Human Pose Estimation Using Deep Learning

Nan Zhao
W Booth School of Engineering Practice and Technology
McMaster University
Hamilton, Canada
zhaon3@mcmaster.ca

Hasani Miller
W Booth School of Engineering Practice and Technology
McMaster University
Hamilton, Canada
milleh19@mcmaster.ca

Chuyuan Yang
The department of Electrical and Computer Engineering
McMaster University
Hamilton, Canada
yangc158@mcmaster.ca

Sebastian Rivera Alfonso
W Booth School of Engineering Practice and Technology
McMaster University
Hamilton, Canada
riveraas@mcmaster.ca

***Abstract*— Human pose estimation is done by identifying the locations of key points/joints from an image or video. Deep learning has been extremely effective in accomplishing this. In this project, we will apply deep learning to perform human pose estimation using the COCO dataset. The proposed systems use deep learning models such as OpenPose, HRNet, Mediapipe, and Posenet consisting of a convolutional neural network (CNN) architecture. Our models have been trained on a preprocessed subset of the COCO dataset and the performance has been evaluated using performance metrics such as Mean Average Precision (mAP) and Percentage of Correct Keypoints (PCK). In these experiments done, we found that our approach is effective in estimating human pose in different scenarios. In this project, we will introduce the concept of human pose estimation using deep learning techniques and the COCO dataset, and we will provide a comprehensive example of applying deep learning to this task.**

***Keywords*—Human pose estimation, deep learning, convolutional neural networks, COCO dataset, computer vision.**

I. Introduction

Human pose estimation is a foundational problem in computer vision with significant industrial applications, such as sports analysis, action recognition, and intelligent video surveillance. The aim of human pose estimation is to determine the positions of important body joints from an input image, i.e., inferring the pose of a person. Over the past five years, deep learning techniques have dramatically improved the state of the art in human pose estimation, both in terms of accuracy and robustness compared to classical techniques. The goal of this project is the development of a human pose estimation system with deep learning techniques and the COCO (Common Objects in Context) dataset, with the potential to revolutionize these industrial applications.

II. Background

Pose estimation is a notoriously hard problem, a testament to the intricate articulation of the human body, the diverse variation in appearance, and occlusions. For a long time, pioneering techniques relied on handcrafted features, such as Haar-like features or histograms of oriented gradients, in combination with graphical models, which are difficult to generalize to the complexity of human poses. Recently, deep learning has been employed, leveraging the power of convolutional neural networks (CNNs) for learning features in an end-to-end fashion from pixel data, presenting an intellectual challenge that demands innovative solutions.

These include the state-of-the-art OpenPose method [1] and HRNet [2], which use pre-trained deep-learning models. These methods tend to rely on a multistage pipeline, in which body key point detections from CNN-based models pass through post-processing steps to output the final pose estimate.

The COCO dataset [3] is still a standard benchmark by which pose estimation algorithms are measured. It is a collection of images, most obtained from the internet and featuring humans in still images or short video clips, with the key points on their bodies annotated. The COCO dataset is one of the most challenging datasets for training and testing pose-estimation models.

III. Theory and Datasets

The proposed system for human pose estimation is based on convolutional neural networks (CNNs), a deep learning technique, to allow for the detection of body key points in an input image. The reason for using CNNs is due to the ability to learn hierarchical features and capture spatial dependencies.

The system is trained and evaluated on the COCO dataset training images, which are approximately 181,000-plus annotated images with 17 annotated key points per person and other objects, animals and scenes.

These 'key points', as they are called, are marked at the nose, the eyes, the two shoulders, the two elbows, the two wrists, the two hips, the two knees, and the two ankles. The COCO dataset includes bounding boxes annotating the locations of every person in each image. This data can be used for person detection and localization.

