca_data = pca.fit_transform(X_train)
    lda = LDA(n_components=2)
    lda_data = lda.fit_transform(X_train, y_train)
    print(lda_data.shape)
    print(lda_data)
    plot_pca_lda(pca_data, lda_data, y_train)


def analysis_kernelpca(X_train, y_train):
    # Preprocess the data using StandardScaler
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    # Perform Kernel PCA with 3 components and an RBF kernel
    # kernel: 'linear', 'poly', 'sigmoid', 'rbf'
    # gamma parameter can be adjusted as needed
    kpca = KernelPCA(n_components=3, kernel='rbf', gamma=0.1)
    kpca_data = kpca.fit_transform(X_train_scaled)
    plot_3d_pca(kpca_data, y_train)
```

Lastly, when the script is running as main module, the input image path, images and labels are loaded, flattened into feature vectors, and LDA analysis is perfoemd to reduce the dimensionality of the feature vectors to 2 components. Then PCA is performed and transformed vectors are used to perform kernel PCA analysis and visualize the results.

[ ]:
```
if __name__ == "__main__":
    directory_path = "./actiondata/images"
    target_size = (100, 100)
    num_components=2
```

```
    images, labels = load_images_from_directory(directory_path, target_size)
    feature_vectors = flatten_images(images)
    X_train = feature_vectors
    y_train = int_labels = [int(label[1:]) for label in labels]
    lda = LDA()
    X_lda = lda.fit_transform(X_train, y_train)
    pca = PCA(n_components=num_components)
    X_pca = pca.fit_transform(X_lda)
    analysis_kernelpca(X_pca, y_train)
```

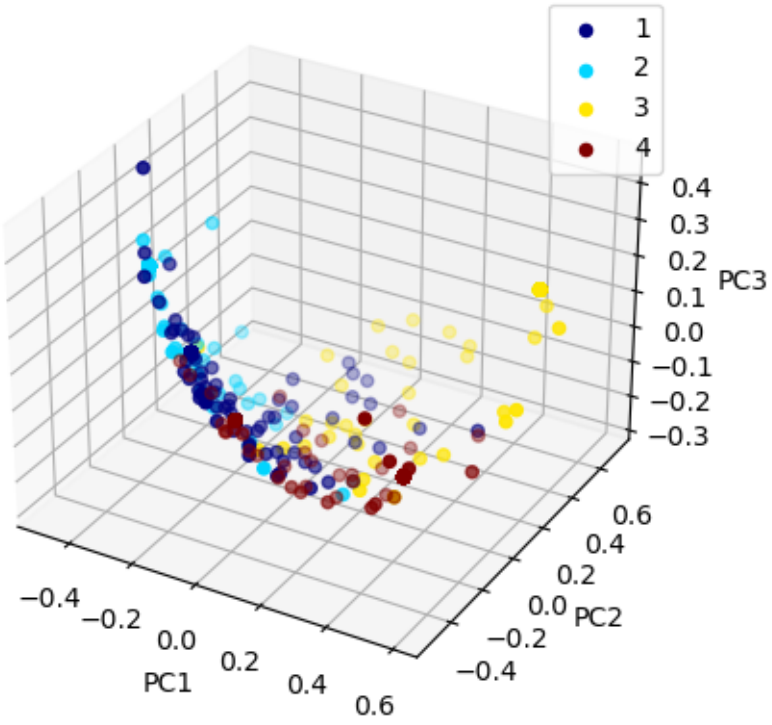
0.1.1 Testing with provided datasets

First 3 videos are about a person walking in different ways. Video A4, a person grabing an imaginary ball. Video A5, a person entering the scene, lying in the ground for a momment and walking off the scene before. A6 is about a person falling down to the floor, and A7 about a person that sits on a chair.

With just 4 first videos:

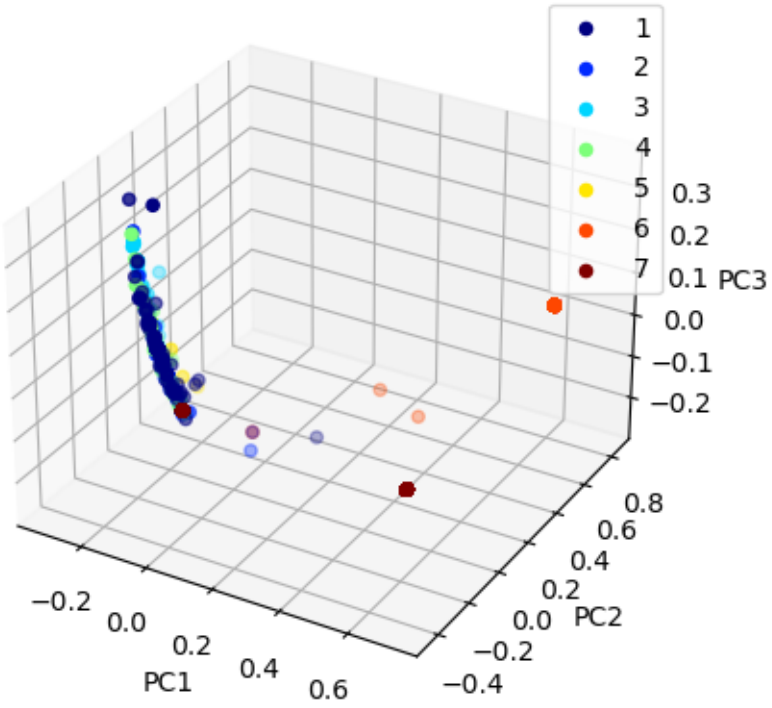
[ ]:
```
capture =  './actiondata/videos/A7/P1/s1.avi'
fillsquares = 1
vec_weight = 10
square_mult = 5
output_dir = "./actiondata/images/A7/P1_1"

