Companion Paper on GolfPose: Implementation and Reproducibility Notes

Ming-Han Lee, Yu-Chen Zhang, Kun-Ru Wu, and Yu-Chee Tseng

Department of Computer Science
National Yang Ming Chiao Tung University
No.1001 University Road, Hsinchu, Taiwan
{mhlee.cs09, yuchen2856.cs10, wufish, yctseng}@nycu.edu.tw

Abstract. This is the companion paper to the ICPR 2024 Paper "GolfPose: From Regular Posture to Golf Swing Posture". It details our *GolfSwing* dataset and *GolfPose* model. For the *GolfSwing* dataset, we introduce the equipment used, data recording steps, and post-processing methods. Regarding the *GolfPose* models, we discuss parameters for fine-tuning both object detectors and 2D/3D pose models, along with integration methods for inference. Finally, we present metrics to evaluate the performance of 2D and 3D pose models and compare the 2D pose model with a pre-trained model.

Keywords: Pose Estimation · Dataset Creation · Golf · Reproducibility.

1 Introduction

In our paper [6], we introduce the *GolfSwing* dataset, which comprises both 2D and 3D golfer-with-club postures, as well as *GolfPose* models that are fine-tuned from 2D and 3D pose models using the *GolfSwing* dataset. The source code is available at [1]. This companion paper focuses on the construction process of the *GolfSwing* dataset, details of fine-tuning *GolfPose*, and evaluation metrics for the models. Section 2 describes the five steps in creating the dataset, including used equipment of the MoCap (motion capture) system and DV (digital video) system, the process of coordinate transformation, and the generation of bounding boxes and keypoints for golfer with club. Section 3 presents the implementation details and loss functions for fine-tuning object detectors, 2D pose models, and a 2D-3D pose lifter, as well as necessary processing steps for inference across all models. Section 4 explains the metrics used to evaluate 2D and 3D pose models and provides details on comparing our GolfPose-2D pose model with the pre-trained pose model.

[1] https://github.com/MingHanLee/GolfPose

2 Dataset

The creation process of the *GolfSwing* dataset includes 5 steps: environment setup, data recording, post-processing, coordinate transformation, and creating the dataset. As shown in Fig. 1

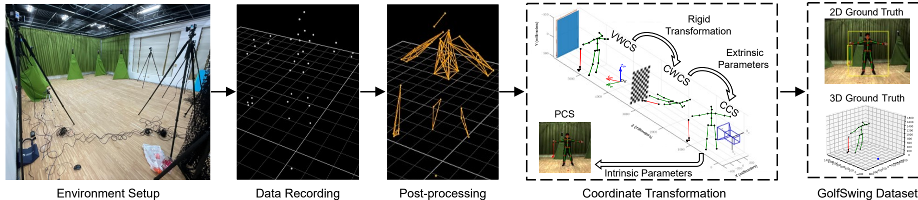

Fig. 1: Workflow to derive the *GolfSwing* dataset.

Environment setup. *GolfSwing* dataset is collected concurrently by a MoCap (motion capture) system and DV (digital video) system. The equipment specifications are shown in Table 1. This step involves setting up the camera, calibrating and synchronizing all Vicon cameras with the Vicon active wand, and establishing a 3D global coordinate origin.

Table 1: Equipment specifications

	MoCap System		DV System	
Camera Type	Vicon Vero v2.2 (x6)	Vicon Vero v1.3 (x3)	Vicon Vue (x1)	iRAYPLE A5201CU150E (x1)
Resolution	2048 x 1088	1280 x 1024	1920 x 1080	1920 x 1200
Frame Rate	100 Hz	100 Hz	100 Hz	150 Hz
Time Sync.	Hardware	Hardware	Hardware	-

Data recording. After the environment setup is complete, we start recording golf swing movements from six volunteers acting as golfers. The MoCap system captures the 3D trajectories of 28 markers placed on each volunteer and 5 markers on the club, while the DV system simultaneously records a 2D video sequence.

Post-processing. Even though each golf swing was recorded by all cameras at the same time, a marker must be captured by at least two infrared cameras to reconstruct its 3D location. However, markers may still be occluded or lost during a swing. Once the raw data was recorded, we had to use Vicon Nexus's algorithms and manual inspection to fill in the 3D location of missing markers.