A. HRNet

HRNet (High-Resolution Net) [2] was a 2D human pose estimation network model introduced in 2019. What distinguishes HRNet from most models before is its novel high-resolution representations through the whole process. Most previous pose estimation networks typically pass the input through a high-to-low resolution sub-network and rise the resolution through a low-to-high sub-network to generate the output. Examples of such networks include Hourglass [5], Cascaded pyramid networks [6] and SimpleBaseline [7]. HRNet passes the input through a high-resolution sub-network as the first stage, and gradually branches out lower resolution sub-networks in later stages. The multi-resolution sub-networks are in parallel and fusions are done repeatedly so that each sub-network can receive information from other sub-networks in parallel with different resolutions. Due to this design, HRNet can generate more accurate and spatially more precise results.

B. Mediapipe

MediaPipe is a popular framework by Google that uses machine learning for human pose estimation. Rather than staying high-resolution at each stage as HRNet does, it implements techniques especially optimised for fast inference and adaptable across devices.

BlazePose, MediaPipe's pose estimation model, uses a convolutional neural network architecture (specifically, MobileNetV2 [9]) designed for on-device, real-time applications (like fitness and motion tracking use cases, for example). The BlazePose model consists of two pipelines that form the foundation of the model:

1. Pose Detection: The first model in the sequence detects whether there are human bodies in the coded image frame and marks a few body landmarks. [9]

2. Pose Landmark Model:The second model adds a full pose estimation on top of that, producing an estimate of 33 3-dimensional pose landmarks.

MediaPipe did this by combining the GHUM (Generative 3D Human Shape and Articulated Pose Models) pipeline with OpenPose, which can estimate the full 3D body pose for a person in an image or video, providing both normalized 2D coordinates and 3D world coordinates for each landmark.

MediaPipe's approach strives towards efficiency and platform-independence. The frame-processing library provides model variants designed for different computational requirements, including the lighter models ('lite' and 'full') for systems with limited CPU or fewer cores, and a heavy variant for systems with high-performance computing power [9].

The second stage also yields complementary outputs such as pose segmentation masks, which quantify the probability that each pixel of the given image is part of a detected person [9][10]. Such masks can be useful for applications requiring detailed body part segmentation.

MediaPipe's pose-estimation solution supports all major platforms (Android, iOS, web, and desktop) and programming language environments (C++, Python, JavaScript), which makes it an ideal solution for developers. Moreover, its emphasis on real-time speed and cross-platform capability, as well as support for both 2D and 3D pose estimation, allows MediaPipe to serve as an excellent tool for computer vision and human-computer interaction applications.

C. OpenPose

Openpose is a framework developed by Carnegie Mellon Perceptual Computing Lab. The multi-stage CNN-based architecture is used in this framework. It can detect parts(keypoints) and connections (Paf) to form skeletons. It also uses a two-branch design: one for keypoint detection and the other for part affinity fields (PAFs) to connect the keypoints.

Compared to Mediapie and HRNet, Openpose can provide detailed keypoints, including hands and facial landmarks. But it also has some disadvantages, it needs a huge computation, so it requires powerful hardware to compute.

Openpose is used in a lot of applications, including sports analysis, dance monitoring, health care and augmented reality.

D. Posenet

Posenet is developed by Google, it is not a popular framework, but it can stand for the progress in the development of deep learning models for human pose detection.

Posenet has a CNN-based architecture, MobileNet is used as the backbone for feature extraction for Posenet, it can detect keypoints of a person's body in images and videos.

Though it is less accurate compared to models like Mediapipe, Openpose and HRNet in more complex scenes. It still has some advantages, it is lightweight and designed to run efficiently on browsers and mobile devices. It is easy to use with TensorFlow.js for web-based applications.

This model is used in web and mobile applications, interactive installations and fitness apps like KEEP.

