MMHuman3D

Release 0.4.0

MMHuman3D Authors

Jan 13, 2022

GET STARTED

CHAPTER

ONE

INSTALLATION

1.1 Requirements

- Linux
- ffmpeg
- Python 3.7+
- PyTorch 1.6.0, 1.7.0, 1.7.1, 1.8.0, 1.8.1, 1.9.0 or 1.9.1.
- CUDA 9.2+
- GCC 5+
- PyTorch3D 0.4+
- MMCV (Please install mmcv-full>=1.3.13, <=1.5.0 for GPU)

Optional:

- MMPOSE (Only for demo.)
- MMDETECTION (Only for demo.)
- MMTRACKING (Only for multi-person demo. If you use mmtrack, please install mmcls<1.18.0, mmcv-full>=1.3.16, <1.4.0 for GPU)

1.2 Prepare environment

a. Install ffmpeg

Install ffmpeg with conda directly and the libx264 will be built automatically.

```
conda install ffmpeg
```

b. Create a conda virtual environment and activate it.

```
conda create -n open-mmlab python=3.8 -y
conda activate open-mmlab
```

c. Install PyTorch and torchvision following the official instructions.

```
conda install pytorch=={torch_version} torchvision cudatoolkit={cu_version} -c pytorch
```

E.g., install PyTorch 1.8.0 & CUDA 10.2.

```
conda install pytorch==1.8.0 torchvision cudatoolkit=10.2 -c pytorch
```

Important: Make sure that your compilation CUDA version and runtime CUDA version match.

d. Install PyTorch3D and dependency libs.

```
conda install -c fvcore -c iopath -c conda-forge fvcore iopath -y
conda install -c bottler nvidiacub -y

conda install pytorch3d -c pytorch3d
```

Please refer to PyTorch3D-install for details.

Your installation is successful if you can do these in command line.

```
echo "import pytorch3d;print(pytorch3d.__version__); \
    from pytorch3d.renderer import MeshRenderer;print(MeshRenderer);\
    from pytorch3d.structures import Meshes;print(Meshes);\
    from pytorch3d.renderer import cameras;print(cameras);\
    from pytorch3d.transforms import Transform3d;print(Transform3d);"|python

echo "import torch;device=torch.device('cuda');\
    from pytorch3d.utils import torus;\
    Torus = torus(r=10, R=20, sides=100, rings=100, device=device);\
    print(Torus.verts_padded());"|python
```

1.3 Install MMHuman3D

a. Build mmcv-full & mmpose & mmdet & mmtrack

- mmcv-full

We recommend you to install the pre-build package as below.

For CPU:

```
pip install mmcv-full -f https://download.openmmlab.com/mmcv/dist/cpu/{torch_version}/
↪index.html
```

Please replace `{torch_version}` in the url to your desired one.

For GPU:

```
pip install "mmcv-full>=1.3.13,<1.4.0" -f https://download.openmmlab.com/mmcv/dist/{cu_
↪version}/{torch_version}/index.html
```

Please replace {cu_version} and {torch_version} in the url to your desired one.

For example, to install mmcv-full with CUDA 10.2 and PyTorch 1.8.0, use the following command:

```
pip install "mmcv-full>=1.3.13,<1.4.0" -f https://download.openmmlab.com/mmcv/dist/cu102/
↪torch1.8.0/index.html
```

See here for different versions of MMCV compatible to different PyTorch and CUDA versions. For more version download link, refer to openmmlab-download.

Optionally you can choose to compile mmcv from source by the following command

```
git clone https://github.com/open-mmlab/mmcv.git -b v1.3.13
cd mmcv
MMCV_WITH_OPS=1 pip install -e .  # package mmcv-full, which contains cuda ops, will be␣
↪installed after this step
# OR pip install -e .  # package mmcv, which contains no cuda ops, will be installed␣
↪after this step
cd ..
```

Important: You need to run `pip uninstall mmcv` first if you have mmcv installed. If mmcv and mmcv-full are both installed, there will be `ModuleNotFoundError`.

- mmdetection (optional)

```
pip install mmdet
```

Optionally, you can also build MMDetection from source in case you want to modify the code:

```
git clone https://github.com/open-mmlab/mmdetection.git
cd mmdetection
pip install -r requirements/build.txt
pip install -v -e .
```

- mmpose (optional)

```
pip install mmpose
```

Optionally, you can also build MMPose from source in case you want to modify the code:

```
git clone https://github.com/open-mmlab/mmpose.git
cd mmpose
pip install -r requirements.txt
pip install -v -e .
```

- mmtracking (optional)

```
pip install "mmcls<0.18.0" "mmtrack<0.9.0,>=0.8.0"
```

Optionally, you can also build MMTracking from source in case you want to modify the code:

```
git clone git@github.com:open-mmlab/mmtracking.git -b v0.8.0
cd mmtracking
pip install -r requirements/build.txt
pip install -v -e .  # or "python setup.py develop"
```

b. Clone the mmhuman3d repository.

```
git clone https://github.com/open-mmlab/mmhuman3d.git
cd mmhuman3d
```

c. Install build requirements and then install mmhuman3d.

```
pip install -v -e .  # or "python setup.py develop"
```

1.4 A from-scratch setup script

```
conda create -n open-mmlab python=3.8 -y
conda activate open-mmlab

conda install pytorch==1.8.0 torchvision cudatoolkit=10.2 -c pytorch -y

# install PyTorch3D

conda install -c fvcore -c iopath -c conda-forge fvcore iopath -y
conda install -c bottler nvidiacub -y

conda install pytorch3d -c pytorch3d

# install mmcv-full

pip install "mmcv-full>=1.3.13,<1.4.0" -f https://download.openmmlab.com/mmcv/dist/cu102/
↪torch1.8.0/index.html


# Optional
# install mmdetection & mmpose & mmtracking

pip install mmdet

pip install mmpose

pip install "mmcls<0.18.0" "mmtrack<0.9.0,>=0.8.0"

# install mmhuman3d

git clone https://github.com/open-mmlab/mmhuman3d.git
cd mmhuman3d
pip install -v -e .
```

CHAPTER

TWO

GETTING STARTED

2.1 Installation

Please refer to *install.md* for installation.

2.2 Data Preparation

Please refer to *data_preparation.md* for data preparation.

2.3 Body Model Preparation

- SMPL v1.0 is used in our experiments.
  - Neutral model can be downloaded from SMPLify.
  - All body models have to be renamed in `SMPL_{GENDER}.pkl` format. For example, `mv basicModel_neutral_lbs_10_207_0_v1.0.0.pkl SMPL_NEUTRAL.pkl`
- J_regressor_extra.npy
- J_regressor_h36m.npy
- smpl_mean_params.npz

Download the above resources and arrange them in the following file structure:

```
mmhuman3d
├── mmhuman3d
├── docs
├── tests
├── tools
├── configs
└── data
    └── body_models
        ├── J_regressor_extra.npy
        ├── J_regressor_h36m.npy
        ├── smpl_mean_params.npz
        └── smpl
            ├── SMPL_FEMALE.pkl
            ├── SMPL_MALE.pkl
            └── SMPL_NEUTRAL.pkl
```

2.4 Inference / Demo

```
python demo/estimate_smpl_image.py ${CONFIG_FILE} ${CHECKPOINT} [optional]
```

2.4.1 Single-person

Optional arguments include:

- `--single_person_demo`: flag for single-person inference
- `--det_config`: MMDetection config
- `--det_checkpoint`: MMDetection checkpoint
- `--input_path`: input path
- `--show_path`: directory to save rendered images or video
- `--smooth_type`: smoothing mode

Example:

```
python demo/estimate_smpl_image.py \
    configs/hmr/resnet50_hmr_pw3d.py \
    data/checkpoints/resnet50_hmr_pw3d.pth \
    --single_person_demo \
    --det_config demo/mmdetection_cfg/faster_rcnn_r50_fpn_coco.py \
    --det_checkpoint https://download.openmmlab.com/mmdetection/v2.0/faster_rcnn/faster_
↪rcnn_r50_fpn_1x_coco/faster_rcnn_r50_fpn_1x_coco_20200130-047c8118.pth \
    --input_path  demo/resources/single_person_demo.mp4 \
    --show_path vis_results/single_person_demo.mp4 \
    --smooth_type savgol
```

Note that the MMHuman3D checkpoints can be downloaded from the *model zoo*. Here we take HMR (resnet50_hmr_pw3d.pth) as an example.

2.4.2 Multi-person

Optional arguments include:

- `--multi_person_demo`: flag for multi_person inference
- `--mmtracking_config`: MMTracking config
- `--input_path`: input path
- `--show_path`: directory to save rendered images or video
- `--smooth_type`: smoothing mode

Example 2: multi-person estimation

```
python demo/estimate_smpl_image.py \
    configs/hmr/resnet50_hmr_pw3d.py \
    data/checkpoints/resnet50_hmr_pw3d.pth \
    --multi_person_demo \
    --tracking_config demo/mmtracking_cfg/deepsort_faster-rcnn_fpn_4e_mot17-private-half.
↪py \
    --input_path  demo/resources/multi_person_demo.mp4 \
    --show_path vis_results/multi_person_demo.mp4 \
    --smooth_type savgol
```

2.5 Evaluation

We provide pretrained models in the respective method folders in config.

2.5.1 Evaluate with a single GPU / multiple GPUs

```
python tools/test.py ${CONFIG} --work-dir=${WORK_DIR} ${CHECKPOINT}
```

Example:

```
python tools/test.py configs/hmr/resnet50_hmr_pw3d.py --work-dir=work_dirs/hmr work_dirs/
↪hmr/latest.pth
```

2.5.2 Evaluate with slurm

If you can run MMHuman3D on a cluster managed with slurm, you can use the script `slurm_test.sh`.

```
./tools/slurm_test.sh ${PARTITION} ${JOB_NAME} ${CONFIG} ${WORK_DIR} ${CHECKPOINT}
```

Example:

```
./tools/slurm_test.sh my_partition test_hmr configs/hmr/resnet50_hmr_pw3d.py work_dirs/
↪hmr work_dirs/hmr/latest.pth 8
```

2.6 Training

2.6.1 Training with a single / multiple GPUs

```
python tools/train.py ${CONFIG_FILE} ${WORK_DIR} --no-validate
```

Example: using 1 GPU to train HMR.

```
python tools/train.py ${CONFIG_FILE} ${WORK_DIR} --gpus 1 --no-validate
```

2.6.2 Training with Slurm

If you can run MMHuman3D on a cluster managed with slurm, you can use the script `slurm_train.sh`.

```
./tools/slurm_train.sh ${PARTITION} ${JOB_NAME} ${CONFIG_FILE} ${WORK_DIR} ${GPU_NUM} --
↪no-validate
```

Common optional arguments include:

- `--resume-from ${CHECKPOINT_FILE}`: Resume from a previous checkpoint file.
- `--no-validate`: Whether not to evaluate the checkpoint during training.

Example: using 8 GPUs to train HMR on a slurm cluster.

```
./tools/slurm_train.sh my_partition my_job configs/hmr/resnet50_hmr_pw3d.py work_dirs/
↪hmr 8 --no-validate
```

You can check slurm_train.sh for full arguments and environment variables.

2.7 More Tutorials

- *Camera conventions*
- *Keypoint conventions*
- *Custom keypoint conventions*
- *HumanData*
- *Keypoint visualization*
- *Mesh visualization*

CHAPTER

THREE

BENCHMARK AND MODEL ZOO

We provide configuration files, log files and pretrained models for all supported methods. Moreover, all pretrain models are evaluated on three common benchmarks: 3DPW, Human3.6M, and MPI-INF-3DHP.

3.1 Baselines

3.1.1 HMR

Please refer to HMR for details.

3.1.2 SPIN

Please refer to SPIN for details.

3.1.3 VIBE

Please refer to VIBE for details.

3.1.4 HybrIK

Please refer to HybrIK for details.

CHAPTER

FOUR

HUMANDATA

4.1 Overview

HumanData is a subclass of python built-in class dict, containing single-view, image-based data for a human being. It has a well-defined base structure for universal data, but it is also compatible with customized data for new features. A native HumanData contains values in numpy.ndarray or python built-in types, it holds no data in torch.Tensor, but you can convert arrays to torch.Tensor(even to GPU Tensor) by `human_data.to()` easily.

4.2 Key/Value definition

4.2.1 The keys and values supported by HumanData are described as below.

- image_path: (N,), list of str, each element is a relative path from the root folder (exclusive) to the image.
- bbox_xywh: (N, 5), numpy array, bounding box with confidence, coordinates of bottom-left point x, y, width w and height h of bbox, score at last.
- config: (), str, the flag name of config for individual dataset.
- keypoints2d: (N, 190, 3), numpy array, 2d joints of smplx model with confidence, joints from each datasets are mapped to HUMAN_DATA joints.
- keypoints3d: (N, 190, 4), numpy array, 3d joints of smplx model with confidence. Same as above.
- smpl: (1,), dict, keys are [‘body_pose’: numpy array, (N, 23, 3), ‘global_orient’: numpy array, (N, 3), ‘betas’: numpy array, (N, 10), ‘transl’: numpy array, (N, 3)].
- smplx: (1,), dict, keys are [‘body_pose’: numpy array, (N, 21, 3),’global_orient’: numpy array, (N, 3), ‘betas’: numpy array, (N, 10), ‘transl’: numpy array, (N, 3), ‘left_hand_pose’: numpy array, (N, 15, 3), ‘right_hand_pose’: numpy array, (N, 15, 3), ‘expression’: numpy array (N, 10), ‘leye_pose’: numpy array (N, 3), ‘reye_pose’: (N, 3), ‘jaw_pose’: numpy array (N, 3)].
- meta: (1,), dict, its keys are meta data from dataset like ‘gender’.
- keypoints2d_mask: (190,), numpy array, mask for which keypoint is valid in keypoints2d. 0 means that the joint in this position cannot be found in original dataset.
- keypoints3d_mask: (190,), numpy array, mask for which keypoint is valid in keypoints3d. 0 means that the joint in this position cannot be found in original dataset.
- misc: (1,), dict, keys and values are defined by user. The space misc takes(sys.getsizeof(misc)) shall be no more than 6MB.

4.2.2 Key check in HumanData.

Only keys above are allowed as top level key in a default HumanData. If you cannot work with that, there's also a way out. Construct a HumanData instance with `__key_strict__ == False`:

```
human_data = HumanData.new(key_strict=False)
human_data['video_path'] = 'test.mp4'
```

The returned human_data will allow any customized keys, logging a warning at the first time HumanData sees a new key. Just ignore the warning if you do know that you are using a customized key, it will not appear again before the program ends.

If you have already constructed a HumanData, and you want to change the strict mode, use `set_key_strict`:

```
human_data = HumanData.fromfile('human_data.npz')
key_strict = human_data.get_key_strict()
human_data.set_key_strict(not key_strict)
```

4.2.3 Value check in HumanData.

Only values above will be check when `human_data[key] == value` is called, and the constraints are defined in `HumanData.SUPPORTED_KEYS`.

For each value, an exclusive type must be specified under its key:

```
'smpl': {
    'type': dict,
},
```

For value as numpy.ndarray, shape and temporal_dim shall be defined:

```
'keypoints3d': {
    'type': np.ndarray,
    'shape': (-1, -1, 4),
    # value.ndim==3, and value.shape[2]==4
    # value.shape[0:2] is arbitrary.
    'temporal_dim': 0
    # dimension 0 marks time(frame index, or second)
},
```

For value which is constant along time axis, set temporal_dim to -1 to ignore temporal check:

```
'keypoints3d_mask': {
    'type': np.ndarray,
    'shape': (-1, ),
    'temporal_dim': -1
},
```

4.3 Data compression

4.3.1 Compression with mask

As the keypoint convention named HUMAN_DATA is a union of keypoint definitions from various datasets, it is common that some keypoints are missing. In this situation, the missing ones are filtered by mask:

```
# keypoints2d_agora is a numpy array in shape [frame_num, 127, 3].
# There are 127 keypoints defined by agora.
keypoints2d_human_data, mask = convert_kps(keypoints2d_agora, 'agora', 'human_data')
# keypoints2d_human_data is a numpy array in shape [frame_num, 190, 3], only 127/190 are␣
↪valid
# mask is a numpy array in shape [190, ], with 127 ones and 63 zeros inside
```

Set `keypoints2d_mask` and `keypoints2d`. It is obvious that there are redundant zeros in keypoints2d:

```
human_data = HumanData()
human_data['keypoints2d_mask'] = mask
human_data['keypoints2d'] = keypoints2d_human_data
```

Call `compress_keypoints_by_mask()` to get rid of the zeros. This method checks if any key containing `keypoints` has a corresponding mask, and performs keypoints compression if both keypoints and masks are present. :

```
human_data.compress_keypoints_by_mask()
```

Call `get_raw_value()` to get the compressed raw value stored in HumanData instance. When getting item with `[]`, the keypoints padded with zeros will be returned:

```
keypoints2d_human_data = human_data.get_raw_value('keypoints2d')
print(keypoints2d_human_data.shape)  # [frame_num, 127, 3]
keypoints2d_human_data = human_data['keypoints2d']
print(keypoints2d_human_data.shape)  # [frame_num, 190, 3]
```

In `keypoints_compressed` mode, keypoints are allowed to be edited. There are two different ways, set with padded data or set the compressed data directly:

```
padded_keypoints2d = np.zeros(shape=[100, 190, 3])
human_data['keypoints2d'] = padded_keypoints2d  # [frame_num, 190, 3]
compressed_keypoints2d = np.zeros(shape=[100, 127, 3])
human_data.set_raw_value('keypoints2d', compressed_keypoints2d)  # [frame_num, 127, 3]
```

When a HumanData instance is in `keypoints_compressed` mode, all masks of keypoints are locked. If you are trying to edit it, a warning will be logged and the value won't change. To modify a mask, de-compress it with `decompress_keypoints()`:

```
human_data.decompress_keypoints()
```

Features above also work with any key pairs like `keypoints*` and `keypoints*_mask`.

4.3.2 Compression for file

Call `dump()` to save HumanData into a compressed `.npz` file.

The dumped file can be load by `load()` :

```
# save
human_data.dump('./dumped_human_data.npz')
# load
another_human_data = HumanData()
another_human_data.load('./dumped_human_data.npz')
```

Sometimes a HumanData instanse is too large to dump, an error will be raised by `numpy.savez_compressed()`. In this case, call `dump_by_pickle` and `load_by_pickle` for file operation.

4.3.3 Compression by key

If a HumanData instance is in not in key_strict mode, it may contains unsupported items which are not necessary. Call `pop_unsupported_items()` to remove those items will save space for you:

```
human_data = HumanData.fromfile('human_data_not_strict.npz')
human_data.pop_unsupported_items()
# set instance.__key_strict__ from True to False will also do
human_data.set_key_strict(True)
```

4.4 Data selection

4.4.1 Select by shape

Assume that `keypoints2d` is an array in shape [200, 190, 3], only the first 10 frames are needed:

```
first_ten_frames = human_data.get_value_in_shape('keypoints2d', shape=[10, -1, -1])
```

In some situation, we need to pad all arrays to a certain size:

```
# pad keypoints2d from [200, 190, 3] to [200, 300, 3] with zeros
padded_keypoints2d = human_data.get_value_in_shape('keypoints2d', shape=[200, 300, -1])
# padding value can be modified
padded_keypoints2d = human_data.get_value_in_shape('keypoints2d', shape=[200, 300, -1],␣
↪padding_constant=1)
```

4.4.2 Select temporal slice

Assume that there are 200 frames in a HumanData instance, only data between 10 and 20 are needed:

```
# all supported values will be sliced
sub_human_data = human_data.get_temporal_slice(10, 21)
```

Downsample is also supported, for example, select 33%:

```
# select [0, 3, 6, 9,..., 198]
sub_human_data = human_data.get_temporal_slice(0, 200, 3)
```

4.5 To torch.Tensor

As introduced, a native HumanData contains values in numpy.ndarray or python built-in types, but the numpy.ndarray can be easily convert to torch.Tensor:

```
# All values as ndarray will be converted to a cpu Tensor.
# Values in other types will not change.
# It returns a dict like HumanData.
dict_of_tensor = human_data.to()
# GPU is also supported
gpu0_device = torch.device('cuda:0')
dict_of_gpu_tensor = human_data.to(gpu0_device)
```

CHAPTER

FIVE

DATA PREPARATION

5.1 Overview

Our data pipeline use *HumanData* structure for storing and loading. The proprocessed npz files can be obtained from raw data using our data converters, and the supported configs can be found here.

These are our supported converters and their respective `dataset-name`:

- AgoraConverter (`agora`)
- AmassConverter (`amass`)
- CocoConverter (`coco`)
- CocoHybrIKConverter (`coco_hybrik`)
- CocoWholebodyConverter (`coco_wholebody`)
- CrowdposeConverter (`crowdpose`)
- EftConverter (`eft`)
- GTAHumanConverter (`gta_human`)
- H36mConverter (`h36m_p1`, `h36m_p2`)
- H36mHybrIKConverter (`h36m_hybrik`)
- H36mSpinConverter (`h36m_spin`)
- InstaVibeConverter (`instavariety_vibe`)
- LspExtendedConverter (`lsp_extended`)
- LspConverter (`lsp_original`, `lsp_dataset`)
- MpiiConverter (`mpii`)
- MpiInf3dhpConverter (`mpi_inf_3dhp`)
- MpiInf3dhpHybrIKConverter (`mpi_inf_3dhp_hybrik`)
- PennActionConverter (`penn_action`)
- PosetrackConverter (`posetrack`)
- Pw3dConverter (`pw3d`)
- Pw3dHybrIKConverter (`pw3d_hybrik`)
- SurrealConverter (`surreal`)
- SpinConverter (`spin`)
- Up3dConverter (`up3d`)

5.2 Datasets for supported algorithms

For all algorithms, the root path for our datasets and output path for our preprocessed npz files are stored in `data/datasets` and `data/preprocessed_datasets`. As such, use this command with the listed `dataset-names`:

```
python tools/convert_datasets.py \
  --datasets <dataset-name> \
  --root_path data/datasets \
  --output_path data/preprocessed_datasets
```

For HMR training and testing, the following datasets are required:

- *COCO*
- *Human3.6M*
- *Human3.6M Mosh*
- *MPI-INF-3DHP*
- *MPII*
- *LSP*
- *LSPET*
- *PW3D*

Convert datasets with the following `dataset-names`:

```
coco, pw3d, mpii, mpi_inf_3dhp, lsp_original, lsp_extended, h36m
```

Alternatively, you may download the preprocessed files directly:

- cmu_mosh.npz
- coco_2014_train.npz
- h36m_train.npz
- lsp_train.npz
- lspet_train.npz
- mpi_inf_3dhp_train.npz
- mpii_train.npz
- pw3d_test.npz

The preprocessed datasets should have this structure:

```
mmhuman3d
├── mmhuman3d
├── docs
├── tests
├── tools
├── configs
└── data
    ├── datasets
    └── preprocessed_datasets
        ├── coco_2014_train.npz
        ├── h36m_train.npz
        ├── lspet_train.npz
        ├── lsp_train.npz
        ├── mpi_inf_3dhp_train.npz
        ├── mpii_train.npz
        └── pw3d_test.npz
```

For SPIN training, the following datasets are required:

- *COCO*
- *Human3.6M*

- *Human3.6M Mosh*
- *MPI-INF-3DHP*
- *MPII*
- *LSP*
- *LSPET*
- *PW3D*
- *SPIN*

Convert datasets with the following `dataset-names`:

```
spin, h36m_spin
```

Alternatively, you may download the preprocessed files directly:

- spin_coco_2014_train.npz
- spin_h36m_train.npz
- spin_lsp_train.npz
- spin_lspet_train.npz
- spin_mpi_inf_3dhp_train.npz
- spin_mpii_train.npz
- spin_pw3d_test.npz

The preprocessed datasets should have this structure:

```
mmhuman3d
├── mmhuman3d
├── docs
├── tests
├── tools
├── configs
└── data
    ├── datasets
    └── preprocessed_datasets
        ├── spin_coco_2014_train.npz
        ├── spin_h36m_train.npz
        ├── spin_lsp_train.npz
        ├── spin_lspet_train.npz
        ├── spin_mpi_inf_3dhp_train.npz
        ├── spin_mpii_train.npz
        └── spin_pw3d_test.npz
```

For VIBE training and testing, the following datasets are required:

- *MPI-INF-3DHP*
- *PW3D*

The data converters are currently not available.

Alternatively, you may download the preprocessed files directly:

- vibe_insta_variety.npz

- vibe_mpi_inf_3dhp_train.npz
- vibe_pw3d_test.npz

The preprocessed datasets should have this structure:

```
mmhuman3d
├── mmhuman3d
├── docs
├── tests
├── tools
├── configs
└── data
    ├── datasets
    └── preprocessed_datasets
        ├── vibe_insta_variety.npz
        ├── vibe_mpi_inf_3dhp_train.npz
        └── vibe_pw3d_test.npz
```

For HYBRIK training and testing, the following datasets are required:

- *HybrIK*
- *COCO*
- *Human3.6M*
- *MPI-INF-3DHP*
- *PW3D*

Convert datasets with the following `dataset-names`:

```
h36m_hybrik, pw3d_hybrik, mpi_inf_3dhp_hybrik, coco_hybrik
```

Alternatively, you may download the preprocessed files directly:

- hybriK_coco_2017_train.npz
- hybrik_h36m_train.npz
- hybrik_mpi_inf_3dhp_train.npz
- hybrik_pw3d_test.npz

The preprocessed datasets should have this structure:

```
mmhuman3d
├── mmhuman3d
├── docs
├── tests
├── tools
├── configs
└── data
    ├── datasets
    └── preprocessed_datasets
        ├── hybriK_coco_2017_train.npz
        ├── hybrik_h36m_train.npz
        ├── hybrik_mpi_inf_3dhp_train.npz
        └── hybrik_pw3d_test.npz
```

5.3 Folder structure

5.3.1 AGORA

```
@inproceedings{Patel:CVPR:2021,
 title = {{AGORA}: Avatars in Geography Optimized for Regression Analysis},
 author = {Patel, Priyanka and Huang, Chun-Hao P. and Tesch, Joachim and Hoffmann, David␣
↪T. and Tripathi, Shashank and Black, Michael J.},
 booktitle = {Proceedings IEEE/CVF Conf.~on Computer Vision and Pattern Recognition (
↪{CVPR})},
 month = jun,
 year = {2021},
 month_numeric = {6}
}
```

For AGORA, please download the dataset and place them in the folder structure below:

```
mmhuman3d
├── mmhuman3d
├── docs
├── tests
├── tools
├── configs
└── data
    └── datasets
        └── agora
            ├── camera_dataframe # smplx annotations
            │   ├── train_0_withjv.pkl
            │   ├── validation_0_withjv.pkl
            │   └── ...
            ├── camera_dataframe_smpl # smpl annotations
            │   ├── train_0_withjv.pkl
            │   ├── validation_0_withjv.pkl
            │   └── ...
            ├── images
            │   ├── train
            │   │   ├── ag_trainset_3dpeople_bfh_archviz_5_10_cam00_00000_1280x720.png
            │   │   ├── ag_trainset_3dpeople_bfh_archviz_5_10_cam00_00001_1280x720.png
            │   │   └── ...
            │   ├── validation
            │   └── test
            ├── smpl_gt
            │   ├── trainset_3dpeople_adults_bfh
            │   │   ├── 10004_w_Amaya_0_0.mtl
            │   │   ├── 10004_w_Amaya_0_0.obj
            │   │   ├── 10004_w_Amaya_0_0.pkl
            │   │   └── ...
            │   └── ...
            └── smplx_gt
                ├── 10004_w_Amaya_0_0.obj
                ├── 10004_w_Amaya_0_0.pkl
                └── ...
```

5.3.2 COCO

```
@inproceedings{lin2014microsoft,
  title={Microsoft coco: Common objects in context},
  author={Lin, Tsung-Yi and Maire, Michael and Belongie, Serge and Hays, James and␣
↪Perona, Pietro and Ramanan, Deva and Doll{\'a}r, Piotr and Zitnick, C Lawrence},
  booktitle={European conference on computer vision},
  pages={740--755},
  year={2014},
  organization={Springer}
}
```

For COCO data, please download from COCO download. COCO'2014 Train is needed for HMR training and COCO'2017 Train is needed for HybrIK trainig. Download and extract them under `$MMHUMAN3D/data/datasets`, and make them look like this:

```
mmhuman3d
├── mmhuman3d
├── docs
├── tests
├── tools
├── configs
└── data
    └── datasets
        └── coco
            ├── annotations
            |   ├── person_keypoints_train2014.json
            |   ├── person_keypoints_val2014.json
            ├── train2014
            |   ├── COCO_train2014_000000000009.jpg
            |   ├── COCO_train2014_000000000025.jpg
            |   ├── COCO_train2014_000000000030.jpg
            |   └── ...
            └── train_2017
                ├── annotations
                |   ├── person_keypoints_train2017.json
                |   └── person_keypoints_val2017.json
                ├── train2017
                |   ├── 000000000009.jpg
                |   ├── 000000000025.jpg
                |   ├── 000000000030.jpg
                |   └── ...
                └── val2017
                    ├── 000000000139.jpg
                    ├── 000000000285.jpg
                    ├── 000000000632.jpg
                    └── ...
```

5.3.3 COCO-WholeBody

```
@inproceedings{jin2020whole,
  title={Whole-Body Human Pose Estimation in the Wild},
  author={Jin, Sheng and Xu, Lumin and Xu, Jin and Wang, Can and Liu, Wentao and Qian,␣
↪Chen and Ouyang, Wanli and Luo, Ping},
  booktitle={Proceedings of the European Conference on Computer Vision (ECCV)},
  year={2020}
}
```

For COCO-WholeBody dataset, images can be downloaded from COCO download, 2017 Train/Val is needed for COCO keypoints training and validation. Download and extract them under `$MMHUMAN3D/data/datasets`, and make them look like this:

```
mmhuman3d
├── mmhuman3d
├── docs
├── tests
├── tools
├── configs
└── data
    └── datasets
        └── coco
            ├── annotations
            |   ├── coco_wholebody_train_v1.0.json
            |   └── coco_wholebody_val_v1.0.json
            └── train_2017
                ├── train2017
                |   ├── 000000000009.jpg
                |   ├── 000000000025.jpg
                |   ├── 000000000030.jpg
                |   └── ...
                └── val2017
                    ├── 000000000139.jpg
                    ├── 000000000285.jpg
                    ├── 000000000632.jpg
                    └── ...
```

5.3.4 CrowdPose

```
@article{li2018crowdpose,
  title={CrowdPose: Efficient Crowded Scenes Pose Estimation and A New Benchmark},
  author={Li, Jiefeng and Wang, Can and Zhu, Hao and Mao, Yihuan and Fang, Hao-Shu and␣
↪Lu, Cewu},
  journal={arXiv preprint arXiv:1812.00324},
  year={2018}
}
```

For CrowdPose data, please download from CrowdPose. Download and extract them under `$MMHUMAN3D/data/datasets`, and make them look like this:

```
mmhuman3d
├── mmhuman3d
├── docs
├── tests
├── tools
├── configs
└── data
    └── datasets
        └── crowdpose
            ├── crowdpose_train.json
            ├── crowdpose_val.json
            ├── crowdpose_trainval.json
            ├── crowdpose_test.json
            └── images
                ├── 100000.jpg
                ├── 100001.jpg
                ├── 100002.jpg
                └── ...
```

5.3.5 EFT

```
@inproceedings{joo2020eft,
 title={Exemplar Fine-Tuning for 3D Human Pose Fitting Towards In-the-Wild 3D Human Pose␣
↪Estimation},
 author={Joo, Hanbyul and Neverova, Natalia and Vedaldi, Andrea},
 booktitle={3DV},
 year={2020}
}
```

For EFT data, please download from EFT. Download and extract them under `$MMHUMAN3D/data/datasets`, and make them look like this:

```
mmhuman3d
├── mmhuman3d
├── docs
├── tests
├── tools
├── configs
└── data
   └── datasets
       └── eft
           ├── coco_2014_train_fit
           |   ├── COCO2014-All-ver01.json
           |   └── COCO2014-Part-ver01.json
           |── LSPet_fit
           |   └── LSPet_ver01.json
           └── MPII_fit
               └── MPII_ver01.json
```

5.3.6 GTA-Human

```
@article{cai2021playing,
  title={Playing for 3D Human Recovery},
  author={Cai, Zhongang and Zhang, Mingyuan and Ren, Jiawei and Wei, Chen and Ren,␣
↪Daxuan and Li, Jiatong and Lin, Zhengyu and Zhao, Haiyu and Yi, Shuai and Yang, Lei␣
↪and others},
  journal={arXiv preprint arXiv:2110.07588},
  year={2021}
}
```

More details are coming soon!

5.3.7 Human3.6M

```
@article{h36m_pami,
  author = {Ionescu, Catalin and Papava, Dragos and Olaru, Vlad and Sminchisescu, ␣
↪Cristian},
  title = {Human3.6M: Large Scale Datasets and Predictive Methods for 3D Human Sensing␣
↪in Natural Environments},
  journal = {IEEE Transactions on Pattern Analysis and Machine Intelligence},
  publisher = {IEEE Computer Society},
  volume = {36},
  number = {7},
  pages = {1325-1339},
  month = {jul},
  year = {2014}
}
```

For Human3.6M, please download from the official website and run the preprocessing script, which will extract pose annotations at downsampled framerate (10 FPS). The processed data should have the following structure:

```
mmhuman3d
├── mmhuman3d
├── docs
├── tests
├── tools
├── configs
└── data
    └── datasets
        └── h36m
            ├── annot
            ├── S1
            |   ├── images
            |   |   |── S1_Directions_1.54138969
            |   |   |   ├── S1_Directions_1.54138969_00001.jpg
            |   |   |   ├── S1_Directions_1.54138969_00002.jpg
            |   |   |   └── ...
            |   |   └── ...
            |   ├── MyPoseFeatures
            |   |   |── D2Positions
            |   |   └── D3_Positions_Mono
```

(continues on next page)

(continued from previous page)

```
|   ├── MySegmentsMat
|   |    └── ground_truth_bs
|   └── Videos
|        |── Directions 1.54138969.mp4
|        |── Directions 1.55011271.mp4
|        └── ...
├── S5
├── S6
├── S7
├── S8
├── S9
├── S11
└── metadata.xml
```

To extract images from Human3.6M original videos, modify the `h36m_p1` config in DATASET_CONFIG:

```
h36m_p1=dict(
    type='H36mConverter',
    modes=['train', 'valid'],
    protocol=1,
    extract_img=True, # set to true to extract images from raw videos
    prefix='h36m'),
```

5.3.8 Human3.6M Mosh

For data preparation of Human3.6M for HMR and SPIN training, we use the MoShed data provided in HMR for training. However, due to license limitations, we are not allowed to redistribute the data. Even if you do not have access to these parameters, you can still generate the preprocessed h36m npz file without mosh parameters using our converter.