M = FBackMVFITemplates(capture, fillsquares, vec_weight, square_mult,␣
 ↪output_dir)
M.run()
```

[6]:
```
if __name__ == "__main__":
    directory_path = "./actiondata/images"
    target_size = (100, 100)
    num_components=2
    images, labels = load_images_from_directory(directory_path, target_size)
    feature_vectors = flatten_images(images)
    X_train = feature_vectors
    y_train = int_labels = [int(label[1:]) for label in labels]
    lda = LDA()
    X_lda = lda.fit_transform(X_train, y_train)
    pca = PCA(n_components=num_components)
    X_pca = pca.fit_transform(X_lda)
    analysis_kernelpca(X_pca, y_train)
```



With the 7:

[18]:
```
if __name__ == "__main__":
    directory_path = "./actiondata/images"
    target_size = (100, 100)
    num_components=2
    images, labels = load_images_from_directory(directory_path, target_size)
    feature_vectors = flatten_images(images)
    X_train = feature_vectors
    y_train = int_labels = [int(label[1:]) for label in labels]
    lda = LDA()
    X_lda = lda.fit_transform(X_train, y_train)
    pca = PCA(n_components=num_components)
    X_pca = pca.fit_transform(X_lda)
    analysis_kernelpca(X_pca, y_train)
```



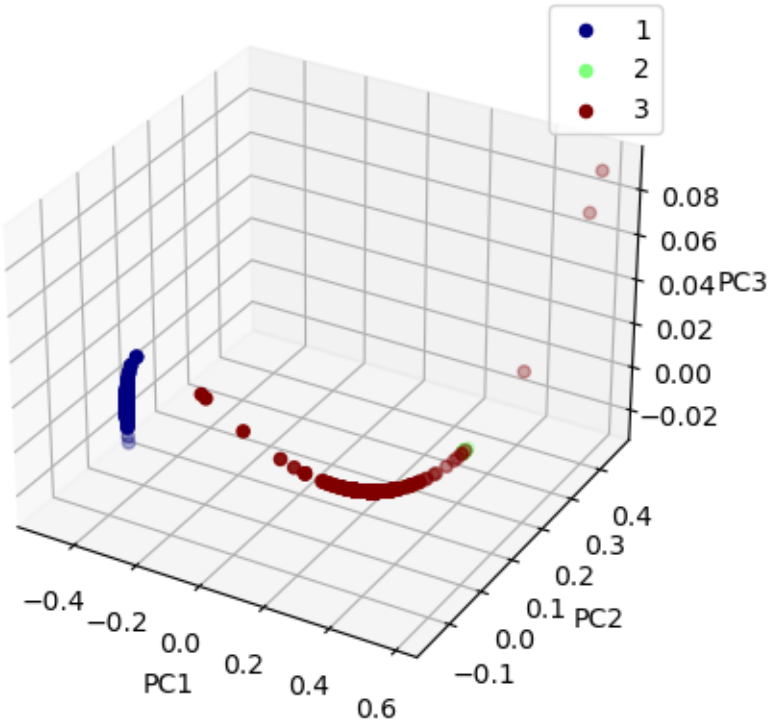
It seems that last videos have different characteristics and quite different feature vectors are find, so while the first four videos are quite together in the plot, the points corresponding to fifth, six and seventh videos are more sparse.

Testing with testing videos, X1 of cars in a highway, X2 of a racing car followed by the camera and X3 of a normal road with cars going slowly and moving leafs and shadows:

[ ]:
```
capture =  './tstvideos/highway_test.mp4'
fillsquares = 1
vec_weight = 10
square_mult = 5
output_dir = "./actiondata/testimages/X3/X1_1"