Coordinate transformation. We integrate the coordinate system from MoCap and Zhang's camera calibration algorithm [10] to define four coordinate systems: Vicon World Coordinate System (VWCS), Checkerboard World Coordinate System (CWCS), Camera Coordinate System (CCS), Pixel Coordinate System (PCS).

```python
def rigid_transform_3D(A, B):
    assert len(A) == len(B)
    N = A.shape[0]
    mu_A = np.mean(A, axis=0)
    mu_B = np.mean(B, axis=0)

    AA = A - np.tile(mu_A, (N, 1))
    BB = B - np.tile(mu_B, (N, 1))
    H = np.transpose(AA) * BB

    U, S, Vt = np.linalg.svd(H)
    R = Vt.T * U.T

    if np.linalg.det(R) < 0:
        print("Reflection detected")
        Vt[2, :] *= -1
        R = Vt.T * U.T
    t = -R * mu_A.T + mu_B.T

    return R, t

(ret_R, ret_T) = rigid_transform_3D(p_vicon, p_cb)
```

Listing 1: Compute the rigid transformation

Fig. 2: One image with pattern drawn for camera calibration

Listing 1 defines a function that computes the rigid transformation (rotation and translation) between VWCS and CWCS. Specifically, as shown by the yellow arrows in Fig. 2, three markers are placed on the chessboard, and these markers are represented as 3D points in both VWCS and CWCS. The two sets of 3D points are utilized to compute the rigid transformation between these two coordinate systems.

```python
rows = 12
cols = 8
mm = 90
criteria = (cv2.TERM_CRITERIA_MAX_ITER + cv2.TERM_CRITERIA_EPS, 30, 0.001)
objectPoints = np.zeros((rows * cols, 3), np.float32)
objectPoints[:, :2] = np.mgrid[0 : rows * mm : mm, 0 : cols * mm : mm].T.reshape(-1, 2)

glob_image = sorted(glob.glob(os.path.join(cb_image, "*")))
for idx, path in enumerate(glob_image):
    # images.append(os.path.basename(path))
    img = cv2.imread(path)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Find the chess board corners
    ret, corners = cv2.findChessboardCorners(gray, (rows, cols), None)

    if ret:
        # Refine the corner position
        corners = cv2.cornerSubPix(gray, corners, (11, 11), (-1, -1), criteria)
        objectPointsArray.append(objectPoints)
        imgPointsArray.append(corners)
        img_dict[os.path.basename(path)] = len(objectPointsArray) - 1

        cv2.drawChessboardCorners(img, (rows, cols), corners, ret)
        img = cv2.circle(img, tuple(corners[0].ravel().astype(np.int32)), 12, (0, 0, 255), -1)
    else:
        print(f"image {path} can't be calibrated.")

# Calibrate the camera and save the results
ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objectPointsArray, imgPointsArray, gray.shape[::-1], None
    , None)
```

Listing 2: Camera calibration

Listing 2 is the code of Zhang's camera calibration algorithm from OpenCV [1]. As shown in Fig. 2, we use a chessboard, consisting of 13 columns and 9 rows, with each square measuring 90 mm on each side. This configuration results in 12x8 internal corners within the board. In order to calibrate camera, a total amount of 48 images are collected and converted to grayscale. Then, we use `cv2.findChessboardCorners` to find chessboard corners and refine them. Both

the object points and image points are stored. With these stored points, we proceed with calibration by using `cv2.calibrateCamera`. This function returns parameters, such as the camera matrix, distortion coefficients, rotation and translation vectors, etc. These parameters, known as external and internal parameters, are utilized for the conversion from CWCS to CCS, and finally to PCS.