To do so, modify the `h36m_p1` config in DATASET_CONFIG:

Config without mosh:

```
h36m_p1=dict(
    type='H36mConverter',
    modes=['train', 'valid'],
    protocol=1,
    prefix='h36m'),
```

Config:

```
h36m_p1=dict(
    type='H36mConverter',
    modes=['train', 'valid'],
    protocol=1,
    mosh_dir='data/datasets/h36m_mosh', # supply the directory to the mosh if available
    prefix='h36m'),
```

If you have MoShed data available, it should have the following structure:

```
mmhuman3d
├── mmhuman3d
```

(continues on next page)

(continued from previous page)

```
├── docs
├── tests
├── tools
├── configs
└── data
    └── datasets
        └── h36m_mosh
            ├── annot
            ├── S1
            |   ├── images
            |   |    ├── Directions 1_cam0_aligned.pkl
            |   |    ├── Directions 1_cam1_aligned.pkl
            |   |    └── ...
            ├── S5
            ├── S6
            ├── S7
            ├── S8
            ├── S9
            └── S11
```

5.3.9 HybrIK

```
@inproceedings{li2020hybrikg,
  author = {Li, Jiefeng and Xu, Chao and Chen, Zhicun and Bian, Siyuan and Yang, Lixin␣
↪and Lu, Cewu},
  title = {HybrIK: A Hybrid Analytical-Neural Inverse Kinematics Solution for 3D Human␣
↪Pose and Shape Estimation},
  booktitle={CVPR 2021},
  pages={3383--3393},
  year={2021},
  organization={IEEE}
}
```

For HybrIK, please download the parsed json annotation files and place them in the folder structure below:

```
mmhuman3d
├── mmhuman3d
├── docs
├── tests
├── tools
├── configs
└── data
    └── datasets
        └── hybrik_data
            ├── Sample_5_train_Human36M_smpl_leaf_twist_protocol_2.json
            ├── Sample_20_test_Human36M_smpl_protocol_2.json
            ├── 3DPW_test_new.json
            ├── annotation_mpi_inf_3dhp_train_v2.json
            └── annotation_mpi_inf_3dhp_test.json
```

To convert the preprocessed json files into npz files used for our pipeline, run the following preprocessing scripts:

- Human3.6M
- PW3D
- Mpi-Inf-3dhp
- COCO

5.3.10 LSP

```
@inproceedings{johnson2010clustered,
  title={Clustered Pose and Nonlinear Appearance Models for Human Pose Estimation.},
  author={Johnson, Sam and Everingham, Mark},
  booktitle={bmvc},
  volume={2},
  number={4},
  pages={5},
  year={2010},
  organization={Citeseer}
}
```

For LSP, please download the high resolution version LSP dataset original. Extract them under `$MMHUMAN3D/data/datasets`, and make them look like this:

```
mmhuman3d
├── mmhuman3d
├── docs
├── tests
├── tools
├── configs
└── data
    └── datasets
        └── lsp
            ├── images
            |   ├── im0001.jpg
            |   ├── im0002.jpg
            |   └── ...
            └── joints.mat
```

5.3.11 LSPET

```
@inproceedings{johnson2011learning,
  title={Learning effective human pose estimation from inaccurate annotation},
  author={Johnson, Sam and Everingham, Mark},
  booktitle={CVPR 2011},
  pages={1465--1472},
  year={2011},
  organization={IEEE}
}
```

For LSPET, please download its high resolution form HR-LSPET. Extract them under `$MMHUMAN3D/data/datasets`, and make them look like this:

```
mmhuman3d
├── mmhuman3d
├── docs
├── tests
├── tools
├── configs
└── data
    └── datasets
        └── lspet
            ├── im00001.jpg
            ├── im00002.jpg
            ├── im00003.jpg
            ├── ...
            └── joints.mat
```

5.3.12 MPI-INF-3DHP

```
@inproceedings{mono-3dhp2017,
 author = {Mehta, Dushyant and Rhodin, Helge and Casas, Dan and Fua, Pascal and␣
↪Sotnychenko, Oleksandr and Xu, Weipeng and Theobalt, Christian},
 title = {Monocular 3D Human Pose Estimation In The Wild Using Improved CNN Supervision},
 booktitle = {3D Vision (3DV), 2017 Fifth International Conference on},
 url = {http://gvv.mpi-inf.mpg.de/3dhp_dataset},
 year = {2017},
 organization={IEEE},
 doi={10.1109/3dv.2017.00064},
}
```

For MPI-INF-3DHP, download and extract them under `$MMHUMAN3D/data/datasets`, and make them look like this:

```
mmhuman3d
├── mmhuman3d
├── docs
├── tests
├── tools
├── configs
└── data
    └── datasets
        └── mpi_inf_3dhp
            ├── mpi_inf_3dhp_test_set
            │   ├── TS1
            │   ├── TS2
            │   ├── TS3
            │   ├── TS4
            │   ├── TS5
            │   └── TS6
            ├── S1
            │   ├── Seq1
            │   └── Seq2
            ├── S2
            │   ├── Seq1
```

(continues on next page)

(continued from previous page)

```
        └── Seq2
    ├── S3
        ├── Seq1
        └── Seq2
    ├── S4
        ├── Seq1
        └── Seq2
    ├── S5
        ├── Seq1
        └── Seq2
    ├── S6
        ├── Seq1
        └── Seq2
    ├── S7
        ├── Seq1
        └── Seq2
    └── S8
        ├── Seq1
        └── Seq2
```

5.3.13 MPII

```
@inproceedings{andriluka14cvpr,
  author = {Mykhaylo Andriluka and Leonid Pishchulin and Peter Gehler and Schiele, Bernt}
↪,
  title = {2D Human Pose Estimation: New Benchmark and State of the Art Analysis},
  booktitle = {IEEE Conference on Computer Vision and Pattern Recognition (CVPR)},
  year = {2014},
  month = {June}
}
```

For MPII data, please download from MPII Human Pose Dataset. Extract them under `$MMHUMAN3D/data/datasets`, and make them look like this:

```
mmhuman3d
├── mmhuman3d
├── docs
├── tests
├── tools
├── configs
└── data
    └── datasets
        └── mpii
            |── train.h5
            └── images
                |── 000001163.jpg
                |── 000003072.jpg
                └── ...
```

5.3.14 PoseTrack18

```
@inproceedings{andriluka2018posetrack,
  title={Posetrack: A benchmark for human pose estimation and tracking},
  author={Andriluka, Mykhaylo and Iqbal, Umar and Insafutdinov, Eldar and Pishchulin,␣
↪Leonid and Milan, Anton and Gall, Juergen and Schiele, Bernt},
  booktitle={Proceedings of the IEEE Conference on Computer Vision and Pattern␣
↪Recognition},
  pages={5167--5176},
  year={2018}
}
```

For PoseTrack18 data, please download from PoseTrack18. Extract them under `$MMHUMAN3D/data/datasets`, and make them look like this:

```
mmhuman3d
├── mmhuman3d
├── docs
├── tests
├── tools
├── configs
└── data
    └── datasets
        └── posetrack
            ├── images
            │   ├── train
            │   │   ├── 000001_bonn_train
            │   │   │   ├── 000000.jpg
            │   │   │   ├── 000001.jpg
            │   │   │   └── ...
            │   │   └── ...
            │   ├── val
            │   │   ├── 000342_mpii_test
            │   │   │   ├── 000000.jpg
            │   │   │   ├── 000001.jpg
            │   │   │   └── ...
            │   │   └── ...
            │   └── test
            │       ├── 000001_mpiinew_test
            │       │   ├── 000000.jpg
            │       │   ├── 000001.jpg
            │       │   └── ...
            │       └── ...
            └── posetrack_data
                └── annotations
                    ├── train
                    │   ├── 000001_bonn_train.json
                    │   ├── 000002_bonn_train.json
                    │   └── ...
                    ├── val
                    │   ├── 000342_mpii_test.json
                    │   ├── 000522_mpii_test.json
                    │   └── ...
```

(continues on next page)

(continued from previous page)

```
                    └── test
                        ├── 000001_mpiinew_test.json
                        ├── 000002_mpiinew_test.json
                        └── ...
```

5.3.15 Penn Action

```
@inproceedings{zhang2013pennaction,
 title={From Actemes to Action: A Strongly-supervised Representation for Detailed Action␣
↪Understanding},
 author={Zhang, Weiyu and Zhu, Menglong and Derpanis, Konstantinos},
 booktitle={ICCV},
 year={2013}
}
```

For Penn Action data, please download from Penn Action. Extract them under `$MMHUMAN3D/data/datasets`, and make them look like this:

```
mmhuman3d
├── mmhuman3d
├── docs
├── tests
├── tools
├── configs
└── data
    └── datasets
        └── penn_action
            ├── frames
            │   ├── 0001
            │   │   ├── 000001.jpg
            │   │   ├── 000002.jpg
            │   │   └── ...
            │   └── ...
            └── labels
                ├── 0001.mat
                ├── 0002.mat
                └── ...
```

5.3.16 PW3D

```
@inproceedings{vonMarcard2018,
title = {Recovering Accurate 3D Human Pose in The Wild Using IMUs and a Moving Camera},
author = {von Marcard, Timo and Henschel, Roberto and Black, Michael and Rosenhahn, Bodo␣
↪and Pons-Moll, Gerard},
booktitle = {European Conference on Computer Vision (ECCV)},
year = {2018},
month = {sep}
}
```

For PW3D data, please download from PW3D Dataset. Extract them under `$MMHUMAN3D/data/datasets`, and make them look like this:

```
mmhuman3d
├── mmhuman3d
├── docs
├── tests
├── tools
├── configs
└── data
    └── datasets
        └── pw3d
            |── imageFiles
            |   |    └── courtyard_arguing_00
            |   |       ├── image_00000.jpg
            |   |       ├── image_00001.jpg
            |   |       └── ...
            └── sequenceFiles
                ├── train
                |   ├── downtown_arguing_00.pkl
                |   └── ...
                ├── val
                |   ├── courtyard_arguing_00.pkl
                |   └── ...
                └── test
                    ├── courtyard_basketball_00.pkl
                    └── ...
```

5.3.17 SPIN

```
@inproceedings{kolotouros2019spin,
  author = {Kolotouros, Nikos and Pavlakos, Georgios and Black, Michael J and Daniilidis,
↪ Kostas},
  title = {Learning to Reconstruct 3D Human Pose and Shape via Model-fitting in the Loop}
↪,
  booktitle={ICCV},
  year={2019}
}
```

For SPIN, please download the preprocessed npz files and place them in the folder structure below:

```
mmhuman3d
├── mmhuman3d
├── docs
├── tests
├── tools
├── configs
└── data
    └── datasets
        └── spin_data
            ├── coco_2014_train.npz
            ├── hr-lspet_train.npz
```

(continues on next page)

(continued from previous page)

```
            ├── lsp_dataset_original_train.npz
            ├── mpi_inf_3dhp_train.npz
            └── mpii_train.npz
```

5.3.18 SURREAL

```
@inproceedings{varol17_surreal,
 title     = {Learning from Synthetic Humans},
 author    = {Varol, G{\"u}l and Romero, Javier and Martin, Xavier and Mahmood, Naureen␣
↪and Black, Michael J. and Laptev, Ivan and Schmid, Cordelia},
 booktitle = {CVPR},
 year      = {2017}
}
```

For SURREAL, please download the [dataset] (https://www.di.ens.fr/willow/research/surreal/data/) and place them in the folder structure below:

```
mmhuman3d
├── mmhuman3d
├── docs
├── tests
├── tools
├── configs
└── data
    └── datasets
        └── surreal
            ├── train
            |   ├── run0
            |   |   ├── 03_01
            |   |   |   ├── 03_01_c0001_depth.mat
            |   |   |   ├── 03_01_c0001_info.mat
            |   |   |   ├── 03_01_c0001_segm.mat
            |   |   |   ├── 03_01_c0001.mp4
            |   |   |   └── ...
            |   |   └── ...
            |   ├── run1
            |   └── run2
            ├── val
            |   ├── run0
            |   ├── run1
            |   └── run2
            └── test
                ├── run0
                ├── run1
                └── run2
```

5.3.19 VIBE

```
@inproceedings{VIBE,
  author    = {Muhammed Kocabas and
               Nikos Athanasiou and
               Michael J. Black},
  title     = {{VIBE}: Video Inference for Human Body Pose and Shape Estimation},
  booktitle = {CVPR},
  year      = {2020}
}
```

For VIBE, please download the preprocessed mpi_inf_3dhp and pw3d npz files from SPIN and pretrained frame feature extractor spin.pth. Place them in the folder structure below:

```
mmhuman3d
├── mmhuman3d
├── docs
├── tests
├── tools
├── configs
└── data
    ├── checkpoints
    |   └── spin.pth
    └── datasets
        └── vibe_data
            ├── mpi_inf_3dhp_train.npz
            └── pw3d_test.npz
```

CHAPTER

SIX

KEYPOINTS CONVENTION

6.1 Overview

Our convention tries to consolidate the different keypoints definition across various commonly used datasets. Due to differences in data-labelling procedures, keypoints across datasets with the same name might not map to semantically similar locations on the human body. Conversely, keypoints with different names might correspond to the same location on the human body. To unify the different keypoints correspondences across datasets, we adopted the `human_data` convention as the base convention for converting and storing our keypoints.

6.2 How to use

6.2.1 Converting between conventions

Keypoints can be converted between different conventions easily using the `convert_kps` function.

To convert a `human_data` keypoints to `coco` convention, specify the source and destination convention for conversion.

```
from mmhuman3d.core.conventions.keypoints_mapping import convert_kps

keypoints_human_data = np.zeros((100, 190, 3))
keypoints_coco, mask = convert_kps(keypoints_human_data, src='human_data', dst='coco')
assert mask.all()==1
```

The output `mask` should be all ones if the `dst` convention is the subset of the `src` convention. You can use the `mask` as the confidence of the keypoints since those keypoints with no correspondence are set to a default value with 0 confidence.

6.2.2 Converting with confidence

If you have confidential information of your keypoints, you can use an original mask to mark it, then the information will be updated into the returned mask. E.g., you want to convert a `smpl` keypoints to `coco` keypoints, and you know its `left_shoulder` is occluded. You want to carry forward this information during the converting. So you can set an original_mask and convert it to `coco` by doing:

```
import numpy as np
from mmhuman3d.core.conventions.keypoints_mapping import KEYPOINTS_FACTORY, convert_kps

keypoints = np.zeros((1, len(KEYPOINTS_FACTORY['smpl']), 3))
```

(continues on next page)

(continued from previous page)

```
confidence = np.ones((len(KEYPOINTS_FACTORY['smpl'])))

# assume that 'left_shoulder' point is invalid.
confidence[KEYPOINTS_FACTORY['smpl'].index('left_shoulder')] = 0

_, conf_coco = convert_kps(
    keypoints=keypoints, confidence=confidence, src='smpl', dst='coco')
_, conf_coco_full = convert_kps(
    keypoints=keypoints, src='smpl', dst='coco')

assert conf_coco[KEYPOINTS_FACTORY['coco'].index('left_shoulder')] == 0
conf_coco[KEYPOINTS_FACTORY['coco'].index('left_shoulder')] = 1
assert (conf_coco == conf_coco_full).all()
```

Our mask represents valid information, its dtype is uint8, while keypoint confidence usually ranges from 0 to 1. E.g., you want to convert a `smpl` keypoints to `coco` keypoints, and you know its `left_shoulder` is occluded. You want to carry forward this information during the converting. So you can set an original_mask and convert it to `coco` by doing:

```
confidence = np.ones((len(KEYPOINTS_FACTORY['smpl'])))
confidence[KEYPOINTS_FACTORY['smpl'].index('left_shoulder')] = 0.5
kp_smpl = np.concatenate([kp_smpl, confidence], -1)
kp_smpl_converted, mask = convert_kps(kp_smpl, src='smpl', dst='coco')
new_confidence =  kp_smpl_converted[..., 2:]
assert new_confidence[KEYPOINTS_FACTORY['smpl'].index('left_shoulder')] == 0.5
```

6.3 Supported Conventions

These are the supported conventions:

- *AGORA*
- *COCO*
- *COCO-WHOLEBODY*
- *CrowdPose*
- *GTA-Human*
- *Human3.6M*
- human_data
- *HybrIK*
- *LSP*
- *MPI-INF-3DHP*
- *MPII*
- openpose
- *PennAction*
- *PoseTrack18*
- *PW3D*

- *SMPL*
- *SMPL-X*

6.3.1 HUMANDATA

The first 144 keypoints in HumanData correspond to that in `SMPL-X`. Keypoints with suffix `_extra` refer to those obtained from Jregressor_extra. Keypoints with suffix `_openpose` refer to those obtained from `OpenPose` predictions.

There are several keypoints from `MPI-INF-3DHP`, `Human3.6M` and `Posetrack` that has the same name but were semantically different from keypoints in `SMPL-X`. As such, we added an extra suffix to differentiate those keypoints i.e. `head_h36m`.

6.3.2 AGORA

```
@inproceedings{Patel:CVPR:2021,
  title = {{AGORA}: Avatars in Geography Optimized for Regression Analysis},
  author = {Patel, Priyanka and Huang, Chun-Hao P. and Tesch, Joachim and Hoffmann,␣
↪David T. and Tripathi, Shashank and Black, Michael J.},
  booktitle = {Proceedings IEEE/CVF Conf.~on Computer Vision and Pattern Recognition (
↪{CVPR})},
  month = jun,
  year = {2021},
  month_numeric = {6}
}
```

6.3.3 COCO

```
@inproceedings{lin2014microsoft,
  title={Microsoft coco: Common objects in context},
  author={Lin, Tsung-Yi and Maire, Michael and Belongie, Serge and Hays, James and␣
↪Perona, Pietro and Ramanan, Deva and Doll{\'a}r, Piotr and Zitnick, C Lawrence},
  booktitle={European conference on computer vision},
  pages={740--755},
  year={2014},
  organization={Springer}
}
```

6.3.4 COCO-WHOLEBODY

```
@inproceedings{jin2020whole,
  title={Whole-Body Human Pose Estimation in the Wild},
  author={Jin, Sheng and Xu, Lumin and Xu, Jin and Wang, Can and Liu, Wentao and Qian,␣
↪Chen and Ouyang, Wanli and Luo, Ping},
  booktitle={Proceedings of the European Conference on Computer Vision (ECCV)},
  year={2020}
}
```

6.3.5 CrowdPose

```
@article{li2018crowdpose,
  title={CrowdPose: Efficient Crowded Scenes Pose Estimation and A New Benchmark},
  author={Li, Jiefeng and Wang, Can and Zhu, Hao and Mao, Yihuan and Fang, Hao-Shu and␣
↪Lu, Cewu},
  journal={Proceedings IEEE/CVF Conf.~on Computer Vision and Pattern Recognition ({CVPR}
↪)},
  year={2019}
}
```

6.3.6 Human3.6M

```
@article{h36m_pami,
  author = {Ionescu, Catalin and Papava, Dragos and Olaru, Vlad and Sminchisescu, ␣
↪Cristian},
  title = {Human3.6M: Large Scale Datasets and Predictive Methods for 3D Human Sensing␣
↪in Natural Environments},
  journal = {IEEE Transactions on Pattern Analysis and Machine Intelligence},
  publisher = {IEEE Computer Society},
  volume = {36},
  number = {7},
  pages = {1325-1339},
  month = {jul},
  year = {2014}
}
```

6.3.7 GTA-Human

```
@article{cai2021playing,
  title={Playing for 3D Human Recovery},
  author={Cai, Zhongang and Zhang, Mingyuan and Ren, Jiawei and Wei, Chen and Ren,␣
↪Daxuan and Li, Jiatong and Lin, Zhengyu and Zhao, Haiyu and Yi, Shuai and Yang, Lei␣
↪and others},
  journal={arXiv preprint arXiv:2110.07588},
  year={2021}
}
```

6.3.8 HybrIK

```
@inproceedings{li2020hybrikg,
  author = {Li, Jiefeng and Xu, Chao and Chen, Zhicun and Bian, Siyuan and Yang, Lixin␣
↪and Lu, Cewu},
  title = {HybrIK: A Hybrid Analytical-Neural Inverse Kinematics Solution for 3D Human␣
↪Pose and Shape Estimation},
  booktitle={CVPR 2021},
  pages={3383--3393},
  year={2021},
```

(continues on next page)

(continued from previous page)

```
  organization={IEEE}
}
```

6.3.9 LSP

```
@inproceedings{johnson2010clustered,
  title={Clustered Pose and Nonlinear Appearance Models for Human Pose Estimation.},
  author={Johnson, Sam and Everingham, Mark},
  booktitle={bmvc},
  volume={2},
  number={4},
  pages={5},
  year={2010},
  organization={Citeseer}
}
```

6.3.10 MPI-INF-3DHP

```
@inproceedings{mono-3dhp2017,
 author = {Mehta, Dushyant and Rhodin, Helge and Casas, Dan and Fua, Pascal and␣
↪Sotnychenko, Oleksandr and Xu, Weipeng and Theobalt, Christian},
 title = {Monocular 3D Human Pose Estimation In The Wild Using Improved CNN Supervision},
 booktitle = {3D Vision (3DV), 2017 Fifth International Conference on},
 url = {http://gvv.mpi-inf.mpg.de/3dhp_dataset},
 year = {2017},
 organization={IEEE},
 doi={10.1109/3dv.2017.00064},
}
```

6.3.11 MPII

```
@inproceedings{andriluka14cvpr,
  author = {Mykhaylo Andriluka and Leonid Pishchulin and Peter Gehler and Schiele, Bernt}
↪,
  title = {2D Human Pose Estimation: New Benchmark and State of the Art Analysis},
  booktitle = {IEEE Conference on Computer Vision and Pattern Recognition (CVPR)},
  year = {2014},
  month = {June}
}
```

6.3.12 PoseTrack18

```
@inproceedings{andriluka2018posetrack,
  title={Posetrack: A benchmark for human pose estimation and tracking},
  author={Andriluka, Mykhaylo and Iqbal, Umar and Insafutdinov, Eldar and Pishchulin,␣
↪Leonid and Milan, Anton and Gall, Juergen and Schiele, Bernt},
  booktitle={Proceedings of the IEEE Conference on Computer Vision and Pattern␣
↪Recognition},
  pages={5167--5176},
  year={2018}
}
```

6.3.13 OpenPose

```
@article{8765346,
  author = {Z. {Cao} and G. {Hidalgo Martinez} and T. {Simon} and S. {Wei} and Y. A.
↪{Sheikh}},
  journal = {IEEE Transactions on Pattern Analysis and Machine Intelligence},
  title = {OpenPose: Realtime Multi-Person 2D Pose Estimation using Part Affinity Fields}
↪,
  year = {2019}
}
```

6.3.14 PennAction

```
@inproceedings{zhang2013,
  title={From Actemes to Action: A Strongly-supervised Representation for Detailed␣
↪Action Understanding},
  author={Zhang, Weiyu and Zhu, Menglong and Derpanis, Konstantinos},
  booktitle={Proceedings of the International Conference on Computer Vision},
  year={2013}
}
```

6.3.15 SMPL

```
@article{SMPL:2015,
      author = {Loper, Matthew and Mahmood, Naureen and Romero, Javier and Pons-Moll,␣
↪Gerard and Black, Michael J.},
      title = {{SMPL}: A Skinned Multi-Person Linear Model},
      journal = {ACM Trans. Graphics (Proc. SIGGRAPH Asia)},
      month = oct,
      number = {6},
      pages = {248:1--248:16},
      publisher = {ACM},
      volume = {34},
      year = {2015}
    }
```

6.3.16 SMPL-X

```
@inproceedings{SMPL-X:2019,
  title = {Expressive Body Capture: {3D} Hands, Face, and Body from a Single Image},
  author = {Pavlakos, Georgios and Choutas, Vasileios and Ghorbani, Nima and Bolkart,␣
↪Timo and Osman, Ahmed A. A. and Tzionas, Dimitrios and Black, Michael J.},
  booktitle = {Proceedings IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)},
  pages     = {10975--10985},
  year = {2019}
}
```

6.3.17 Customizing keypoint convention

Please refer to *customize_keypoints_convention*.

CHAPTER

SEVEN

CUSTOMIZE KEYPOINTS CONVENTION

7.1 Overview

If your dataset use an unsupported convention, a new convention can be added following this documentation.

These are the conventions that our project currently support:

- agora
- coco
- coco_wholebody
- crowdpose
- h36m
- human_data
- hybrik
- lsp
- mpi_inf_3dhp
- mpii
- openpose
- penn_action
- posetrack
- pw3d
- smpl
- smplx

1. Create a new convention

Please follow `mmhuman3d/core/conventions/keypoints_mapping/human_data.py` to create a file named NEW_CONVENTION.py. In this file, `NEW_KEYPOINTS` is a list containing keypoints naming and order specific to the new convention.

For instance, if we want to create a new convention for AGORA dataset, `agora.py` would contain:

```
AGORA_KEYPOINTS = [
  'pelvis',
  'left_hip',
```

(continues on next page)

(continued from previous page)

```
  'right_hip'
  ...
]
```

2. Search for keypoint names in human_data.

In this project, keypoints that share the same naming across datasets should have the exact same semantic definition in the human body. human_data convention has already consolidated the different keypoints naming and correspondences across our supported datasets.

For each keypoint in NEW_KEYPOINTS, we have to check (1) if the keypoint name exists in mmhuman3d/core/conventions/keypoints_mapping/human_data.py and (2) if the keypoint has a correspondence i.e. maps to the same location as the ones defined in human_data.

If both conditions are met, retain the keypoint name in NEW_CONVENTION.py.

3. Search for keypoints correspondence in human_data.

If a keypoint in NEW_KEYPOINTS shares the same correspondence as a keypoint that is named differently in the human_data convention i.e. head in NEW_CONVENTION.py maps to head_extra in human_data, rename the keypoint to follow the new one in our convention i.e. head-> head_extra.

3. Add a new keypoint to human_data

If the keypoint has no correspondence nor share an existing name to the ones defined in human_data, please list it as well but add a prefix to the original name to differentiate it from those with existing correspondences i.e. spine_3dhp

We may expand human_data to the new keypoint if necessary. However, this can only be done after checking that the new keypoint do not have a correspondence and there is no conflicting names.

4. Initialise the new set of keypoint convention

Add import for NEW_CONVENTION.py in mmhuman3d/core/conventions/keypoints_mapping/__init__.py, and add the identifier to dict KEYPOINTS_FACTORY.

For instance, if our new convention is agora:

```
# add import
from mmhuman3d.core.conventions.keypoints_mapping import (
    agora,
    ...
)

# add to factory
KEYPOINTS_FACTORY = {
    'agora': agora.AGORA_KEYPOINTS,
    ...
}
```

5. Using keypoints convention for keypoints mapping

To convert keypoints from any existing convention to your newly defined convention (or vice versa), you can use the convert_kps function mmhuman3d/core/conventions/keypoints_mapping/__init__.py, which produce a mask containing 0 or 1 indicating if the corresponding point should be filtered or retained.

To convert from coco to new convention:

```
new_kps, mask = convert_kps(smplx_keypoints, src='coco', dst='NEW_CONVENTION')
```

To convert from new convention to human_data:

```
new_kps, mask = convert_kps(smplx_keypoints, src='NEW_CONVENTION', dst='human_data')
```

CHAPTER

EIGHT

CAMERAS

8.1 Camera Initialization

We follow `Pytorch3D` cameras. The camera extrinsic matrix is defined as the camera to world transformation, and uses right matrix multiplication, whereas the intrinsic matrix uses left matrix multiplication. Nevertheless, our interface provides `opencv` convention that defines the camera the same way as an `OpenCV` camera, would be helpful if you are more familiar with that.

- **Slice cameras:**

  In mmhuman3d, the recommended way to initialize a camera is by passing `K`, `R`, `T` matrix directly. You can slice the cameras by index. You can also concat the cameras in batch dim.

```
from mmhuman3d.core.cameras import PerspectiveCameras
import torch
K = torch.eye(4, 4)[None]
R = torch.eye(3, 3)[None]
T = torch.zeros(100, 3)
# Batch of K, R, T should all be the same or some of them could be 1. The final␣
↪batch size will be the biggest one.
cam = PerspectiveCameras(K=K, R=R, T=T)
assert cam.R.shape == (100, 3, 3)
assert cam.K.shape == (100, 4, 4)
assert cam.T.shape == (100, 3)
assert (cam[:10].K == cam.K[:10]).all()
```

- **Build cameras:**

  Wrapped by mmcv.Registry. In mmhuman3d, the recommended way to initialize a camera is by passing `K`, `R`, `T` matrix directly, but you also have the options to pass `focal_length` and `principle_point` as the input.

  Take the usually used `PerspectiveCameras` as examples. If `K`, `R`, `T` are not specified, the `K` will use default `K` by `compute_default_projection_matrix` with default `focal_length` and `principal_point` and `R` will be identical matrix, `T` will be zeros. You can also specify by overwriting the parameters for `compute_default_projection_matrix`.

```
from mmhuman3d.core.cameras import build_cameras

# Initialize a perspective camera with given K, R, T matrix.
# It is recommended that the batches of K, R, T either the same or be 1.
K = torch.eye(4, 4)[None]
R = torch.eye(3, 3)[None]
T = torch.zeros(10, 3)
```

(continues on next page)

(continued from previous page)

```
height, width = 1000
cam1 = build_cameras(
    dict(
        type='PerspectiveCameras',
        K=K,
        R=R,
        T=T,
        in_ndc=True,
        image_size=(height, width),
        convention='opencv',
        ))

# This is the same as:
cam2 = PerspectiveCameras(
        K=K,
        R=R,
        T=T,
        in_ndc=True,
        image_size=1000, # single number represents square images.
        convention='opencv',
        )
assert cam1.K.shape == cam2.K.shape == (10, 4, 4)
assert cam1.R.shape == cam2.R.shape == (10, 3, 3)
assert cam1.T.shape == cam2.T.shape == (10, 3)

# Initialize a perspective camera with specific `image_size`, `principal_points`,␣
↪`focal_length`.
# `in_ndc = False` means the intrinsic matrix `K` defined in screen space. The `focal_
↪length` and `principal_point` in `K` is defined in scale of pixels. This `principal_
↪points` is (500, 500) pixels and `focal_length` is 1000 pixels.
cam = build_cameras(
    dict(
        type='PerspectiveCameras',
        in_ndc=False,
        image_size=(1000, 1000),
        principal_points=(500, 500),
        focal_length=1000,
        convention='opencv',
        ))

assert (cam.K[0] == torch.Tensor([[1000., 0.,  500.,    0.],
                                  [0., 1000.,  500.,    0.],
                                  [0.,    0.,    0.,    1.],
                                  [0.,    0.,    1.,    0.]]).view(4, 4)).all()

# Initialize a weakperspective camera with given K, R, T. weakperspective camera␣
↪support `in_ndc = True` only.
cam = build_cameras(
    dict(
        type='WeakPerspectiveCameras',
        K=K,
```

(continues on next page)

(continued from previous page)

```
        R=R,
        T=T,
        image_size=(1000, 1000)
        ))

# If no `K`, `R`, `T` information provided
# Initialize a `in_ndc` perspective camera with default matrix.
cam = build_cameras(
    dict(
        type='PerspectiveCameras',
        in_ndc=True,
        image_size=(1000, 1000),
        ))
# Then convert it to screen. This operation requires `image_size`.
cam.to_screen_()
```

8.2 Camera Projection Matrixs

- **Perspective:**

  format of intrinsic matrix: fx, fy is focal_length, px, py is principal_point.

```
K = [
        [fx,   0,   px,   0],
        [0,   fy,   py,   0],
        [0,    0,    0,   1],
        [0,    0,    1,   0],
    ]
```

  Detailed information refer to Pytorch3D.

- **WeakPerspective:**

  format of intrinsic matrix:

```
K = [
        [sx*r,   0,    0,   tx*sx*r],
        [0,     sy,    0,   ty*sy],
        [0,     0,     1,       0],
        [0,     0,     0,       1],
    ]
```

  `WeakPerspectiveCameras` is orthographics indeed, mainly for SMPL(x) projection. Detailed information refer to mmhuman3d cameras. This can be converted from SMPL predicted camera parameter by:

```
from mmhuman3d.core.cameras import WeakPerspectiveCameras
K = WeakPerspectiveCameras.convert_orig_cam_to_matrix(orig_cam)
```

  The pred_cam is array/tensor of shape (frame, 4) consists of [scale_x, scale_y, transl_x, transl_y]. See in VIBE.

- **FoVPerspective:**

```
format of intrinsic matrix:
K = [
        [s1,   0,   w1,   0],
        [0,   s2,   h1,   0],
        [0,    0,   f1,  f2],
        [0,    0,    1,   0],
    ]
```

s1, s2, w1, h1, f1, f2 are defined by FoV parameters (`fov`, `znear`, `zfar`, etc.), detailed information refer to Pytorch3D.

- **Orthographics:**

  format of intrinsic matrix:

```
K = [
        [fx,   0,    0,  px],
        [0,   fy,    0,  py],
        [0,    0,    1,   0],
        [0,    0,    0,   1],
]
```

  Detailed information refer to Pytorch3D.

- **FoVOrthographics:**

```
K = [
        [scale_x,        0,         0,  -mid_x],
        [0,        scale_y,         0,  -mix_y],
        [0,              0,  -scale_z,  -mid_z],
        [0,              0,         0,       1],
]
```

  scale_x, scale_y, scale_z, mid_x, mid_y, mid_z are defined by FoV parameters(`min_x`, `min_y`, `max_x`, `max_y`, `znear`, `zfar`, etc.), related information refer to Pytorch3D.

8.3 Camera Conventions

- **Convert between different cameras:**

  We name intrinsic matrix as `K`, rotation matrix as `R` and translation matrix as `T`. Different camera conventions have different axis directions, and some use left matrix multiplication and some use right matrix multiplication. Intrinsic and extrinsic matrix should be of the same multiplication convention, but some conventions like `Pytorch3D` uses right matrix multiplication in computation procedure but passes left matrix multiplication `K` when initializing the cameras(mainly for better understanding). Conversion between `NDC` (normalized device coordinate) and `screen` also influence the intrinsic matrix, this is independent of camera conventions but should also be included. If you want to use an existing convention, choose in `['opengl', 'opencv', 'pytorch3d', 'pyrender', 'open3d']`. E.g., you want to convert your opencv calibrated camera to Pytorch3D NDC defined camera for rendering, you can do:

```
from mmhuman3d.core.conventions.cameras import convert_cameras
import torch
```

(continues on next page)

(continued from previous page)

```
K = torch.eye(4, 4)[None]
R = torch.eye(3, 3)[None]
T = torch.zeros(10, 3)
height, width = 1080, 1920
K, R, T = convert_cameras(
    K=K,
    R=R,
    T=T,
    in_ndc_src=False,
    in_ndc_dst=True,
    resolution_src=(height, width),
    convention_src='opencv',
    convention_dst='pytorch3d')
```

Input K could be None, or `array/tensor` of shape (batch_size, 3, 3) or (batch_size, 4, 4). Input R could be None, or `array/tensor` of shape (batch_size, 3, 3). Input T could be None, or `array/tensor` of shape (batch_size, 3). If the original `K` is `None`, it will remain `None`. If the original `R` is `None`, it will be set as identity matrix. If the original `T` is `None`, it will be set as zeros matrix. Please refer to Pytorch3D for more information about cameras in `NDC` and in `screen` space..

- **Define your new camera convention:**

  If want to use a new convention, define your convention in CAMERA_CONVENTION_FACTORY by the order of right to, up to, and off screen. E.g., the first one is pyrender and its convention should be '+x+y+z'. '+' could be ignored. The second one is opencv and its convention should be '+x-y-z'. The third one is Pytorch3D and its convention should be '-xyz'.

```
OpenGL(PyRender)        OpenCV              Pytorch3D
    y                     z                     y
    |                    /                      |
    |                   /                       |
    |_______x         /________x     x________  |
    /                 |                       /
   /                  |                      /
z /                 y |                   z /
```

8.4 Some Conversion Functions

Convert functions are also defined in conventions.cameras.

- **NDC & screen:**

```
from mmhuman3d.core.conventions.cameras import (convert_ndc_to_screen,
                                                convert_screen_to_ndc)

K = convert_ndc_to_screen(K, resolution=(1080, 1920), is_perspective=True)
K = convert_screen_to_ndc(K, resolution=(1080, 1920), is_perspective=True)
```

- **3x3 & 4x4 intrinsic matrix**

```
from mmhuman3d.core.conventions.cameras import (convert_K_3x3_to_4x4,
                                                convert_K_4x4_to_3x3)
```

(continues on next page)

(continued from previous page)

```
K = convert_K_3x3_to_4x4(K, is_perspective=True)
K = convert_K_4x4_to_3x3(K, is_perspective=True)
```

- **world & view:**

  Convert between world & view coordinates.

```
from mmhuman3d.core.conventions.cameras import convert_world_view
R, T = convert_world_view(R, T)
```

- **weakperspective & perspective:**

  Convert between weakperspective & perspective. zmean is needed. WeakperspectiveCameras is in_ndc, so you should pass resolution if perspective not in ndc.

```
from mmhuman3d.core.conventions.cameras import (
    convert_perspective_to_weakperspective,
    convert_weakperspective_to_perspective)

K = convert_perspective_to_weakperspective(
    K, zmean, in_ndc=False, resolution, convention='opencv')
K = convert_weakperspective_to_perspective(
    K, zmean, in_ndc=False, resolution, convention='pytorch3d')
```

8.5 Some Compute Functions

- **Project 3D coordinates to screen:**

```
points_xydepth = cameras.transform_points_screen(points)
points_xy = points_xydepth[..., :2]
```

- **Compute depth of points:**

  You can simply convert points to the view coordinates and get the z value as depth. Example could be found in DepthRenderer.

```
points_depth = cameras.compute_depth_of_points(points)
```

- **Compute normal of meshes:**

  Use `Pytorch3D` to compute normal of meshes. Example could be found in NormalRenderer.

```
normals = cameras.compute_normal_of_meshes(meshes)
```

- **Get camera plane normal:**

  Get the normalized normal tensor which points out of the camera plane from camera center.

```
normals = cameras.get_camera_plane_normals()
```

CHAPTER

NINE

VISUALIZE KEYPOINTS

9.1 Visualize 2d keypoints

- **simple example for visualize 2d keypoints:**

  You have 2d coco_wholebody keypoints of shape(10, 133, 2).

```
from mmhuman3d.core.visualization.visualize_keypoints2d import visualize_kp2d

visualize_kp2d(
    kp2d_coco_wholebody,
    data_source='coco_wholebody',
    output_path='some_video.mp4',
    resolution=(1024, 1024))
```

  Then a 1024x1024 sized video with 10 frames would be save as 'some_video.mp4'

- **data_source and mask:**

  If your keypoints have some nonsense points, you should provide the mask. `data_source` is mainly used to search the limb connections and palettes. You should specify the data_source if you dataset is in convention. E.g., convert coco_wholebody keypoints to the convention of smpl and visualize it:

```
from mmhuman3d.core.conventions.keypoints_mapping import convert_kps
from mmhuman3d.core.visualization.visualize_keypoints2d import visualize_kp2d

kp2d_smpl, mask = convert_kps(kp2d_coco_wholebody, src='coco_wholebody', dst='smpl')
visualize_kp2d(
    kp2d_smpl,
    mask=mask,
    output_path='some_video.mp4',
    resolution=(1024, 1024))
```

  mask is `None` by default. This is the same as all ones mask, then no keypoints will be excluded. Ignore it when you are sure that all the keypoints are valid.

- **whether plot on backgrounds:**

  Maybe you want to use numpy input backgrounds.

  E.g., you want to visualize you coco_wholebody kp2d as smpl convention. You have 2d coco_wholebody keypoints of shape(10, 133, 2).

```
from mmhuman3d.core.conventions.keypoints_mapping import convert_kps
from mmhuman3d.core.visualization.visualize_keypoints2d import visualize_kp2d

background = np.random.randint(low=0, high=255, shape=(10, 1024, 1024, 4))
# multi_person, shape is (num_person, num_joints, 2)
out_image = visualize_kp2d(
    kp2d=kp2d, image_array=background, data_source='coco_wholebody', return_
↪array=True)
```

This is just an example, you can use this function flexibly.

If want to plot keypoints on frame files, you could provide `frame_list`(list of image path). **Be aware that the order of the frame will be sorted by name.** or `origin_frames`(mp4 path or image folder path), **Be aware that you should provide the correct `img_format` for `ffmpeg` to read the images.**.

```
frame_list = ['im1.png', 'im2.png', ...]
visualize_kp2d(
    kp2d_coco_wholebody,
    data_source='coco_wholebody',
    output_path='some_video.mp4',
    resolution=(1024, 1024),
    frame_list=frame_list)

origin_frames = 'some_folder'
visualize_kp2d(
    kp2d_coco_wholebody,
    data_source='coco_wholebody',
    output_path='some_video.mp4',
    resolution=(1024, 1024),
    origin_frames=origin_frames)

origin_frames = 'some.mp4'
array = visualize_kp2d(
    kp2d_coco_wholebody,
    data_source='coco_wholebody',
    output_path='some_video.mp4',
    resolution=(1024, 1024),
    return_array=True,
    origin_frames=origin_frames)
```

The superiorty of background images: `frame_list`

- **output a video or frames:**

  If `output_path` is a folder, this function will output frames. If `output_path` is a '.mp4' path, this function will output a video. `output_path` could be set as `None` when `return_array` is True. The function will return an array of shape (frame, width, height, 3).

- **whether plot origin file name on images:**

  Specify `with_file_name=True` then origin frame name will be plotted on the image.

- **dataset not in existing convention or want to visualize some specific limbs:**

  You should provide limbs like `limbs=[[0, 1], ..., [10, 11]]` if you dataset is not in convention.

- **other parameters:**

  Easy to understand, please read the doc strings in the function.

9.2 Visualize 3d keypoints

- **simple example for visualize single person:**

  You have kp3d in smplx convention of shape (num_frame, 144, 3).

```
visualize_kp3d(kp3d=kp3d, data_source='smplx', output_path='some_video.mp4')
```

  The result video would have one person dancing, each body part has its own color.

- **simple example for visualize multi person:**

  You have kp3d_1 and kp3d_2 which are both in smplx convention of shape (num_frame, 144, 3).

```
kp3d = np.concatenate([kp3d_1[:, np.newaxis], kp3d_2[:, np.newaxis]], axis=1)
# kp3d.shape is now (num_frame, num_person, 144, 3)
visualize_kp3d(kp3d=kp3d, data_source='smplx', output_path='some_video.mp4')
```

  The result video would have two person dancing, each in a pure color, and the there will be a color legend describing the index of each person.