M = FBackMVFITemplates(capture, fillsquares, vec_weight, square_mult,␣
 ↪output_dir)
M.run()
```

[37]:
```
if __name__ == "__main__":
    directory_path = "./actiondata/testimages"
    target_size = (100, 100)
    num_components=1
    images, labels = load_images_from_directory(directory_path, target_size)
    feature_vectors = flatten_images(images)
    X_train = feature_vectors
```

```
    y_train = int_labels = [int(label[1:]) for label in labels]
    lda = LDA()
    X_lda = lda.fit_transform(X_train, y_train)
    pca = PCA(n_components=num_components)
    X_pca = pca.fit_transform(X_lda)
    analysis_kernelpca(X_pca, y_train)
```



Simple and smooth actions with predictable patterns, such as walking, may result in accurate motion estimation. However, complex actions with rapid changes, such as the video where the man suddenly falls into the ground, may be challenging to the algorithm. Also, videos with changing or poor lighting conditions, such as changing illumination, shadows, or reflections, may be challenging for the optical flow method. The complexity of backgrounds in videos can also affect the performance of the optical flow method. Videos with simple and static backgrounds, such as the walking person, may result in accurate motion estimation, and those with moving backgrounds (such as racing car or leaves moving) may cause the algorithm to struggle when distinguing between the action beeing performed and the background.

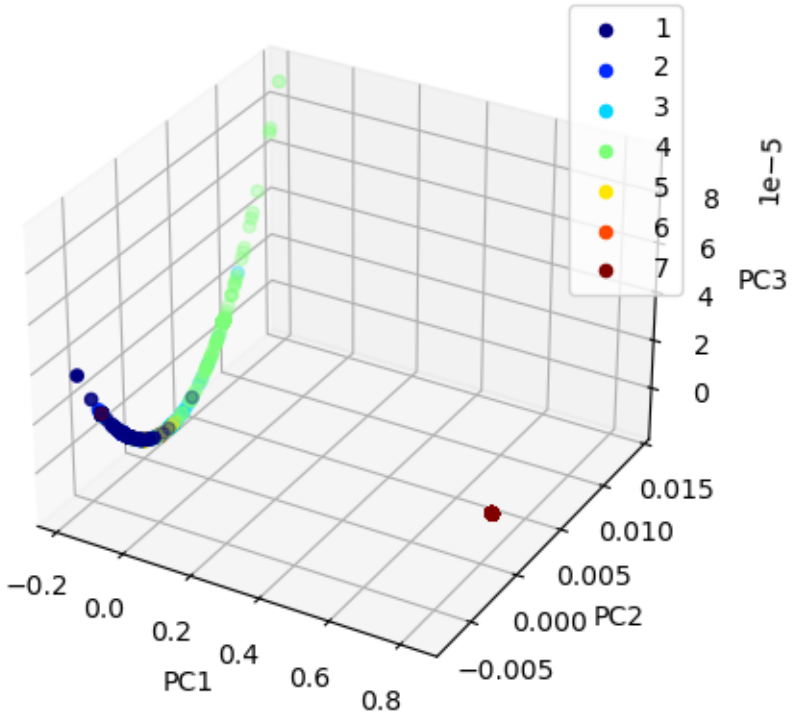
0.1.2 Modification

One simple modification that could potentially improve the performance of the optical flow algorithm is to apply Gaussian blur to the video frames before computing the optical flow. This can be achieved using the cv2.GaussianBlur() function from OpenCV. Gaussian blur can help to reduce noise in the video frames, which can improve the accuracy of optical flow estimation.

[38]:
```
def new_run(self):
    first_frame = True
    ix = 0
    while True:
        ret, frame = self.capture.read()
        print(ix, "-----")
        if ret:
            if first_frame:
                gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
                prev_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
                flow = np.zeros_like(frame)
                self.cflow = np.zeros_like(frame)

            gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

            ksize = (5, 5)
            sigma = 0
            gray = cv2.GaussianBlur(gray, ksize, sigma)
            prev_gray = cv2.GaussianBlur(prev_gray, ksize, sigma)

            if not first_frame:
                flow = cv2.calcOpticalFlowFarneback(prev_gray, gray, flow,
                                                    pyr_scale=0.5, levels=3,␣
 ↪winsize=15,
                                                    iterations=3, poly_n=5,␣
 ↪poly_sigma=1.2, flags=0)

                self.draw_flow(flow, prev_gray)
                c = cv2.waitKey(7)
                if c in [27, ord('q'), ord('Q')]:
                    break

            prev_gray = gray
            ix = ix + 1
            first_frame = False
        else:
            break

        k = cv2.waitKey(30) & 0xff
        if k == 27:
            break
    # Release video capture and close windows
    self.capture.release()
    cv2.destroyAllWindows()

FBackMVFITemplates.run = new_run
```

[ ]:
```
capture =  './actiondata/videos/A7/P1/s1.avi'
fillsquares = 1
vec_weight = 10
square_mult = 5
output_dir = "./actiondata/images/A7/P1_1"

M = FBackMVFITemplates(capture, fillsquares, vec_weight, square_mult,␣
 ↪output_dir)
M.run()
```

[48]:
```
if __name__ == "__main__":
    directory_path = "./actiondata/images"
    target_size = (100, 100)
    num_components=1
    images, labels = load_images_from_directory(directory_path, target_size)
    feature_vectors = flatten_images(images)
    X_train = feature_vectors
    y_train = int_labels = [int(label[1:]) for label in labels]
    lda = LDA()
    X_lda = lda.fit_transform(X_train, y_train)
    pca = PCA(n_components=num_components)
    X_pca = pca.fit_transform(X_lda)
    analysis_kernelpca(X_pca, y_train)
```



After applying the gaussian blur, most of the videos seem to fit into a curve (except for the 7th one, the one with the chair, which can mean that the video pressents quite different patterns respect to the others). This suggests that the videos share some common patterns or behaviors that are captured by the blurred optical flow map, resulting in a more consistent representation of the overall movement after applying this technique in order to remove the noise. However, some fine-grained details are losed in the videos, and for example hands and toes seems to be more generally captured.