Creating Dataset Via MoCap system, the above steps determine the ground truth of 3D keypoints. Then, through coordinate transformation, the 3D keypoints can be projected onto each image to produce the ground truth of the 2D keypoints.

```
def calc_bboxes(kps_2d, shift=0):
    # bboxes: [x1, y1, x2, y2] => coco bbox [x, y , width, height]
    bboxes = np.stack([np.min(kps_2d[:, :, 0], axis=1), np.min(kps_2d[:, :, 1], axis=1), np.max(kps_2d[:, :,
        0], axis=1), np.max(kps_2d[:, :, 1], axis=1)], axis=1)

    if shift != 0:
        bboxes[:, 0] = bboxes[:, 0] - shift*1
        bboxes[:, 1] = bboxes[:, 1] - shift*1
        bboxes[:, 2] = bboxes[:, 2] + shift*1.2
        bboxes[:, 3] = bboxes[:, 3] + shift*1.2

    bboxes_coco = np.stack([bboxes[:, 0], bboxes[:, 1], (bboxes[:, 2] - bboxes[:, 0]), (bboxes[:, 3] -
      bboxes[:, 1])], axis=1)
    area = bboxes_coco[:, 2] * bboxes_coco[:, 3]

    centers = np.stack([(bboxes[:, 0] + bboxes[:, 2]) / 2, (bboxes[:, 1] + bboxes[:, 3]) / 2], axis=1)

    return bboxes_coco, area, centers

person_bboxes, person_area, golfer_centers= calc_bboxes(person_kps, shift=50)
club_bboxes, club_area, club_centers = calc_bboxes(club_kps, shift=15)
```

Listing 3: Caculate the bounding box

After obtaining the 2D keypoints, we use these keypoints to calculate the bounding boxes for both the golfer and the club. In the Listing 3, we first determine the initial bounding box by finding the minimum and maximum coordinates of all keypoints. Then, we apply a 50-pixel shift for the golfer and a 15-pixel shift for the club to ensure that the bounding box fully covers the object. These bounding boxes will be used to fine-tune our object detectors and pose models.

3 Implementation

GolfPose contains an object detector, 2D pose model, and 2D-3D lifting models. In the beginning, the object detector detects the golfer and the club; then the 2D pose model estimates their 2D keypoints. Finally, the 2D-3D lifting model is adopted to convert 2D keypoints to 3D coordinates. Implementation details are provided below.

3.1 Object detector

MMDetection[2] is adopted as object detector with [2] version 3.1.0. It is an open-source toolbox based on PyTorch. The feature of MMDetection is its modular design, which supports dataset construction as well as model invocation and modification.

```
# model settings
load_from = 'yolox_s_8x8_300e_coco_20211121_095711-4592a793.pth'
model = dict(
```

```
    type='YOLOX',
    data_preprocessor=dict(...
    backbone=dict(...
    neck=dict(...
    bbox_head=dict(
        type='YOLOXHead',
        num_classes=2,
        ...),
)

# dataset settings
data_root = 'GolfSwing'
dataset_type = 'CocoDataset'
classes = ('person', 'club')

# training settings
max_epochs = 30
```

Listing 4: Object Detector config file

Listing 4 is our object detector config file. Here, we take YOLOx from MMDetection as an example. In order to detect both the golfer and the club separately, we load the detector which is pretrained on the COCO dataset using `load_from`. Then, set `model.bbox_head.num_classes=2` and specify `classes=('person', 'club')`. The detector is fine-tuned for 30 epochs to detect bounding boxes for both person and club. If we want to detect them jointly, we should set `model.bbox_head.num_classes=1` and `classes=('golfer')`. Then, fine-tune the detector for 30 epochs to identify bounding boxes for the single object "Golfer-with-club".