- **data_source and mask:**

  The same as *visualize_kp2d*

- **dataset not in existing convention or want to visualize some specific limbs:**

  The same as *visualize_kp2d*

- **output:** If `output_path` is a folder, this function will output frames. If `output_path` is a '.mp4' path, this function will output a video. `output_path` could be set as `None` when `return_array` is True. The function will return an array of shape (frame, width, height, 3).

- **other parameters:**

  Easy to understand, please read the doc strings in the function.

9.3 About ffmpeg_utils

- In ffmpeg_utils , each function has abundant doc strings, and the semantically defined function names could be easily understood.

- **read files:**

  images_to_array, video_to_array

- **write files:**

  array_to_images, array_to_video

- **convert formats:**

  gif_to_images, gif_to_video, video_to_images, video_to_gif, images_to_gif, images_to_video

- **temporally crop/concat:**

  slice_video, temporal_concat_video

- **spatially crop/concat:**

  crop_video, spatial_concat_video

- **compress:**

  compress_gif, compress_video

CHAPTER

TEN

VISUALIZE SMPL MESH

- **fast visualize smpl(x) pose without background images:**

  You have smpl pose tensor or array shape of which is (frame, 72)

```
from mmhuman3d.core.visualization import visualize_smpl_pose

visualize_smpl_pose(
    poses=poses,
    model_path=model_path,
    output_path='smpl.mp4',
    model_type='smpl',
    resolution=(1024, 1024))
```

  Or you have smplx pose tensor or array shape of which is (frame, 165)

```
visualize_smpl_pose(
    poses=poses,
    model_path=model_path,
    output_path='smplx.mp4',
    model_type='smplx',
    resolution=(1024, 1024))
```

  You could also feed dict tensor of smplx definitions. You could check that in visualize_smpl or original smplx.

- **visualize T-pose:** If you want to visualize a T-pose smpl or your poses do not have global_orient, you can do:

```
import torch
from mmhuman3d.core.visualization import visualize_T_pose

visualize_T_pose(
    num_frames=100,
    model_type='smpl',
    model_path=model_path,
    output_path='smpl_tpose.mp4',
    orbit_speed=(1, 0.5),
    resolution=(1024, 1024))
```

- **visualize smpl with predicted VIBE camera:** You have poses (numpy/tensor) of shape (frame, 72), betas of shape (frame, 10), pred_cam of shape (10, 4). E.g., we use vibe sample_video.mp4 as an example.

```
import pickle
from mmhuman3d.core.visualization import visualize_smpl_vibe
```

(continues on next page)

(continued from previous page)

```
with open('vibe_output.pkl', 'rb') as f:
    d = pickle.load(f, encoding='latin1')
poses = d[1]['pose']
orig_cam = d[1]['orig_cam']
pred_cam = d[1]['pred_cam']
bbox = d[1]['bboxes']
gender = 'female'

# pass pred_cam & bbox
visualize_smpl_vibe(
    poses=poses,
    betas=betas,
    gender=gender,
    pred_cam=pred_cam,
    bbox=bbox,
    model_type='smpl',
    model_path=model_path,
    output_path='vibe_demo.mp4',
    origin_frames='sample_video.mp4',
    resolution=(1024, 1024))

# or pass orig_cam
visualize_smpl_vibe(
    poses=poses,
    betas=betas,
    gender=gender,
    orig_cam=orig_cam,
    model_type='smpl',
    model_path=model_path,
    output_path='vibe_demo.mp4',
    origin_frames='sample_video.mp4',
    resolution=(1024, 1024))
```

- **visualize smpl with predicted HMR/SPIN camera:** You have poses (numpy/tensor) of shape (frame, 72), betas of shape (frame, 10), cam_translation of shape (10, 4). E.g., we use vibe sample_video.mp4 as an example.

```
import pickle
from mmhuman3d.core.visualization import visualize_smpl_hmr
gender = 'female'
focal_length = 5000
det_width = 224
det_height = 224

# you can pass smpl poses & betas & gender
visualize_smpl_hmr(
    poses=poses,
    betas=betas,
    gender=gender,
    bbox=bbox,
    focal_length=focal_length,
    det_width=det_width,
```

(continues on next page)

(continued from previous page)

```
    det_height=det_height,
    T=cam_translation,
    model_type='smpl',
    model_path=model_path,
    output_path='hmr_demo.mp4',
    origin_frames=image_folder,
    resolution=(1024, 1024))

# or you can pass verts
visualize_smpl_hmr(
    verts=verts,
    bbox=bbox,
    focal_length=focal_length,
    det_width=det_width,
    det_height=det_height,
    T=cam_translation,
    model_type='smpl',
    model_path=model_path,
    output_path='hmr_demo.mp4',
    origin_frames=image_folder,
    resolution=(1024, 1024))

# you can also pass kp2d in replace of bbox.
visualize_smpl_hmr(
    verts=verts,
    kp2d=kp2d,
    focal_length=focal_length,
    det_width=det_width,
    det_height=det_height,
    T=cam_translation,
    model_type='smpl',
    model_path=model_path,
    output_path='hmr_demo.mp4',
    origin_frames=image_folder,
    resolution=(1024, 1024))
```

- **visualize smpl with opencv camera:** You should pass the opencv defined intrinsic matrix K and extrinsic matrix R, T.

```
from mmhuman3d.core.visualization import visualize_smpl_calibration

visualize_smpl_calibration(
    poses=poses,
    betas=betas,
    gender=gender,
    transl=transl,
    model_type='smpl',
    K=K,
    R=R,
    T=T,
    model_path=model_path,
    output_path='opencv.mp4',
```

(continues on next page)

(continued from previous page)

```
    origin_frames='bg_video.mp4',
    resolution=(1024, 1024))
```

10.1 Different render_choice:

- **visualize mesh:** This is independent of cameras and you could directly set `render_choice` as `hq`(high quality), `mq`(medium quality) or `lq`(low quality).
- **visualize binary silhouettes:** This is independent of cameras and you could directly set `render_choice` as `silhouette`. The output video/images will be binary masks.
- **visualize body part silhouette:** This is independent of cameras and you could directly set `render_choice` as `part_silhouette`. The output video/images will be body part segmentation masks.
- **visualize depth map:** This is independent of cameras and you could directly set `render_choice` as `depth`. The output video/images will be gray depth maps.
- **visualize normal map:** This is independent of cameras and you could directly set `render_choice` as `normal`. The output video/images will be colorful normal maps.
- **visualize point clouds:** This is independent of cameras and you could directly set `render_choice` as `pointcloud`. The output video/images will be point clouds with keypoints.
- **Choose your color:** Set palette as 'white', 'black', 'blue', 'green', 'red', 'yellow', and pass a list of string with the length of num_person. Or send a numpy.ndarray of shape (num_person, 3). Should be normalized color: (1.0, 1.0, 1.0) represents white. The color channel is RGB.
- **Differentiable render:** Set `no_grad=False` and `return_tensor=True`.

10.2 Important parameters:

- **background images:** You could pass `image_array`(`numpy.ndarray` of shape (frame, h, w, 3)) or `frame_list`(`list` of paths of images(.png or .jpg)) or `origin_frames`(str of video path or image folder path). The priority order is `image_array` > `frame_list` > `origin_frames`. If the background images are too big, you should set `read_frames_batch` as `True` to relieve the IO burden. This will be done automatically in the code when you number of frame is large than 500.
- **smpl pose & verts:** There area two ways to pass smpl mesh information: 1). You pass `poses`, `betas`(optional) and `transl`(optional) and `gender`(optional). 2). You pass `verts` directly and the above three will be ignored. The `body_model` or `model_path` is still required if you pass`verts` since we need to get the `faces`. The priority order is `verts` > (`poses` & `betas` & `transl` & `gender`). Check the docstring for details. 3) for multi-person, you should have an extra dim for num_person. E.g., shape of smpl `verts` should be (num_frame, num_person, 6890, 3), shape of smpl `poses` should be (num_frame, num_person, 72), shape of smpl `betas` should be (num_frame, num_person, 10), shape of vibe `pred_cam` should be (num_frame, num_person, 3). This doesn't have influence on `K`, `R`, `T` since they are for every frame.
- **body model:** There area two ways to pass body model: 1). You pass `model_path`, `model_type`(optional) and `gender`(optional). 2). You pass `body_model` directly and the above three will be ignored. The priority order is `body_model` > (`model_path` & `model_type` & `gender`). Check the docstring for details.
- **output path:** Output_path could be `None` or `str` of video path or `str` of image folder path. 1). If `None`, no output file will be wrote. 1). If a video path like `xxx.mp4`, a video file will be wrote. Make sure you have enough space for temporal images. The images will be removed automatically. 1). If a image folder path like `xxx/`, a folder will be created and the images will be wrote into it.

CHAPTER

ELEVEN

ADDITIONAL LICENSES

We would like to pay tribute to open-source implementations to which we make reference. Note that they may carry additional license requiresments.

11.1 SMPLify-X

License

Software Copyright License for non-commercial scientific research purposes Please read carefully the following terms and conditions and any accompanying documentation before you download and/or use the SMPL-X/SMPLify-X model, data and software, (the "Model & Software"), including 3D meshes, blend weights, blend shapes, textures, software, scripts, and animations. By downloading and/or using the Model & Software (including downloading, cloning, installing, and any other use of this github repository), you acknowledge that you have read these terms and conditions, understand them, and agree to be bound by them. If you do not agree with these terms and conditions, you must not download and/or use the Model & Software. Any infringement of the terms of this agreement will automatically terminate your rights under this License

Ownership / Licensees The Software and the associated materials has been developed at the

Max Planck Institute for Intelligent Systems (hereinafter "MPI").

Any copyright or patent right is owned by and proprietary material of the

Max-Planck-Gesellschaft zur Förderung der Wissenschaften e.V. (hereinafter "MPG"; MPI and MPG hereinafter collectively "Max-Planck")

hereinafter the "Licensor".

License Grant Licensor grants you (Licensee) personally a single-user, non-exclusive, non-transferable, free of charge right:

To install the Model & Software on computers owned, leased or otherwise controlled by you and/or your organization; To use the Model & Software for the sole purpose of performing non-commercial scientific research, non-commercial education, or non-commercial artistic projects; Any other use, in particular any use for commercial, pornographic, military, or surveillance, purposes is prohibited. This includes, without limitation, incorporation in a commercial product, use in a commercial service, or production of other artifacts for commercial purposes. The Data & Software may not be used to create fake, libelous, misleading, or defamatory content of any kind excluding analyses in peer-reviewed scientific research. The Data & Software may not be reproduced, modified and/or made available in any form to any third party without Max-Planck's prior written permission.

The Data & Software may not be used for pornographic purposes or to generate pornographic material whether commercial or not. This license also prohibits the use of the Software to train methods/algorithms/neural networks/etc. for commercial, pornographic, military, surveillance, or defamatory use of any kind. By downloading the Data & Software, you agree not to reverse engineer it.

No Distribution The Model & Software and the license herein granted shall not be copied, shared, distributed, re-sold, offered for re-sale, transferred or sub-licensed in whole or in part except that you may make one copy for archive purposes only.

Disclaimer of Representations and Warranties You expressly acknowledge and agree that the Model & Software results from basic research, is provided "AS IS", may contain errors, and that any use of the Model & Software is at your sole risk. LICENSOR MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE MODEL & SOFTWARE, NEITHER EXPRESS NOR IMPLIED, AND THE ABSENCE OF ANY LEGAL OR ACTUAL DEFECTS, WHETHER DISCOVERABLE OR NOT. Specifically, and not to limit the foregoing, licensor makes no representations or warranties (i) regarding the merchantability or fitness for a particular purpose of the Model & Software, (ii) that the use of the Model & Software will not infringe any patents, copyrights or other intellectual property rights of a third party, and (iii) that the use of the Model & Software will not cause any damage of any kind to you or a third party.

Limitation of Liability Because this Model & Software License Agreement qualifies as a donation, according to Section 521 of the German Civil Code (Bürgerliches Gesetzbuch – BGB) Licensor as a donor is liable for intent and gross negligence only. If the Licensor fraudulently conceals a legal or material defect, they are obliged to compensate the Licensee for the resulting damage. Licensor shall be liable for loss of data only up to the amount of typical recovery costs which would have arisen had proper and regular data backup measures been taken. For the avoidance of doubt Licensor shall be liable in accordance with the German Product Liability Act in the event of product liability. The foregoing applies also to Licensor's legal representatives or assistants in performance. Any further liability shall be excluded. Patent claims generated through the usage of the Model & Software cannot be directed towards the copyright holders. The Model & Software is provided in the state of development the licensor defines. If modified or extended by Licensee, the Licensor makes no claims about the fitness of the Model & Software and is not responsible for any problems such modifications cause.

No Maintenance Services You understand and agree that Licensor is under no obligation to provide either maintenance services, update services, notices of latent defects, or corrections of defects with regard to the Model & Software. Licensor nevertheless reserves the right to update, modify, or discontinue the Model & Software at any time.

Defects of the Model & Software must be notified in writing to the Licensor with a comprehensible description of the error symptoms. The notification of the defect should enable the reproduction of the error. The Licensee is encouraged to communicate any use, results, modification or publication.

Publications using the Model & Software You acknowledge that the Model & Software is a valuable scientific resource and agree to appropriately reference the following paper in any publication making use of the Model & Software.

Citation:

@inproceedings{SMPL-X:2019, title = {Expressive Body Capture: 3D Hands, Face, and Body from a Single Image}, author = {Pavlakos, Georgios and Choutas, Vasileios and Ghorbani, Nima and Bolkart, Timo and Osman, Ahmed A. A. and Tzionas, Dimitrios and Black, Michael J.}, booktitle = {Proceedings IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)}, year = {2019} } Commercial licensing opportunities For commercial uses of the Software, please send email to ps-license@tue.mpg.de

This Agreement shall be governed by the laws of the Federal Republic of Germany except for the UN Sales Convention.

11.2 VIBE

License

Software Copyright License for non-commercial scientific research purposes Please read carefully the following terms and conditions and any accompanying documentation before you download and/or use the VIBE model, data and software, (the "Model & Software"), including 3D meshes, software, and scripts. By downloading and/or using the Model & Software (including downloading, cloning, installing, and any other use of this github repository), you acknowledge that you have read these terms and conditions, understand them, and agree to be bound by them. If you do not agree

with these terms and conditions, you must not download and/or use the Model & Software. Any infringement of the terms of this agreement will automatically terminate your rights under this License

Ownership / Licensees The Software and the associated materials has been developed at the

Max Planck Institute for Intelligent Systems (hereinafter "MPI").

Any copyright or patent right is owned by and proprietary material of the

Max-Planck-Gesellschaft zur Förderung der Wissenschaften e.V. (hereinafter "MPG"; MPI and MPG hereinafter collectively "Max-Planck")

hereinafter the "Licensor".

This software includes the SMPL Body Model. By downloading this software, you are agreeing to be bound by the terms of the SMPL Model License

```
https://smpl.is.tue.mpg.de/modellicense
```

which is necessary to create SMPL body models.

SMPL bodies that are generated with VIBE can be distributed freely under the SMPL Body License

```
https://smpl.is.tue.mpg.de/bodylicense
```

License Grant Licensor grants you (Licensee) personally a single-user, non-exclusive, non-transferable, free of charge right:

To install the Model & Software on computers owned, leased or otherwise controlled by you and/or your organization; To use the Model & Software for the sole purpose of performing non-commercial scientific research, non-commercial education, or non-commercial artistic projects; Any other use, in particular any use for commercial purposes, is prohibited. This includes, without limitation, incorporation in a commercial product, use in a commercial service, or production of other artifacts for commercial purposes. The Model & Software may not be reproduced, modified and/or made available in any form to any third party without Max-Planck's prior written permission.

The Model & Software may not be used for pornographic purposes or to generate pornographic material whether commercial or not. This license also prohibits the use of the Model & Software to train methods/algorithms/neural networks/etc. for commercial use of any kind. By downloading the Model & Software, you agree not to reverse engineer it.

No Distribution The Model & Software and the license herein granted shall not be copied, shared, distributed, re-sold, offered for re-sale, transferred or sub-licensed in whole or in part except that you may make one copy for archive purposes only.

Disclaimer of Representations and Warranties You expressly acknowledge and agree that the Model & Software results from basic research, is provided "AS IS", may contain errors, and that any use of the Model & Software is at your sole risk. LICENSOR MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE MODEL & SOFTWARE, NEITHER EXPRESS NOR IMPLIED, AND THE ABSENCE OF ANY LEGAL OR ACTUAL DEFECTS, WHETHER DISCOVERABLE OR NOT. Specifically, and not to limit the foregoing, licensor makes no representations or warranties (i) regarding the merchantability or fitness for a particular purpose of the Model & Software, (ii) that the use of the Model & Software will not infringe any patents, copyrights or other intellectual property rights of a third party, and (iii) that the use of the Model & Software will not cause any damage of any kind to you or a third party.

Limitation of Liability Because this Model & Software License Agreement qualifies as a donation, according to Section 521 of the German Civil Code (Bürgerliches Gesetzbuch – BGB) Licensor as a donor is liable for intent and gross negligence only. If the Licensor fraudulently conceals a legal or material defect, they are obliged to compensate the Licensee for the resulting damage.

Licensor shall be liable for loss of data only up to the amount of typical recovery costs which would have arisen had proper and regular data backup measures been taken. For the avoidance of doubt Licensor shall be liable in accordance

with the German Product Liability Act in the event of product liability. The foregoing applies also to Licensor's legal representatives or assistants in performance. Any further liability shall be excluded. Patent claims generated through the usage of the Model & Software cannot be directed towards the copyright holders. The Model & Software is provided in the state of development the licensor defines. If modified or extended by Licensee, the Licensor makes no claims about the fitness of the Model & Software and is not responsible for any problems such modifications cause.

No Maintenance Services You understand and agree that Licensor is under no obligation to provide either maintenance services, update services, notices of latent defects, or corrections of defects with regard to the Model & Software. Licensor nevertheless reserves the right to update, modify, or discontinue the Model & Software at any time.

Defects of the Model & Software must be notified in writing to the Licensor with a comprehensible description of the error symptoms. The notification of the defect should enable the reproduction of the error. The Licensee is encouraged to communicate any use, results, modification or publication.

Publications using the Model & Software You acknowledge that the Model & Software is a valuable scientific resource and agree to appropriately reference the following paper in any publication making use of the Model & Software.

Citation:

@inproceedings{VIBE:CVPR:2020, title = {{VIBE}: Video Inference for Human Body Pose and Shape Estimation}, author = {Kocabas, Muhammed and Athanasiou, Nikos and Black, Michael J.}, booktitle = {Computer Vision and Pattern Recognition (CVPR)}, month = jun, year = {2020}, month_numeric = {6} }

Commercial licensing opportunities For commercial uses of the Software, please send email to ps-license@tue.mpg.de

This Agreement shall be governed by the laws of the Federal Republic of Germany except for the UN Sales Convention.

11.3 SPIN

Copyright (c) 2019, University of Pennsylvania, Max Planck Institute for Intelligent Systems All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

CHAPTER

TWELVE

MMHUMAN3D.APIS

mmhuman3d.apis.**collect_results_cpu**(*result_part*, *size*, *tmpdir=None*)
Collect results in cpu.

mmhuman3d.apis.**collect_results_gpu**(*result_part*, *size*)
Collect results in gpu.

mmhuman3d.apis.**feature_extract**(*model*, *img_or_path*, *det_results*, *bbox_thr=None*, *format='xywh'*)
Extract image features with a list of person bounding boxes.

Parameters

- **model** (*nn.Module*) – The loaded feature extraction model.
- **img_or_path** (*Union[str, np.ndarray]*) – Image filename or loaded image.
- **det_results** (*List(dict)*) – the item in the dict may contain 'bbox' and/or 'track_id'. 'bbox' (4,) or (5,): The person bounding box, which contains 4 box coordinates (and score). 'track_id' (int): The unique id for each human instance.
- **bbox_thr** (*float, optional*) – Threshold for bounding boxes. If bbox_thr is None, ignore it. Defaults to None.
- **format** (*str, optional*) – bbox format. Default: 'xywh'. 'xyxy' means (left, top, right, bottom), 'xywh' means (left, top, width, height).

Returns

The bbox & pose info, containing the bbox: (left, top, right, bottom, [score]) and the features.

Return type list[dict]

mmhuman3d.apis.**inference_image_based_model**(*model*, *img_or_path*, *det_results*, *bbox_thr=None*, *format='xywh'*)
Inference a single image with a list of person bounding boxes.

Parameters

- **model** (*nn.Module*) – The loaded pose model.
- **img_or_path** (*Union[str, np.ndarray]*) – Image filename or loaded image.
- **det_results** (*List(dict)*) – the item in the dict may contain 'bbox' and/or 'track_id'. 'bbox' (4,) or (5,): The person bounding box, which contains 4 box coordinates (and score). 'track_id' (int): The unique id for each human instance.
- **bbox_thr** (*float, optional*) – Threshold for bounding boxes. Only bboxes with higher scores will be fed into the pose detector. If bbox_thr is None, ignore it. Defaults to None.
- **format** (*str, optional*) – bbox format ('xyxy' | 'xywh'). Default: 'xywh'. 'xyxy' means (left, top, right, bottom), 'xywh' means (left, top, width, height).

Returns

Each item in the list is a dictionary, containing the bbox: (left, top, right, bottom, [score]), SMPL parameters, vertices, kp3d, and camera.

Return type list[dict]

mmhuman3d.apis.**inference_video_based_model**(*model*, *extracted_results*, *with_track_id=True*, *causal=True*)

Inference SMPL parameters from extracted featutres using a video-based model.

Parameters

- **model** (*nn.Module*) – The loaded mesh estimation model.
- **extracted_results** (*List[List[Dict]]*) – Multi-frame feature extraction results stored in a nested list. Each element of the outer list is the feature extraction results of a single frame, and each element of the inner list is the feature information of one person, which contains:

  features (ndarray): extracted features track_id (int): unique id of each person, required when

  `with_track_id==True`

  bbox ((4,) or (5,)): left, right, top, bottom, [score]
- **with_track_id** – If True, the element in extracted_results is expected to contain "track_id", which will be used to gather the feature sequence of a person from multiple frames. Otherwise, the extracted results in each frame are expected to have a consistent number and order of identities. Default is True.
- **causal** (*bool*) – If True, the target frame is the first frame in a sequence. Otherwise, the target frame is in the middle of a sequence.

Returns

Each item in the list is a dictionary, which contains: SMPL parameters, vertices, kp3d, and camera.

Return type list[dict]

mmhuman3d.apis.**init_model**(*config*, *checkpoint=None*, *device='cuda:0'*)

Initialize a model from config file.

Parameters

- **config** (str or mmcv.Config) – Config file path or the config object.
- **checkpoint** (*str, optional*) – Checkpoint path. If left as None, the model will not load any weights.

Returns The constructed model. (nn.Module, None): The constructed extractor model

Return type nn.Module

mmhuman3d.apis.**multi_gpu_test**(*model*, *data_loader*, *tmpdir=None*, *gpu_collect=False*)

Test model with multiple gpus.

This method tests model with multiple gpus and collects the results under two different modes: gpu and cpu modes. By setting 'gpu_collect=True' it encodes results to gpu tensors and use gpu communication for results collection. On cpu mode it saves the results on different gpus to 'tmpdir' and collects them by the rank 0 worker.

Parameters

- **model** (*nn.Module*) – Model to be tested.
- **data_loader** (*nn.Dataloader*) – Pytorch data loader.

- **tmpdir** (*str*) – Path of directory to save the temporary results from different gpus under cpu mode.
- **gpu_collect** (*bool*) – Option to use either gpu or cpu to collect results.

Returns The prediction results.

Return type list

mmhuman3d.apis.**set_random_seed**(*seed*, *deterministic=False*)
Set random seed.

Parameters

- **seed** (*int*) – Seed to be used.
- **deterministic** (*bool*) – Whether to set the deterministic option for CUDNN backend, i.e., set *torch.backends.cudnn.deterministic* to True and *torch.backends.cudnn.benchmark* to False. Default: False.

mmhuman3d.apis.**single_gpu_test**(*model*, *data_loader*, *show=False*, *out_dir=None*, ***show_kwargs*)
Test with single gpu.

mmhuman3d.apis.**train_model**(*model*, *dataset*, *cfg*, *distributed=False*, *validate=False*, *timestamp=None*, *device='cuda'*, *meta=None*)
Main api for training model.

CHAPTER

THIRTEEN

MMHUMAN3D.CORE

13.1 cameras

class mmhuman3d.core.cameras.**FoVOrthographicCameras**(**args: Any*, ***kwargs: Any*)
Inherited from Pytorch3D *FoVOrthographicCameras*.

classmethod get_default_projection_matrix(***args*) → torch.Tensor
Class method. Calculate the projective transformation matrix by default parameters.

```
scale_x = 2 / (max_x - min_x)
scale_y = 2 / (max_y - min_y)
scale_z = 2 / (far-near)
mid_x = (max_x + min_x) / (max_x - min_x)
mix_y = (max_y + min_y) / (max_y - min_y)
mid_z = (far + near) / (far - near)

K = [[scale_x,        0,         0,  -mid_x],
     [0,        scale_y,         0,  -mix_y],
     [0,              0,  -scale_z,  -mid_z],
     [0,              0,         0,       1],]
```

Parameters ****kwargs** – parameters for the projection can be passed in as keyword arguments to override the default values.

Returns a *torch.Tensor* which represents a batch of projection matrices K of shape (N, 4, 4)

to_ndc(***kwargs*)
Not implemented.

to_ndc_(***kwargs*)
Not implemented.

to_screen(***kwargs*)
Not implemented.

to_screen_(***kwargs*)
Not implemented.

class mmhuman3d.core.cameras.**FoVPerspectiveCameras**(**args: Any*, ***kwargs: Any*)
Inherited from Pytorch3D *FoVPerspectiveCameras*.

classmethod get_default_projection_matrix(***args*) → torch.Tensor
Class method. Calculate the projective transformation matrix by default parameters.

Parameters ****kwargs** – parameters for the projection can be passed in as keyword arguments to override the default values set in *__init__*.

Returns a *torch.Tensor* which represents a batch of projection matrices K of shape (N, 4, 4)

to_ndc(***kwargs*)
Not implemented.

to_ndc_(***kwargs*)
Not implemented.

to_screen(***kwargs*)
Not implemented.

to_screen_(***kwargs*)
Not implemented.

class mmhuman3d.core.cameras.**NewAttributeCameras**(**args: Any*, ***kwargs: Any*)
Inherited from Pytorch3D CamerasBase and provide some new functions.

compute_depth_of_points(*points: torch.Tensor*) → torch.Tensor
Compute depth of points to the camera plane.

Parameters **points** (*[torch.Tensor]*) – shape should be (batch_size, . . . , 3).

Returns shape will be (batch_size, 1)

Return type torch.Tensor

compute_normal_of_meshes(*meshes: pytorch3d.structures.Meshes*) → torch.Tensor
Compute normal of meshes in the camera view.

Parameters **points** (*[torch.Tensor]*) – shape should be (batch_size, 3).

Returns shape will be (batch_size, 1)

Return type torch.Tensor

extend(*N*)
Create new camera class which contains each input camera N times.

Parameters **N** – number of new copies of each camera.

Returns NewAttributeCameras object.

extend_(*N*)
extend camera inplace.

get_camera_plane_normals(***kwargs*) → torch.Tensor
Get the identity normal vector which stretchs out of the camera plane.

Could pass *R* to override the camera extrinsic rotation matrix. :returns: shape will be (N, 3) :rtype: torch.Tensor

classmethod get_default_projection_matrix()
Class method. Calculate the projective transformation matrix by default parameters.

Parameters ****kwargs** – parameters for the projection can be passed in as keyword arguments to override the default values set in *__init__*.

Returns a *torch.Tensor* which represents a batch of projection matrices K of shape (N, 4, 4)

get_image_size()
Returns the image size, if provided, expected in the form of (height, width) The image size is used for conversion of projected points to screen coordinates.

to_ndc(***kwargs*)
Convert to ndc.

to_ndc_(***kwargs*)
Convert to ndc inplace.

to_screen(***kwargs*)
Convert to screen.

to_screen_(***kwargs*)
Convert to screen inplace.

class mmhuman3d.core.cameras.**OrthographicCameras**(**args: Any*, ***kwargs: Any*)
Inherited from Pytorch3D *OrthographicCameras*.

classmethod get_default_projection_matrix(***args*) → torch.Tensor
Class method. Calculate the projective transformation matrix by default parameters.

```
fx = focal_length[:,0]
fy = focal_length[:,1]
px = principal_point[:,0]
py = principal_point[:,1]

K = [[fx,   0,    0,  px],
     [0,   fy,    0,  py],
     [0,    0,    1,   0],
     [0,    0,    0,   1],]
```

Parameters ****kwargs** – parameters for the projection can be passed in as keyword arguments to override the default values.

Returns a *torch.Tensor* which represents a batch of projection matrices K of shape (N, 4, 4)

class mmhuman3d.core.cameras.**PerspectiveCameras**(**args: Any*, ***kwargs: Any*)
Inherited from Pytorch3D *PerspectiveCameras*.

classmethod get_default_projection_matrix(***args*) → torch.Tensor
Class method. Calculate the projective transformation matrix by default parameters.

Parameters ****kwargs** – parameters for the projection can be passed in as keyword arguments to override the default values set in *__init__*.

Returns a *torch.Tensor* which represents a batch of projection matrices K of shape (N, 4, 4)

class mmhuman3d.core.cameras.**WeakPerspectiveCameras**(**args: Any*, ***kwargs: Any*)
Inherited from [Pytorch3D cameras](https://github.com/facebookresearch/ pytorch3d/blob/main/pytorch3d/renderer/cameras.py) and mimiced the code style. And re-inmplemented functions: compute_projection_matrix, get_projection_transform, unproject_points, is_perspective, in_ndc for render.

K modified from [VIBE](https://github.com/mkocabas/VIBE/blob/master/ lib/utils/renderer.py) and changed to opencv convention. Original license please see docs/additional_license/md.

This intrinsic matrix is orthographics indeed, but could serve as weakperspective for single smpl mesh.

compute_projection_matrix(*scale_x*, *scale_y*, *transl_x*, *transl_y*, *aspect_ratio*) → torch.Tensor
Compute the calibration matrix K of shape (N, 4, 4)

Parameters

- **scale_x** (*Union[torch.Tensor, float], optional*) – Scale in x direction.

- **scale_y** (*Union[torch.Tensor, float], optional*) – Scale in y direction.
- **transl_x** (*Union[torch.Tensor, float], optional*) – Translation in x direction.
- **transl_y** (*Union[torch.Tensor, float], optional*) – Translation in y direction.
- **aspect_ratio** (*Union[torch.Tensor, float], optional*) – aspect ratio of the image pixels. 1.0 indicates square pixels.

Returns torch.FloatTensor of the calibration matrix with shape (N, 4, 4)

static convert_K_to_orig_cam(*K: torch.Tensor*, *aspect_ratio: Union[torch.Tensor, float] = 1.0*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor]

Compute intrinsic camera matrix from pred camera parameter of smpl.

Parameters

- **K** (*torch.Tensor*) – opencv orthographics intrinsic matrix: (N, 4, 4)
- **code-block:** (.) – python: K = [[sx*r, 0, 0, tx*sx*r],

  [0, sy, 0, ty*sy], [0, 0, 1, 0], [0, 0, 0, 1],]
- **aspect_ratio** (*Union[torch.Tensor, float], optional*) – aspect ratio of the image pixels. 1.0 indicates square pixels. Defaults to 1.0.

Returns shape should be (N, 4).

Return type orig_cam (torch.Tensor)

static convert_orig_cam_to_matrix(*orig_cam: torch.Tensor*, ***kwargs*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor]

Compute intrinsic camera matrix from orig_cam parameter of smpl.

```
r > 1::

    K = [[sx*r,   0,    0,   tx*sx*r],
         [0,     sy,    0,     ty*sy],
         [0,      0,    1,         0],
         [0,      0,    0,         1]]

or r < 1::

    K = [[sx,    0,     0,   tx*sx],
         [0,   sy/r,    0,  ty*sy/r],
         [0,     0,     1,      0],
         [0,     0,     0,      1],]

rotation matrix: (N, 3, 3)::

    [[1, 0, 0],
     [0, 1, 0],
     [0, 0, 1]]

translation matrix: (N, 3)::

    [0, 0, -znear]
```

Parameters

- **orig_cam** (*torch.Tensor*) – shape should be (N, 4).

- **znear** (*Union[torch.Tensor, float], optional*) – near clipping plane of the view frustrum. Defaults to 0.0.
- **aspect_ratio** (*Union[torch.Tensor, float], optional*) – aspect ratio of the image pixels. 1.0 indicates square pixels. Defaults to 1.0.

Returns opencv intrinsic matrix: (N, 4, 4)

Return type Tuple[torch.Tensor, torch.Tensor, torch.Tensor]

classmethod get_default_projection_matrix(***args*)
Class method. Calculate the projective transformation matrix by default parameters.

Parameters ****kwargs** – parameters for the projection can be passed in as keyword arguments to override the default values set in *__init__*.

Returns a *torch.Tensor* which represents a batch of projection matrices K of shape (N, 4, 4)

get_projection_transform(***kwargs*) → pytorch3d.transforms.Transform3d
Calculate the orthographic projection matrix. Use column major order.

Parameters ****kwargs** – parameters for the projection can be passed in to override the default values set in __init__.

Returns

a Transform3d object which represents a batch of projection matrices of shape (N, 4, 4)

in_ndc()
Boolean of whether in NDC.

is_perspective()
Boolean of whether is perspective.

to_ndc(***kwargs*)
Not implemented.

to_ndc_(***kwargs*)
Not implemented.

to_screen(***kwargs*)
Not implemented.

to_screen_(***kwargs*)
Not implemented.

unproject_points(*xy_depth: torch.Tensor*, *world_coordinates: bool = True*, ***kwargs*) → torch.Tensor
Sends points from camera coordinates (NDC or screen) back to camera view or world coordinates depending on the *world_coordinates* boolean argument of the function.

mmhuman3d.core.cameras.**build_cameras**(*cfg*)
Build cameras.

mmhuman3d.core.cameras.**compute_orbit_cameras**(*elev: float = 0*, *azim: float = 0*, *dist: float = 2.7*, *at: Union[torch.Tensor, List, Tuple] = (0, 0, 0)*, *batch_size: int = 1*, *orbit_speed: Union[float, Tuple[float, float]] = 0*, *dist_speed: Optional[float] = 0*, *convention: str = 'pytorch3d'*)
Generate a sequence of moving cameras following an orbit.

Parameters

- **elev** (*float, optional*) – This is the angle between the vector from the object to the camera, and the horizontal plane y = 0 (xz-plane). Defaults to 0.

- **azim** (*float, optional*) – angle in degrees or radians. The vector from the object to the camera is projected onto a horizontal plane y = 0. azim is the angle between the projected vector and a reference vector at (0, 0, 1) on the reference plane (the horizontal plane). Defaults to 0.
- **dist** (*float, optional*) – distance of the camera from the object. Defaults to 2.7.
- **at** (*Union[torch.Tensor, List, Tuple], optional*) – the position of the object(s) in world coordinates. Defaults to (0, 0, 0).
- **batch_size** (*int, optional*) – batch size. Defaults to 1.
- **orbit_speed** (*Union[float, Tuple[float, float]], optional*) – degree speed of camera moving along the orbit. Could be one or two number. One number for only elev speed, two number for both. Defaults to 0.
- **dist_speed** (*Optional[float], optional*) – speed of camera moving along the center line. Defaults to 0.
- **convention** (*str, optional*) – Camera convention. Defaults to 'pytorch3d'.

Returns computed K, R, T.

Return type Union[torch.Tensor, torch.Tensor, torch.Tensor]

13.2 conventions

class mmhuman3d.core.conventions.**body_segmentation**(*model_type='smpl'*)

SMPL(X) body mesh vertex segmentation.

mmhuman3d.core.conventions.**compress_converted_kps**(*zero_pad_array: Union[numpy.ndarray, torch.Tensor]*, *mask_array: Union[numpy.ndarray, torch.Tensor]*) → Union[numpy.ndarray, torch.Tensor]

Compress keypoints that are zero-padded after applying convert_kps.

Parameters

- **keypoints** (*np.ndarray*) – input keypoints array, could be (f * n * J * 3/2) or (f * J * 3/2). You can set keypoints as np.zeros((1, J, 2)) if you only need mask.
- **[Union[np.ndarray** (*mask*) – The original mask to mark the existence of the keypoints.
- **torch.Tensor]]** – The original mask to mark the existence of the keypoints.

Returns out_keypoints

Return type Union[np.ndarray, torch.Tensor]

mmhuman3d.core.conventions.**convert_K_3x3_to_4x4**(*K: Union[torch.Tensor, numpy.ndarray]*, *is_perspective: bool = True*) → Union[torch.Tensor, numpy.ndarray]

Convert opencv 3x3 intrinsic matrix to 4x4.

Parameters

- **K** (*Union[torch.Tensor, np.ndarray]*) – Input 3x3 intrinsic matrix, left mm defined. [[fx, 0, px],

  [0, fy, py], [0, 0, 1]]

- **is_perspective** (*bool, optional*) – whether is perspective projection. Defaults to True.

Raises

- **TypeError** – K is not *Tensor* or *array*.
- **ValueError** – Shape is not (batch, 3, 3) or (3, 3)

Returns

Output intrinsic matrix. for perspective:

[[fx, 0, px, 0], [0, fy, py, 0], [0, 0, 0, 1], [0, 0, 1, 0]]

for orthographics: [[fx, 0, 0, px], [0, fy, 0, py], [0, 0, 1, 0], [0, 0, 0, 1]]

Return type Union[torch.Tensor, np.ndarray]

mmhuman3d.core.conventions.**convert_K_4x4_to_3x3**(*K: Union[torch.Tensor, numpy.ndarray]*, *is_perspective: bool = True*) → Union[torch.Tensor, numpy.ndarray]

Convert opencv 4x4 intrinsic matrix to 3x3.

Parameters

- **K** (*Union[torch.Tensor, np.ndarray]*) – Input 4x4 intrinsic matrix, left mm defined. for perspective:

  [[fx, 0, px, 0], [0, fy, py, 0], [0, 0, 0, 1], [0, 0, 1, 0]]

  for orthographics: [[fx, 0, 0, px], [0, fy, 0, py], [0, 0, 1, 0], [0, 0, 0, 1]]

- **is_perspective** (*bool, optional*) – whether is perspective projection. Defaults to True.

Raises

- **TypeError** – type K should be *Tensor* or *array*.
- **ValueError** – Shape is not (batch, 3, 3) or (3, 3).

Returns

Output 3x3 intrinsic matrix, left mm defined. [[fx, 0, px],

[0, fy, py], [0, 0, 1]]

Return type Union[torch.Tensor, np.ndarray]

mmhuman3d.core.conventions.**convert_cameras**(*K: Optional[Union[numpy.ndarray, torch.Tensor]] = None*, *R: Optional[Union[numpy.ndarray, torch.Tensor]] = None*, *T: Optional[Union[numpy.ndarray, torch.Tensor]] = None*, *is_perspective: bool = True*, *convention_src: str = 'opencv'*, *convention_dst: str = 'pytorch3d'*, *in_ndc_src: bool = True*, *in_ndc_dst: bool = True*, *resolution_src: Optional[Union[int, Tuple[int, int], torch.Tensor, numpy.ndarray]] = None*, *resolution_dst: Optional[Union[int, Tuple[int, int], torch.Tensor, numpy.ndarray]] = None*, *camera_conventions: dict = {'blender': {'axis': 'xy-z', 'left_mm_extrinsic': True, 'left_mm_intrinsic': True, 'view_to_world': False}, 'maya': {'axis': 'xy-z', 'left_mm_extrinsic': True, 'left_mm_intrinsic': True, 'view_to_world': False}, 'open3d': {'axis': 'x-yz', 'left_mm_extrinsic': False, 'left_mm_intrinsic': False, 'view_to_world': False}, 'opencv': {'axis': 'x-yz', 'left_mm_extrinsic': True, 'left_mm_intrinsic': True, 'view_to_world': True}, 'opengl': {'axis': 'xy-z', 'left_mm_extrinsic': True, 'left_mm_intrinsic': True, 'view_to_world': False}, 'pyrender': {'axis': 'xy-z', 'left_mm_extrinsic': True, 'left_mm_intrinsic': True, 'view_to_world': False}, 'pytorch3d': {'axis': '-xyz', 'left_mm_extrinsic': False, 'left_mm_intrinsic': True, 'view_to_world': False}, 'unity': {'axis': 'xyz', 'left_mm_extrinsic': True, 'left_mm_intrinsic': True, 'view_to_world': False}}*) → Tuple[Union[torch.Tensor, numpy.ndarray], Union[torch.Tensor, numpy.ndarray], Union[torch.Tensor, numpy.ndarray]]

Convert the intrinsic matrix K and extrinsic matrix [R|T] from source convention to destination convention.

Parameters

- **K** (*Union[torch.Tensor, np.ndarray]*) – Intrinsic matrix, shape should be (batch_size, 4, 4) or (batch_size, 3, 3). Will be ignored if None.
- **R** (*Optional[Union[torch.Tensor, np.ndarray]], optional*) – Extrinsic rotation matrix. Shape should be (batch_size, 3, 3). Will be identity if None. Defaults to None.
- **T** (*Optional[Union[torch.Tensor, np.ndarray]], optional*) – Extrinsic translation matrix. Shape should be (batch_size, 3). Will be zeros if None. Defaults to None.
- **is_perspective** (*bool, optional*) – whether is perspective projection. Defaults to True.
- **——** –
- **Camera dependent args** (*#*) –
- **convention_src** (*str, optional*) – convention of source camera,
- **convention_dst** (*str, optional*) – convention of destination camera,
- **define the convention of cameras by the order of right** (*We*) –
- **and** (*front*) –
- **up.** –

- **E.g.** – '+x+z+y'. '+' could be ignored. The second one is opencv and its convention should be '+x-z-y'. The third one is pytorch3d and its convention should be '-xzy'.

  opengl(pyrender) opencv pytorch3d y z y | / | | / | |_____x /_____x
  x_____ | / | /

  / | /

  z / y | z /

- **first one is pyrender and its convention should be** (*the*) – '+x+z+y'. '+' could be ignored. The second one is opencv and its convention should be '+x-z-y'. The third one is pytorch3d and its convention should be '-xzy'.

  opengl(pyrender) opencv pytorch3d y z y | / | | / | |_____x /_____x
  x_____ | / | /

  / | /

  z / y | z /

- **in_ndc_src** (*bool, optional*) – Whether is the source camera defined in ndc. Defaults to True.
- **in_ndc_dst** (*bool, optional*) – Whether is the destination camera defined in ndc. Defaults to True.
- **camera_convention** (*in*) –

  1). *left_mm_ex* means extrinsic matrix *K* is left matrix multiplcation defined.

  2). *left_mm_in* means intrinsic matrix [*R* ⍀ *T*] is left matrix multiplcation defined.

  3) ***view_to_world* means extrinsic matrix [*R* ⍀ *T*] is defined** as view to world.

- **define these args as** (*we*) –

  1). *left_mm_ex* means extrinsic matrix *K* is left matrix multiplcation defined.

  2). *left_mm_in* means intrinsic matrix [*R* ⍀ *T*] is left matrix multiplcation defined.

  3) ***view_to_world* means extrinsic matrix [*R* ⍀ *T*] is defined** as view to world.

- **(Optional[Union[int** (*resolution_src*) – np.ndarray]], optional): Source camera image size of (height, width). Required if defined in screen. Will be square if int. Shape should be (2,) if *array* or *tensor*. Defaults to None.
- **Tuple[int** – np.ndarray]], optional): Source camera image size of (height, width). Required if defined in screen. Will be square if int. Shape should be (2,) if *array* or *tensor*. Defaults to None.
- **int]** – np.ndarray]], optional): Source camera image size of (height, width). Required if defined in screen. Will be square if int. Shape should be (2,) if *array* or *tensor*. Defaults to None.
- **torch.Tensor** – np.ndarray]], optional): Source camera image size of (height, width). Required if defined in screen. Will be square if int. Shape should be (2,) if *array* or *tensor*. Defaults to None.

:param [np.ndarray]], optional):] Source camera image size of (height, width). Required if defined in screen. Will be square if int. Shape should be (2,) if *array* or *tensor*. Defaults to None.

Parameters

- **(Optional[Union[int** (*resolution_dst*) – np.ndarray]], optional): Destination camera image size of (height, width). Required if defined in screen. Will be square if int. Shape should be (2,) if *array* or *tensor*. Defaults to None.
- **Tuple[int** – np.ndarray]], optional): Destination camera image size of (height, width). Required if defined in screen. Will be square if int. Shape should be (2,) if *array* or *tensor*. Defaults to None.
- **int]** – np.ndarray]], optional): Destination camera image size of (height, width). Required if defined in screen. Will be square if int. Shape should be (2,) if *array* or *tensor*. Defaults to None.
- **torch.Tensor** – np.ndarray]], optional): Destination camera image size of (height, width). Required if defined in screen. Will be square if int. Shape should be (2,) if *array* or *tensor*. Defaults to None.

:param [np.ndarray]], optional):] Destination camera image size of (height, width). Required if defined in screen. Will be square if int. Shape should be (2,) if *array* or *tensor*. Defaults to None.

Parameters **camera_conventions** – (dict, optional): *dict* containing pre-defined camera convention information. Defaults to CAMERA_CONVENTIONS.

Raises **TypeError** – K, R, T should all be *torch.Tensor* or *np.ndarray*.

Returns

Tuple[Union[torch.Tensor, None], Union[torch.Tensor, None], Union[torch.Tensor, None]]: Converted K, R, T matrix of *tensor*.

mmhuman3d.core.conventions.**convert_kps**(*keypoints: Union[numpy.ndarray, torch.Tensor]*, *src: str*, *dst: str*, *approximate: bool = False*, *mask: Optional[Union[numpy.ndarray, torch.Tensor]] = None*, *keypoints_factory: dict = {'agora': ['pelvis', 'left_hip', 'right_hip', 'spine_1', 'left_knee', 'right_knee', 'spine_2', 'left_ankle', 'right_ankle', 'spine_3', 'left_foot', 'right_foot', 'neck', 'left_collar', 'right_collar', 'head', 'left_shoulder', 'right_shoulder', 'left_elbow', 'right_elbow', 'left_wrist', 'right_wrist', 'jaw', 'left_eyeball', 'right_eyeball', 'left_index_1', 'left_index_2', 'left_index_3', 'left_middle_1', 'left_middle_2', 'left_middle_3', 'left_pinky_1', 'left_pinky_2', 'left_pinky_3', 'left_ring_1', 'left_ring_2', 'left_ring_3', 'left_thumb_1', 'left_thumb_2', 'left_thumb_3', 'right_index_1', 'right_index_2', 'right_index_3', 'right_middle_1', 'right_middle_2', 'right_middle_3', 'right_pinky_1', 'right_pinky_2', 'right_pinky_3', 'right_ring_1', 'right_ring_2', 'right_ring_3', 'right_thumb_1', 'right_thumb_2', 'right_thumb_3', 'nose', 'right_eye', 'left_eye', 'right_ear', 'left_ear', 'left_bigtoe', 'left_smalltoe', 'left_heel', 'right_bigtoe', 'right_smalltoe', 'right_heel', 'left_thumb', 'left_index', 'left_middle', 'left_ring', 'left_pinky', 'right_thumb', 'right_index', 'right_middle', 'right_ring', 'right_pinky', 'right_eyebrow_1', 'right_eyebrow_2', 'right_eyebrow_3', 'right_eyebrow_4', 'right_eyebrow_5', 'left_eyebrow_5', 'left_eyebrow_4', 'left_eyebrow_3', 'left_eyebrow_2', 'left_eyebrow_1', 'nosebridge_1', 'nosebridge_2', 'nosebridge_3', 'nosebridge_4', 'nose_1', 'nose_2', 'nose_3', 'nose_4', 'nose_5', 'right_eye_1', 'right_eye_2', 'right_eye_3', 'right_eye_4', 'right_eye_5', 'right_eye_6', 'left_eye_4', 'left_eye_3', 'left_eye_2', 'left_eye_1', 'left_eye_6', 'left_eye_5', 'mouth_1', 'mouth_2', 'mouth_3', 'mouth_4', 'mouth_5', 'mouth_6', 'mouth_7', 'mouth_8', 'mouth_9', 'mouth_10', 'mouth_11', 'mouth_12', 'lip_1', 'lip_2', 'lip_3', 'lip_4', 'lip_5', 'lip_6', 'lip_7', 'lip_8'], 'coco': ['nose', 'left_eye', 'right_eye', 'left_ear', 'right_ear', 'left_shoulder', 'right_shoulder', 'left_elbow', 'right_elbow', 'left_wrist', 'right_wrist', 'left_hip_extra', 'right_hip_extra', 'left_knee', 'right_knee', 'left_ankle', 'right_ankle'], 'coco_wholebody': ['nose', 'left_eye', 'right_eye', 'left_ear', 'right_ear', 'left_shoulder', 'right_shoulder', 'left_elbow', 'right_elbow', 'left_wrist', 'right_wrist', 'left_hip', 'right_hip', 'left_knee', 'right_knee', 'left_ankle', 'right_ankle', 'left_bigtoe', 'left_smalltoe', 'left_heel', 'right_bigtoe', 'right_smalltoe', 'right_heel', 'face_contour_1', 'face_contour_2', 'face_contour_3', 'face_contour_4', 'face_contour_5', 'face_contour_6', 'face_contour_7', 'face_contour_8', 'face_contour_9', 'face_contour_10', 'face_contour_11', 'face_contour_12', 'face_contour_13', 'face_contour_14', 'face_contour_15', 'face_contour_16', 'face_contour_17', 'right_eyebrow_1', 'right_eyebrow_2', 'right_eyebrow_3', 'right_eyebrow_4', 'right_eyebrow_5', 'left_eyebrow_5', 'left_eyebrow_4', 'left_eyebrow_3', 'left_eyebrow_2', 'left_eyebrow_1', 'nosebridge_1', 'nosebridge_2', 'nosebridge_3', 'nosebridge_4', 'nose_1', 'nose_2', 'nose_3', 'nose_4', 'nose_5', 'right_eye_1', 'right_eye_2', 'right_eye_3', 'right_eye_4', 'right_eye_5', 'right_eye_6', 'left_eye_4', 'left_eye_3', 'left_eye_2', 'left_eye_1', 'left_eye_6', 'left_eye_5', 'mouth_1', 'mouth_2', 'mouth_3', 'mouth_4', 'mouth_5', 'mouth_6', 'mouth_7', 'mouth_8', 'mouth_9', 'mouth_10', 'mouth_11', 'mouth_12', 'lip_1', 'lip_2', 'lip_3', 'lip_4', 'lip_5', 'lip_6', 'lip_7', 'lip_8', 'left_hand_root', 'left_thumb_1', 'left_thumb_2', 'left_thumb_3', 'left_thumb', 'left_index_1', 'left_index_2', 'left_index_3', 'left_index', 'left_middle_1', 'left_middle_2', 'left_middle_3', 'left_middle',

conventions by now: agora, coco, smplx, smpl, mpi_inf_3dhp, mpi_inf_3dhp_test, h36m, h36m_mmpose, pw3d, mpii, lsp. :param keypoints [Union[np.ndarray: input keypoints array,

could be (f * n * J * 3/2) or (f * J * 3/2). You can set keypoints as np.zeros((1, J, 2)) if you only need mask.

Parameters

- **torch.Tensor]]** – input keypoints array, could be (f * n * J * 3/2) or (f * J * 3/2). You can set keypoints as np.zeros((1, J, 2)) if you only need mask.
- **src** (*str*) – source data type from keypoints_factory.
- **dst** (*str*) – destination data type from keypoints_factory.
- **approximate** (*bool*) – control whether approximate mapping is allowed.
- **mask** (*Optional[Union[np.ndarray, torch.Tensor]], optional*) – The original mask to mark the existence of the keypoints. None represents all ones mask. Defaults to None.
- **keypoints_factory** (*dict, optional*) – A class to store the attributes. Defaults to keypoints_factory.

Returns

Tuple[Union[np.ndarray, torch.Tensor], Union[np.ndarray, torch.Tensor]] : tuple of (out_keypoints, mask). out_keypoints and mask will be of the same type.

mmhuman3d.core.conventions.**convert_ndc_to_screen**(*K: Union[torch.Tensor, numpy.ndarray]*, *resolution: Union[int, Tuple[int, int], List[int], torch.Tensor, numpy.ndarray]*, *sign: Optional[Iterable[int]] = None*, *is_perspective: bool = True*) → Union[torch.Tensor, numpy.ndarray]

Convert intrinsic matrix from ndc to screen.

Parameters

- **K** (*Union[torch.Tensor, np.ndarray]*) – Input 4x4 intrinsic matrix, left mm defined.
- **resolution** (*Union[int, Tuple[int, int], torch.Tensor, np.ndarray]*) – (height, width) of image.
- **sign** (*Optional[Union[Iterable[int]]], optional*) – xyz axis sign. Defaults to None.
- **is_perspective** (*bool, optional*) – whether is perspective projection. Defaults to True.

Raises

- **TypeError** – K should be Tensor or array.
- **ValueError** – shape of K should be (batch, 4, 4)

Returns output intrinsic matrix.

Return type Union[torch.Tensor, np.ndarray]

mmhuman3d.core.conventions.**convert_perspective_to_weakperspective**(*K: Union[torch.Tensor, numpy.ndarray]*, *zmean: Union[torch.Tensor, numpy.ndarray, float, int]*, *resolution: Optional[Union[int, Tuple[int, int], torch.Tensor, numpy.ndarray]] = None*, *in_ndc: bool = False*, *convention: str = 'opencv'*) → Union[torch.Tensor, numpy.ndarray]

Convert perspective to weakperspective intrinsic matrix.

Parameters

- **K** (*Union[torch.Tensor, np.ndarray]*) – input intrinsic matrix, shape should be (batch, 4, 4) or (batch, 3, 3).
- **zmean** (*Union[torch.Tensor, np.ndarray, int, float]*) – zmean for object. shape should be (batch,) or singleton number.
- **(Union[int** (*resolution*) – optional): (height, width) of image. Defaults to None.
- **Tuple[int** – optional): (height, width) of image. Defaults to None.
- **int]** – optional): (height, width) of image. Defaults to None.
- **torch.Tensor** – optional): (height, width) of image. Defaults to None.
- **np.ndarray]** – optional): (height, width) of image. Defaults to None.

:param : optional): (height, width) of image. Defaults to None. :param in_ndc: whether defined in ndc. Defaults to False. :type in_ndc: bool, optional :param convention: camera convention. Defaults to 'opencv'. :type convention: str, optional

Returns

output weakperspective pred_cam, shape is (batch, 4)

Return type Union[torch.Tensor, np.ndarray]

mmhuman3d.core.conventions.**convert_screen_to_ndc**(*K: Union[torch.Tensor, numpy.ndarray]*, *resolution: Union[int, Tuple[int, int], torch.Tensor, numpy.ndarray]*, *sign: Optional[Iterable[int]] = None*, *is_perspective: bool = True*) → Union[torch.Tensor, numpy.ndarray]

Convert intrinsic matrix from screen to ndc.

Parameters

- **K** (*Union[torch.Tensor, np.ndarray]*) – input intrinsic matrix.
- **resolution** (*Union[int, Tuple[int, int], torch.Tensor, np.ndarray]*) – (height, width) of image.
- **sign** (*Optional[Union[Iterable[int]]], optional*) – xyz axis sign. Defaults to None.
- **is_perspective** (*bool, optional*) – whether is perspective projection. Defaults to True.

Raises

- **TypeError** – K should be Tensor or array.
- **ValueError** – shape of K should be (batch, 4, 4)

Returns output intrinsic matrix.

Return type Union[torch.Tensor, np.ndarray]

mmhuman3d.core.conventions.**convert_weakperspective_to_perspective**(*K: Union[torch.Tensor, numpy.ndarray]*, *zmean: Union[torch.Tensor, numpy.ndarray, int, float]*, *resolution: Optional[Union[int, Tuple[int, int], torch.Tensor, numpy.ndarray]] = None*, *in_ndc: bool = False*, *convention: str = 'opencv'*) → Union[torch.Tensor, numpy.ndarray]

Convert perspective to weakperspective intrinsic matrix.

Parameters

- **K** (*Union[torch.Tensor, np.ndarray]*) – input intrinsic matrix, shape should be (batch, 4, 4) or (batch, 3, 3).
- **zmean** (*Union[torch.Tensor, np.ndarray, int, float]*) – zmean for object. shape should be (batch,) or singleton number.
- **(Union[int** (*resolution*) – optional): (height, width) of image. Defaults to None.
- **Tuple[int** – optional): (height, width) of image. Defaults to None.
- **int]** – optional): (height, width) of image. Defaults to None.
- **torch.Tensor** – optional): (height, width) of image. Defaults to None.
- **np.ndarray]** – optional): (height, width) of image. Defaults to None.

:param : optional): (height, width) of image. Defaults to None. :param in_ndc: whether defined in ndc. Defaults to False. :type in_ndc: bool, optional :param convention: camera convention. Defaults to 'opencv'. :type convention: str, optional

Returns

output weakperspective pred_cam, shape is (batch, 4)

Return type Union[torch.Tensor, np.ndarray]

mmhuman3d.core.conventions.**convert_world_view**(*R: Union[torch.Tensor, numpy.ndarray]*, *T: Union[torch.Tensor, numpy.ndarray]*) → Tuple[Union[numpy.ndarray, torch.Tensor], Union[numpy.ndarray, torch.Tensor]]

Convert between view_to_world and world_to_view defined extrinsic matrix.

Parameters

- **R** (*Union[torch.Tensor, np.ndarray]*) – extrinsic rotation matrix. shape should be (batch, 3, 4)
- **T** (*Union[torch.Tensor, np.ndarray]*) – extrinsic translation matrix.

Raises **TypeError** – R and T should be of the same type.

Returns

Tuple[Union[torch.Tensor, np.ndarray], Union[torch.Tensor, np.ndarray]]: output R, T.

mmhuman3d.core.conventions.**enc_camera_convention**(*convention*, *camera_conventions={'blender': {'axis': 'xy-z', 'left_mm_extrinsic': True, 'left_mm_intrinsic': True, 'view_to_world': False}, 'maya': {'axis': 'xy-z', 'left_mm_extrinsic': True, 'left_mm_intrinsic': True, 'view_to_world': False}, 'open3d': {'axis': 'x-yz', 'left_mm_extrinsic': False, 'left_mm_intrinsic': False, 'view_to_world': False}, 'opencv': {'axis': 'x-yz', 'left_mm_extrinsic': True, 'left_mm_intrinsic': True, 'view_to_world': True}, 'opengl': {'axis': 'xy-z', 'left_mm_extrinsic': True, 'left_mm_intrinsic': True, 'view_to_world': False}, 'pyrender': {'axis': 'xy-z', 'left_mm_extrinsic': True, 'left_mm_intrinsic': True, 'view_to_world': False}, 'pytorch3d': {'axis': '-xyz', 'left_mm_extrinsic': False, 'left_mm_intrinsic': True, 'view_to_world': False}, 'unity': {'axis': 'xyz', 'left_mm_extrinsic': True, 'left_mm_intrinsic': True, 'view_to_world': False}}*)

convert camera convention to axis direction and order.

mmhuman3d.core.conventions.**get_flip_pairs**(*convention: str = 'smplx', keypoints_factory: dict = {'agora': ['pelvis', 'left_hip', 'right_hip', 'spine_1', 'left_knee', 'right_knee', 'spine_2', 'left_ankle', 'right_ankle', 'spine_3', 'left_foot', 'right_foot', 'neck', 'left_collar', 'right_collar', 'head', 'left_shoulder', 'right_shoulder', 'left_elbow', 'right_elbow', 'left_wrist', 'right_wrist', 'jaw', 'left_eyeball', 'right_eyeball', 'left_index_1', 'left_index_2', 'left_index_3', 'left_middle_1', 'left_middle_2', 'left_middle_3', 'left_pinky_1', 'left_pinky_2', 'left_pinky_3', 'left_ring_1', 'left_ring_2', 'left_ring_3', 'left_thumb_1', 'left_thumb_2', 'left_thumb_3', 'right_index_1', 'right_index_2', 'right_index_3', 'right_middle_1', 'right_middle_2', 'right_middle_3', 'right_pinky_1', 'right_pinky_2', 'right_pinky_3', 'right_ring_1', 'right_ring_2', 'right_ring_3', 'right_thumb_1', 'right_thumb_2', 'right_thumb_3', 'nose', 'right_eye', 'left_eye', 'right_ear', 'left_ear', 'left_bigtoe', 'left_smalltoe', 'left_heel', 'right_bigtoe', 'right_smalltoe', 'right_heel', 'left_thumb', 'left_index', 'left_middle', 'left_ring', 'left_pinky', 'right_thumb', 'right_index', 'right_middle', 'right_ring', 'right_pinky', 'right_eyebrow_1', 'right_eyebrow_2', 'right_eyebrow_3', 'right_eyebrow_4', 'right_eyebrow_5', 'left_eyebrow_5', 'left_eyebrow_4', 'left_eyebrow_3', 'left_eyebrow_2', 'left_eyebrow_1', 'nosebridge_1', 'nosebridge_2', 'nosebridge_3', 'nosebridge_4', 'nose_1', 'nose_2', 'nose_3', 'nose_4', 'nose_5', 'right_eye_1', 'right_eye_2', 'right_eye_3', 'right_eye_4', 'right_eye_5', 'right_eye_6', 'left_eye_4', 'left_eye_3', 'left_eye_2', 'left_eye_1', 'left_eye_6', 'left_eye_5', 'mouth_1', 'mouth_2', 'mouth_3', 'mouth_4', 'mouth_5', 'mouth_6', 'mouth_7', 'mouth_8', 'mouth_9', 'mouth_10', 'mouth_11', 'mouth_12', 'lip_1', 'lip_2', 'lip_3', 'lip_4', 'lip_5', 'lip_6', 'lip_7', 'lip_8'], 'coco': ['nose', 'left_eye', 'right_eye', 'left_ear', 'right_ear', 'left_shoulder', 'right_shoulder', 'left_elbow', 'right_elbow', 'left_wrist', 'right_wrist', 'left_hip_extra', 'right_hip_extra', 'left_knee', 'right_knee', 'left_ankle', 'right_ankle'], 'coco_wholebody': ['nose', 'left_eye', 'right_eye', 'left_ear', 'right_ear', 'left_shoulder', 'right_shoulder', 'left_elbow', 'right_elbow', 'left_wrist', 'right_wrist', 'left_hip', 'right_hip', 'left_knee', 'right_knee', 'left_ankle', 'right_ankle', 'left_bigtoe', 'left_smalltoe', 'left_heel', 'right_bigtoe', 'right_smalltoe', 'right_heel', 'face_contour_1', 'face_contour_2', 'face_contour_3', 'face_contour_4', 'face_contour_5', 'face_contour_6', 'face_contour_7', 'face_contour_8', 'face_contour_9', 'face_contour_10', 'face_contour_11', 'face_contour_12', 'face_contour_13', 'face_contour_14', 'face_contour_15', 'face_contour_16', 'face_contour_17', 'right_eyebrow_1', 'right_eyebrow_2', 'right_eyebrow_3', 'right_eyebrow_4', 'right_eyebrow_5', 'left_eyebrow_5', 'left_eyebrow_4', 'left_eyebrow_3', 'left_eyebrow_2', 'left_eyebrow_1', 'nosebridge_1', 'nosebridge_2', 'nosebridge_3', 'nosebridge_4', 'nose_1', 'nose_2', 'nose_3', 'nose_4', 'nose_5', 'right_eye_1', 'right_eye_2', 'right_eye_3', 'right_eye_4', 'right_eye_5', 'right_eye_6', 'left_eye_4', 'left_eye_3', 'left_eye_2', 'left_eye_1', 'left_eye_6', 'left_eye_5', 'mouth_1', 'mouth_2', 'mouth_3', 'mouth_4', 'mouth_5', 'mouth_6', 'mouth_7', 'mouth_8', 'mouth_9', 'mouth_10', 'mouth_11', 'mouth_12', 'lip_1', 'lip_2', 'lip_3', 'lip_4', 'lip_5', 'lip_6', 'lip_7', 'lip_8', 'left_hand_root', 'left_thumb_1', 'left_thumb_2', 'left_thumb_3', 'left_thumb', 'left_index_1', 'left_index_2',*

Parameters

- **convention** (*str*) – data type from keypoints_factory.
- **keypoints_factory** (*dict, optional*) – A class to store the attributes. Defaults to keypoints_factory.

Returns left, right keypoint indices

Return type List[int]

mmhuman3d.core.conventions.**get_keypoint_idx**(*name: str*, *convention: str = 'smplx'*, *approximate: bool = False*, *keypoints_factory: dict = {'agora': ['pelvis', 'left_hip', 'right_hip', 'spine_1', 'left_knee', 'right_knee', 'spine_2', 'left_ankle', 'right_ankle', 'spine_3', 'left_foot', 'right_foot', 'neck', 'left_collar', 'right_collar', 'head', 'left_shoulder', 'right_shoulder', 'left_elbow', 'right_elbow', 'left_wrist', 'right_wrist', 'jaw', 'left_eyeball', 'right_eyeball', 'left_index_1', 'left_index_2', 'left_index_3', 'left_middle_1', 'left_middle_2', 'left_middle_3', 'left_pinky_1', 'left_pinky_2', 'left_pinky_3', 'left_ring_1', 'left_ring_2', 'left_ring_3', 'left_thumb_1', 'left_thumb_2', 'left_thumb_3', 'right_index_1', 'right_index_2', 'right_index_3', 'right_middle_1', 'right_middle_2', 'right_middle_3', 'right_pinky_1', 'right_pinky_2', 'right_pinky_3', 'right_ring_1', 'right_ring_2', 'right_ring_3', 'right_thumb_1', 'right_thumb_2', 'right_thumb_3', 'nose', 'right_eye', 'left_eye', 'right_ear', 'left_ear', 'left_bigtoe', 'left_smalltoe', 'left_heel', 'right_bigtoe', 'right_smalltoe', 'right_heel', 'left_thumb', 'left_index', 'left_middle', 'left_ring', 'left_pinky', 'right_thumb', 'right_index', 'right_middle', 'right_ring', 'right_pinky', 'right_eyebrow_1', 'right_eyebrow_2', 'right_eyebrow_3', 'right_eyebrow_4', 'right_eyebrow_5', 'left_eyebrow_5', 'left_eyebrow_4', 'left_eyebrow_3', 'left_eyebrow_2', 'left_eyebrow_1', 'nosebridge_1', 'nosebridge_2', 'nosebridge_3', 'nosebridge_4', 'nose_1', 'nose_2', 'nose_3', 'nose_4', 'nose_5', 'right_eye_1', 'right_eye_2', 'right_eye_3', 'right_eye_4', 'right_eye_5', 'right_eye_6', 'left_eye_4', 'left_eye_3', 'left_eye_2', 'left_eye_1', 'left_eye_6', 'left_eye_5', 'mouth_1', 'mouth_2', 'mouth_3', 'mouth_4', 'mouth_5', 'mouth_6', 'mouth_7', 'mouth_8', 'mouth_9', 'mouth_10', 'mouth_11', 'mouth_12', 'lip_1', 'lip_2', 'lip_3', 'lip_4', 'lip_5', 'lip_6', 'lip_7', 'lip_8'], 'coco': ['nose', 'left_eye', 'right_eye', 'left_ear', 'right_ear', 'left_shoulder', 'right_shoulder', 'left_elbow', 'right_elbow', 'left_wrist', 'right_wrist', 'left_hip_extra', 'right_hip_extra', 'left_knee', 'right_knee', 'left_ankle', 'right_ankle'], 'coco_wholebody': ['nose', 'left_eye', 'right_eye', 'left_ear', 'right_ear', 'left_shoulder', 'right_shoulder', 'left_elbow', 'right_elbow', 'left_wrist', 'right_wrist', 'left_hip', 'right_hip', 'left_knee', 'right_knee', 'left_ankle', 'right_ankle', 'left_bigtoe', 'left_smalltoe', 'left_heel', 'right_bigtoe', 'right_smalltoe', 'right_heel', 'face_contour_1', 'face_contour_2', 'face_contour_3', 'face_contour_4', 'face_contour_5', 'face_contour_6', 'face_contour_7', 'face_contour_8', 'face_contour_9', 'face_contour_10', 'face_contour_11', 'face_contour_12', 'face_contour_13', 'face_contour_14', 'face_contour_15', 'face_contour_16', 'face_contour_17', 'right_eyebrow_1', 'right_eyebrow_2', 'right_eyebrow_3', 'right_eyebrow_4', 'right_eyebrow_5', 'left_eyebrow_5', 'left_eyebrow_4', 'left_eyebrow_3', 'left_eyebrow_2', 'left_eyebrow_1', 'nosebridge_1', 'nosebridge_2', 'nosebridge_3', 'nosebridge_4', 'nose_1', 'nose_2', 'nose_3', 'nose_4', 'nose_5', 'right_eye_1', 'right_eye_2', 'right_eye_3', 'right_eye_4', 'right_eye_5', 'right_eye_6', 'left_eye_4', 'left_eye_3', 'left_eye_2', 'left_eye_1', 'left_eye_6', 'left_eye_5', 'mouth_1', 'mouth_2', 'mouth_3', 'mouth_4', 'mouth_5', 'mouth_6', 'mouth_7', 'mouth_8', 'mouth_9', 'mouth_10', 'mouth_11', 'mouth_12', 'lip_1', 'lip_2', 'lip_3',*

Parameters

- **name** (*str*) – keypoint name
- **convention** (*str*) – data type from keypoints_factory.
- **approximate** (*bool*) – control whether approximate mapping is allowed.
- **keypoints_factory** (*dict, optional*) – A class to store the attributes. Defaults to keypoints_factory.

Returns keypoint index

Return type List[int]

mmhuman3d.core.conventions.**get_keypoint_idxs_by_part**(*part: str*, *convention: str = 'smplx'*, *keypoints_factory: dict = {'agora': ['pelvis', 'left_hip', 'right_hip', 'spine_1', 'left_knee', 'right_knee', 'spine_2', 'left_ankle', 'right_ankle', 'spine_3', 'left_foot', 'right_foot', 'neck', 'left_collar', 'right_collar', 'head', 'left_shoulder', 'right_shoulder', 'left_elbow', 'right_elbow', 'left_wrist', 'right_wrist', 'jaw', 'left_eyeball', 'right_eyeball', 'left_index_1', 'left_index_2', 'left_index_3', 'left_middle_1', 'left_middle_2', 'left_middle_3', 'left_pinky_1', 'left_pinky_2', 'left_pinky_3', 'left_ring_1', 'left_ring_2', 'left_ring_3', 'left_thumb_1', 'left_thumb_2', 'left_thumb_3', 'right_index_1', 'right_index_2', 'right_index_3', 'right_middle_1', 'right_middle_2', 'right_middle_3', 'right_pinky_1', 'right_pinky_2', 'right_pinky_3', 'right_ring_1', 'right_ring_2', 'right_ring_3', 'right_thumb_1', 'right_thumb_2', 'right_thumb_3', 'nose', 'right_eye', 'left_eye', 'right_ear', 'left_ear', 'left_bigtoe', 'left_smalltoe', 'left_heel', 'right_bigtoe', 'right_smalltoe', 'right_heel', 'left_thumb', 'left_index', 'left_middle', 'left_ring', 'left_pinky', 'right_thumb', 'right_index', 'right_middle', 'right_ring', 'right_pinky', 'right_eyebrow_1', 'right_eyebrow_2', 'right_eyebrow_3', 'right_eyebrow_4', 'right_eyebrow_5', 'left_eyebrow_5', 'left_eyebrow_4', 'left_eyebrow_3', 'left_eyebrow_2', 'left_eyebrow_1', 'nosebridge_1', 'nosebridge_2', 'nosebridge_3', 'nosebridge_4', 'nose_1', 'nose_2', 'nose_3', 'nose_4', 'nose_5', 'right_eye_1', 'right_eye_2', 'right_eye_3', 'right_eye_4', 'right_eye_5', 'right_eye_6', 'left_eye_4', 'left_eye_3', 'left_eye_2', 'left_eye_1', 'left_eye_6', 'left_eye_5', 'mouth_1', 'mouth_2', 'mouth_3', 'mouth_4', 'mouth_5', 'mouth_6', 'mouth_7', 'mouth_8', 'mouth_9', 'mouth_10', 'mouth_11', 'mouth_12', 'lip_1', 'lip_2', 'lip_3', 'lip_4', 'lip_5', 'lip_6', 'lip_7', 'lip_8'], 'coco': ['nose', 'left_eye', 'right_eye', 'left_ear', 'right_ear', 'left_shoulder', 'right_shoulder', 'left_elbow', 'right_elbow', 'left_wrist', 'right_wrist', 'left_hip_extra', 'right_hip_extra', 'left_knee', 'right_knee', 'left_ankle', 'right_ankle'], 'coco_wholebody': ['nose', 'left_eye', 'right_eye', 'left_ear', 'right_ear', 'left_shoulder', 'right_shoulder', 'left_elbow', 'right_elbow', 'left_wrist', 'right_wrist', 'left_hip', 'right_hip', 'left_knee', 'right_knee', 'left_ankle', 'right_ankle', 'left_bigtoe', 'left_smalltoe', 'left_heel', 'right_bigtoe', 'right_smalltoe', 'right_heel', 'face_contour_1', 'face_contour_2', 'face_contour_3', 'face_contour_4', 'face_contour_5', 'face_contour_6', 'face_contour_7', 'face_contour_8', 'face_contour_9',*

Parameters

- **part** (*str*) – part to search from
- **convention** (*str*) – data type from keypoints_factory.
- **keypoints_factory** (*dict, optional*) – A class to store the attributes. Defaults to keypoints_factory.

Returns part keypoint indices

Return type List[int]

mmhuman3d.core.conventions.**get_keypoint_num**(*convention: str = 'smplx'*, *keypoints_factory: dict = {'agora': ['pelvis', 'left_hip', 'right_hip', 'spine_1', 'left_knee', 'right_knee', 'spine_2', 'left_ankle', 'right_ankle', 'spine_3', 'left_foot', 'right_foot', 'neck', 'left_collar', 'right_collar', 'head', 'left_shoulder', 'right_shoulder', 'left_elbow', 'right_elbow', 'left_wrist', 'right_wrist', 'jaw', 'left_eyeball', 'right_eyeball', 'left_index_1', 'left_index_2', 'left_index_3', 'left_middle_1', 'left_middle_2', 'left_middle_3', 'left_pinky_1', 'left_pinky_2', 'left_pinky_3', 'left_ring_1', 'left_ring_2', 'left_ring_3', 'left_thumb_1', 'left_thumb_2', 'left_thumb_3', 'right_index_1', 'right_index_2', 'right_index_3', 'right_middle_1', 'right_middle_2', 'right_middle_3', 'right_pinky_1', 'right_pinky_2', 'right_pinky_3', 'right_ring_1', 'right_ring_2', 'right_ring_3', 'right_thumb_1', 'right_thumb_2', 'right_thumb_3', 'nose', 'right_eye', 'left_eye', 'right_ear', 'left_ear', 'left_bigtoe', 'left_smalltoe', 'left_heel', 'right_bigtoe', 'right_smalltoe', 'right_heel', 'left_thumb', 'left_index', 'left_middle', 'left_ring', 'left_pinky', 'right_thumb', 'right_index', 'right_middle', 'right_ring', 'right_pinky', 'right_eyebrow_1', 'right_eyebrow_2', 'right_eyebrow_3', 'right_eyebrow_4', 'right_eyebrow_5', 'left_eyebrow_5', 'left_eyebrow_4', 'left_eyebrow_3', 'left_eyebrow_2', 'left_eyebrow_1', 'nosebridge_1', 'nosebridge_2', 'nosebridge_3', 'nosebridge_4', 'nose_1', 'nose_2', 'nose_3', 'nose_4', 'nose_5', 'right_eye_1', 'right_eye_2', 'right_eye_3', 'right_eye_4', 'right_eye_5', 'right_eye_6', 'left_eye_4', 'left_eye_3', 'left_eye_2', 'left_eye_1', 'left_eye_6', 'left_eye_5', 'mouth_1', 'mouth_2', 'mouth_3', 'mouth_4', 'mouth_5', 'mouth_6', 'mouth_7', 'mouth_8', 'mouth_9', 'mouth_10', 'mouth_11', 'mouth_12', 'lip_1', 'lip_2', 'lip_3', 'lip_4', 'lip_5', 'lip_6', 'lip_7', 'lip_8'], 'coco': ['nose', 'left_eye', 'right_eye', 'left_ear', 'right_ear', 'left_shoulder', 'right_shoulder', 'left_elbow', 'right_elbow', 'left_wrist', 'right_wrist', 'left_hip_extra', 'right_hip_extra', 'left_knee', 'right_knee', 'left_ankle', 'right_ankle'], 'coco_wholebody': ['nose', 'left_eye', 'right_eye', 'left_ear', 'right_ear', 'left_shoulder', 'right_shoulder', 'left_elbow', 'right_elbow', 'left_wrist', 'right_wrist', 'left_hip', 'right_hip', 'left_knee', 'right_knee', 'left_ankle', 'right_ankle', 'left_bigtoe', 'left_smalltoe', 'left_heel', 'right_bigtoe', 'right_smalltoe', 'right_heel', 'face_contour_1', 'face_contour_2', 'face_contour_3', 'face_contour_4', 'face_contour_5', 'face_contour_6', 'face_contour_7', 'face_contour_8', 'face_contour_9', 'face_contour_10', 'face_contour_11', 'face_contour_12', 'face_contour_13', 'face_contour_14', 'face_contour_15', 'face_contour_16', 'face_contour_17', 'right_eyebrow_1', 'right_eyebrow_2', 'right_eyebrow_3', 'right_eyebrow_4', 'right_eyebrow_5', 'left_eyebrow_5', 'left_eyebrow_4', 'left_eyebrow_3', 'left_eyebrow_2', 'left_eyebrow_1', 'nosebridge_1', 'nosebridge_2', 'nosebridge_3', 'nosebridge_4', 'nose_1', 'nose_2', 'nose_3', 'nose_4', 'nose_5', 'right_eye_1', 'right_eye_2', 'right_eye_3', 'right_eye_4', 'right_eye_5', 'right_eye_6', 'left_eye_4', 'left_eye_3', 'left_eye_2', 'left_eye_1', 'left_eye_6', 'left_eye_5', 'mouth_1', 'mouth_2', 'mouth_3', 'mouth_4', 'mouth_5', 'mouth_6', 'mouth_7', 'mouth_8', 'mouth_9', 'mouth_10', 'mouth_11', 'mouth_12', 'lip_1', 'lip_2', 'lip_3',*

Parameters

- **convention** (*str*) – data type from keypoints_factory.
- **keypoints_factory** (*dict, optional*) – A class to store the attributes. Defaults to keypoints_factory.