3.2 2D pose models

With the bounding boxes of the golfer and club objects, the next step is to estimate the 2D keypoints of them. We train Golfer, Club, and Golfer-with-club pose models separately by MMPose [3] with version 1.3.0.

```
dataset_info = dict(
    keypoint_info=dict(
        {
            0: dict(name="root", id=0, color=[51, 153, 255], type="lower", swap=""),
            1: dict(name="right_hip", id=1, color=[255, 128, 0], type="lower", swap="left_hip"),
            2: dict(name="right_knee", id=2, color=[255, 128, 0], type="lower", swap="left_knee"),
            3: dict(name="right_foot", id=3, color=[255, 128, 0], type="lower", swap="left_foot"),
            4: dict(name="left_hip", id=4, color=[0, 255, 0], type="lower", swap="right_hip"),
            5: dict(name="left_knee", id=5, color=[0, 255, 0], type="lower", swap="right_knee"),
            6: dict(name="left_foot", id=6, color=[0, 255, 0], type="lower", swap="right_foot"),
            7: dict(name="spine", id=7, color=[51, 153, 255], type="upper", swap=""),
            8: dict(name="thorax", id=8, color=[51, 153, 255], type="upper", swap=""),
            9: dict(name="neck_base", id=9, color=[51, 153, 255], type="upper", swap=""),
            10: dict(name="head", id=10, color=[51, 153, 255], type="upper", swap=""),
            11: dict(name="left_shoulder", id=11, color=[0, 255, 0], type="upper", swap="right_shoulder"),
            12: dict(name="left_elbow", id=12, color=[0, 255, 0], type="upper", swap="right_elbow"),
            13: dict(name="left_wrist", id=13, color=[0, 255, 0], type="upper", swap="right_wrist"),
            14: dict(name="right_shoulder", id=14, color=[255, 128, 0], type="upper", swap="left_shoulder"),
            15: dict(name="right_elbow", id=15, color=[255, 128, 0], type="upper", swap="left_elbow"),
            16: dict(name="right_wrist", id=16, color=[255, 128, 0], type="upper", swap="left_wrist"),
            17: dict(name="shaft", id=17, color=[255, 255, 255], type="upper", swap=""),
            18: dict(name="hosel", id=18, color=[255, 255, 255], type="lower", swap=""),
            19: dict(name="heel", id=19, color=[255, 255, 255], type="lower", swap=""),
            20: dict(name="toe_down", id=20, color=[255, 255, 255], type="lower", swap=""),
            21: dict(name="toe_up", id=21, color=[255, 255, 255], type="lower", swap=""),
        }
    ),
    skeleton_info=dict(
        {
            0: dict(link=("root", "left_hip"), id=0, color=[0, 255, 0]),
            1: dict(link=("left_hip", "left_knee"), id=1, color=[0, 255, 0]),
            2: dict(link=("left_knee", "left_foot"), id=2, color=[0, 255, 0]),
            3: dict(link=("root", "right_hip"), id=3, color=[255, 128, 0]),
            4: dict(link=("right_hip", "right_knee"), id=4, color=[255, 128, 0]),
            5: dict(link=("right_knee", "right_foot"), id=5, color=[255, 128, 0]),
            6: dict(link=("root", "spine"), id=6, color=[51, 153, 255]),
            7: dict(link=("spine", "thorax"), id=7, color=[51, 153, 255]),
            8: dict(link=("thorax", "neck_base"), id=8, color=[51, 153, 255]),
```

```
            9: dict(link=("neck_base", "head"), id=9, color=[51, 153, 255]),
            10: dict(link=("thorax", "left_shoulder"), id=10, color=[0, 255, 0]),
            11: dict(link=("left_shoulder", "left_elbow"), id=11, color=[0, 255, 0]),
            12: dict(link=("left_elbow", "left_wrist"), id=12, color=[0, 255, 0]),
            13: dict(link=("thorax", "right_shoulder"), id=13, color=[255, 128, 0]),
            14: dict(link=("right_shoulder", "right_elbow"), id=14, color=[255, 128, 0]),
            15: dict(link=("right_elbow", "right_wrist"), id=15, color=[255, 128, 0]),
            16: dict(link=("shaft", "hosel"), id=16, color=[255, 255, 255]),
            17: dict(link=("hosel", "heel"), id=17, color=[255, 255, 255]),
            18: dict(link=("heel", "toe_down"), id=18, color=[255, 255, 255]),
            19: dict(link=("toe_down", "toe_up"), id=19, color=[255, 255, 255]),
        }
    ),
    joint_weights=[1.0, 1.0, 1.2, 1.5, 1.0, 1.2, 1.5, 1.0, 1.0, 1.0, 1.0, 1.0, 1.2, 1.5, 1.0, 1.2, 1.5, 1.6,
       1.9, 2.0, 2.0, 2.0],
    sigmas=[0.02] * 22,
)
```

Listing 5: *GolfSwing* Dataset - Golfer-with-club config file

Taking the Golfer-with-club model as an example, Listing 5 is the Golfer-with-club dataset config file. It includes 22 keypoints, skeleton links, weights for each keypoint, and sigma values for calculating OKS (Object Keypoints Similarity). 17 of these keypoints are based on Human3.6M for the golfer, and 5 are custom-defined for the club: shaft, hosel, heel, toe down, and toe up. The keypoints farther from the root have greater weight due to the faster movement speed. The keypoints of club experience more significant speed changes and are therefore given higher weights than the keypoints of the golfer. Additionally, we use a Motion Capture System to record the ground truth of 3D keypoints and convert them to 2D keypoints using camera parameters. The converted 2D keypoints are more precise than manual annotations. Therefore, the sigma is set to 0.02, which is stricter than used in the COCO dataset.

```
# model settings
load_from = 'td-hm_hrnet-w48_8xb32-210e_coco-384x288-c161b7de_20220915.pth'
model = dict(
    type='TopdownPoseEstimator',
    data_preprocessor=dict(...
    backbone=dict(...
    head=dict(
        type='HeatmapHead',
        in_channels=48,
        out_channels=22,
        deconv_out_channels=None,
        loss=dict(type='KeypointMSELoss', use_target_weight=True),
        decoder=codec),
)

# dataset settings
dataset_type = 'CocoDataset'
metainfo = dict(from_file='configs/mmpose/_base_/datasets/golfswing_golfer.py')
data_mode = 'topdown'

# training settings
train_cfg = dict(max_epochs=20, val_interval=1)
```

Listing 6: *GolfPose*-2D(GC) Model config file

In the *GolfPose*-2D(GC) model config file 6, we set the output channels to 22 to estimate the 22 custom keypoints of Golfer-with-club (abbreviated as GC) and import the dataset config file 5. By loading a pre-trained model and fine-tuning it over 20 epochs, we can estimate these keypoints for the Golfer-with-club.

```
from mmpose.apis import _track_by_iou, _track_by_oks,

if args.use_oks_tracking:
    _track = partial(_track_by_oks)
else:
    _track = _track_by_iou

track_id, pose_est_results_last, _ = _track(data_sample, pose_est_results_last, args.tracking_thr, sigmas=
      pose_estimator.dataset_meta['sigmas'])
```

Listing 7: Object tracking

After estimating the 2D keypoints for each frame, we utilize the MMPose API to calculate IOU and OKS. This ensures consistent object tracking and 2D keypoint extraction across all frames.

```
def merge_person_club(person_pose_est_results_list, golf_club_pose_est_results_list):
    assert len(person_pose_est_results_list) == len(golf_club_pose_est_results_list), f"The quantity not
      equal."

    kps_all = []
    bboxes_all = []
    for idx, _ in enumerate(person_pose_est_results_list):
        # bbox
        bboxes = np.stack((person_pose_est_results_list[idx][0].get('pred_instances').get('bboxes'),
      golf_club_pose_est_results_list[idx][0].get('pred_instances').get('bboxes')), axis=1)
        bboxes_all.append(bboxes)

        # keypoints
        keypoints = np.concatenate((person_pose_est_results_list[idx][0].get('pred_instances').get('
      keypoints'), golf_club_pose_est_results_list[idx][0].get('pred_instances').get('keypoints')), axis=1)
        kps_all.append(keypoints)

    bboxes_all = np.concatenate(bboxes_all)
    kps_all = np.concatenate(kps_all)

    return bboxes_all, kps_all
```

Listing 8: Concate golfer and club keypoints

In the previous stage, we treated the golfer and club as a single object, named Golfer-with-club. We estimated their 22 keypoints jointly. On the contrary, we detect them separately and track each across all frames. We need to use the function in Listing 8 to extract 17 keypoints from the golfer and 5 from the club, then combine them.