Returns part keypoint indices

Return type List[int]

mmhuman3d.core.conventions.**get_mapping**(*src: str*, *dst: str*, *approximate: bool = False*, *keypoints_factory: dict = {'agora': ['pelvis', 'left_hip', 'right_hip', 'spine_1', 'left_knee', 'right_knee', 'spine_2', 'left_ankle', 'right_ankle', 'spine_3', 'left_foot', 'right_foot', 'neck', 'left_collar', 'right_collar', 'head', 'left_shoulder', 'right_shoulder', 'left_elbow', 'right_elbow', 'left_wrist', 'right_wrist', 'jaw', 'left_eyeball', 'right_eyeball', 'left_index_1', 'left_index_2', 'left_index_3', 'left_middle_1', 'left_middle_2', 'left_middle_3', 'left_pinky_1', 'left_pinky_2', 'left_pinky_3', 'left_ring_1', 'left_ring_2', 'left_ring_3', 'left_thumb_1', 'left_thumb_2', 'left_thumb_3', 'right_index_1', 'right_index_2', 'right_index_3', 'right_middle_1', 'right_middle_2', 'right_middle_3', 'right_pinky_1', 'right_pinky_2', 'right_pinky_3', 'right_ring_1', 'right_ring_2', 'right_ring_3', 'right_thumb_1', 'right_thumb_2', 'right_thumb_3', 'nose', 'right_eye', 'left_eye', 'right_ear', 'left_ear', 'left_bigtoe', 'left_smalltoe', 'left_heel', 'right_bigtoe', 'right_smalltoe', 'right_heel', 'left_thumb', 'left_index', 'left_middle', 'left_ring', 'left_pinky', 'right_thumb', 'right_index', 'right_middle', 'right_ring', 'right_pinky', 'right_eyebrow_1', 'right_eyebrow_2', 'right_eyebrow_3', 'right_eyebrow_4', 'right_eyebrow_5', 'left_eyebrow_5', 'left_eyebrow_4', 'left_eyebrow_3', 'left_eyebrow_2', 'left_eyebrow_1', 'nosebridge_1', 'nosebridge_2', 'nosebridge_3', 'nosebridge_4', 'nose_1', 'nose_2', 'nose_3', 'nose_4', 'nose_5', 'right_eye_1', 'right_eye_2', 'right_eye_3', 'right_eye_4', 'right_eye_5', 'right_eye_6', 'left_eye_4', 'left_eye_3', 'left_eye_2', 'left_eye_1', 'left_eye_6', 'left_eye_5', 'mouth_1', 'mouth_2', 'mouth_3', 'mouth_4', 'mouth_5', 'mouth_6', 'mouth_7', 'mouth_8', 'mouth_9', 'mouth_10', 'mouth_11', 'mouth_12', 'lip_1', 'lip_2', 'lip_3', 'lip_4', 'lip_5', 'lip_6', 'lip_7', 'lip_8'], 'coco': ['nose', 'left_eye', 'right_eye', 'left_ear', 'right_ear', 'left_shoulder', 'right_shoulder', 'left_elbow', 'right_elbow', 'left_wrist', 'right_wrist', 'left_hip_extra', 'right_hip_extra', 'left_knee', 'right_knee', 'left_ankle', 'right_ankle'], 'coco_wholebody': ['nose', 'left_eye', 'right_eye', 'left_ear', 'right_ear', 'left_shoulder', 'right_shoulder', 'left_elbow', 'right_elbow', 'left_wrist', 'right_wrist', 'left_hip', 'right_hip', 'left_knee', 'right_knee', 'left_ankle', 'right_ankle', 'left_bigtoe', 'left_smalltoe', 'left_heel', 'right_bigtoe', 'right_smalltoe', 'right_heel', 'face_contour_1', 'face_contour_2', 'face_contour_3', 'face_contour_4', 'face_contour_5', 'face_contour_6', 'face_contour_7', 'face_contour_8', 'face_contour_9', 'face_contour_10', 'face_contour_11', 'face_contour_12', 'face_contour_13', 'face_contour_14', 'face_contour_15', 'face_contour_16', 'face_contour_17', 'right_eyebrow_1', 'right_eyebrow_2', 'right_eyebrow_3', 'right_eyebrow_4', 'right_eyebrow_5', 'left_eyebrow_5', 'left_eyebrow_4', 'left_eyebrow_3', 'left_eyebrow_2', 'left_eyebrow_1', 'nosebridge_1', 'nosebridge_2', 'nosebridge_3', 'nosebridge_4', 'nose_1', 'nose_2', 'nose_3', 'nose_4', 'nose_5', 'right_eye_1', 'right_eye_2', 'right_eye_3', 'right_eye_4', 'right_eye_5', 'right_eye_6', 'left_eye_4', 'left_eye_3', 'left_eye_2', 'left_eye_1', 'left_eye_6', 'left_eye_5', 'mouth_1', 'mouth_2', 'mouth_3', 'mouth_4', 'mouth_5', 'mouth_6', 'mouth_7', 'mouth_8', 'mouth_9', 'mouth_10', 'mouth_11', 'mouth_12', 'lip_1', 'lip_2', 'lip_3', 'lip_4', 'lip_5', 'lip_6', 'lip_7', 'lip_8', 'left_hand_root', 'left_thumb_1', 'left_thumb_2', 'left_thumb_3', 'left_thumb', 'left_index_1', 'left_index_2', 'left_index_3', 'left_index', 'left_middle_1', 'left_middle_2', 'left_middle_3', 'left_middle', 'left_ring_1', 'left_ring_2', 'left_ring_3', 'left_ring', 'left_pinky_1', 'left_pinky_2', 'left_pinky_3', 'left_pinky', 'right_hand_root',*

Parameters

- **src** (*str*) – source data type from keypoints_factory.
- **dst** (*str*) – destination data type from keypoints_factory.
- **approximate** (*bool*) – control whether approximate mapping is allowed.
- **keypoints_factory** (*dict, optional*) – A class to store the attributes. Defaults to keypoints_factory.

Returns

[src_to_intersection_idx, dst_to_intersection_index, intersection_names]

Return type list

13.3 evaluation

mmhuman3d.core.evaluation.**compute_similarity_transform**(*source_points*, *target_points*)

Computes a similarity transform (sR, t) that takes a set of 3D points source_points (N x 3) closest to a set of 3D points target_points, where R is an 3x3 rotation matrix, t 3x1 translation, s scale.

And return the transformed 3D points source_points_hat (N x 3). i.e. solves the orthogonal Procrutes problem. .. rubric:: Notes

Points number: N

Parameters

- **source_points** (*np.ndarray([N, 3])*) – Source point set.
- **target_points** (*np.ndarray([N, 3])*) – Target point set.

Returns Transformed source point set.

Return type source_points_hat (np.ndarray([N, 3]))

mmhuman3d.core.evaluation.**keypoint_mpjpe**(*pred*, *gt*, *mask*, *alignment='none'*)

Calculate the mean per-joint position error (MPJPE) and the error after rigid alignment with the ground truth (P-MPJPE). batch_size: N num_keypoints: K keypoint_dims: C :param pred: Predicted keypoint location. :type pred: np.ndarray[N, K, C] :param gt: Groundtruth keypoint location. :type gt: np.ndarray[N, K, C] :param mask: Visibility of the target. False for invisible

joints, and True for visible. Invisible joints will be ignored for accuracy calculation.

Parameters **alignment** (*str, optional*) – method to align the prediction with the groundtruth. Supported options are: - `'none'`: no alignment will be applied - `'scale'`: align in the least-square sense in scale - `'procrustes'`: align in the least-square sense in scale,

rotation and translation.

Returns

A tuple containing joint position errors - mpjpe (float|np.ndarray[N]): mean per-joint position error. - p-mpjpe (float|np.ndarray[N]): mpjpe after rigid alignment with the

ground truth

Return type tuple

13.4 filter

class mmhuman3d.core.filter.**Gaus1dFilter**(*window_size=11*, *sigma=4*)
Applies median filter and then gaussian filter. code from: https://github.com/akanazawa/human_dynamics/blob/mas ter/src/util/smooth_bbox.py.

Parameters

- **x** (*np.ndarray*) – input pose
- **window_size** (*int, optional*) – for median filters (must be odd).
- **sigma** (*float, optional*) – Sigma for gaussian smoothing.

Returns Smoothed poses

Return type np.ndarray

class mmhuman3d.core.filter.**OneEuroFilter**(*min_cutoff=0.004*, *beta=0.7*)
Oneeuro filter, source code: https://github.com/mkocabas/VIBE/blob/c0c3f77d587351c806e901221a9dc05d1ffade4b/lib/utils/smooth_pose.py.

Parameters

- **min_cutoff** (*float, optional*) –
- **the minimum cutoff frequency decreases slow speed jitter** (*Decreasing*) –
- **beta** (*float, optional*) –
- **the speed coefficient** (*Increasing*) –

Returns smoothed poses

Return type np.ndarray

class mmhuman3d.core.filter.**SGFilter**(*window_size=11*, *polyorder=2*)
savgol_filter lib is from: https://docs.scipy.org/doc/scipy/reference/generated/ scipy.signal.savgol_filter.html.

Parameters

- **window_size** (*float*) – The length of the filter window (i.e., the number of coefficients). window_length must be a positive odd integer.
- **polyorder** (*int*) – The order of the polynomial used to fit the samples. polyorder must be less than window_length.

Returns smoothed poses (np.ndarray, torch.tensor)

mmhuman3d.core.filter.**build_filter**(*cfg*)
Build filters function.

13.5 optimizer

mmhuman3d.core.optimizer.**build_optimizers**(*model*, *cfgs*)

Build multiple optimizers from configs. If *cfgs* contains several dicts for optimizers, then a dict for each constructed optimizers will be returned. If *cfgs* only contains one optimizer config, the constructed optimizer itself will be returned. For example,

1) Multiple optimizer configs:

```
optimizer_cfg = dict(
    model1=dict(type='SGD', lr=lr),
    model2=dict(type='SGD', lr=lr))
```

The return dict is `dict('model1': torch.optim.Optimizer, 'model2': torch.optim.Optimizer)`

2) Single optimizer config:

```
optimizer_cfg = dict(type='SGD', lr=lr)
```

The return is `torch.optim.Optimizer`.

Parameters

- **model** (`nn.Module`) – The model with parameters to be optimized.
- **cfgs** (*dict*) – The config dict of the optimizer.

Returns The initialized optimizers.

Return type dict[`torch.optim.Optimizer`] | `torch.optim.Optimizer`

13.6 parametric_model

13.7 visualization

mmhuman3d.core.visualization.**render_smpl**(*poses: Optional[Union[torch.Tensor, numpy.ndarray, dict]] = None*, *betas: Optional[Union[numpy.ndarray, torch.Tensor]] = None*, *transl: Optional[Union[numpy.ndarray, torch.Tensor]] = None*, *verts: Optional[Union[numpy.ndarray, torch.Tensor]] = None*, *model_type: typing_extensions.Literal[smpl, smplx] = 'smpl'*, *body_model: Optional[torch.nn.modules.module.Module] = None*, *body_model_config: Optional[dict] = None*, *R: Optional[Union[numpy.ndarray, torch.Tensor]] = None*, *T: Optional[Union[numpy.ndarray, torch.Tensor]] = None*, *K: Optional[Union[numpy.ndarray, torch.Tensor]] = None*, *orig_cam: Optional[Union[numpy.ndarray, torch.Tensor]] = None*, *Ks: Optional[Union[numpy.ndarray, torch.Tensor]] = None*, *in_ndc: bool = True*, *convention: str = 'pytorch3d'*, *projection: typing_extensions.Literal[weakperspective, perspective, fovperspective, orthographics, fovorthographics] = 'perspective'*, *orbit_speed: Union[float, Tuple[float, float]] = 0.0*, *render_choice: typing_extensions.Literal[lq, mq, hq, silhouette, depth, normal, pointcloud, part_silhouette] = 'hq'*, *palette: Union[List[str], str, numpy.ndarray] = 'white'*, *resolution: Optional[Union[List[int], Tuple[int, int]]] = None*, *start: int = 0*, *end: Optional[int] = None*, *alpha: float = 1.0*, *no_grad: bool = True*, *batch_size: int = 10*, *device: Union[torch.device, str] = 'cuda'*, *return_tensor: bool = False*, *output_path: Optional[str] = None*, *origin_frames: Optional[str] = None*, *frame_list: Optional[List[str]] = None*, *image_array: Optional[Union[numpy.ndarray, torch.Tensor]] = None*, *img_format: str = 'frame_%06d.jpg'*, *overwrite: bool = False*, *mesh_file_path: Optional[str] = None*, *read_frames_batch: bool = False*, *plot_kps: bool = False*, *kp3d: Optional[Union[numpy.ndarray, torch.Tensor]] = None*, *mask: Optional[Union[numpy.ndarray, List[int]]] = None*, *vis_kp_index: bool = False*) → Union[None, torch.Tensor]

Render SMPL or SMPL-X mesh or silhouette into differentiable tensors, and export video or images.

Parameters

- **smpl parameters** (*#*) –
- **poses** (*Union[torch.Tensor, np.ndarray, dict]*) – 1). *tensor* or *array* and ndim is 2, shape should be (frame, 72).

  2). *tensor* or *array* and ndim is 3, shape should be (frame, num_person, 72/165). num_person equals 1 means single-person. Rendering predicted multi-person should feed together with multi-person weakperspective cameras. meshes would be computed and use an identity intrinsic matrix.

  3). *dict*, standard dict format defined in smplx.body_models. will be treated as single-person.

  Lower priority than *verts*.

Defaults to None.

- **betas** (*Optional[Union[torch.Tensor, np.ndarray]], optional*) – 1). ndim is 2, shape should be (frame, 10).

  2). ndim is 3, shape should be (frame, num_person, 10). num_person equals 1 means single-person. If poses are multi-person, betas should be set to the same person number.

  None will use default betas.

  Defaults to None.

- **transl** (*Optional[Union[torch.Tensor, np.ndarray]], optional*) – translations of smpl(x).

  1). ndim is 2, shape should be (frame, 3).

  2). ndim is 3, shape should be (frame, num_person, 3). num_person equals 1 means single-person. If poses are multi-person, transl should be set to the same person number.

  Defaults to None.

- **verts** (*Optional[Union[torch.Tensor, np.ndarray]], optional*) – 1). ndim is 3, shape should be (frame, num_verts, 3).

  2). ndim is 4, shape should be (frame, num_person, num_verts, 3). num_person equals 1 means single-person.

  Higher priority over *poses* & *betas* & *transl*.

  Defaults to None.

- **model_type** (*Literal[, optional*) – choose in 'smpl' or 'smplx'.

  Defaults to 'smpl'.

  Defaults to None.

- **body_model** (*nn.Module, optional*) – body_model created from smplx.create. Higher priority than *body_model_config*. Should not both be None.

  Defaults to None.

- **body_model_config** (*dict, optional*) – body_model_config for build_model. Lower priority than *body_model*. Should not both be None. Defaults to None.

- **camera parameters** (*#*) –

- **K** (*Optional[Union[torch.Tensor, np.ndarray]], optional*) – shape should be (frame, 4, 4) or (frame, 3, 3), frame could be 1. if (4, 4) or (3, 3), dim 0 will be added automatically. Will be default *FovPerspectiveCameras* intrinsic if None. Lower priority than *orig_cam*.

- **R** (*Optional[Union[torch.Tensor, np.ndarray]], optional*) – shape should be (frame, 3, 3), If f equals 1, camera will have identical rotation. If *K* and *orig_cam* is None, will be generated by *look_at_view*. If have *K* or *orig_cam* and *R* is None, will be generated by *convert_cameras*.

  Defaults to None.

- **T** (*Optional[Union[torch.Tensor, np.ndarray]], optional*) – shape should be (frame, 3). If f equals 1, camera will have identical translation. If *K* and *orig_cam* is None, will be generated by *look_at_view*. If have *K* or *orig_cam* and *T* is None, will be generated by *convert_cameras*.

  Defaults to None.

- **orig_cam** (*Optional[Union[torch.Tensor, np.ndarray]], optional*) – shape should be (frame, 4) or (frame, num_person, 4). If f equals 1, will be repeated to num_frames. num_person should be 1 if single person. Usually for HMR, VIBE predicted cameras. Higher priority than *K* & *R* & *T*.

  Defaults to None.

- **Ks** (*Optional[Union[torch.Tensor, np.ndarray]], optional*) – shape should be (frame, 4, 4). This is for HMR or SPIN multi-person demo.

- **in_ndc** (*bool, optional*) – . Defaults to True.

- **convention** (*str, optional*) – If want to use an existing convention, choose in ['opengl', 'opencv', 'pytorch3d', 'pyrender', 'open3d', 'maya', 'blender', 'unity']. If want to use a new convention, define your convention in (CAMERA_CONVENTION_FACTORY)[mmhuman3d/core/conventions/cameras/__init__.py] by the order of right, front and up.

  Defaults to 'pytorch3d'.

- **projection** (*Literal[, optional*) – projection mode of camers. Choose in ['orthographics, fovperspective', 'perspective', 'weakperspective', 'fovorthographics'] Defaults to 'perspective'.

- **orbit_speed** (*float, optional*) – orbit speed for viewing when no *K* provided. *float* for only azim speed and Tuple for *azim* and *elev*.

- **render choice parameters** (*#*) –

- **render_choice** (*Literal[, optional*) – choose in ['lq', 'mq', 'hq', 'silhouette', 'depth', 'normal', 'pointcloud', 'part_silhouette'] .

  lq, *mq*, *hq* would output (frame, h, w, 4) tensor.

  lq means low quality, *mq* means medium quality, h`q means high quality.

  silhouette would output (frame, h, w) binary tensor.

  part_silhouette would output (frame, h, w, n_class) tensor.

  n_class is the body segmentation classes.

  depth will output a depth map of (frame, h, w, 1) tensor and 'normal' will output a normal map of (frame, h, w, 1).

  pointcloud will output a (frame, h, w, 4) tensor.

  Defaults to 'mq'.

- **palette** (*Union[List[str], str, np.ndarray], optional*) – color theme str or list of color str or *array*.

  1). If use str to represent the color, should choose in ['segmentation', 'random'] or color from Colormap https://en.wikipedia.org/wiki/X11_color_names. If choose 'segmentation', will get a color for each part.

  2). If you have multi-person, better give a list of str or all will be in the same color.

  3). If you want to define your specific color, use an *array* of shape (3,) for single person and (N, 3) for multiple persons.

  If (3,) for multiple persons, all will be in the same color.

  Your *array* should be in range [0, 255] for 8 bit color.

  Defaults to 'white'.

- **resolution** (*Union[Iterable[int], int], optional*) – 1). If iterable, should be (height, width) of output images.

  2). If int, would be taken as (resolution, resolution).

  Defaults to (1024, 1024).

  This will influence the overlay results when render with backgrounds. The output video will be rendered following the size of background images and finally resized to resolution.

- **start** (*int, optional*) – start frame index. Defaults to 0.

- **end** (*int, optional*) –

  end frame index. Exclusive. Could be positive int or negative int or None. None represents include all the frames.

  Defaults to None.

- **alpha** (*float, optional*) – Transparency of the mesh. Range in [0.0, 1.0]

  Defaults to 1.0.

- **no_grad** (*bool, optional*) – Set to True if do not need differentiable render.

  Defaults to False.

- **batch_size** (*int, optional*) – Batch size for render. Related to your gpu memory.

  Defaults to 10.

- **file io parameters** (*#*) –

- **return_tensor** (*bool, optional*) – Whether return the result tensors.

  Defaults to False, will return None.

- **output_path** (*str, optional*) – output video or gif or image folder.

  Defaults to None, pass export procedure.

- **background frames** (*#*) – image_array > frame_list > origin_frames

- **priority** – image_array > frame_list > origin_frames

- **origin_frames** (*Optional[str], optional*) – origin background frame path, could be *.mp4*, `.gif` (will be sliced into a folder) or an image folder.

  Defaults to None.

- **frame_list** (*Optional[List[str]], optional*) – list of origin background frame paths, element in list each should be a image path like **.jpg* or **.png*. Use this when your file names is hard to sort or you only want to render a small number frames.

  Defaults to None.

- **image_array** – (Optional[Union[np.ndarray, torch.Tensor]], optional): origin background frame *tensor* or *array*, use this when you want your frames in memory as array or tensor.

- **overwrite** (*bool, optional*) – whether overwriting the existing files.

  Defaults to False.

- **mesh_file_path** (*bool, optional*) – the directory path to store the *.ply* or '.ply' files. Will be named like 'frame_idx_person_idx.ply'.

  Defaults to None.

- **read_frames_batch** (*bool, optional*) – Whether read frames by batch. Set it as True if your video is large in size.

  Defaults to False.

- **visualize keypoints** (*#*) –
- **plot_kps** (*bool, optional*) – whether plot keypoints on the output video.

  Defaults to False.

- **kp3d** (*Optional[Union[np.ndarray, torch.Tensor]], optional*) – the keypoints of any convention, should pass *mask* if have any none-sense points. Shape should be (frame,)

  Defaults to None.

- **mask** (*Optional[Union[np.ndarray, List[int]]], optional*) – Mask of keypoints existence.

  Defaults to None.

- **vis_kp_index** (*bool, optional*) – Whether plot keypoint index number on human mesh.

  Defaults to False.

Returns return the rendered image tensors or None.

Return type Union[None, torch.Tensor]

mmhuman3d.core.visualization.**visualize_T_pose**(*num_frames*, *orbit_speed=1.0*, *model_type='smpl'*, ***kwargs*) → None

Simplest way to visualize a sequence of T pose.

mmhuman3d.core.visualization.**visualize_kp2d**(*kp2d: numpy.ndarray*, *output_path: Optional[str] = None*, *frame_list: Optional[List[str]] = None*, *origin_frames: Optional[str] = None*, *image_array: Optional[numpy.ndarray] = None*, *limbs: Optional[Union[numpy.ndarray, List[int]]] = None*, *palette: Optional[Iterable[int]] = None*, *data_source: str = 'coco'*, *mask: Optional[Union[list, numpy.ndarray]] = None*, *img_format: str = '%06d.png'*, *start: int = 0*, *end: Optional[int] = None*, *overwrite: bool = False*, *with_file_name: bool = True*, *resolution: Optional[Union[Tuple[int, int], list]] = None*, *fps: Union[float, int] = 30*, *draw_bbox: bool = False*, *with_number: bool = False*, *pop_parts: Optional[Iterable[str]] = None*, *disable_tqdm: bool = False*, *disable_limbs: bool = False*, *return_array: Optional[bool] = False*, *keypoints_factory: dict = {'agora': ['pelvis', 'left_hip', 'right_hip', 'spine_1', 'left_knee', 'right_knee', 'spine_2', 'left_ankle', 'right_ankle', 'spine_3', 'left_foot', 'right_foot', 'neck', 'left_collar', 'right_collar', 'head', 'left_shoulder', 'right_shoulder', 'left_elbow', 'right_elbow', 'left_wrist', 'right_wrist', 'jaw', 'left_eyeball', 'right_eyeball', 'left_index_1', 'left_index_2', 'left_index_3', 'left_middle_1', 'left_middle_2', 'left_middle_3', 'left_pinky_1', 'left_pinky_2', 'left_pinky_3', 'left_ring_1', 'left_ring_2', 'left_ring_3', 'left_thumb_1', 'left_thumb_2', 'left_thumb_3', 'right_index_1', 'right_index_2', 'right_index_3', 'right_middle_1', 'right_middle_2', 'right_middle_3', 'right_pinky_1', 'right_pinky_2', 'right_pinky_3', 'right_ring_1', 'right_ring_2', 'right_ring_3', 'right_thumb_1', 'right_thumb_2', 'right_thumb_3', 'nose', 'right_eye', 'left_eye', 'right_ear', 'left_ear', 'left_bigtoe', 'left_smalltoe', 'left_heel', 'right_bigtoe', 'right_smalltoe', 'right_heel', 'left_thumb', 'left_index', 'left_middle', 'left_ring', 'left_pinky', 'right_thumb', 'right_index', 'right_middle', 'right_ring', 'right_pinky', 'right_eyebrow_1', 'right_eyebrow_2', 'right_eyebrow_3', 'right_eyebrow_4', 'right_eyebrow_5', 'left_eyebrow_5', 'left_eyebrow_4', 'left_eyebrow_3', 'left_eyebrow_2', 'left_eyebrow_1', 'nosebridge_1', 'nosebridge_2', 'nosebridge_3', 'nosebridge_4', 'nose_1', 'nose_2', 'nose_3', 'nose_4', 'nose_5', 'right_eye_1', 'right_eye_2', 'right_eye_3', 'right_eye_4', 'right_eye_5', 'right_eye_6', 'left_eye_4', 'left_eye_3', 'left_eye_2', 'left_eye_1', 'left_eye_6', 'left_eye_5', 'mouth_1', 'mouth_2', 'mouth_3', 'mouth_4', 'mouth_5', 'mouth_6', 'mouth_7', 'mouth_8', 'mouth_9', 'mouth_10', 'mouth_11', 'mouth_12', 'lip_1', 'lip_2', 'lip_3', 'lip_4', 'lip_5', 'lip_6', 'lip_7', 'lip_8'], 'coco': ['nose', 'left_eye', 'right_eye', 'left_ear', 'right_ear', 'left_shoulder', 'right_shoulder', 'left_elbow', 'right_elbow', 'left_wrist', 'right_wrist', 'left_hip_extra', 'right_hip_extra', 'left_knee', 'right_knee', 'left_ankle', 'right_ankle'], 'coco_wholebody': ['nose', 'left_eye', 'right_eye', 'left_ear', 'right_ear', 'left_shoulder', 'right_shoulder', 'left_elbow', 'right_elbow', 'left_wrist', 'right_wrist', 'left_hip', 'right_hip', 'left_knee', 'right_knee', 'left_ankle', 'right_ankle', 'left_bigtoe', 'left_smalltoe', 'left_heel', 'right_bigtoe', 'right_smalltoe', 'right_heel', 'face_contour_1', 'face_contour_2', 'face_contour_3', 'face_contour_4', 'face_contour_5', 'face_contour_6', 'face_contour_7', 'face_contour_8', 'face_contour_9',*

Parameters

- **kp2d** (*np.ndarray*) – should be array of shape (f * J * 2) or (f * n * J * 2)]
- **output_path** (*str*) – output video path or image folder.
- **frame_list** (*Optional[List[str]], optional*) – list of origin background frame paths, element in list each should be a image path like **.jpg* or **.png*. Higher priority than *origin_frames*. Use this when your file names is hard to sort or you only want to render a small number frames. Defaults to None.
- **origin_frames** (*Optional[str], optional*) – origin background frame path, could be *.mp4*, *.gif`(will be sliced into a folder) or an image folder. Lower priority than `frame_list*. Defaults to None.
- **limbs** (*Optional[Union[np.ndarray, List[int]]], optional*) – if not specified, the limbs will be searched by search_limbs, this option is for free skeletons like BVH file. Defaults to None.
- **palette** (*Iterable, optional*) – specified palette, three int represents (B, G, R). Should be tuple or list. Defaults to None.
- **data_source** (*str, optional*) – data source type. Defaults to 'coco'.
- **mask** (*Optional[Union[list, np.ndarray]], optional*) – mask to mask out the incorrect point. Pass a *np.ndarray* of shape (J,) or *list* of length J. Defaults to None.
- **img_format** (*str, optional*) – input image format. Default to '%06d.png',
- **start** (*int, optional*) – start frame index. Defaults to 0.
- **end** (*int, optional*) – end frame index. Exclusive. Could be positive int or negative int or None. None represents include all the frames.
- **overwrite** (*bool, optional*) – whether replace the origin frames. Defaults to False.
- **with_file_name** (*bool, optional*) – whether write origin frame name on the images. Defaults to True.
- **resolution** (*Optional[Union[Tuple[int, int], list]], optional*) – (height, width) of the output video will be the same size as the original images if not specified. Defaults to None.
- **fps** (*Union[float, int], optional*) – fps. Defaults to 30.
- **draw_bbox** (*bool, optional*) – whether need to draw bounding boxes. Defaults to False.
- **with_number** (*bool, optional*) – whether draw index number. Defaults to False.
- **pop_parts** (*Iterable[str], optional*) – The body part names you do not want to visualize. Supported parts are ['left_eye','right_eye' ,'nose', 'mouth', 'face', 'left_hand', 'right_hand']. Defaults to [].frame_list
- **disable_tqdm** (*bool, optional*) – Whether to disable the entire progressbar wrapper. Defaults to False.
- **disable_limbs** (*bool, optional*) – whether need to disable drawing limbs. Defaults to False.
- **return_array** (*bool, optional*) – Whether to return images as a opencv array. Defaults to None.
- **keypoints_factory** (*dict, optional*) – Dict of all the conventions. Defaults to KEYPOINTS_FACTORY.

Raises

- `FileNotFoundError` – check output video path.
- `FileNotFoundError` – check input frame paths.

Returns Union[None, np.ndarray].

mmhuman3d.core.visualization.**visualize_kp3d**(*kp3d: numpy.ndarray*, *output_path: Optional[str] = None*, *limbs: Optional[Union[numpy.ndarray, List[int]]] = None*, *palette: Optional[Iterable[int]] = None*, *data_source: str = 'coco'*, *mask: Optional[Union[numpy.ndarray, tuple, list]] = None*, *start: int = 0*, *end: Optional[int] = None*, *resolution: Union[list, Tuple[int, int]] = (1024, 1024)*, *fps: Union[float, int] = 30*, *frame_names: Optional[Union[List[str], str]] = None*, *orbit_speed: Union[float, int] = 0.5*, *value_range: Union[Tuple[int, int], list] = (- 100, 100)*, *pop_parts: Iterable[str] = ()*, *disable_limbs: bool = False*, *return_array: Optional[bool] = None*, *convention: str = 'opencv'*, *keypoints_factory: dict = {'agora': ['pelvis', 'left_hip', 'right_hip', 'spine_1', 'left_knee', 'right_knee', 'spine_2', 'left_ankle', 'right_ankle', 'spine_3', 'left_foot', 'right_foot', 'neck', 'left_collar', 'right_collar', 'head', 'left_shoulder', 'right_shoulder', 'left_elbow', 'right_elbow', 'left_wrist', 'right_wrist', 'jaw', 'left_eyeball', 'right_eyeball', 'left_index_1', 'left_index_2', 'left_index_3', 'left_middle_1', 'left_middle_2', 'left_middle_3', 'left_pinky_1', 'left_pinky_2', 'left_pinky_3', 'left_ring_1', 'left_ring_2', 'left_ring_3', 'left_thumb_1', 'left_thumb_2', 'left_thumb_3', 'right_index_1', 'right_index_2', 'right_index_3', 'right_middle_1', 'right_middle_2', 'right_middle_3', 'right_pinky_1', 'right_pinky_2', 'right_pinky_3', 'right_ring_1', 'right_ring_2', 'right_ring_3', 'right_thumb_1', 'right_thumb_2', 'right_thumb_3', 'nose', 'right_eye', 'left_eye', 'right_ear', 'left_ear', 'left_bigtoe', 'left_smalltoe', 'left_heel', 'right_bigtoe', 'right_smalltoe', 'right_heel', 'left_thumb', 'left_index', 'left_middle', 'left_ring', 'left_pinky', 'right_thumb', 'right_index', 'right_middle', 'right_ring', 'right_pinky', 'right_eyebrow_1', 'right_eyebrow_2', 'right_eyebrow_3', 'right_eyebrow_4', 'right_eyebrow_5', 'left_eyebrow_5', 'left_eyebrow_4', 'left_eyebrow_3', 'left_eyebrow_2', 'left_eyebrow_1', 'nosebridge_1', 'nosebridge_2', 'nosebridge_3', 'nosebridge_4', 'nose_1', 'nose_2', 'nose_3', 'nose_4', 'nose_5', 'right_eye_1', 'right_eye_2', 'right_eye_3', 'right_eye_4', 'right_eye_5', 'right_eye_6', 'left_eye_4', 'left_eye_3', 'left_eye_2', 'left_eye_1', 'left_eye_6', 'left_eye_5', 'mouth_1', 'mouth_2', 'mouth_3', 'mouth_4', 'mouth_5', 'mouth_6', 'mouth_7', 'mouth_8', 'mouth_9', 'mouth_10', 'mouth_11', 'mouth_12', 'lip_1', 'lip_2', 'lip_3', 'lip_4', 'lip_5', 'lip_6', 'lip_7', 'lip_8'], 'coco': ['nose', 'left_eye', 'right_eye', 'left_ear', 'right_ear', 'left_shoulder', 'right_shoulder', 'left_elbow', 'right_elbow', 'left_wrist', 'right_wrist', 'left_hip_extra', 'right_hip_extra', 'left_knee', 'right_knee', 'left_ankle', 'right_ankle'], 'coco_wholebody': ['nose', 'left_eye', 'right_eye', 'left_ear', 'right_ear', 'left_shoulder', 'right_shoulder', 'left_elbow', 'right_elbow', 'left_wrist', 'right_wrist', 'left_hip', 'right_hip', 'left_knee', 'right_knee', 'left_ankle', 'right_ankle', 'left_bigtoe', 'left_smalltoe', 'left_heel', 'right_bigtoe', 'right_smalltoe', 'right_heel', 'face_contour_1', 'face_contour_2', 'face_contour_3', 'face_contour_4', 'face_contour_5', 'face_contour_6', 'face_contour_7', 'face_contour_8', 'face_contour_9', 'face_contour_10', 'face_contour_11', 'face_contour_12', 'face_contour_13', 'face_contour_14', 'face_contour_15', 'face_contour_16', 'face_contour_17', 'right_eyebrow_1',

Parameters

- **kp3d** (*np.ndarray*) – shape could be (f * J * 4/3/2) or (f * num_person * J * 4/3/2)
- **output_path** (*str*) – output video path image folder.
- **limbs** (*Optional[Union[np.ndarray, List[int]]], optional*) – if not specified, the limbs will be searched by search_limbs, this option is for free skeletons like BVH file. Defaults to None.
- **palette** (*Iterable, optional*) – specified palette, three int represents (B, G, R). Should be tuple or list. Defaults to None.
- **data_source** (*str, optional*) – data source type. Defaults to 'coco'. choose in ['coco', 'smplx', 'smpl', 'coco_wholebody', 'mpi_inf_3dhp', 'mpi_inf_3dhp_test', 'h36m', 'pw3d', 'mpii']
- **mask** (*Optional[Union[list, tuple, np.ndarray]], optional*) – mask to mask out the incorrect points. Defaults to None.
- **start** (*int, optional*) – start frame index. Defaults to 0.
- **end** (*int, optional*) – end frame index. Could be positive int or negative int or None. None represents include all the frames. Defaults to None.
- **resolution** (*Union[list, Tuple[int, int]], optional*) – (width, height) of the output video will be the same size as the original images if not specified. Defaults to None.
- **fps** (*Union[float, int], optional*) – fps. Defaults to 30.
- **frame_names** (*Optional[Union[List[str], str]], optional*) – List(should be the same as frame numbers) or single string or string format (like 'frame%06d')for frame title, no title if None. Defaults to None.
- **orbit_speed** (*Union[float, int], optional*) – orbit speed of camera. Defaults to 0.5.
- **value_range** (*Union[Tuple[int, int], list], optional*) – range of axis value. Defaults to (-100, 100).
- **pop_parts** (*Iterable[str], optional*) – The body part names you do not want to visualize. Choose in ['left_eye','right_eye', 'nose', 'mouth', 'face', 'left_hand', 'right_hand']Defaults to [].
- **disable_limbs** (*bool, optional*) – whether need to disable drawing limbs. Defaults to False.
- **return_array** (*bool, optional*) – Whether to return images as opencv array .If None, an array will be returned when frame number is below 100. Defaults to None.
- **keypoints_factory** (*dict, optional*) – Dict of all the conventions. Defaults to KEYPOINTS_FACTORY.

Raises

- **TypeError** – check the type of input keypoints.
- **FileNotFoundError** – check the output video path.

Returns Union[None, np.ndarray].

mmhuman3d.core.visualization.**visualize_smpl_calibration**(*K*, *R*, *T*, *resolution*, ***kwargs*) → None

Visualize a smpl mesh which has opencv calibration matrix defined in screen.

mmhuman3d.core.visualization.**visualize_smpl_hmr**(*cam_transl*, *bbox=None*, *kp2d=None*, *focal_length=5000*, *det_width=224*, *det_height=224*, *bbox_format='xyxy'*, ***kwargs*) → None

Simplest way to visualize HMR or SPIN or Smplify pred smpl with origin frames and predicted cameras.

mmhuman3d.core.visualization.**visualize_smpl_pose**(*poses=None*, *verts=None*, ***kwargs*) → None

Simplest way to visualize a sequence of smpl pose.

Cameras will focus on the center of smpl mesh. *orbit speed* is recommended.

mmhuman3d.core.visualization.**visualize_smpl_vibe**(*orig_cam=None*, *pred_cam=None*, *bbox=None*, *output_path='sample.mp4'*, *resolution=None*, *aspect_ratio=1.0*, *bbox_scale_factor=1.25*, *bbox_format='xyxy'*, ***kwargs*) → None

Simplest way to visualize pred smpl with origin frames and predicted cameras.