3.3 2D-3D Pose Lifter

After concating these 22 keypoints of the golfer and club, we train a 2D-3D pose lifter to convert them into 3D coordinates. We utilize MixSTE [9] as our 2D-3D pose lifter model.

```
# Prepare 3d data
from common.golf_dataset import GolfDataset
dataset = GolfDataset(dataset_path)

for subject in sorted(dataset.subjects()):
    for action in sorted(dataset[subject].keys()):
        anim = dataset[subject][action]
        if 'positions' in anim:
            anim['positions'] = anim['positions'][:, :total_num, :] ##
            positions_3d = []
            for cam in anim['cameras']:
                pos_3d_mm = anim['positions'] * 1000
                pos_3d_world = vicon_to_world_golf(pos_3d_mm, cam['vicon_to_world_basis_dots'], cam['
      square_size'])
                pos_3d_cam = world_to_camera_golf(pos_3d_world, cam['orientation'], cam['translation_mm'])
                pos_3d = pos_3d_cam / 1000

                pos_3d[:, 1:] -= pos_3d[:, :1] # Remove global offset, but keep trajectory in first position
                positions_3d.append(pos_3d)
            anim['positions_3d'] = positions_3d

# Prepare 2d data
keypoints = np.load('data/data_2d_' + args.dataset + '_' + args.keypoints + '.npz', allow_pickle=True)
keypoints = keypoints['positions_2d'].item()

for subject in sorted(keypoints.keys()):
    for action in sorted(keypoints[subject]):
        for cam_idx, kps in enumerate(keypoints[subject][action]):
            # Normalize camera frame
            cam = dataset.cameras()[subject][cam_idx]
            kps[..., :2] = normalize_screen_coordinates(kps[..., :2], w=cam['res_w'], h=cam['res_h'])
            keypoints[subject][action][cam_idx] = kps

# Loading model
human_num = 17
club_num = 5
```

```
total_num = human_num + club_num

model_pos_train =  MixSTE2(num_frame=receptive_field, num_joints=total_num, in_chans=2, embed_dim_ratio=args
      .cs, depth=args.dep, num_heads=8, mlp_ratio=2., qkv_bias=True, qk_scale=None,drop_path_rate=0.1)
```

Listing 9: 2D-3D pose lifter

In Listing 9, we transform the 3D keypoints ground truth based on the above camera parameters. This transformation was executed in sequence from the Vicon World Coordinate System (VWCS) to the Checkerboard World Coordinate System (CWCS), and subsequently to the Camera Coordinate System (CCS). After this operation, the global offset was eliminated from all keypoints, except the root keypoint, by subtracting the root keypoint. Furthermore, the 2D keypoints ground truth was normalized by mapping the original values to a range of [-1, 1] while preserving the frame's aspect ratio. We then import these ground truth of 2D and 3D keypoints, expand the keypoint dimensions from 17 to 22, load the pre-trained model, and train it for 200 epochs.

Loss Functions. The loss functions of 2D-3D pose lifter model in *GolfPose* are adopted from MixSTE [9]. There are two loss functions: Weight Mean Per Joint Position Error (W-MPJPE) L_w and Mean Per Joint Velocity Error (MPJVE) L_v [8]. Additionally, we adopt Temporal Consistency Loss (TCLoss) L_c to improve motion smoothness [4]. The overall loss function is defined as follows:

$$L = L_w + \lambda_v L_v + \lambda_c L_c, \tag{1}$$

where λ_v and λ_c are weighting factors. The W-MPJPE L_w compares the predicted 3D keypoints with the ground truth, where the weighting vector $W = [w_1, w_2, ..., w_{(N+K)}] \in \mathbb{R}^{(N+K)}$ is applied to keypoints:

$$L_w = \frac{1}{T \times (N+K)} \sum_{t=1}^{T} \sum_{n=1}^{N+K} w_n \times ||y_n^t - gt_n^t||_2^2, \tag{2}$$

where $||\cdot||_2$ denotes the Euclidean distance. The MPJVE L_v considers the velocity differences between predicted keypoints and the ground truth. The TCLoss L_c evaluates the movement distances of predicted keypoints to improve smoothness (with the same W):

$$L_c = \frac{1}{(T-1) \times (N+K)} \sum_{t=2}^{T} \sum_{n=1}^{N+K} w_n \times ||y_n^t - y_n^{t-1}||_2^2. \tag{3}$$