CHAPTER

FOURTEEN

MMHUMAN3D.MODELS

14.1 models

mmhuman3d.models.**build_architecture**(*cfg*)
Build framework.

mmhuman3d.models.**build_backbone**(*cfg*)
Build backbone.

mmhuman3d.models.**build_body_model**(*cfg*)
Build body model.

mmhuman3d.models.**build_discriminator**(*cfg*)
Build discriminator.

mmhuman3d.models.**build_head**(*cfg*)
Build head.

mmhuman3d.models.**build_loss**(*cfg*)
Build loss.

mmhuman3d.models.**build_neck**(*cfg*)
Build neck.

14.2 architectures

class mmhuman3d.models.architectures.**HybrIK_trainer**(*backbone=None*, *neck=None*, *head=None*, *body_model=None*, *loss_beta=None*, *loss_theta=None*, *loss_twist=None*, *loss_uvd=None*, *init_cfg=None*)

Hybrik_trainer Architecture.

Parameters

- **backbone** (*dict | None, optional*) – Backbone config dict. Default: None.
- **neck** (*dict | None, optional*) – Neck config dict. Default: None
- **head** (*dict | None, optional*) – Regressor config dict. Default: None.
- **body_model** (*dict | None, optional*) – SMPL config dict. Default: None.
- **loss_beta** (*dict | None, optional*) – Losses config dict for beta (shape parameters) estimation. Default: None

- **loss_theta** (*dict | None, optional*) – Losses config dict for theta (pose parameters) estimation. Default: None
- **loss_twist** (*dict | None, optional*) – Losses config dict for twist angles estimation. Default: None
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Default: None

compute_losses(*predictions*, *targets*)
Compute regression losses for beta, theta, twist and uvd.

forward_test(*img*, *img_metas*, ***kwargs*)
Test step function.

In this function, train step is carried out with following the pipeline:

1. extract features with the backbone
2. **feed the extracted features into the head to** predicte beta, theta, twist angle, and heatmap (uvd map)

3. store predictions for evaluation :param img: Batch of data as input. :type img: torch.Tensor :param img_metas: Dict with image metas i.e. path :type img_metas: dict :param kwargs: Dict with ground-truth :type kwargs: dict

Returns Dict with image_path, vertices, xyz_17, uvd_jts, xyz_24 for predictions.

Return type all_preds (dict)

forward_train(*img*, *img_metas*, ***kwargs*)
Train step function.

In this function, train step is carried out with following the pipeline:

1. extract features with the backbone
2. **feed the extracted features into the head to** predicte beta, theta, twist angle, and heatmap (uvd map)
3. **compute regression losses of the predictions** and optimize backbone and head

Parameters

- **img** (*torch.Tensor*) – Batch of data as input.
- **kwargs** (*dict*) – Dict with ground-truth

Returns Dict with loss, information for logger, the number of samples.

Return type output (dict)

class mmhuman3d.models.architectures.**ImageBodyModelEstimator**(*backbone: Optional[dict] = None*, *neck: Optional[dict] = None*, *head: Optional[dict] = None*, *disc: Optional[dict] = None*, *registrant: Optional[dict] = None*, *body_model_train: Optional[dict] = None*, *body_model_test: Optional[dict] = None*, *convention: Optional[str] = 'human_data'*, *loss_keypoints2d: Optional[dict] = None*, *loss_keypoints3d: Optional[dict] = None*, *loss_vertex: Optional[dict] = None*, *loss_smpl_pose: Optional[dict] = None*, *loss_smpl_betas: Optional[dict] = None*, *loss_camera: Optional[dict] = None*, *loss_adv: Optional[dict] = None*, *init_cfg: Optional[Union[list, dict]] = None*)

forward_test(*img: torch.Tensor*, *img_metas: dict*, ***kwargs*)

Defines the computation performed at every call when testing.

class mmhuman3d.models.architectures.**VideoBodyModelEstimator**(*backbone: Optional[dict] = None*, *neck: Optional[dict] = None*, *head: Optional[dict] = None*, *disc: Optional[dict] = None*, *registrant: Optional[dict] = None*, *body_model_train: Optional[dict] = None*, *body_model_test: Optional[dict] = None*, *convention: Optional[str] = 'human_data'*, *loss_keypoints2d: Optional[dict] = None*, *loss_keypoints3d: Optional[dict] = None*, *loss_vertex: Optional[dict] = None*, *loss_smpl_pose: Optional[dict] = None*, *loss_smpl_betas: Optional[dict] = None*, *loss_camera: Optional[dict] = None*, *loss_adv: Optional[dict] = None*, *init_cfg: Optional[Union[list, dict]] = None*)

forward_test(*img_metas: dict*, ***kwargs*)

Defines the computation performed at every call when testing.

14.3 backbones

class mmhuman3d.models.backbones.**ResNet**(*depth*, *in_channels=3*, *stem_channels=None*, *base_channels=64*, *num_stages=4*, *strides=(1, 2, 2, 2)*, *dilations=(1, 1, 1, 1)*, *out_indices=(0, 1, 2, 3)*, *style='pytorch'*, *deep_stem=False*, *avg_down=False*, *frozen_stages=- 1*, *conv_cfg=None*, *norm_cfg={'requires_grad': True, 'type': 'BN'}*, *norm_eval=True*, *dcn=None*, *stage_with_dcn=(False, False, False, False)*, *plugins=None*, *with_cp=False*, *zero_init_residual=True*, *pretrained=None*, *init_cfg=None*)

ResNet backbone. :param depth: Depth of resnet, from {18, 34, 50, 101, 152}. :type depth: int :param stem_channels: Number of stem channels. If not specified,

it will be the same as *base_channels*. Default: None.

Parameters

- **base_channels** (*int*) – Number of base channels of res layer. Default: 64.
- **in_channels** (*int*) – Number of input image channels. Default: 3.
- **num_stages** (*int*) – Resnet stages. Default: 4.
- **strides** (*Sequence[int]*) – Strides of the first block of each stage.
- **dilations** (*Sequence[int]*) – Dilation of each stage.
- **out_indices** (*Sequence[int]*) – Output from which stages.
- **style** (*str*) – *pytorch* or *caffe*. If set to "pytorch", the stride-two layer is the 3x3 conv layer, otherwise the stride-two layer is the first 1x1 conv layer.
- **deep_stem** (*bool*) – Replace 7x7 conv in input stem with 3 3x3 conv
- **avg_down** (*bool*) – Use AvgPool instead of stride conv when downsampling in the bottleneck.
- **frozen_stages** (*int*) – Stages to be frozen (stop grad and set eval mode). -1 means not freezing any parameters.
- **norm_cfg** (*dict*) – Dictionary to construct and config norm layer.
- **norm_eval** (*bool*) – Whether to set norm layers to eval mode, namely, freeze running stats (mean and var). Note: Effect on Batch Norm and its variants only.
- **plugins** (*list[dict]*) – List of plugins for stages, each dict contains: - cfg (dict, required): Cfg dict to build plugin. - position (str, required): Position inside block to insert

  plugin, options are 'after_conv1', 'after_conv2', 'after_conv3'.

  - stages (tuple[bool], optional): Stages to apply plugin, length should be same as 'num_stages'.

- **with_cp** (*bool*) – Use checkpoint or not. Using checkpoint will save some memory while slowing down the training speed.
- **zero_init_residual** (*bool*) – Whether to use zero init for last norm layer in resblocks to let them behave as identity.
- **pretrained** (*str, optional*) – model pretrained path. Default: None

- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Default: None

Example

```
>>> from mmhuman3d.models import ResNet
>>> import torch
>>> self = ResNet(depth=18)
>>> self.eval()
>>> inputs = torch.rand(1, 3, 32, 32)
>>> level_outputs = self.forward(inputs)
>>> for level_out in level_outputs:
...     print(tuple(level_out.shape))
(1, 64, 8, 8)
(1, 128, 4, 4)
(1, 256, 2, 2)
(1, 512, 1, 1)
```

forward(*x*)
Forward function.

make_res_layer(***kwargs*)
Pack all blocks in a stage into a `ResLayer`.

make_stage_plugins(*plugins*, *stage_idx*)
Make plugins for ResNet `stage_idx` th stage. Currently we support to insert `context_block`, `empirical_attention_block`, `nonlocal_block` into the backbone like ResNet/ResNeXt. They could be inserted after conv1/conv2/conv3 of Bottleneck. An example of plugins format could be: .. rubric:: Examples

```
>>> plugins=[
...     dict(cfg=dict(type='xxx', arg1='xxx'),
...          stages=(False, True, True, True),
...          position='after_conv2'),
...     dict(cfg=dict(type='yyy'),
...          stages=(True, True, True, True),
...          position='after_conv3'),
...     dict(cfg=dict(type='zzz', postfix='1'),
...          stages=(True, True, True, True),
...          position='after_conv3'),
...     dict(cfg=dict(type='zzz', postfix='2'),
...          stages=(True, True, True, True),
...          position='after_conv3')
... ]
>>> self = ResNet(depth=18)
>>> stage_plugins = self.make_stage_plugins(plugins, 0)
>>> assert len(stage_plugins) == 3
```

Suppose `stage_idx=0`, the structure of blocks in the stage would be: .. code-block:: none

conv1-> conv2->conv3->yyy->zzz1->zzz2

Suppose 'stage_idx=1', the structure of blocks in the stage would be: .. code-block:: none

conv1-> conv2->xxx->conv3->yyy->zzz1->zzz2

If stages is missing, the plugin would be applied to all stages. :param plugins: List of plugins cfg to build. The postfix is

required if multiple same type plugins are inserted.

Parameters **stage_idx** (*int*) – Index of stage to build

Returns Plugins for current stage

Return type list[dict]

property norm1

the normalization layer named "norm1"

Type nn.Module

train(*mode=True*)

Convert the model into training mode while keep normalization layer freezed.

class mmhuman3d.models.backbones.**ResNetV1d**(***kwargs*)

ResNetV1d variant described in Bag of Tricks. Compared with default ResNet(ResNetV1b), ResNetV1d replaces the 7x7 conv in the input stem with three 3x3 convs. And in the downsampling block, a 2x2 avg_pool with stride 2 is added before conv, whose stride is changed to 1.

14.4 discriminators

class mmhuman3d.models.discriminators.**SMPLDiscriminator**(*beta_channel=(10, 5, 1)*, *per_joint_channel=(9, 32, 32, 1)*, *full_pose_channel=(736, 1024, 1024, 1)*)

Discriminator for SMPL pose and shape parameters.

It is composed of a discriminator for SMPL shape parameters, a discriminator for SMPL pose parameters of all joints and a discriminator for SMPL pose parameters of each joint. :param beta_channel: Tuple of neuron count of the

discriminator of shape parameters. Defaults to (10, 5, 1)

Parameters

- **per_joint_channel** (*tuple of int*) – Tuple of neuron count of the discriminator of each joint. Defaults to (9, 32, 32, 1)
- **full_pose_channel** (*tuple of int*) – Tuple of neuron count of the discriminator of full pose. Defaults to (23*32, 1024, 1024, 1)

forward(*thetas*)

Forward function.

init_weights()

Initialize model weights.

14.5 necks

class mmhuman3d.models.necks.**TemporalGRUEncoder**(*input_size: Optional[int] = 2048*, *num_layers: Optional[int] = 1*, *hidden_size: Optional[int] = 2048*, *init_cfg: Optional[Union[list, dict]] = None*)

TemporalEncoder used for VIBE. Adapted from https://github.com/mkocabas/VIBE.

Parameters

- **input_size** (*int, optional*) – dimension of input feature. Default: 2048.
- **num_layer** (*int, optional*) – number of layers for GRU. Default: 1.
- **hidden_size** (*int, optional*) – hidden size for GRU. Default: 2048.
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Default: None.

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

14.6 heads

class mmhuman3d.models.heads.**HMRHead**(*feat_dim*, *smpl_mean_params=None*, *npose=144*, *nbeta=10*, *ncam=3*, *hdim=1024*, *init_cfg=None*)

forward(*x*, *init_pose=None*, *init_shape=None*, *init_cam=None*, *n_iter=3*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmhuman3d.models.heads.**HybrIKHead**(*feature_channel=512*, *deconv_dim=[256, 256, 256]*, *num_joints=29*, *depth_dim=64*, *height_dim=64*, *width_dim=64*, *smpl_mean_params=None*)

HybrIK parameters regressor head.

Parameters

- **feature_channel** (*int*) – Number of input channels
- **deconv_dim** (*List[int]*) – List of deconvolution dimensions
- **num_joints** (*int*) – Number of keypoints
- **depth_dim** (*int*) – Depth dimension

- **height_dim** (*int*) – Height dimension
- **width_dim** (*int*) – Width dimension
- **smpl_mean_params** (*str*) – file name of the mean SMPL parameters

flip_phi(*pred_phi*)
Flip phi.

Parameters **pred_phi** (*torch.Tensor*) – phi in shape (Num_twistx2)

Returns flipped phi in shape (Num_twistx2)

Return type pred_phi (torch.Tensor)

flip_uvd_coord(*pred_jts*, *flip=False*, *flatten=True*)
Flip uvd coordinates.

Parameters

- **pred_jts** (*torch.Tensor*) – predicted uvd coordinates with shape (Bx87)
- **flip** (*bool*) – Store True to flip uvd coordinates. Default: False.
- **flatten** (*bool*) – Store True to reshape uvd_coordinates to shape (Bx29x3) Default: True

Returns flipped uvd coordinates with shape (Bx29x3)

Return type pred_jts (torch.Tensor)

forward(*feature*, *trans_inv*, *intrinsic_param*, *joint_root*, *depth_factor*, *smpl_layer*, *flip_item=None*, *flip_output=False*)
Forward function.

Parameters

- **feature** (*torch.Tensor*) – features extracted from backbone
- **trans_inv** (*torch.Tensor*) – inverse affine transformation matrix with shape (Bx2x3)
- **intrinsic_param** (*torch.Tensor*) – camera intrinsic matrix with shape (Bx3x3)
- **joint_root** (*torch.Tensor*) – root joint coordinate with shape (Bx3)
- **depth_factor** (*float*) – depth factor with shape (Bx1)
- **smpl_layer** (*torch.Tensor*) – smpl body model
- **flip_item** (*List[torch.Tensor]|None*) – list containing items to flip
- **flip_output** (*bool*) – Store True to flip output. Default: False

Returns Dict containing model predictions.

Return type output (dict)

uvd_to_cam(*uvd_jts*, *trans_inv*, *intrinsic_param*, *joint_root*, *depth_factor*, *return_relative=True*)
Project uvd coordinates to camera frame.

Parameters

- **uvd_jts** (*torch.Tensor*) – uvd coordinates with shape (BxNum_jointsx3)
- **trans_inv** (*torch.Tensor*) – inverse affine transformation matrix with shape (Bx2x3)
- **intrinsic_param** (*torch.Tensor*) – camera intrinsic matrix with shape (Bx3x3)

- **joint_root** (*torch.Tensor*) – root joint coordinate with shape (Bx3)
- **depth_factor** (*float*) – depth factor with shape (Bx1)
- **return_relative** (*bool*) – Store True to return root normalized relative coordinates. Default: True.

Returns uvd coordinates in camera frame with shape (BxNum_jointsx3)

Return type xyz_jts (torch.Tensor)

14.7 losses

class mmhuman3d.models.losses.**CameraPriorLoss**(*scale=10*, *reduction='mean'*, *loss_weight=1.0*)

Prior loss for predicted camera.

Parameters

- **reduction** (*str, optional*) – The method that reduces the loss to a scalar. Options are "none", "mean" and "sum".
- **scale** (*float, optional*) – The scale coefficient for regularizing camera parameters. Defaults to 10
- **loss_weight** (*float, optional*) – The weight of the loss. Defaults to 1.0

forward(*cameras*, *loss_weight_override=None*, *reduction_override=None*)

Forward function of loss.

Parameters

- **cameras** (*torch.Tensor*) – The predicted camera parameters
- **loss_weight_override** (*float, optional*) – The weight of loss used to override the original weight of loss
- **reduction_override** (*str, optional*) – The reduction method used to override the original reduction method of the loss. Defaults to None

Returns The calculated loss

Return type torch.Tensor

class mmhuman3d.models.losses.**GANLoss**(*gan_type*, *real_label_val=1.0*, *fake_label_val=0.0*, *loss_weight=1.0*)

Define GAN loss.

Parameters

- **gan_type** (*str*) – Support 'vanilla', 'lsgan', 'wgan', 'hinge'.
- **real_label_val** (*float*) – The value for real label. Default: 1.0.
- **fake_label_val** (*float*) – The value for fake label. Default: 0.0.
- **loss_weight** (*float*) – Loss weight. Default: 1.0. Note that loss_weight is only for generators; and it is always 1.0 for discriminators.

forward(*input*, *target_is_real*, *is_disc=False*)

Parameters

- **input** (*Tensor*) – The input for the loss module, i.e., the network prediction.

- **target_is_real** (*bool*) – Whether the targe is real or fake.
- **is_disc** (*bool*) – Whether the loss for discriminators or not. Default: False.

Returns GAN loss value.

Return type Tensor

get_target_label(*input*, *target_is_real*)
Get target label.

Parameters

- **input** (*Tensor*) – Input tensor.
- **target_is_real** (*bool*) – Whether the target is real or fake.

Returns

Target tensor. Return bool for wgan, otherwise, return Tensor.

Return type (bool | Tensor)

class mmhuman3d.models.losses.**JointPriorLoss**(*reduction='mean'*, *loss_weight=1.0*, *use_full_body=False*, *smooth_spine=False*, *smooth_spine_loss_weight=1.0*)
Prior loss for joint angles.

Parameters

- **reduction** (*str, optional*) – The method that reduces the loss to a scalar. Options are "none", "mean" and "sum".
- **loss_weight** (*float, optional*) – The weight of the loss. Defaults to 1.0
- **use_full_body** (*bool, optional*) – Use full set of joint constraints (in standard joint angles).
- **smooth_spine** (*bool, optional*) – Ensuring smooth spine rotations
- **smooth_spine_loss_weight** (*float, optional*) – An additional weight factor multiplied on smooth spine loss

forward(*body_pose*, *loss_weight_override=None*, *reduction_override=None*)
Forward function of loss.

Parameters

- **body_pose** (*torch.Tensor*) – The body pose parameters
- **loss_weight_override** (*float, optional*) – The weight of loss used to override the original weight of loss
- **reduction_override** (*str, optional*) – The reduction method used to override the original reduction method of the loss. Defaults to None

Returns The calculated loss

Return type torch.Tensor

class mmhuman3d.models.losses.**KeypointMSELoss**(*reduction='mean'*, *loss_weight=1.0*, *sigma=1.0*)
MSELoss for 2D and 3D keypoints.

Parameters

- **reduction** (*str, optional*) – The method that reduces the loss to a scalar. Options are "none", "mean" and "sum".
- **loss_weight** (*float, optional*) – The weight of the loss. Defaults to 1.0

- **sigma** (*float, optional*) – Weighing parameter of Geman-McClure error function. Defaults to 1.0 (no effect).

forward(*pred*, *target*, *pred_conf=None*, *target_conf=None*, *keypoint_weight=None*, *avg_factor=None*, *loss_weight_override=None*, *reduction_override=None*)

Forward function of loss.

Parameters

- **pred** (*torch.Tensor*) – The prediction. Shape should be (N, K, 2/3) B: batch size. K: number of keypoints.
- **target** (*torch.Tensor*) – The learning target of the prediction. Shape should be the same as pred.
- **pred_conf** (*optional, torch.Tensor*) – Confidence of predicted keypoints. Shape should be (N, K).
- **target_conf** (*optional, torch.Tensor*) – Confidence of target keypoints. Shape should be the same as pred_conf.
- **keypoint_weight** (*optional, torch.Tensor*) – keypoint-wise weight. shape should be (K,). This weight allow different weights to be assigned at different body parts.
- **avg_factor** (*int, optional*) – Average factor that is used to average the loss. Defaults to None.
- **loss_weight_override** (*float, optional*) – The overall weight of loss used to override the original weight of loss.
- **reduction_override** (*str, optional*) – The reduction method used to override the original reduction method of the loss. Defaults to None.

Returns The calculated loss

Return type torch.Tensor

class mmhuman3d.models.losses.**L1Loss**(*reduction='mean'*, *loss_weight=1.0*)

L1 loss.

Parameters

- **reduction** (*str, optional*) – The method to reduce the loss. Options are "none", "mean" and "sum".
- **loss_weight** (*float, optional*) – The weight of loss.

forward(*pred*, *target*, *weight=None*, *avg_factor=None*, *reduction_override=None*)

Forward function.

Parameters

- **pred** (*torch.Tensor*) – The prediction.
- **target** (*torch.Tensor*) – The learning target of the prediction.
- **weight** (*torch.Tensor, optional*) – The weight of loss for each prediction. Defaults to None.
- **avg_factor** (*int, optional*) – Average factor that is used to average the loss. Defaults to None.
- **reduction_override** (*str, optional*) – The reduction method used to override the original reduction method of the loss. Defaults to None.

class mmhuman3d.models.losses.**MSELoss**(*reduction='mean'*, *loss_weight=1.0*)
MSELoss.

Parameters

- **reduction** (*str, optional*) – The method that reduces the loss to a scalar. Options are "none", "mean" and "sum".
- **loss_weight** (*float, optional*) – The weight of the loss. Defaults to 1.0

forward(*pred*, *target*, *weight=None*, *avg_factor=None*, *reduction_override=None*)
Forward function of loss.

Parameters

- **pred** (*torch.Tensor*) – The prediction.
- **target** (*torch.Tensor*) – The learning target of the prediction.
- **weight** (*torch.Tensor, optional*) – Weight of the loss for each prediction. Defaults to None.
- **avg_factor** (*int, optional*) – Average factor that is used to average the loss. Defaults to None.
- **reduction_override** (*str, optional*) – The reduction method used to override the original reduction method of the loss. Defaults to None.

Returns The calculated loss

Return type torch.Tensor

class mmhuman3d.models.losses.**MaxMixturePrior**(*prior_folder='data'*, *num_gaussians=8*, *dtype=torch.float32*, *epsilon=1e-16*, *use_merged=True*, *reduction=None*, *loss_weight=1.0*)
Ref: SMPLify-X https://github.com/vchoutas/smplify-x/blob/master/smplifyx/prior.py

forward(*body_pose*, *loss_weight_override=None*, *reduction_override=None*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

get_mean()
Returns the mean of the mixture.

log_likelihood(*pose*)
Create graph operation for negative log-likelihood calculation.

class mmhuman3d.models.losses.**ShapePriorLoss**(*reduction='mean'*, *loss_weight=1.0*)
Prior loss for body shape parameters.

Parameters

- **reduction** (*str, optional*) – The method that reduces the loss to a scalar. Options are "none", "mean" and "sum".
- **loss_weight** (*float, optional*) – The weight of the loss. Defaults to 1.0

forward(*betas*, *loss_weight_override=None*, *reduction_override=None*)
Forward function of loss.

Parameters

- **betas** (*torch.Tensor*) – The body shape parameters
- **loss_weight_override** (*float, optional*) – The weight of loss used to override the original weight of loss
- **reduction_override** (*str, optional*) – The reduction method used to override the original reduction method of the loss. Defaults to None

Returns The calculated loss

Return type torch.Tensor

class mmhuman3d.models.losses.**SmoothJointLoss**(*reduction='mean'*, *loss_weight=1.0*, *degree=False*)
Smooth loss for joint angles.

Parameters

- **reduction** (*str, optional*) – The method that reduces the loss to a scalar. Options are "none", "mean" and "sum".
- **loss_weight** (*float, optional*) – The weight of the loss. Defaults to 1.0
- **degree** (*bool, optional*) – The flag which represents whether the input tensor is in degree or radian.

forward(*body_pose*, *loss_weight_override=None*, *reduction_override=None*)
Forward function of loss.

Parameters

- **body_pose** (*torch.Tensor*) – The body pose parameters
- **loss_weight_override** (*float, optional*) – The weight of loss used to override the original weight of loss
- **reduction_override** (*str, optional*) – The reduction method used to override the original reduction method of the loss. Defaults to None

Returns The calculated loss

Return type torch.Tensor

class mmhuman3d.models.losses.**SmoothL1Loss**(*beta=1.0*, *reduction='mean'*, *loss_weight=1.0*)
Smooth L1 loss.

Parameters

- **beta** (*float, optional*) – The threshold in the piecewise function. Defaults to 1.0.
- **reduction** (*str, optional*) – The method to reduce the loss. Options are "none", "mean" and "sum". Defaults to "mean".
- **loss_weight** (*float, optional*) – The weight of loss.

forward(*pred*, *target*, *weight=None*, *avg_factor=None*, *reduction_override=None*, ***kwargs*)
Forward function.

Parameters

- **pred** (*torch.Tensor*) – The prediction.
- **target** (*torch.Tensor*) – The learning target of the prediction.

- **weight** (*torch.Tensor, optional*) – The weight of loss for each prediction. Defaults to None.
- **avg_factor** (*int, optional*) – Average factor that is used to average the loss. Defaults to None.
- **reduction_override** (*str, optional*) – The reduction method used to override the original reduction method of the loss. Defaults to None.

class mmhuman3d.models.losses.**SmoothPelvisLoss**(*reduction='mean'*, *loss_weight=1.0*, *degree=False*)

Smooth loss for pelvis angles.

Parameters

- **reduction** (*str, optional*) – The method that reduces the loss to a scalar. Options are "none", "mean" and "sum".
- **loss_weight** (*float, optional*) – The weight of the loss. Defaults to 1.0
- **degree** (*bool, optional*) – The flag which represents whether the input tensor is in degree or radian.

forward(*global_orient*, *loss_weight_override=None*, *reduction_override=None*)

Forward function of loss.

Parameters

- **global_orient** (*torch.Tensor*) – The global orientation parameters
- **loss_weight_override** (*float, optional*) – The weight of loss used to override the original weight of loss
- **reduction_override** (*str, optional*) – The reduction method used to override the original reduction method of the loss. Defaults to None

Returns The calculated loss

Return type torch.Tensor

class mmhuman3d.models.losses.**SmoothTranslationLoss**(*reduction='mean'*, *loss_weight=1.0*)

Smooth loss for translations.

Parameters

- **reduction** (*str, optional*) – The method that reduces the loss to a scalar. Options are "none", "mean" and "sum".
- **loss_weight** (*float, optional*) – The weight of the loss. Defaults to 1.0

forward(*translation*, *loss_weight_override=None*, *reduction_override=None*)

Forward function of loss.

Parameters

- **translation** (*torch.Tensor*) – The body translation parameters
- **loss_weight_override** (*float, optional*) – The weight of loss used to override the original weight of loss
- **reduction_override** (*str, optional*) – The reduction method used to override the original reduction method of the loss. Defaults to None

Returns The calculated loss

Return type torch.Tensor

mmhuman3d.models.losses.**convert_to_one_hot**(*targets: torch.Tensor*, *classes*) → torch.Tensor
This function converts target class indices to one-hot vectors, given the number of classes.

Parameters

- **targets** (*Tensor*) – The ground truth label of the prediction with shape (N, 1)
- **classes** (*int*) – the number of classes.

Returns Processed loss values.

Return type Tensor

mmhuman3d.models.losses.**reduce_loss**(*loss*, *reduction*)
Reduce loss as specified.

Parameters

- **loss** (*Tensor*) – Elementwise loss tensor.
- **reduction** (*str*) – Options are "none", "mean" and "sum".

Returns Reduced loss tensor.

Return type Tensor

mmhuman3d.models.losses.**weight_reduce_loss**(*loss*, *weight=None*, *reduction='mean'*, *avg_factor=None*)
Apply element-wise weight and reduce loss.

Parameters

- **loss** (*Tensor*) – Element-wise loss.
- **weight** (*Tensor*) – Element-wise weights.
- **reduction** (*str*) – Same as built-in losses of PyTorch.
- **avg_factor** (*float*) – Average factor when computing the mean of losses.

Returns Processed loss values.

Return type Tensor

mmhuman3d.models.losses.**weighted_loss**(*loss_func*)
Create a weighted version of a given loss function.

To use this decorator, the loss function must have the signature like *loss_func(pred, target, **kwargs)*. The function only needs to compute element-wise loss without any reduction. This decorator will add weight and reduction arguments to the function. The decorated function will have the signature like *loss_func(pred, target, weight=None, reduction='mean', avg_factor=None, **kwargs)*.

Example

```
>>> import torch
>>> @weighted_loss
>>> def l1_loss(pred, target):
>>>     return (pred - target).abs()
```

```
>>> pred = torch.Tensor([0, 2, 3])
>>> target = torch.Tensor([1, 1, 1])
>>> weight = torch.Tensor([1, 0, 1])
```

```
>>> l1_loss(pred, target)
tensor(1.3333)
>>> l1_loss(pred, target, weight)
tensor(1.)
>>> l1_loss(pred, target, reduction='none')
tensor([1., 1., 2.])
>>> l1_loss(pred, target, weight, avg_factor=2)
tensor(1.5000)
```

14.8 utils

class mmhuman3d.models.utils.**FitsDict**(*fits='static'*)

Dictionary keeping track of the best fit per image in the training set.

Ref: https://github.com/nkolot/SPIN/blob/master/train/fits_dict.py

flip_pose(*pose*, *is_flipped*)

flip SMPL pose parameters.

rotate_pose(*pose*, *rot*)

Rotate SMPL pose parameters by rot degrees.

save()

Save dictionary state to disk.

class mmhuman3d.models.utils.**ResLayer**(*block*, *inplanes*, *planes*, *num_blocks*, *stride=1*, *avg_down=False*, *conv_cfg=None*, *norm_cfg={'type': 'BN'}*, *downsample_first=True*, ***kwargs*)

ResLayer to build ResNet style backbone.

Parameters

- **block** (*nn.Module*) – block used to build ResLayer.
- **inplanes** (*int*) – inplanes of block.
- **planes** (*int*) – planes of block.
- **num_blocks** (*int*) – number of blocks.
- **stride** (*int*) – stride of the first block. Default: 1
- **avg_down** (*bool*) – Use AvgPool instead of stride conv when downsampling in the bottleneck. Default: False
- **conv_cfg** (*dict*) – dictionary to construct and config conv layer. Default: None
- **norm_cfg** (*dict*) – dictionary to construct and config norm layer. Default: dict(type='BN')
- **downsample_first** (*bool*) – Downsample at the first block or last block. False for Hourglass, True for ResNet. Default: True

class mmhuman3d.models.utils.**SimplifiedBasicBlock**(*inplanes*, *planes*, *stride=1*, *dilation=1*, *downsample=None*, *style='pytorch'*, *with_cp=False*, *conv_cfg=None*, *norm_cfg={'type': 'BN'}*, *dcn=None*, *plugins=None*, *init_fg=None*)

Simplified version of original basic residual block. This is used in SCNet.

- Norm layer is now optional

- Last ReLU in forward function is removed

forward(*x*)
Forward function.

property norm1
normalization layer after the first convolution layer

Type nn.Module

property norm2
normalization layer after the second convolution layer

Type nn.Module

mmhuman3d.models.utils.**batch_inverse_kinematics_transform**(*pose_skeleton*, *global_orient*, *phis*, *rest_pose*, *children*, *parents*, *dtype=torch.float32*, *train=False*, *leaf_thetas=None*)

Applies inverse kinematics transform to joints in a batch.

Parameters

- **pose_skeleton** (*torch.tensor*) – Locations of estimated pose skeleton with shape (Bx29x3)
- **global_orient** (*torch.tensor|none*) – Tensor of global rotation matrices with shape (Bx1x3x3)
- **phis** (*torch.tensor*) – Rotation on bone axis parameters with shape (Bx23x2)
- **rest_pose** (*torch.tensor*) – Locations of rest (Template) pose with shape (Bx29x3)
- **children** (*List[int]*) – list of indexes of kinematic children with len 29
- **parents** (*List[int]*) – list of indexes of kinematic parents with len 29
- **dtype** (*torch.dtype, optional*) – Data type of the created tensors. Default: torch.float32
- **train** (*bool*) – Store True in train mode. Default: False
- **leaf_thetas** (*torch.tensor, optional*) – Rotation matrixes for 5 leaf joints (Bx5x3x3). Default: None

Returns

Rotation matrics of all joints with shape (Bx29x3x3) rotate_rest_pose (torch.tensor):

Locations of rotated rest/ template pose with shape (Bx29x3)

Return type rot_mats (torch.tensor)

CHAPTER

FIFTEEN

MMHUMAN3D.DATA

15.1 data

15.2 datasets

class mmhuman3d.data.datasets.**AdversarialDataset**(*train_dataset: torch.utils.data.dataset.Dataset*, *adv_dataset: torch.utils.data.dataset.Dataset*)

Mix Dataset for the adversarial training in 3D human mesh estimation task.

The dataset combines data from two datasets and return a dict containing data from two datasets. :param train_dataset: Dataset for 3D human mesh estimation. :type train_dataset: `Dataset` :param adv_dataset: Dataset for adversarial learning. :type adv_dataset: `Dataset`

class mmhuman3d.data.datasets.**BaseDataset**(*data_prefix: str*, *pipeline: list*, *ann_file: Optional[str] = None*, *test_mode: Optional[bool] = False*, *dataset_name: Optional[str] = None*)

Base dataset.

Parameters

- **data_prefix** (*str*) – the prefix of data path.
- **pipeline** (*list*) – a list of dict, where each element represents a operation defined in *mmhuman3d.datasets.pipelines*.
- **ann_file** (*str | None, optional*) – the annotation file. When ann_file is str, the subclass is expected to read from the ann_file. When ann_file is None, the subclass is expected to read according to data_prefix.
- **test_mode** (*bool*) – in train mode or test mode. Default: None.
- **dataset_name** (*str | None, optional*) – the name of dataset. It is used to identify the type of evaluation metric. Default: None.

abstract load_annotations()

Load annotations from `ann_file`

prepare_data(*idx: int*)

"Prepare raw data for the f'{idx'}-th data.

class mmhuman3d.data.datasets.**Compose**(*transforms*)

Compose a data pipeline with a sequence of transforms.

Parameters **transforms** (*list[dict | callable]*) – Either config dicts of transforms or transform objects.

class mmhuman3d.data.datasets.**ConcatDataset**(*datasets: list*)
A wrapper of concatenated dataset.

Same as `torch.utils.data.dataset.ConcatDataset`, but add *get_cat_ids* function.

Parameters **datasets** (list[`Dataset`]) – A list of datasets.

class mmhuman3d.data.datasets.**DistributedSampler**(*dataset*, *num_replicas=None*, *rank=None*, *shuffle=True*, *round_up=True*)

class mmhuman3d.data.datasets.**HumanImageDataset**(*data_prefix: str*, *pipeline: list*, *dataset_name: str*, *body_model: Optional[dict] = None*, *ann_file: Optional[str] = None*, *convention: Optional[str] = 'human_data'*, *test_mode: Optional[bool] = False*)

Human Image Dataset.

Parameters

- **data_prefix** (*str*) – the prefix of data path.
- **pipeline** (*list*) – a list of dict, where each element represents a operation defined in *mmhuman3d.datasets.pipelines*.
- **dataset_name** (*str | None*) – the name of dataset. It is used to identify the type of evaluation metric. Default: None.
- **body_model** (*dict | None, optional*) – the config for body model, which will be used to generate meshes and keypoints. Default: None.
- **ann_file** (*str | None, optional*) – the annotation file. When ann_file is str, the subclass is expected to read from the ann_file. When ann_file is None, the subclass is expected to read according to data_prefix.
- **convention** (*str, optional*) – keypoints convention. Keypoints will be converted from "human_data" to the given one. Default: "human_data"
- **test_mode** (*bool, optional*) – in train mode or test mode. Default: False.

evaluate(*outputs: list*, *res_folder: str*, *metric: Optional[str] = 'joint_error'*)
Evaluate 3D keypoint results.

Parameters

- **outputs** (*list*) – results from model inference.
- **res_folder** (*str*) – path to store results.
- **metric** (*str*) – the type of metric. Default: 'joint_error'

Returns A dict of all evaluation results.

Return type dict

get_annotation_file()
Get path of the annotation file.

load_annotations()
Load annotation from the annotation file.

Here we simply use `HumanData` to parse the annotation.

prepare_data(*idx: int*)
Generate and transform data.

prepare_raw_data(*idx: int*)
Get item from self.human_data.

class mmhuman3d.data.datasets.**HumanVideoDataset**(*data_prefix: str*, *pipeline: list*, *dataset_name: str*, *seq_len: Optional[int] = 16*, *overlap: Optional[float] = 0.0*, *only_vid_name: Optional[bool] = False*, *body_model: Optional[dict] = None*, *ann_file: Optional[str] = None*, *convention: Optional[str] = 'human_data'*, *test_mode: Optional[bool] = False*)

Human Video Dataset.

Parameters

- **data_prefix** (*str*) – the prefix of data path.
- **pipeline** (*list*) – a list of dict, where each element represents a operation defined in *mmhuman3d.datasets.pipelines*.
- **dataset_name** (*str | None*) – the name of dataset. It is used to identify the type of evaluation metric. Default: None.
- **seq_len** (*int, optional*) – the length of input sequence. Default: 16.
- **overlap** (*float, optional*) – the overlap between different sequences. Default: 0
- **only_vid_name** (*bool, optional*) – the format of image_path. If only_vid_name is true, image_path only contains the video name. Otherwise, image_path contains both video_name and frame index.
- **body_model** (*dict | None, optional*) – the config for body model, which will be used to generate meshes and keypoints. Default: None.
- **ann_file** (*str | None, optional*) – the annotation file. When ann_file is str, the subclass is expected to read from the ann_file. When ann_file is None, the subclass is expected to read according to data_prefix.
- **convention** (*str, optional*) – keypoints convention. Keypoints will be converted from "human_data" to the given one. Default: "human_data"
- **test_mode** (*bool, optional*) – in train mode or test mode. Default: False.

prepare_data(*idx: int*)

Prepare data for each chunk.

Step 1: get annotation from each frame. Step 2: add metas of each chunk.

class mmhuman3d.data.datasets.**HybrIKHumanImageDataset**(*data_prefix*, *pipeline*, *dataset_name*, *ann_file*, *test_mode=False*)

Dataset for HybrIK training. The dataset loads raw features and apply specified transforms to return a dict containing the image tensors and other information.

Parameters

- **data_prefix** (*str*) – Path to a directory where preprocessed datasets are held.
- **pipeline** (*list[dict | callable]*) – A sequence of data transforms.
- **dataset_name** (*str*) – accepted names include 'h36m', 'pw3d', 'mpi_inf_3dhp', 'coco'
- **ann_file** (*str*) – Name of annotation file.
- **test_mode** (*bool*) – Store True when building test dataset. Default: False.

evaluate(*outputs*, *res_folder*, *metric='joint_error'*, *logger=None*)

Evaluate 3D keypoint results.

static get_3d_keypoints_vis(*keypoints*)

Get 3d keypoints and visibility mask :param keypoints: 2d (NxKx3) or 3d (NxKx4) keypoints with

visibility. N refers to number of datapoints, K refers to number of keypoints.

Returns (NxKx3) 3d keypoints joint_vis (np.ndarray): (NxKx3) visibility mask for keypoints

Return type joint_img (np.ndarray)

`get_annotation_file()`
Obtain annotation file path from data prefix.

`load_annotations()`
Load annotations.

class mmhuman3d.data.datasets.**MeshDataset**(*data_prefix: str*, *pipeline: list*, *dataset_name: str*, *ann_file: Optional[str] = None*, *test_mode: Optional[bool] = False*)

Mesh Dataset. This dataset only contains smpl data.

Parameters

- **data_prefix** (*str*) – the prefix of data path.
- **pipeline** (*list*) – a list of dict, where each element represents a operation defined in *mmhuman3d.datasets.pipelines*.
- **dataset_name** (*str | None*) – the name of dataset. It is used to identify the type of evaluation metric. Default: None.
- **ann_file** (*str | None, optional*) – the annotation file. When ann_file is str, the subclass is expected to read from the ann_file. When ann_file is None, the subclass is expected to read according to data_prefix.
- **test_mode** (*bool, optional*) – in train mode or test mode. Default: False.

`load_annotations()`
Load annotations from ann_file

class mmhuman3d.data.datasets.**MixedDataset**(*configs: list*, *partition: list*, *num_data: Optional[int] = None*)

Mixed Dataset.

Parameters

- **config** (*list*) – the list of different datasets.
- **partition** (*list*) – the ratio of datasets in each batch.
- **num_data** (*int | None, optional*) – if num_data is not None, the number of iterations is set to this fixed value. Otherwise, the number of iterations is set to the maximum size of each single dataset. Default: None.

class mmhuman3d.data.datasets.**RepeatDataset**(*dataset: torch.utils.data.dataset.Dataset*, *times: int*)

A wrapper of repeated dataset.

The length of repeated dataset will be *times* larger than the original dataset. This is useful when the data loading time is long but the dataset is small. Using RepeatDataset can reduce the data loading time between epochs.

Parameters

- **dataset** (Dataset) – The dataset to be repeated.
- **times** (*int*) – Repeat times.

mmhuman3d.data.datasets.**build_dataloader**(*dataset: torch.utils.data.dataset.Dataset*, *samples_per_gpu: int*, *workers_per_gpu: int*, *num_gpus: Optional[int] = 1*, *dist: Optional[bool] = True*, *shuffle: Optional[bool] = True*, *round_up: Optional[bool] = True*, *seed: Optional[int] = None*, ***kwargs*)

Build PyTorch DataLoader.

In distributed training, each GPU/process has a dataloader. In non-distributed training, there is only one dataloader for all GPUs.

Parameters

- **dataset** (Dataset) – A PyTorch dataset.
- **samples_per_gpu** (*int*) – Number of training samples on each GPU, i.e., batch size of each GPU.
- **workers_per_gpu** (*int*) – How many subprocesses to use for data loading for each GPU.
- **num_gpus** (*int, optional*) – Number of GPUs. Only used in non-distributed training.
- **dist** (*bool, optional*) – Distributed training/test or not. Default: True.
- **shuffle** (*bool, optional*) – Whether to shuffle the data at every epoch. Default: True.
- **round_up** (*bool, optional*) – Whether to round up the length of dataset by adding extra samples to make it evenly divisible. Default: True.
- **kwargs** – any keyword argument to be used to initialize DataLoader

Returns A PyTorch dataloader.

Return type DataLoader

mmhuman3d.data.datasets.**build_dataset**(*cfg: Union[dict, list, tuple]*, *default_args: Optional[dict] = None*)

"Build dataset by the given config.

15.3 data_converters

15.4 data_structures

class mmhuman3d.data.data_structures.**HumanData**(**args: Any*, ***kwargs: Any*)

check_keypoints_compressed() → bool

Check whether the keypoints are compressed.

Returns Whether the keypoints are compressed.

Return type bool

compress_keypoints_by_mask()

If a key contains 'keypoints', and f'{key}_mask' is in self.keys(), invalid zeros will be removed and f'{key}_mask' will be locked.

Raises **KeyError** – A key contains 'keypoints' has been found but its corresponding mask is missing.

decompress_keypoints()

If a key contains 'keypoints', and f'{key}_mask' is in self.keys(), invalid zeros will be inserted to the right places and f'{key}_mask' will be unlocked.

Raises **KeyError** – A key contains 'keypoints' has been found but its corresponding mask is missing.

dump(*npz_path: str*, *overwrite: bool = True*)
Dump keys and items to an npz file.

Parameters

- **npz_path** (*str*) – Path to a dumped npz file.
- **overwrite** (*bool, optional*) – Whether to overwrite if there is already a file. Defaults to True.

Raises

- **ValueError** – npz_path does not end with '.npz'.
- **FileExistsError** – When overwrite is False and file exists.

dump_by_pickle(*pkl_path: str*, *overwrite: bool = True*)
Dump keys and items to a pickle file. It's a secondary dump method, when a HumanData instance is too large to be dumped by self.dump()

Parameters

- **pkl_path** (*str*) – Path to a dumped pickle file.
- **overwrite** (*bool, optional*) – Whether to overwrite if there is already a file. Defaults to True.

Raises

- **ValueError** – npz_path does not end with '.pkl'.
- **FileExistsError** – When overwrite is False and file exists.

classmethod fromfile(*npz_path: str*)
Construct a HumanData instance from an npz file.

Parameters **npz_path** (*str*) – Path to a dumped npz file.

Returns A HumanData instance load from file.

Return type *HumanData*

get_key_strict() → bool
Get value of attribute key_strict.

Returns Whether to raise error when setting unsupported keys.

Return type bool

get_raw_value(*key: mmhuman3d.data.data_structures.human_data._KT*) → mmhuman3d.data.data_structures.human_data._VT
Get raw value from the dict. It acts the same as dict.__getitem__(k).

Parameters **key** (*_KT*) – Key in dict.

Returns Value to the key.

Return type _VT

get_temporal_slice(*stop: int*)
get_temporal_slice(*start: int*, *stop: int*)
get_temporal_slice(*start: int*, *stop: int*, *step: int*)
Slice all temporal values along timeline dimension.

Parameters

- **arg_0** (*int*) – When arg_1 is None, arg_0 is stop and start=0. When arg_1 is not None, arg_0 is start.
- **arg_1** (*Union[int, Any], optional*) – None or where to stop. Defaults to None.
- **step** (*int, optional*) – Length of step. Defaults to 1.

Returns A new HumanData instance with sliced values.

Return type *HumanData*

get_value_in_shape(*key: mmhuman3d.data.data_structures.human_data._KT*, *shape: Union[list, tuple]*, *padding_constant: int = 0*) → numpy.ndarray

Get value in a specific shape. For each dim, if the required shape is smaller than current shape, ndarray will be sliced. Otherwise, it will be padded with padding_constant at the end.

Parameters

- **key** (*_KT*) – Key in dict. The value of this key must be an instance of numpy.ndarray.
- **shape** (*Union[list, tuple]*) – Shape of the returned array. Its length must be equal to value.ndim. Set -1 for a dimension if you do not want to edit it.
- **padding_constant** (*int, optional*) – The value to set the padded values for each axis. Defaults to 0.

Raises **ValueError** – A value in shape is neither positive integer nor -1.

Returns An array in required shape.

Return type np.ndarray

load(*npz_path: str*)

Load data from npz_path and update them to self.

Parameters **npz_path** (*str*) – Path to a dumped npz file.

load_by_pickle(*pkl_path: str*)

Load data from pkl_path and update them to self.

When a HumanData Instance was dumped by self.dump_by_pickle(), use this to load. :param npz_path: Path to a dumped npz file. :type npz_path: str

classmethod new(*source_dict: Optional[dict] = None*, *key_strict: bool = False*)

Construct a HumanData instance from a dict.

Parameters

- **source_dict** (*dict, optional*) – A dict with items in HumanData fashion. Defaults to None.
- **key_strict** (*bool, optional*) – Whether to raise error when setting unsupported keys. Defaults to False.

Returns A HumanData instance.

Return type *HumanData*

pop_unsupported_items()

Find every item with a key not in HumanData.SUPPORTED_KEYS, and pop it to save memory.

set_key_strict(*value: bool*)

Set value of attribute key_strict.

Parameters **value** (*bool, optional*) – Whether to raise error when setting unsupported keys. Defaults to True.

classmethod set_logger(*logger: Optional[Union[logging.Logger, str]] = None*)
Set logger of HumanData class.

Parameters **logger** (*logging.Logger | str | None, optional*) – The way to print summary. See *mmcv.utils.print_log()* for details. Defaults to None.

set_raw_value(*key: mmhuman3d.data.data_structures.human_data._KT*, *val: mmhuman3d.data.data_structures.human_data._VT*) → None
Set the raw value of self[key] to val after key check. It acts the same as dict.__setitem__(self, key, val) if the key satisfied constraints.

Parameters

- **key** (*_KT*) – Key in dict.
- **val** (*_VT*) – Value to the key.

Raises

- **KeyError** – self.get_key_strict() is True and key cannot be found in HumanData.SUPPORTED_KEYS.
- **ValueError** – Value is supported but doesn't match definition.

property temporal_len: int
Get the temporal length of this HumanData instance.

Returns Number of frames related to this instance.

Return type int

to(*device: Optional[Union[torch.device, str]] = device(type='cpu')*, *dtype: Optional[torch.dtype] = None*, *non_blocking: Optional[bool] = False*, *copy: Optional[bool] = False*, *memory_format: Optional[torch.memory_format] = None*) → dict
Convert values in numpy.ndarray type to torch.Tensor, and move Tensors to the target device. All keys will exist in the returned dict.

Parameters

- **device** (*Union[torch.device, str], optional*) – A specified device. Defaults to CPU_DEVICE.
- **dtype** (*torch.dtype, optional*) – The data type of the expected torch.Tensor. If dtype is None, it is decided according to numpy.ndarry. Defaults to None.
- **non_blocking** (*bool, optional*) – When non_blocking, tries to convert asynchronously with respect to the host if possible, e.g., converting a CPU Tensor with pinned memory to a CUDA Tensor. Defaults to False.
- **copy** (*bool, optional*) – When copy is set, a new Tensor is created even when the Tensor already matches the desired conversion. No matter what value copy is, Tensor constructed from numpy will not share the same memory with the source numpy.ndarray. Defaults to False.
- **memory_format** (*torch.memory_format, optional*) – The desired memory format of returned Tensor. Not supported by pytorch-cpu. Defaults to None.

Returns A dict with all numpy.ndarray values converted into torch.Tensor and all Tensors moved to the target device.

Return type dict

CHAPTER

SIXTEEN

MMHUMAN3D.UTILS

class mmhuman3d.utils.**DistOptimizerHook**(*grad_clip=None*, *coalesce=True*, *bucket_size_mb=- 1*)

class mmhuman3d.utils.**Existence**(*value*)

State of file existence.

mmhuman3d.utils.**aa_to_ee**(*axis_angle: Union[torch.Tensor, numpy.ndarray]*, *convention: str = 'xyz'*) → Union[torch.Tensor, numpy.ndarray]

Convert axis angles to euler angle.

Parameters

- **axis_angle** (*Union[torch.Tensor, numpy.ndarray]*) – input shape should be (..., 3). ndim of input is unlimited.
- **convention** (*str, optional*) – Convention string of three letters from {"x", "y", and "z"}. Defaults to 'xyz'.

Returns shape would be (..., 3).

Return type Union[torch.Tensor, numpy.ndarray]

mmhuman3d.utils.**aa_to_quat**(*axis_angle: Union[torch.Tensor, numpy.ndarray]*) → Union[torch.Tensor, numpy.ndarray]

Convert axis_angle to quaternions. :param axis_angle: input shape

should be (..., 3). ndim of input is unlimited.

Returns shape would be (..., 4).

Return type Union[torch.Tensor, numpy.ndarray]

mmhuman3d.utils.**aa_to_rot6d**(*axis_angle: Union[torch.Tensor, numpy.ndarray]*) → Union[torch.Tensor, numpy.ndarray]

Convert axis angles to rotation 6d representations.

Parameters **axis_angle** (*Union[torch.Tensor, numpy.ndarray]*) – input shape should be (..., 3). ndim of input is unlimited.

Returns shape would be (..., 6).

Return type Union[torch.Tensor, numpy.ndarray]

[1] Zhou, Y., Barnes, C., Lu, J., Yang, J., & Li, H. On the Continuity of Rotation Representations in Neural Networks. IEEE Conference on Computer Vision and Pattern Recognition, 2019. Retrieved from http://arxiv.org/abs/1812.07035

mmhuman3d.utils.**aa_to_rotmat**(*axis_angle: Union[torch.Tensor, numpy.ndarray]*) → Union[torch.Tensor, numpy.ndarray]

Convert axis_angle to rotation matrixs. :param axis_angle: input shape

should be (..., 3). ndim of input is unlimited.

Returns shape would be (..., 3, 3).

Return type Union[torch.Tensor, numpy.ndarray]

mmhuman3d.utils.**aa_to_sja**(*axis_angle: Union[torch.Tensor, numpy.ndarray], R_t: Union[torch.Tensor, numpy.ndarray] = tensor([[[1.0, 0.0, 0.0], [0.0, 0.0, 1.0], [0.0, - 1.0, 0.0]], [[1.0, 0.0, 0.0], [0.0, 0.0, 1.0], [0.0, - 1.0, 0.0]], [[1.0, 0.0, 0.0], [0.0, 0.0, - 1.0], [0.0, 1.0, 0.0]], [[1.0, 0.0, 0.0], [0.0, 0.0, 1.0], [0.0, - 1.0, 0.0]], [[1.0, 0.0, 0.0], [0.0, 0.0, 1.0], [0.0, - 1.0, 0.0]], [[1.0, 0.0, 0.0], [0.0, 0.0, - 1.0], [0.0, 1.0, 0.0]], [[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]], [[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]], [[1.0, 0.0, 0.0], [0.0, 0.0, - 1.0], [0.0, 1.0, 0.0]], [[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]], [[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]], [[1.0, 0.0, 0.0], [0.0, 0.0, - 1.0], [0.0, 1.0, 0.0]], [[0.0, 0.0, - 1.0], [0.0, 1.0, 0.0], [1.0, 0.0, 0.0]], [[0.0, 0.0, 1.0], [0.0, 1.0, 0.0], [- 1.0, 0.0, 0.0]], [[1.0, 0.0, 0.0], [0.0, 0.0, - 1.0], [0.0, 1.0, 0.0]], [[0.0, 0.0, - 1.0], [0.0, 1.0, 0.0], [1.0, 0.0, 0.0]], [[0.0, 0.0, 1.0], [0.0, 1.0, 0.0], [- 1.0, 0.0, 0.0]], [[0.0, 0.0, - 1.0], [0.0, 1.0, 0.0], [1.0, 0.0, 0.0]], [[0.0, 0.0, 1.0], [0.0, 1.0, 0.0], [- 1.0, 0.0, 0.0]], [[0.0, 0.0, - 1.0], [0.0, 1.0, 0.0], [1.0, 0.0, 0.0]], [[0.0, 0.0, 1.0], [0.0, 1.0, 0.0], [- 1.0, 0.0, 0.0]]]), R_t_inv: Union[torch.Tensor, numpy.ndarray] = tensor([[[1.0, - 0.0, 0.0], [0.0, 0.0, - 1.0], [0.0, 1.0, 0.0]], [[1.0, - 0.0, 0.0], [0.0, 0.0, - 1.0], [0.0, 1.0, 0.0]], [[1.0, 0.0, - 0.0], [0.0, 0.0, 1.0], [0.0, - 1.0, 0.0]], [[1.0, - 0.0, 0.0], [0.0, 0.0, - 1.0], [0.0, 1.0, 0.0]], [[1.0, - 0.0, 0.0], [0.0, 0.0, - 1.0], [0.0, 1.0, 0.0]], [[1.0, 0.0, - 0.0], [0.0, 0.0, 1.0], [0.0, - 1.0, 0.0]], [[1.0, 0.0, - 0.0], [0.0, 1.0, - 0.0], [0.0, 0.0, 1.0]], [[1.0, 0.0, - 0.0], [0.0, 1.0, - 0.0], [0.0, 0.0, 1.0]], [[1.0, 0.0, - 0.0], [0.0, 0.0, 1.0], [0.0, - 1.0, 0.0]], [[1.0, 0.0, - 0.0], [0.0, 1.0, - 0.0], [0.0, 0.0, 1.0]], [[1.0, 0.0, - 0.0], [0.0, 1.0, - 0.0], [0.0, 0.0, 1.0]], [[1.0, 0.0, - 0.0], [0.0, 0.0, 1.0], [0.0, - 1.0, 0.0]], [[0.0, - 0.0, 1.0], [0.0, 1.0, 0.0], [- 1.0, 0.0, 0.0]], [[- 0.0, 0.0, - 1.0], [- 0.0, 1.0, 0.0], [1.0, 0.0, 0.0]], [[1.0, 0.0, - 0.0], [0.0, 0.0, 1.0], [0.0, - 1.0, 0.0]], [[0.0, - 0.0, 1.0], [0.0, 1.0, 0.0], [- 1.0, 0.0, 0.0]], [[- 0.0, 0.0, - 1.0], [- 0.0, 1.0, 0.0], [1.0, 0.0, 0.0]], [[0.0, - 0.0, 1.0], [0.0, 1.0, 0.0], [- 1.0, 0.0, 0.0]], [[- 0.0, 0.0, - 1.0], [- 0.0, 1.0, 0.0], [1.0, 0.0, 0.0]], [[0.0, - 0.0, 1.0], [0.0, 1.0, 0.0], [- 1.0, 0.0, 0.0]], [[- 0.0, 0.0, - 1.0], [- 0.0, 1.0, 0.0], [1.0, 0.0, 0.0]]])*) → Union[torch.Tensor, numpy.ndarray]

Convert axis-angles to standard joint angles.

Parameters

- **axis_angle** (*Union[torch.Tensor, numpy.ndarray]*) – input shape should be (..., 21, 3), ndim of input is unlimited.
- **R_t** (*Union[torch.Tensor, numpy.ndarray]*) – input shape should be (..., 21, 3, 3). Transformation matrices from original axis-angle coordinate system to standard joint angle coordinate system, ndim of input is unlimited.
- **R_t_inv** (*Union[torch.Tensor, numpy.ndarray]*) – input shape should be (..., 21, 3, 3). Transformation matrices from standard joint angle coordinate system to original axis-angle coordinate system, ndim of input is unlimited.

Returns shape would be (..., 3).

Return type Union[torch.Tensor, numpy.ndarray]

mmhuman3d.utils.**array_to_images**(*image_array: numpy.ndarray*, *output_folder: str*, *img_format: str = '%06d.png'*, *resolution: Optional[Union[Tuple[int, int], Tuple[float, float]]] = None*, *disable_log: bool = False*) → None

Convert an array to images directly.

Parameters

- **image_array** (*np.ndarray*) – shape should be (f * h * w * 3).
- **output_folder** (*str*) – output folder for the images.
- **img_format** (*str, optional*) – format of the images. Defaults to '%06d.png'.
- **(Optional[Union[Tuple[int** (*resolution*) – optional): resolution(height, width) of output. Defaults to None.
- **int]** – optional): resolution(height, width) of output. Defaults to None.
- **Tuple[float** – optional): resolution(height, width) of output. Defaults to None.
- **float]]]** – optional): resolution(height, width) of output. Defaults to None.

:param [optional): resolution(height, width) of output.] Defaults to None.

Parameters **disable_log** (*bool, optional*) – whether close the ffmepg command info. Defaults to False.

Raises

- **FileNotFoundError** – check output folder.
- **TypeError** – check input array.

Returns None

mmhuman3d.utils.**array_to_video**(*image_array: numpy.ndarray*, *output_path: str*, *fps: Union[int, float] = 30*, *resolution: Optional[Union[Tuple[int, int], Tuple[float, float]]] = None*, *disable_log: bool = False*) → None

Convert an array to a video directly, gif not supported.

Parameters

- **image_array** (*np.ndarray*) – shape should be (f * h * w * 3).
- **output_path** (*str*) – output video file path.
- **fps** (*Union[int, float, optional*) – fps. Defaults to 30.
- **(Optional[Union[Tuple[int** (*resolution*) – optional): (height, width) of the output video. Defaults to None.
- **int]** – optional): (height, width) of the output video. Defaults to None.
- **Tuple[float** – optional): (height, width) of the output video. Defaults to None.
- **float]]]** – optional): (height, width) of the output video. Defaults to None.

:param [optional): (height, width) of the output video.] Defaults to None.

Parameters **disable_log** (*bool, optional*) – whether close the ffmepg command info. Defaults to False.

Raises

- **FileNotFoundError** – check output path.

- **TypeError** – check input array.

Returns None.

mmhuman3d.utils.**batch_rodrigues**(*theta*)
Convert axis-angle representation to rotation matrix.

Parameters **theta** – size = [B, 3]

Returns Rotation matrix corresponding to the quaternion – size = [B, 3, 3]

mmhuman3d.utils.**box2cs**(*bbox_xywh*, *aspect_ratio=1.0*, *bbox_scale_factor=1.25*)
Convert xywh coordinates to center and scale.

Args: bbox_xywh (numpy.ndarray): the height of the bbox_xywh aspect_ratio (int, optional): Defaults to 1.0 bbox_scale_factor (float, optional): Defaults to 1.25 :returns: center of the bbox

numpy.ndarray: the scale of the bbox w & h

Return type numpy.ndarray

mmhuman3d.utils.**check_input_path**(*input_path: str*, *allowed_suffix: List[str] = []*, *tag: str = 'input file'*, *path_type: typing_extensions.Literal[file, dir, auto] = 'auto'*)
Check input folder or file.

Parameters

- **input_path** (*str*) – input folder or file path.
- **allowed_suffix** (*List[str], optional*) – Check the suffix of *input_path*. If folder, should be [] or ['']. If could both be folder or file, should be [suffixs..., '']. Defaults to [].
- **tag** (*str, optional*) – The *string* tag to specify the output type. Defaults to 'output file'.
- **path_type** (*Literal[, optional*) – Choose *file* for file and *directory* for folder. Choose *auto* if allowed to be both. Defaults to 'auto'.

Raises **FileNotFoundError** – file does not exists or suffix does not match.

Returns None

mmhuman3d.utils.**check_path_existence**(*path_str: str*, *path_type: typing_extensions.Literal[file, dir, auto] = 'auto'*) → *mmhuman3d.utils.path_utils.Existence*
Check whether a file or a directory exists at the expected path.

Parameters

- **path_str** (*str*) – Path to check.
- **path_type** (*Literal[, optional*) – What kind of file do we expect at the path. Choose among *file*, *dir*, *auto*. Defaults to 'auto'. path_type = path_type.lower()

Raises **KeyError** – if *path_type* conflicts with *path_str*

Returns

0. FileExist: file at path_str exists.
1. DirectoryExistEmpty: folder at path exists and.
2. DirectoryExistNotEmpty: folder at path_str exists and not empty.
3. MissingParent: its parent doesn't exist.

4. DirectoryNotExist: expect a folder at path_str, but not found.
5. FileNotExist: expect a file at path_str, but not found.

Return type *Existence*

mmhuman3d.utils.**check_path_suffix**(*path_str: str*, *allowed_suffix: Union[str, List[str]] = ''*) → bool
Check whether the suffix of the path is allowed.

Parameters

- **path_str** (*str*) – Path to check.
- **allowed_suffix** (*List[str], optional*) – What extension names are allowed. Offer a list like ['.jpg', '.jpeg']. When it's [], all will be received. Use [''] then directory is allowed. Defaults to [].

Returns True: suffix test passed False: suffix test failed

Return type bool

mmhuman3d.utils.**collect_env**()
Collect the information of the running environments.

mmhuman3d.utils.**compress_video**(*input_path: str*, *output_path: str*, *compress_rate: int = 1*, *down_sample_scale: Union[float, int] = 1*, *fps: int = 30*, *disable_log: bool = False*) → None
Compress a video file.

Parameters

- **input_path** (*str*) – input video file path.
- **output_path** (*str*) – output video file path.
- **compress_rate** (*int, optional*) – compress rate, influents the bit rate. Defaults to 1.
- **down_sample_scale** (*Union[float, int], optional*) – spatial down sample scale. Defaults to 1.
- **fps** (*int, optional*) – Frames per second. Defaults to 30.
- **disable_log** (*bool, optional*) – whether close the ffmepg command info. Defaults to False.

Raises

- **FileNotFoundError** – check the input path.
- **FileNotFoundError** – check the output path.

Returns None.

mmhuman3d.utils.**conver_verts_to_cam_coord**(*verts*, *pred_cams*, *bboxes_xy*, *focal_length=5000.0*, *bbox_scale_factor=1.25*, *bbox_format='xyxy'*)
Convert vertices from the world coordinate to camera coordinate.

Parameters

- **verts** (*[np.ndarray]*) – The vertices in the world coordinate. The shape is (frame,num_person,6890,3) or (frame,6890,3).
- **pred_cams** (*[np.ndarray]*) – Camera parameters estimated by HMR or SPIN. The shape is (frame,num_person,3) or (frame,6890,3).
- **bboxes_xy** (*[np.ndarray]*) – (frame, num_person, 4|5) or (frame, 4|5)

- **focal_length** (*[float],optional*) – Defined same as your training.
- **bbox_scale_factor** (*float*) – scale factor for expanding the bbox.
- **bbox_format** (*Literal['xyxy', 'xywh']*) – 'xyxy' means the left-up point and right-bottomn point of the bbox. 'xywh' means the left-up point and the width and height of the bbox.

Returns

The vertices in the camera coordinate. The shape is (frame,num_person,6890,3) or (frame,6890,3).

np.ndarray: The intrinsic parameters of the pred_cam. The shape is (num_frame, 3, 3).

Return type np.ndarray

mmhuman3d.utils.**convert_bbox_to_intrinsic**(*bboxes: numpy.ndarray*, *img_width: int = 224*, *img_height: int = 224*, *bbox_scale_factor: float = 1.25*, *bbox_format: typing_extensions.Literal[xyxy, xywh] = 'xyxy'*)

Convert bbox to intrinsic parameters.

Parameters

- **bbox** (*np.ndarray*) – (frame, num_person, 4) or (frame, 4)
- **img_width** (*int*) – image width of training data.
- **img_height** (*int*) – image height of training data.
- **bbox_scale_factor** (*float*) – scale factor for expanding the bbox.
- **bbox_format** (*Literal['xyxy', 'xywh']*) – 'xyxy' means the left-up point and right-bottomn point of the bbox. 'xywh' means the left-up point and the width and height of the bbox.

Returns (frame, num_person, 3, 3) or (frame, 3, 3)

Return type np.ndarray

mmhuman3d.utils.**convert_crop_cam_to_orig_img**(*cam: numpy.ndarray*, *bbox: numpy.ndarray*, *img_width: int*, *img_height: int*, *aspect_ratio: float = 1.0*, *bbox_scale_factor: float = 1.25*, *bbox_format: typing_extensions.Literal[xyxy, xywh, cs] = 'xyxy'*)

This function is modified from [VIBE](https://github.com/ mkocabas/VIBE/blob/master/lib/utils/demo_utils.py#L242-L259). Original license please see docs/additional_licenses.md.

Parameters

- **cam** (*np.ndarray*) – cam (ndarray, shape=(frame, 3) or
- **(frame** –
- **num_person** –
- **3))** –
- **perspective camera in cropped img coordinates** (*weak*) –
- **bbox** (*np.ndarray*) – bbox coordinates
- **img_width** (*int*) – original image width
- **img_height** (*int*) – original image height
- **aspect_ratio** (*float, optional*) – Defaults to 1.0.

- **bbox_scale_factor** (*float, optional*) – Defaults to 1.25.
- **bbox_format** (*Literal['xyxy', 'xywh', 'cs']*) – Defaults to 'xyxy'. 'xyxy' means the left-up point and right-bottomn point of the bbox. 'xywh' means the left-up point and the width and height of the bbox. 'cs' means the center of the bbox (x,y) and the scale of the bbox w & h.

Returns shape = (frame, 4) or (frame, num_person, 4)

Return type orig_cam

mmhuman3d.utils.**convert_kp2d_to_bbox**(*kp2d: numpy.ndarray*, *bbox_format: typing_extensions.Literal[xyxy, xywh] = 'xyxy'*) → numpy.ndarray

Convert kp2d to bbox.

Parameters

- **kp2d** (*np.ndarray*) – shape should be (num_frame, num_points, 2/3) or (num_frame, num_person, num_points, 2/3).
- **bbox_format** (*Literal['xyxy', 'xywh'], optional*) – Defaults to 'xyxy'.

Returns shape will be (num_frame, num_person, 4)

Return type np.ndarray

mmhuman3d.utils.**crop_video**(*input_path: str*, *output_path: str*, *box: Optional[Union[List[int], Tuple[int, int, int, int]]] = None*, *resolution: Optional[Union[Tuple[int, int], Tuple[float, float]]] = None*, *disable_log: bool = False*) → None

Spatially or temporally crop a video or gif file.

Parameters

- **input_path** (*str*) – input video or gif file path.
- **output_path** (*str*) – output video or gif file path.
- **box** (*Iterable[int], optional*) – [x, y of the crop region left. corner and width and height]. Defaults to [0, 0, 100, 100].
- **(Optional[Union[Tuple[int** (*resolution*) – optional): (height, width) of output. Defaults to None.
- **int]** – optional): (height, width) of output. Defaults to None.
- **Tuple[float** – optional): (height, width) of output. Defaults to None.
- **float]]]** – optional): (height, width) of output. Defaults to None.

:param : optional): (height, width) of output. Defaults to None. :param disable_log: whether close the ffmepg command info.

Defaults to False.

Raises

- **FileNotFoundError** – check the input path.
- **FileNotFoundError** – check the output path.

Returns None'-start_number', f'{start}',

mmhuman3d.utils.**ee_to_aa**(*euler_angle: Union[torch.Tensor, numpy.ndarray]*, *convention: str = 'xyz'*) → Union[torch.Tensor, numpy.ndarray]

Convert euler angles to axis angles.

Parameters

- **euler_angle** (*Union[torch.Tensor, numpy.ndarray]*) – input shape should be (..., 3). ndim of input is unlimited.
- **convention** (*str, optional*) – Convention string of three letters from {"x", "y", and "z"}. Defaults to 'xyz'.

Returns shape would be (..., 3).

Return type Union[torch.Tensor, numpy.ndarray]

mmhuman3d.utils.**ee_to_quat**(*euler_angle: Union[torch.Tensor, numpy.ndarray]*, *convention='xyz'*) → Union[torch.Tensor, numpy.ndarray]

Convert euler angles to quaternions.

Parameters

- **euler_angle** (*Union[torch.Tensor, numpy.ndarray]*) – input shape should be (..., 3). ndim of input is unlimited.
- **convention** (*str, optional*) – Convention string of three letters from {"x", "y", and "z"}. Defaults to 'xyz'.

Returns shape would be (..., 4).

Return type Union[torch.Tensor, numpy.ndarray]

mmhuman3d.utils.**ee_to_rot6d**(*euler_angle: Union[torch.Tensor, numpy.ndarray]*, *convention='xyz'*) → Union[torch.Tensor, numpy.ndarray]

Convert euler angles to rotation 6d representation.

Parameters

- **euler_angle** (*Union[torch.Tensor, numpy.ndarray]*) – input shape should be (..., 3). ndim of input is unlimited.
- **convention** (*str, optional*) – Convention string of three letters from {"x", "y", and "z"}. Defaults to 'xyz'.

Returns shape would be (..., 6).

Return type Union[torch.Tensor, numpy.ndarray]

[1] Zhou, Y., Barnes, C., Lu, J., Yang, J., & Li, H. On the Continuity of Rotation Representations in Neural Networks. IEEE Conference on Computer Vision and Pattern Recognition, 2019. Retrieved from http://arxiv.org/abs/1812.07035

mmhuman3d.utils.**ee_to_rotmat**(*euler_angle: Union[torch.Tensor, numpy.ndarray]*, *convention='xyz'*) → Union[torch.Tensor, numpy.ndarray]

Convert euler angle to rotation matrixs.

Parameters

- **euler_angle** (*Union[torch.Tensor, numpy.ndarray]*) – input shape should be (..., 3). ndim of input is unlimited.
- **convention** (*str, optional*) – Convention string of three letters from {"x", "y", and "z"}. Defaults to 'xyz'.

Returns shape would be (..., 3, 3).

Return type Union[torch.Tensor, numpy.ndarray]

mmhuman3d.utils.**estimate_translation**(*S*, *joints_2d*, *focal_length=5000.0*, *img_size=224.0*)

Find camera translation that brings 3D joints S closest to 2D the corresponding joints_2d.

Input: S: (B, 49, 3) 3D joint locations joints: (B, 49, 3) 2D joint locations and confidence

Returns (B, 3) camera translation vectors

mmhuman3d.utils.**estimate_translation_np**(*S*, *joints_2d*, *joints_conf*, *focal_length=5000*, *img_size=224*)
Find camera translation that brings 3D joints S closest to 2D the corresponding joints_2d.

Input: S: (25, 3) 3D joint locations joints: (25, 3) 2D joint locations and confidence

Returns (3,) camera translation vector

mmhuman3d.utils.**get_default_hmr_intrinsic**(*num_frame=1*, *focal_length=1000*, *det_width=224*, *det_height=224*) → numpy.ndarray
Get default hmr intrinsic, defined by how you trained.

Parameters

- **num_frame** (*int, optional*) – num of frames. Defaults to 1.
- **focal_length** (*int, optional*) – defined same as your training. Defaults to 1000.
- **det_width** (*int, optional*) – the size you used to detect. Defaults to 224.
- **det_height** (*int, optional*) – the size you used to detect. Defaults to 224.

Returns shape of (N, 3, 3)

Return type np.ndarray

mmhuman3d.utils.**get_different_colors**(*number_of_colors*, *flag=0*, *alpha: float = 1.0*, *mode: str = 'bgr'*, *int_dtype: bool = True*)
Get a numpy of colors of shape (N, 3).

mmhuman3d.utils.**gif_to_images**(*input_path: str*, *output_folder: str*, *fps: int = 30*, *img_format: str = '%06d.png'*, *resolution: Optional[Union[Tuple[int, int], Tuple[float, float]]] = None*, *disable_log: bool = False*) → None
Convert a gif file to a folder of images.

Parameters

- **input_path** (*str*) – input gif file path.
- **output_folder** (*str*) – output folder to save the images.
- **fps** (*int, optional*) – fps. Defaults to 30.
- **img_format** (*str, optional*) – output image name format. Defaults to '%06d.png'.
- **(Optional[Union[Tuple[int** (*resolution*) – optional): (height, width) of output. Defaults to None.
- **int]** – optional): (height, width) of output. Defaults to None.
- **Tuple[float** – optional): (height, width) of output. Defaults to None.
- **float]]]** – optional): (height, width) of output. Defaults to None.

:param [optional): (height, width) of output.] Defaults to None.

Parameters **disable_log** (*bool, optional*) – whether close the ffmepg command info. Defaults to False.

Raises

- **FileNotFoundError** – check the input path.
- **FileNotFoundError** – check the output path.

Returns None

mmhuman3d.utils.**gif_to_video**(*input_path: str*, *output_path: str*, *fps: int = 30*, *remove_raw_file: bool = False*, *resolution: Optional[Union[Tuple[int, int], Tuple[float, float]]] = None*, *disable_log: bool = False*) → None

Convert a gif file to a video.

Parameters

- **input_path** (*str*) – input gif file path.
- **output_path** (*str*) – output video file path.
- **fps** (*int, optional*) – fps. Defaults to 30.
- **remove_raw_file** (*bool, optional*) – whether remove original input file. Defaults to False.
- **down_sample_scale** (*Union[int, float], optional*) – down sample scale. Defaults to 1.
- **(Optional[Union[Tuple[int** (*resolution*) – optional): (height, width) of output. Defaults to None.
- **int]** – optional): (height, width) of output. Defaults to None.
- **Tuple[float** – optional): (height, width) of output. Defaults to None.
- **float]]]** – optional): (height, width) of output. Defaults to None.

:param : optional): (height, width) of output. Defaults to None. :param disable_log: whether close the ffmepg command info.

Defaults to False.

Raises

- **FileNotFoundError** – check the input path.
- **FileNotFoundError** – check the output path.

Returns None

mmhuman3d.utils.**images_to_array**(*input_folder: str*, *resolution: Optional[Union[Tuple[int, int], Tuple[float, float]]] = None*, *img_format: str = '%06d.png'*, *start: int = 0*, *end: Optional[int] = None*, *remove_raw_files: bool = False*, *disable_log: bool = False*) → numpy.ndarray

Read a folder of images as an array of (f * h * w * 3).

Parameters

- **input_folder** (*str*) – folder of input images.
- **(Optional[Union[Tuple[int** (*resolution*) – resolution(height, width) of output. Defaults to None.
- **int]** – resolution(height, width) of output. Defaults to None.
- **Tuple[float** – resolution(height, width) of output. Defaults to None.
- **float]]]** – resolution(height, width) of output. Defaults to None.

- **img_format** (*str, optional*) – format of images to be read. Defaults to '%06d.png'.
- **start** (*int, optional*) –

  start frame index. Inclusive. If < 0, will be converted to frame_index range in [0, frame_num].

  Defaults to 0.
- **end** (*int, optional*) – end frame index. Exclusive. Could be positive int or negative int or None. If None, all frames from start till the last frame are included. Defaults to None.
- **remove_raw_files** (*bool, optional*) – whether remove raw images. Defaults to False.
- **disable_log** (*bool, optional*) – whether close the ffmepg command info. Defaults to False.

Raises **FileNotFoundError** – check the input path.

Returns shape will be (f * h * w * 3).

Return type np.ndarray

mmhuman3d.utils.**images_to_gif**(*input_folder: str*, *output_path: str*, *remove_raw_file: bool = False*, *img_format: str = '%06d.png'*, *fps: int = 15*, *resolution: Optional[Union[Tuple[int, int], Tuple[float, float]]] = None*, *start: int = 0*, *end: Optional[int] = None*, *disable_log: bool = False*) → None

Convert series of images to a video, similar to images_to_video, but provide more suitable parameters.

Parameters

- **input_folder** (*str*) – input image folder.
- **output_path** (*str*) – output gif file path.
- **remove_raw_file** (*bool, optional*) – whether remove raw images. Defaults to False.
- **img_format** (*str, optional*) – format to name the images. Defaults to '%06d.png'.
- **fps** (*int, optional*) – output video fps. Defaults to 15.
- **(Optional[Union[Tuple[int** (*resolution*) – optional): (height, width) of output. Defaults to None.
- **int]** – optional): (height, width) of output. Defaults to None.
- **Tuple[float** – optional): (height, width) of output. Defaults to None.
- **float]]]** – optional): (height, width) of output. Defaults to None.