4 Evaluation

4.1 Metrics

2D Golfer, Club, Golfer-with-club Pose Models. We follow the evaluation metric (Object Keypoint Similarity, OKS) used in COCO keypoints evaluation [7]:

$$OKS = \frac{\sum_i [exp\{-d_i^2/2s^2k_i^2\}\delta(v_i > 0)]}{\sum_i [\delta(v_i > 0)]}, \tag{4}$$

where d_i is the Euclidean distances between each ground truth and corresponding detected keypoint, s is the object scale, that is, the square-root of the object's area $\sqrt{(x_2 - x_1)(y_2 - y_1)}$, k_i is a per-keypont constant that controls falloff, v_i is the visibility flags of the ground truth (v=0: not labeled, v=1: labeled but not visible, and v=2: labeled and visible). In our experiment, since our 2D keypoint ground truth was obtained by projecting the 3D keypoint ground truth, we set all $k_i, i = 1...(17+5)$ to 0.02. Subsequently, based on OKS, we compute the following 6 AP and AR metrics: AP, AP^{50}, AP^{75}, AR, AR^{50}, AR^{75}.

2D-3D Lifter. We follow the metric, Mean Per Joint Position Error (MPJPE), from Human3.6m [5] to evaluate the performance difference of the 2D-3D lifter both before and after the fine-tuning process. For all frames of a single subject, MPJPE is computed as

$$l_m = \frac{1}{T \times (N+K)} \sum_{t=1}^{T} \sum_{n=1}^{N+K} ||y_n^t - gt_n^t||_2^2, \quad (5)$$

where T is the number of frames for a single subject and $(N+K)$ is the number of golfer's and club's keypoints.

4.2 Comparison between GolfPose-2D and Pre-Trained Pose Models

In our main paper, we compared 2D pose estimation results for three cases: golfer-only, club-only, and golfer-with-club. The results indicate that the golfer-with-club outperformed both the golfer-only and club-only scenarios. We then compared pre-trained models with golfer-only and golfer-with-club models. However, our model was trained on our *GolfSwing* dataset, and the golfer's keypoint format follows the Human3.6M dataset, which differs from that used in the COCO dataset. Therefore, for comparison purposes, we select 12 common keypoints (shoulders, elbows, wrists, hips, knees, and ankles), as shown in Fig. 3.

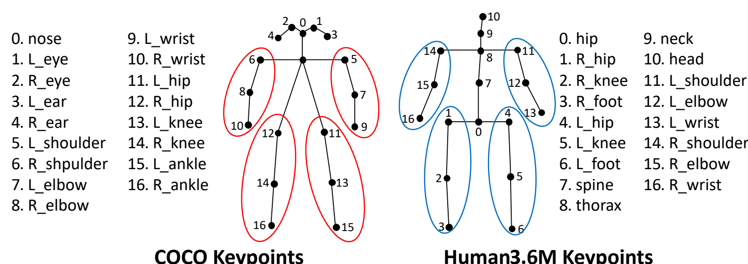

Fig. 3: The keypoint format comparison between COCO and Human3.6M.

```
# Initial set
coco_to_h36m_converter = dict(
    type="KeypointConverter",
    num_keypoints=17,
```

```
    mapping=[(12, 1), (14, 2), (16, 3), (11, 4), (13, 5), (15, 6), (5, 11), (7, 12), (9, 13), (6, 14), (8,
      15), (10, 16)],
)

H36M_partitions = dict(
    limb=[1, 2, 3, 4, 5, 6, 11, 12, 13, 14, 15, 16],
)

# Build the evaluator
evaluator = Evaluator(
    metrics=dict(
        type="KeypointPartitionMetric",
        metric=dict(
            type="CocoMetric",
        ),
        partitions=H36M_partitions,
    )
)

metainfo = parse_pose_metainfo({'from_file': 'configs/mmpose/_base_/datasets/coco.py'})
metainfo["CLASSES"] = COCO_CLASSES
evaluator.dataset_meta = metainfo

# Transform sigmas
from mmpose.evaluation.functional.transforms import transform_sigmas

source_sigmas = evaluator.dataset_meta['sigmas']
print(source_sigmas)
target_sigmas= transform_sigmas(source_sigmas, coco_to_h36m_converter["num_keypoints"],
      coco_to_h36m_converter["mapping"])
evaluator.dataset_meta['sigmas'] = target_sigmas
print(evaluator.dataset_meta['sigmas'])

# Transform predictions
from mmpose.evaluation.functional.transforms import transform_pred

# Load inference results
data_samples = load(pred_file)

for idx, instance in enumerate(data_samples):
    source_instance = instance['pred_instances']
    # print(source_instance)
    target_instance = transform_pred(source_instance, coco_to_h36m_converter["num_keypoints"],
      coco_to_h36m_converter["mapping"])
    # print(target_instance)

print(data_samples[0])

# Call the evaluator offline evaluation
results = evaluator.offline_evaluate(data_samples, chunk_size=128)
print(results)
```

Listing 10: Keypoint Transformation

The code for keypoint transformation is shown in Listing 10. In the beginning, we match the 12 common keypoint indices between the COCO dataset and our dataset. We then choose these common keypoint for performance evaluation. Next, we set up an evaluator and import the COCO dataset's meta-info, which includes the sigmas used for calculating OKS. Subsequently, we convert these sigmas to match the relevant keypoints. Following this, we load the inference results predicted by the pre-trained model on the *GolfSwing* test set and perform a keypoint mapping transformation. Finally, using the mapped sigmas and inference results, we proceed to the evaluation. The results are discussed in our main paper.

5 Conclusion

In this companion paper, we describe the creation of our *GolfSwing* dataset using a MoCap system and DV system. The process involves environment setup, data recording, post-processing, coordinate transformation, and dataset creation. We generate 2D bounding boxes and keypoints by transforming 3D keypoint coordinates, which is faster and more accurate than manual labeling. Additionally,

we detail the implementation and loss functions used for fine-tuning the *GolfPose* model with the *GolfSwing* dataset. We also elaborate on the integration of various models in sequence during inference. Finally, we discuss the metrics for evaluating the 2D pose model and 2D-3D lifter, as well as comparing the 2D pose models with pre-trained models.

References

1. Bradski, G.: The OpenCV Library. Dr. Dobb's Journal of Software Tools (2000)
2. Chen, K., Wang, J., Pang, J., Cao, Y., Xiong, Y., Li, X., Sun, S., Feng, W., Liu, Z., Xu, J., Zhang, Z., Cheng, D., Zhu, C., Cheng, T., Zhao, Q., Li, B., Lu, X., Zhu, R., Wu, Y., Dai, J., Wang, J., Shi, J., Ouyang, W., Loy, C.C., Lin, D.: MMDetection: Open mmlab detection toolbox and benchmark. arXiv preprint arXiv:1906.07155 (2019)
3. Contributors, M.: Openmmlab pose estimation toolbox and benchmark. `https://github.com/open-mmlab/mmpose` (2020)
4. Hossain, M.R.I., Little, J.J.: Exploiting Temporal Information for 3D Human Pose Estimation. In: Proceedings of the European Conference on Computer Vision (ECCV). pp. 68–84 (2018)
5. Ionescu, C., Papava, D., Olaru, V., Sminchisescu, C.: Human3.6m: Large Scale Datasets and Predictive Methods for 3D Human Sensing in Natural Environments. IEEE Transactions on Pattern Analysis and Machine Intelligence **36**(7), 1325–1339 (2013)
6. Lee, M.H., Zhang, Y.C., Wu, K.R., Tseng, Y.C.: Golfpose: From regular posture to golf swing posture. In: 2024 27th International Conference on Pattern Recognition (ICPR) (2024)
7. MSCOCO: MSCOCO keypoint evaluation metric. `https://cocodataset.org/#keypoints-eval` (2017), accessed: 2023-07-12
8. Pavllo, D., Feichtenhofer, C., Grangier, D., Auli, M.: 3D Human Pose Estimation in Video with Temporal Convolutions and Semi-supervised Training. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). pp. 7753–7762 (2019)
9. Zhang, J., Tu, Z., Yang, J., Chen, Y., Yuan, J.: MixSTE: Seq2seq Mixed Spatio-Temporal Encoder for 3D Human Pose Estimation in Video. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). pp. 13232–13242 (2022)
10. Zhang, Z.: A Flexible New Technique for Camera Calibration. IEEE Transactions on Pattern Analysis and Machine Intelligence **22**(11), 1330–1334 (2000)