:param : optional): (height, width) of output. Defaults to None. :param start: start frame index. Inclusive.

If < 0, will be converted to frame_index range in [0, frame_num]. Defaults to 0.

Parameters

- **end** (*int, optional*) – end frame index. Exclusive. Could be positive int or negative int or None. If None, all frames from start till the last frame are included. Defaults to None.
- **disable_log** (*bool, optional*) – whether close the ffmepg command info. Defaults to False.

Raises

- **FileNotFoundError** – check the input path.
- **FileNotFoundError** – check the output path.

Returns None

mmhuman3d.utils.**images_to_sorted_images**(*input_folder*, *output_folder*, *img_format='%06d'*)

Copy and rename a folder of images into a new folder following the *img_format*.

Parameters

- **input_folder** (*str*) – input folder.
- **output_folder** (*str*) – output folder.
- **img_format** (*str, optional*) – image format name, do not need extension. Defaults to '%06d'.

Returns image format of the rename images.

Return type str

mmhuman3d.utils.**images_to_video**(*input_folder: str*, *output_path: str*, *remove_raw_file: bool = False*, *img_format: str = '%06d.png'*, *fps: Union[int, float] = 30*, *resolution: Optional[Union[Tuple[int, int], Tuple[float, float]]] = None*, *start: int = 0*, *end: Optional[int] = None*, *disable_log: bool = False*) → None

Convert a folder of images to a video.

Parameters

- **input_folder** (*str*) – input image folder
- **output_path** (*str*) – output video file path
- **remove_raw_file** (*bool, optional*) – whether remove raw images. Defaults to False.
- **img_format** (*str, optional*) – format to name the images]. Defaults to '%06d.png'.
- **fps** (*Union[int, float], optional*) – output video fps. Defaults to 30.
- **(Optional[Union[Tuple[int** (*resolution*) – optional): (height, width) of output. defaults to None.
- **int]** – optional): (height, width) of output. defaults to None.
- **Tuple[float** – optional): (height, width) of output. defaults to None.
- **float]]]** – optional): (height, width) of output. defaults to None.

:param [optional): (height, width) of output.] defaults to None.

Parameters

- **start** (*int, optional*) – start frame index. Inclusive. If < 0, will be converted to frame_index range in [0, frame_num]. Defaults to 0.
- **end** (*int, optional*) – end frame index. Exclusive. Could be positive int or negative int or None. If None, all frames from start till the last frame are included. Defaults to None.
- **disable_log** (*bool, optional*) – whether close the ffmepg command info. Defaults to False.

Raises

- **FileNotFoundError** – check the input path.

- **FileNotFoundError** – check the output path.

Returns None

mmhuman3d.utils.**join_batch_meshes_as_scene**(*meshes: List[pytorch3d.structures.Meshes]*, *include_textures: bool = True*) → pytorch3d.structures.Meshes

Join meshes as a scene each batch. Only for pytorch3d meshes. The Meshes must share the same batch size, and arbitrary topology. They must all be on the same device. If include_textures is true, they must all be compatible, either all or none having textures, and all the Textures objects being the same type. If include_textures is False, textures are ignored. If not, ValueError would be raised in join_meshes_as_batch and join_meshes_as_scene.

Parameters

- **meshes** (*List[Meshes]*) – A *list* of *Meshes* with the same batches. Required.
- **include_textures** – (bool) whether to try to join the textures.

Returns New Meshes which has join different Meshes by each batch.

mmhuman3d.utils.**mesh_to_pointcloud_vc**(*meshes: pytorch3d.structures.Meshes*, *include_textures: bool = True*, *alpha: float = 1.0*) → pytorch3d.structures.Pointclouds

Convert pytorch3d *Meshes* to *PointClouds*.

Parameters

- **meshes** (*Meshes*) – input meshes.
- **include_textures** (*bool, optional*) – Whether include colors. Require the texture of input meshes is vertex color. Defaults to True.
- **alpha** (*float, optional*) – transparency. Defaults to 1.0.

Returns output pointclouds.

Return type Pointclouds

mmhuman3d.utils.**pad_for_libx264**(*image_array*)

Pad zeros if width or height of image_array is not divisible by 2. Otherwise you will get.

"[libx264 @ 0x1b1d560] width not divisible by 2 "

Parameters **image_array** (*np.ndarray*) – Image or images load by cv2.imread(). Possible shapes: 1. [height, width] 2. [height, width, channels] 3. [images, height, width] 4. [images, height, width, channels]

Returns A image with both edges divisible by 2.

Return type np.ndarray

mmhuman3d.utils.**perspective_projection**(*points*, *rotation*, *translation*, *focal_length*, *camera_center*)

This function computes the perspective projection of a set of points.

Input: points (bs, N, 3): 3D points rotation (bs, 3, 3): Camera rotation translation (bs, 3): Camera translation focal_length (bs,) or scalar: Focal length camera_center (bs, 2): Camera center

mmhuman3d.utils.**prepare_frames**(*input_path=None*)

Prepare frames from input_path.

Parameters **input_path** (*str, optional*) – Defaults to None.

Raises **ValueError** – check the input path.

Returns prepared frames

Return type List[np.ndarray]

mmhuman3d.utils.**prepare_output_path**(*output_path: str*, *allowed_suffix: List[str] = []*, *tag: str = 'output file'*, *path_type: typing_extensions.Literal[file, dir, auto] = 'auto'*, *overwrite: bool = True*) → None

Check output folder or file.

Parameters

- **output_path** (*str*) – could be folder or file.
- **allowed_suffix** (*List[str], optional*) – Check the suffix of *output_path*. If folder, should be [] or ['']. If could both be folder or file, should be [suffixs..., '']. Defaults to [].
- **tag** (*str, optional*) – The *string* tag to specify the output type. Defaults to 'output file'.
- **path_type** (*Literal[, optional*) – Choose *file* for file and *dir* for folder. Choose *auto* if allowed to be both. Defaults to 'auto'.
- **overwrite** (*bool, optional*) – Whether overwrite the existing file or folder. Defaults to True.

Raises

- **FileNotFoundError** – suffix does not match.
- **FileExistsError** – file or folder already exists and *overwrite* is False.

Returns None

mmhuman3d.utils.**process_mmdet_results**(*mmdet_results*, *cat_id=1*)

Process mmdet results, and return a list of bboxes.

Parameters

- **mmdet_results** (*list|tuple*) – mmdet results.
- **cat_id** (*int*) – category id (default: 1 for human)

Returns a list of detected bounding boxes

Return type person_results (list)

mmhuman3d.utils.**process_mmtracking_results**(*mmtracking_results*, *max_track_id*)

Process mmtracking results.

Parameters **mmtracking_results** (*[list]*) – mmtracking_results.

Returns a list of tracked bounding boxes

Return type list

mmhuman3d.utils.**quat_to_aa**(*quaternions: Union[torch.Tensor, numpy.ndarray]*) → Union[torch.Tensor, numpy.ndarray]

Convert quaternions to axis angles.

Parameters **quaternions** (*Union[torch.Tensor, numpy.ndarray]*) – input shape should be (..., 3). ndim of input is unlimited.

Returns shape would be (..., 3).

Return type Union[torch.Tensor, numpy.ndarray]

mmhuman3d.utils.**quat_to_ee**(*quaternions: Union[torch.Tensor, numpy.ndarray]*, *convention: str = 'xyz'*) → Union[torch.Tensor, numpy.ndarray]

Convert quaternions to euler angles.

Parameters

- **quaternions** (*Union[torch.Tensor, numpy.ndarray]*) – input shape should be (..., 4). ndim of input is unlimited.
- **convention** (*str, optional*) – Convention string of three letters from {"x", "y", and "z"}. Defaults to 'xyz'.

Returns shape would be (..., 3).

Return type Union[torch.Tensor, numpy.ndarray]

mmhuman3d.utils.**quat_to_rot6d**(*quaternions: Union[torch.Tensor, numpy.ndarray]*) → Union[torch.Tensor, numpy.ndarray]

Convert quaternions to rotation 6d representations.

Parameters **quaternions** (*Union[torch.Tensor, numpy.ndarray]*) – input shape should be (..., 4). ndim of input is unlimited.

Returns shape would be (..., 6).

Return type Union[torch.Tensor, numpy.ndarray]

[1] Zhou, Y., Barnes, C., Lu, J., Yang, J., & Li, H. On the Continuity of Rotation Representations in Neural Networks. IEEE Conference on Computer Vision and Pattern Recognition, 2019. Retrieved from http://arxiv.org/abs/1812.07035

mmhuman3d.utils.**quat_to_rotmat**(*quaternions: Union[torch.Tensor, numpy.ndarray]*) → Union[torch.Tensor, numpy.ndarray]

Convert quaternions to rotation matrixs.

Parameters **quaternions** (*Union[torch.Tensor, numpy.ndarray]*) – input shape should be (..., 3). ndim of input is unlimited.

Returns shape would be (..., 3, 3).

Return type Union[torch.Tensor, numpy.ndarray]

mmhuman3d.utils.**quaternion_to_angle_axis**(*quaternion: torch.Tensor*) → torch.Tensor

This function is borrowed from https://github.com/kornia/kornia Convert quaternion vector to angle axis of rotation. Adapted from ceres C++ library: ceres-solver/include/ceres/rotation.h :param quaternion: tensor with quaternions. :type quaternion: torch.Tensor

Returns tensor with angle axis of rotation.

Return type torch.Tensor

Shape:

- Input: $(*, 4)$ where * means, any number of dimensions
- Output: $(*, 3)$

Example

```
>>> quaternion = torch.rand(2, 4)  # Nx4
>>> angle_axis = tgm.quaternion_to_angle_axis(quaternion)  # Nx3
```

mmhuman3d.utils.**rot6d_to_aa**(*rotation_6d: Union[torch.Tensor, numpy.ndarray]*) → Union[torch.Tensor, numpy.ndarray]

Convert rotation 6d representations to axis angles.

Parameters **rotation_6d** (*Union[torch.Tensor, numpy.ndarray]*) – input shape should be (..., 6). ndim of input is unlimited.

Returns shape would be (..., 3).

Return type Union[torch.Tensor, numpy.ndarray]

[1] Zhou, Y., Barnes, C., Lu, J., Yang, J., & Li, H. On the Continuity of Rotation Representations in Neural Networks. IEEE Conference on Computer Vision and Pattern Recognition, 2019. Retrieved from http://arxiv.org/abs/1812.07035

mmhuman3d.utils.**rot6d_to_ee**(*rotation_6d: Union[torch.Tensor, numpy.ndarray]*, *convention: str = 'xyz'*) → Union[torch.Tensor, numpy.ndarray]

Convert rotation 6d representations to euler angles.

Parameters **rotation_6d** (*Union[torch.Tensor, numpy.ndarray]*) – input shape should be (..., 6). ndim of input is unlimited.

Returns shape would be (..., 3).

Return type Union[torch.Tensor, numpy.ndarray]

[1] Zhou, Y., Barnes, C., Lu, J., Yang, J., & Li, H. On the Continuity of Rotation Representations in Neural Networks. IEEE Conference on Computer Vision and Pattern Recognition, 2019. Retrieved from http://arxiv.org/abs/1812.07035

mmhuman3d.utils.**rot6d_to_quat**(*rotation_6d: Union[torch.Tensor, numpy.ndarray]*) → Union[torch.Tensor, numpy.ndarray]

Convert rotation 6d representations to quaternions.

Parameters **rotation** (*Union[torch.Tensor, numpy.ndarray]*) – input shape should be (..., 6). ndim of input is unlimited.

Returns shape would be (..., 4).

Return type Union[torch.Tensor, numpy.ndarray]

[1] Zhou, Y., Barnes, C., Lu, J., Yang, J., & Li, H. On the Continuity of Rotation Representations in Neural Networks. IEEE Conference on Computer Vision and Pattern Recognition, 2019. Retrieved from http://arxiv.org/abs/1812.07035

mmhuman3d.utils.**rot6d_to_rotmat**(*rotation_6d: Union[torch.Tensor, numpy.ndarray]*) → Union[torch.Tensor, numpy.ndarray]

Convert rotation 6d representations to rotation matrixs.

Parameters **rotation_6d** (*Union[torch.Tensor, numpy.ndarray]*) – input shape should be (..., 6). ndim of input is unlimited.

Returns shape would be (..., 3, 3).

Return type Union[torch.Tensor, numpy.ndarray]

[1] Zhou, Y., Barnes, C., Lu, J., Yang, J., & Li, H. On the Continuity of Rotation Representations in Neural Networks. IEEE Conference on Computer Vision and Pattern Recognition, 2019. Retrieved from http://arxiv.org/abs/1812.07035

mmhuman3d.utils.**rotation_matrix_to_angle_axis**(*rotation_matrix*)

This function is borrowed from https://github.com/kornia/kornia Convert 3x4 rotation matrix to Rodrigues vector :param rotation_matrix: rotation matrix. :type rotation_matrix: Tensor

Returns Rodrigues vector transformation.

Return type Tensor

Shape:

- Input: $(N, 3, 4)$
- Output: $(N, 3)$

Example

```
>>> input = torch.rand(2, 3, 4)  # Nx3x4
>>> output = tgm.rotation_matrix_to_angle_axis(input)  # Nx3
```

mmhuman3d.utils.**rotation_matrix_to_quaternion**(*rotation_matrix*, *eps=1e-06*)

This function is borrowed from https://github.com/kornia/kornia Convert 3x4 rotation matrix to 4d quaternion vector This algorithm is based on algorithm described in https://github.com/KieranWynn/pyquaternion/blob/master/pyquaternion/quaternion.py#L201 :param rotation_matrix: the rotation matrix to convert. :type rotation_matrix: Tensor

Returns the rotation in quaternion

Return type Tensor

Shape:

- Input: $(N, 3, 4)$
- Output: $(N, 4)$

Example

```
>>> input = torch.rand(4, 3, 4)  # Nx3x4
>>> output = tgm.rotation_matrix_to_quaternion(input)  # Nx4
```

mmhuman3d.utils.**rotmat_to_aa**(*matrix: Union[torch.Tensor, numpy.ndarray]*) → Union[torch.Tensor, numpy.ndarray]

Convert rotation matrixs to axis angles.

Parameters

- **matrix** (*Union[torch.Tensor, numpy.ndarray]*) – input shape should be (..., 3, 3). ndim of input is unlimited.
- **convention** (*str, optional*) – Convention string of three letters from {“x”, “y”, and “z”}. Defaults to ‘xyz’.

Returns shape would be (..., 3).

Return type Union[torch.Tensor, numpy.ndarray]

mmhuman3d.utils.**rotmat_to_ee**(*matrix: Union[torch.Tensor, numpy.ndarray]*, *convention: str = 'xyz'*) → Union[torch.Tensor, numpy.ndarray]

Convert rotation matrixs to euler angle.

Parameters

- **matrix** (*Union[torch.Tensor, numpy.ndarray]*) – input shape should be (..., 3, 3). ndim of input is unlimited.
- **convention** (*str, optional*) – Convention string of three letters from {"x", "y", and "z"}. Defaults to 'xyz'.

Returns shape would be (..., 3).

Return type Union[torch.Tensor, numpy.ndarray]

mmhuman3d.utils.**rotmat_to_quat**(*matrix: Union[torch.Tensor, numpy.ndarray]*) → Union[torch.Tensor, numpy.ndarray]

Convert rotation matrixs to quaternions.

Parameters **matrix** (*Union[torch.Tensor, numpy.ndarray]*) – input shape should be (..., 3, 3). ndim of input is unlimited.

Returns shape would be (..., 4).

Return type Union[torch.Tensor, numpy.ndarray]

mmhuman3d.utils.**rotmat_to_rot6d**(*matrix: Union[torch.Tensor, numpy.ndarray]*) → Union[torch.Tensor, numpy.ndarray]

Convert rotation matrixs to rotation 6d representations.

Parameters **matrix** (*Union[torch.Tensor, numpy.ndarray]*) – input shape should be (..., 3, 3). ndim of input is unlimited.

Returns shape would be (..., 6).

Return type Union[torch.Tensor, numpy.ndarray]

[1] Zhou, Y., Barnes, C., Lu, J., Yang, J., & Li, H. On the Continuity of Rotation Representations in Neural Networks. IEEE Conference on Computer Vision and Pattern Recognition, 2019. Retrieved from http://arxiv.org/abs/1812.07035

mmhuman3d.utils.**save_meshes_as_objs**(*meshes: Optional[pytorch3d.structures.Meshes] = None*, *paths: List[str] = []*) → None

Save meshes as .obj files. Mainly for uv texture meshes.

Parameters

- **meshes** (*Meshes, optional*) – Defaults to None.
- **paths** (*List[str], optional*) – Output .obj file list. Defaults to [].

mmhuman3d.utils.**save_meshes_as_plys**(*meshes: Optional[pytorch3d.structures.Meshes] = None*, *verts: Optional[torch.Tensor] = None*, *faces: Optional[torch.Tensor] = None*, *verts_rgb: Optional[torch.Tensor] = None*, *paths: List[str] = []*) → None

Save meshes as .ply files. Mainly for vertex color meshes.

Parameters

- **meshes** (*Meshes, optional*) – higher priority than (verts & faces & verts_rgb). Defaults to None.
- **verts** (*torch.Tensor, optional*) – lower priority than meshes. Defaults to None.

- **faces** (*torch.Tensor, optional*) – lower priority than meshes. Defaults to None.
- **verts_rgb** (*torch.Tensor, optional*) – lower priority than meshes. Defaults to None.
- **paths** (*List[str], optional*) – Output .ply file list. Defaults to [].

mmhuman3d.utils.**search_limbs**(*data_source: str*, *mask: Optional[Union[numpy.ndarray, tuple, list]] = None*, *keypoints_factory: dict = {'agora': ['pelvis', 'left_hip', 'right_hip', 'spine_1', 'left_knee', 'right_knee', 'spine_2', 'left_ankle', 'right_ankle', 'spine_3', 'left_foot', 'right_foot', 'neck', 'left_collar', 'right_collar', 'head', 'left_shoulder', 'right_shoulder', 'left_elbow', 'right_elbow', 'left_wrist', 'right_wrist', 'jaw', 'left_eyeball', 'right_eyeball', 'left_index_1', 'left_index_2', 'left_index_3', 'left_middle_1', 'left_middle_2', 'left_middle_3', 'left_pinky_1', 'left_pinky_2', 'left_pinky_3', 'left_ring_1', 'left_ring_2', 'left_ring_3', 'left_thumb_1', 'left_thumb_2', 'left_thumb_3', 'right_index_1', 'right_index_2', 'right_index_3', 'right_middle_1', 'right_middle_2', 'right_middle_3', 'right_pinky_1', 'right_pinky_2', 'right_pinky_3', 'right_ring_1', 'right_ring_2', 'right_ring_3', 'right_thumb_1', 'right_thumb_2', 'right_thumb_3', 'nose', 'right_eye', 'left_eye', 'right_ear', 'left_ear', 'left_bigtoe', 'left_smalltoe', 'left_heel', 'right_bigtoe', 'right_smalltoe', 'right_heel', 'left_thumb', 'left_index', 'left_middle', 'left_ring', 'left_pinky', 'right_thumb', 'right_index', 'right_middle', 'right_ring', 'right_pinky', 'right_eyebrow_1', 'right_eyebrow_2', 'right_eyebrow_3', 'right_eyebrow_4', 'right_eyebrow_5', 'left_eyebrow_5', 'left_eyebrow_4', 'left_eyebrow_3', 'left_eyebrow_2', 'left_eyebrow_1', 'nosebridge_1', 'nosebridge_2', 'nosebridge_3', 'nosebridge_4', 'nose_1', 'nose_2', 'nose_3', 'nose_4', 'nose_5', 'right_eye_1', 'right_eye_2', 'right_eye_3', 'right_eye_4', 'right_eye_5', 'right_eye_6', 'left_eye_4', 'left_eye_3', 'left_eye_2', 'left_eye_1', 'left_eye_6', 'left_eye_5', 'mouth_1', 'mouth_2', 'mouth_3', 'mouth_4', 'mouth_5', 'mouth_6', 'mouth_7', 'mouth_8', 'mouth_9', 'mouth_10', 'mouth_11', 'mouth_12', 'lip_1', 'lip_2', 'lip_3', 'lip_4', 'lip_5', 'lip_6', 'lip_7', 'lip_8'], 'coco': ['nose', 'left_eye', 'right_eye', 'left_ear', 'right_ear', 'left_shoulder', 'right_shoulder', 'left_elbow', 'right_elbow', 'left_wrist', 'right_wrist', 'left_hip_extra', 'right_hip_extra', 'left_knee', 'right_knee', 'left_ankle', 'right_ankle'], 'coco_wholebody': ['nose', 'left_eye', 'right_eye', 'left_ear', 'right_ear', 'left_shoulder', 'right_shoulder', 'left_elbow', 'right_elbow', 'left_wrist', 'right_wrist', 'left_hip', 'right_hip', 'left_knee', 'right_knee', 'left_ankle', 'right_ankle', 'left_bigtoe', 'left_smalltoe', 'left_heel', 'right_bigtoe', 'right_smalltoe', 'right_heel', 'face_contour_1', 'face_contour_2', 'face_contour_3', 'face_contour_4', 'face_contour_5', 'face_contour_6', 'face_contour_7', 'face_contour_8', 'face_contour_9', 'face_contour_10', 'face_contour_11', 'face_contour_12', 'face_contour_13', 'face_contour_14', 'face_contour_15', 'face_contour_16', 'face_contour_17', 'right_eyebrow_1', 'right_eyebrow_2', 'right_eyebrow_3', 'right_eyebrow_4', 'right_eyebrow_5', 'left_eyebrow_5', 'left_eyebrow_4', 'left_eyebrow_3', 'left_eyebrow_2', 'left_eyebrow_1', 'nosebridge_1', 'nosebridge_2', 'nosebridge_3', 'nosebridge_4', 'nose_1', 'nose_2', 'nose_3', 'nose_4', 'nose_5', 'right_eye_1', 'right_eye_2', 'right_eye_3', 'right_eye_4', 'right_eye_5', 'right_eye_6', 'left_eye_4', 'left_eye_3', 'left_eye_2', 'left_eye_1', 'left_eye_6', 'left_eye_5', 'mouth_1', 'mouth_2', 'mouth_3', 'mouth_4', 'mouth_5', 'mouth_6', 'mouth_7', 'mouth_8', 'mouth_9', 'mouth_10', 'mouth_11', 'mouth_12', 'lip_1', 'lip_2', 'lip_3', 'lip_4', 'lip_5', 'lip_6', 'lip_7', 'lip_8', 'left_hand_root', 'left_thumb_1', 'left_thumb_2', 'left_thumb_3', 'left_thumb', 'left_index_1', 'left_index_2', 'left_index_3', 'left_index', 'left_middle_1', 'left_middle_2', 'left_middle_3', 'left_middle', 'left_ring_1', 'left_ring_2', 'left_ring_3', 'left_ring', 'left_pinky_1', 'left_pinky_2', 'left_pinky_3', 'left_pinky', 'right_hand_root', 'right_thumb_1', 'right_thumb_2', 'right_thumb_3', 'right_thumb', 'right_index_1', 'right_index_2', 'right_index_3', 'right_index', 'right_middle_1', 'right_middle_2', 'right_middle_3', 'right_middle', 'right_ring_1', 'right_ring_2', 'right_ring_3', 'right_ring', 'right_pinky_1', 'right_pinky_2', 'right_pinky_3', 'right_pinky'], 'crowdpose': ['left_shoulder', 'right_shoulder', 'left_elbow', 'right_elbow', 'left_wrist', 'right_wrist', 'left_hip', 'right_hip', 'left_knee', 'right_knee', 'left_ankle', 'right_ankle', 'head', 'neck'], 'gta': ['gta_head_top', 'head', 'neck', 'gta_right_clavicle', 'right_shoulder', 'right_elbow', 'right_wrist', 'gta_left_clavicle', 'left_shoulder', 'left_elbow', 'left_wrist', 'spine_2', 'gta_spine1', 'spine_1', 'pelvis', 'gta_spine4', 'right_hip', 'right_knee', 'right_ankle', 'left_hip', 'left_knee', 'left_ankle',*

keypoints.

Parameters

- **data_source** (*str*) – data source type.
- **mask** (*Optional[Union[np.ndarray, tuple, list]], optional*) – refer to keypoints_mapping. Defaults to None.
- **keypoints_factory** (*dict, optional*) – Dict of all the conventions. Defaults to KEYPOINTS_FACTORY.

Returns (limbs_target, limbs_palette).

Return type Tuple[dict, dict]

mmhuman3d.utils.**sja_to_aa**(*sja: Union[torch.Tensor, numpy.ndarray], R_t: Union[torch.Tensor, numpy.ndarray] = tensor([[[1.0, 0.0, 0.0], [0.0, 0.0, 1.0], [0.0, - 1.0, 0.0]], [[1.0, 0.0, 0.0], [0.0, 0.0, 1.0], [0.0, - 1.0, 0.0]], [[1.0, 0.0, 0.0], [0.0, 0.0, - 1.0], [0.0, 1.0, 0.0]], [[1.0, 0.0, 0.0], [0.0, 0.0, 1.0], [0.0, - 1.0, 0.0]], [[1.0, 0.0, 0.0], [0.0, 0.0, 1.0], [0.0, - 1.0, 0.0]], [[1.0, 0.0, 0.0], [0.0, 0.0, - 1.0], [0.0, 1.0, 0.0]], [[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]], [[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]], [[1.0, 0.0, 0.0], [0.0, 0.0, - 1.0], [0.0, 1.0, 0.0]], [[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]], [[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]], [[1.0, 0.0, 0.0], [0.0, 0.0, - 1.0], [0.0, 1.0, 0.0]], [[0.0, 0.0, - 1.0], [0.0, 1.0, 0.0], [1.0, 0.0, 0.0]], [[0.0, 0.0, 1.0], [0.0, 1.0, 0.0], [- 1.0, 0.0, 0.0]], [[1.0, 0.0, 0.0], [0.0, 0.0, - 1.0], [0.0, 1.0, 0.0]], [[0.0, 0.0, - 1.0], [0.0, 1.0, 0.0], [1.0, 0.0, 0.0]], [[0.0, 0.0, 1.0], [0.0, 1.0, 0.0], [- 1.0, 0.0, 0.0]], [[0.0, 0.0, - 1.0], [0.0, 1.0, 0.0], [1.0, 0.0, 0.0]], [[0.0, 0.0, 1.0], [0.0, 1.0, 0.0], [- 1.0, 0.0, 0.0]], [[0.0, 0.0, - 1.0], [0.0, 1.0, 0.0], [1.0, 0.0, 0.0]], [[0.0, 0.0, 1.0], [0.0, 1.0, 0.0], [- 1.0, 0.0, 0.0]]]), R_t_inv: Union[torch.Tensor, numpy.ndarray] = tensor([[[1.0, - 0.0, 0.0], [0.0, 0.0, - 1.0], [0.0, 1.0, 0.0]], [[1.0, - 0.0, 0.0], [0.0, 0.0, - 1.0], [0.0, 1.0, 0.0]], [[1.0, 0.0, - 0.0], [0.0, 0.0, 1.0], [0.0, - 1.0, 0.0]], [[1.0, - 0.0, 0.0], [0.0, 0.0, - 1.0], [0.0, 1.0, 0.0]], [[1.0, - 0.0, 0.0], [0.0, 0.0, - 1.0], [0.0, 1.0, 0.0]], [[1.0, 0.0, - 0.0], [0.0, 0.0, 1.0], [0.0, - 1.0, 0.0]], [[1.0, 0.0, - 0.0], [0.0, 1.0, - 0.0], [0.0, 0.0, 1.0]], [[1.0, 0.0, - 0.0], [0.0, 1.0, - 0.0], [0.0, 0.0, 1.0]], [[1.0, 0.0, - 0.0], [0.0, 0.0, 1.0], [0.0, - 1.0, 0.0]], [[1.0, 0.0, - 0.0], [0.0, 1.0, - 0.0], [0.0, 0.0, 1.0]], [[1.0, 0.0, - 0.0], [0.0, 1.0, - 0.0], [0.0, 0.0, 1.0]], [[1.0, 0.0, - 0.0], [0.0, 0.0, 1.0], [0.0, - 1.0, 0.0]], [[0.0, - 0.0, 1.0], [0.0, 1.0, 0.0], [- 1.0, 0.0, 0.0]], [[- 0.0, 0.0, - 1.0], [- 0.0, 1.0, 0.0], [1.0, 0.0, 0.0]], [[1.0, 0.0, - 0.0], [0.0, 0.0, 1.0], [0.0, - 1.0, 0.0]], [[0.0, - 0.0, 1.0], [0.0, 1.0, 0.0], [- 1.0, 0.0, 0.0]], [[- 0.0, 0.0, - 1.0], [- 0.0, 1.0, 0.0], [1.0, 0.0, 0.0]], [[0.0, - 0.0, 1.0], [0.0, 1.0, 0.0], [- 1.0, 0.0, 0.0]], [[- 0.0, 0.0, - 1.0], [- 0.0, 1.0, 0.0], [1.0, 0.0, 0.0]], [[0.0, - 0.0, 1.0], [0.0, 1.0, 0.0], [- 1.0, 0.0, 0.0]], [[- 0.0, 0.0, - 1.0], [- 0.0, 1.0, 0.0], [1.0, 0.0, 0.0]]])*) → Union[torch.Tensor, numpy.ndarray]

Convert standard joint angles to axis angles.

Parameters

- **sja** (*Union[torch.Tensor, numpy.ndarray]*) – input shape should be (. . . , 21, 3). ndim of input is unlimited.
- **R_t** (*Union[torch.Tensor, numpy.ndarray]*) – input shape should be (. . . , 21, 3, 3). Transformation matrices from original axis-angle coordinate system to standard joint angle coordinate system
- **R_t_inv** (*Union[torch.Tensor, numpy.ndarray]*) – input shape should be (. . . , 21, 3, 3). Transformation matrices from standard joint angle coordinate system to original axis-angle coordinate system

Returns shape would be (. . . , 3).

Return type Union[torch.Tensor, numpy.ndarray]

mmhuman3d.utils.**slice_video**(*input_path: str*, *output_path: str*, *start: int = 0*, *end: Optional[int] = None*, *resolution: Optional[Union[Tuple[int, int], Tuple[float, float]]] = None*, *disable_log: bool = False*) → None

Temporally crop a video/gif into another video/gif.

Parameters

- **input_path** (*str*) – input video or gif file path.
- **output_path** (*str*) – output video of gif file path.
- **start** (*int, optional*) – start frame index. Defaults to 0.
- **end** (*int, optional*) – end frame index. Exclusive. Could be positive int or negative int or None. If None, all frames from start till the last frame are included. Defaults to None.
- **(Optional[Union[Tuple[int** (*resolution*) – optional): (height, width) of output. Defaults to None.
- **int]** – optional): (height, width) of output. Defaults to None.
- **Tuple[float** – optional): (height, width) of output. Defaults to None.
- **float]]]** – optional): (height, width) of output. Defaults to None.

:param : optional): (height, width) of output. Defaults to None. :param disable_log: whether close the ffmepg command info.

Defaults to False.

Raises

- **FileNotFoundError** – check the input path.
- **FileNotFoundError** – check the output path.

Returns NoReturn

mmhuman3d.utils.**smooth_process**(*x*, *smooth_type='savgol'*)

Smooth the array with the specified smoothing type.

Parameters

- **x** (*np.ndarray*) – Shape should be (frame,num_person,K,C) or (frame,K,C).
- **smooth_type** (*str, optional*) – Smooth type. choose in ['oneeuro', 'gaus1d', 'savgol']. Defaults to 'savgol'.

Raises **ValueError** – check the input smoothing type.

Returns

Smoothed data. The shape should be (frame,num_person,K,C) or (frame,K,C).

Return type np.ndarray

mmhuman3d.utils.**spatial_concat_video**(*input_path_list: List[str]*, *output_path: str*, *array: List[int] = [1, 1]*, *direction: typing_extensions.Literal[h, w] = 'h'*, *resolution: Union[Tuple[int, int], List[int], List[float], Tuple[float, float]] = (512, 512)*, *remove_raw_files: bool = False*, *padding: int = 0*, *disable_log: bool = False*) → None

Spatially concat some videos as an array video.

Parameters

- **input_path_list** (*list*) – input video or gif file list.
- **output_path** (*str*) – output video or gif file path.
- **array** (*List[int], optional*) – line number and column number of the video array]. Defaults to [1, 1].
- **direction** (*str, optional*) – [choose in 'h' or 'v', represent horizontal and vertical separately]. Defaults to 'h'.
- **(Optional[Union[Tuple[int** (*resolution*) – optional): (height, width) of output. Defaults to (512, 512).
- **int]** – optional): (height, width) of output. Defaults to (512, 512).
- **Tuple[float** – optional): (height, width) of output. Defaults to (512, 512).
- **float]]]** – optional): (height, width) of output. Defaults to (512, 512).

:param [optional): (height, width) of output.] Defaults to (512, 512).

Parameters

- **remove_raw_files** (*bool, optional*) – whether remove raw images. Defaults to False.
- **padding** (*int, optional*) – width of pixels between videos. Defaults to 0.
- **disable_log** (*bool, optional*) – whether close the ffmepg command info. Defaults to False.

Raises

- **FileNotFoundError** – check the input path.
- **FileNotFoundError** – check the output path.

Returns None

mmhuman3d.utils.**temporal_concat_video**(*input_path_list: List[str]*, *output_path: str*, *resolution: Union[Tuple[int, int], Tuple[float, float]] = (512, 512)*, *remove_raw_files: bool = False*, *disable_log: bool = False*) → None

Concat no matter videos or gifs into a temporal sequence, and save as a new video or gif file.

Parameters

- **input_path_list** (*List[str]*) – list of input video paths.
- **output_path** (*str*) – output video file path.
- **(Optional[Union[Tuple[int** (*resolution*) – , optional): (height, width) of output]. Defaults to (512,512).
- **int]** – , optional): (height, width) of output]. Defaults to (512,512).
- **Tuple[float** – , optional): (height, width) of output]. Defaults to (512,512).
- **float]]]** – , optional): (height, width) of output]. Defaults to (512,512).
- **remove_raw_files** (*bool, optional*) – whether remove the input videos. Defaults to False.

- **disable_log** (*bool, optional*) – whether close the ffmepg command info. Defaults to False.

Raises

- **FileNotFoundError** – check the input path.
- **FileNotFoundError** – check the output path.

Returns None.

mmhuman3d.utils.**video_to_array**(*input_path: str*, *resolution: Optional[Union[Tuple[int, int], Tuple[float, float]]] = None*, *start: int = 0*, *end: Optional[int] = None*, *disable_log: bool = False*) → numpy.ndarray

Read a video/gif as an array of (f * h * w * 3).

Parameters

- **input_path** (*str*) – input path.
- **(Optional[Union[Tuple[int** (*resolution*) – optional): resolution(height, width) of output. Defaults to None.
- **int]** – optional): resolution(height, width) of output. Defaults to None.
- **Tuple[float** – optional): resolution(height, width) of output. Defaults to None.
- **float]]]** – optional): resolution(height, width) of output. Defaults to None.

:param [optional): resolution(height, width) of output.] Defaults to None.

Parameters

- **start** (*int, optional*) –

  start frame index. Inclusive. If < 0, will be converted to frame_index range in [0, frame_num].

  Defaults to 0.

- **end** (*int, optional*) – end frame index. Exclusive. Could be positive int or negative int or None. If None, all frames from start till the last frame are included. Defaults to None.
- **disable_log** (*bool, optional*) – whether close the ffmepg command info. Defaults to False.

Raises **FileNotFoundError** – check the input path.

Returns shape will be (f * h * w * 3).

Return type np.ndarray

mmhuman3d.utils.**video_to_gif**(*input_path: str*, *output_path: str*, *resolution: Optional[Union[Tuple[int, int], Tuple[float, float]]] = None*, *fps: Union[float, int] = 15*, *disable_log: bool = False*) → None

Convert a video to a gif file.

Parameters

- **input_path** (*str*) – video file path.
- **output_path** (*str*) – gif file path.

- **(Optional[Union[Tuple[int** (*resolution*) – optional): (height, width) of the output video. Defaults to None.
- **int]** – optional): (height, width) of the output video. Defaults to None.
- **Tuple[float** – optional): (height, width) of the output video. Defaults to None.
- **float]]]** – optional): (height, width) of the output video. Defaults to None.

:param [optional): (height, width) of the output video.] Defaults to None.

Parameters

- **fps** (*Union[float, int], optional*) – frames per second. Defaults to 15.
- **disable_log** (*bool, optional*) – whether close the ffmepg command info. Defaults to False.

Raises

- **FileNotFoundError** – check the input path.
- **FileNotFoundError** – check the output path.

Returns None.

mmhuman3d.utils.**video_to_images**(*input_path: str*, *output_folder: str*, *resolution: Optional[Union[Tuple[int, int], Tuple[float, float]]] = None*, *img_format: str = '%06d.png'*, *start: int = 0*, *end: Optional[int] = None*, *disable_log: bool = False*) → None

Convert a video to a folder of images.

Parameters

- **input_path** (*str*) – video file path
- **output_folder** (*str*) – output folder to store the images
- **resolution** (*Optional[Tuple[int, int]], optional*) – (height, width) of output. defaults to None.
- **img_format** (*str, optional*) – format of images to be read. Defaults to '%06d.png'.
- **start** (*int, optional*) –

  start frame index. Inclusive. If < 0, will be converted to frame_index range in [0, frame_num].

  Defaults to 0.
- **end** (*int, optional*) – end frame index. Exclusive. Could be positive int or negative int or None. If None, all frames from start till the last frame are included. Defaults to None.
- **disable_log** (*bool, optional*) – whether close the ffmepg command info. Defaults to False.

Raises

- **FileNotFoundError** – check the input path
- **FileNotFoundError** – check the output path

Returns None

mmhuman3d.utils.**xywh2xyxy**(*bbox_xywh*)

Transform the bbox format from xywh to x1y1x2y2.

Parameters

- **bbox_xywh** (*np.ndarray*) – Bounding boxes (with scores), shaped
- **(n** (*n, 5) (left, top, width, height, [score]*) –
- **or** (*4*)) –

Returns

Bounding boxes (with scores), shaped (n, 4) or (n, 5). (left, top, right, bottom, [score])

Return type np.ndarray

mmhuman3d.utils.**xyxy2xywh**(*bbox_xyxy*)

Transform the bbox format from x1y1x2y2 to xywh.

Parameters **bbox_xyxy** (*np.ndarray*) – Bounding boxes (with scores), shaped (n, 4) or (n, 5). (left, top, right, bottom, [score])

Returns

Bounding boxes (with scores), shaped (n, 4) or (n, 5). (left, top, width, height, [score])

Return type np.ndarray

CHAPTER

SEVENTEEN

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

m

INDEX

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X