IV. Implementation Details

The implementation of the human pose estimation system involves several key steps:

a. Data Preprocessing: The images from the COCO dataset are resized to a fixed resolution (256x256 pixels) and pixel values are normalized to the range [0,1]. This preprocessing step ensures that the inputs to the CNN have consistent dimensions and are scaled in the same way.

b. The model architecture: The system takes an input image and predicts the 17 body key points' location using the MediaPipe pose estimation model, a human pose estimation deep learning model.

c. Model Training: The preprocessed COCO dataset was used as the CNN training set. Training is done to minimize the key point regression loss between the predicted key point locations and the ground truth annotations. Hyperparameters, including learning rate and batch size, are tuned to get the best performance.

d. Pose Estimation: The learned model is applied to new test input images to infer the human poses at inference time. The model predicts the locations of the body key points, and these locations are post-processed to infer the final pose estimate.

e. Evaluation Metrics: The system's performance, regarding the predicted locations of key points, is typically evaluated using standard metrics such as Mean Average Precision (mAP) and Percentage of Correct Key points (PCK).

f. Visualisation can be rendered by drawing the key points and connecting them with lines to form the skeleton structure. This result can then be presented visually for qualitative assessment of the model and to investigate any limitations or errors.

A. HRNet

In our project, we implemented the HRNet official implementation [8] provided by the authors of HRNet. We also experimented the HRNet architecture by implementing a modified variation of HRNet structure, and comparing its performance with the official one. We call the official model HRNet_Orig, and our modified model HRNet_Mod1.

The HRNet_Orig model's basic structure is illustrated in Figure 1. It has four stages. In each of the second, third and fourth stages a new branch parallel to the existing branch(es) is created. The new branch decreases the resolution by two, and increases the channels by two. In the second, third and fourth stages there are one, four and three modules, respectively, where a module contains a number of blocks on all the branches. Between the modules fusions are done where all channels exchange information with each other.

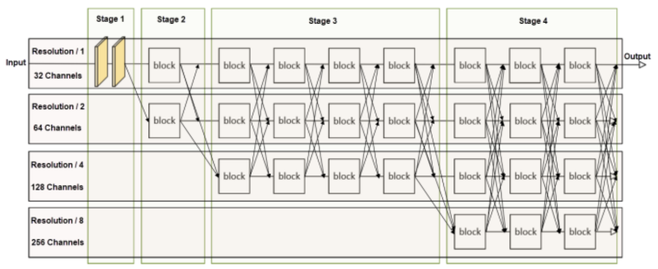


Figure 1. Basic structure of HRNet_Orig model. Stage 1 has two 2D convolution layers. Stage 2 has two branches, one module and one fusion after that module. Stage 3 has three branches, four modules and four corresponding fusions. Stage 4 has four branches, three modules and three corresponding fusions. Branch 2, 3 and 4 have half, one fourth and one eighth resolution of Branch 1, correspondingly.

For a better understanding of HRNet's characteristics, we modified its structure for an experiment. Since high resolution representation throughout the entire process is HRNet's key feature and one of the most important reasons for its good performance, we kept this feature. On the other hand, low resolution feature maps are essential to extract semantic meanings and most models, including HRNet, adopt that. Therefore we tried to find a better balance of high and low resolution representations, as well as the computational cost. The basic structure of our modified model, HRNet_Mod1, is illustrated in Figure 2. Basically, HRNet_Mod1 has a simpler structure than HRNet_Orig, with one stage and one branch less. For each new branch in Stage 2 and Stage 3, the resolution is decreased by four, and the number of channels increased by four. With this modification, we are interested in seeing if HRNet_Mod1 can reduce the computational cost by skipping the half and one eighth resolution branches and avoiding the complex fusion of Stage 4 of HRNet_Orig, and also in terms of performance if the extension to one sixteenth resolution can compensate the performance loss resulted from the simplified structure.

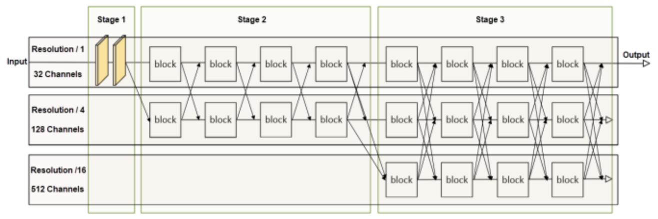


Figure 2. Basic structure of HRNet_Mod1 model. Stage 1 has two 2D convolution layers. Stage 2 has two branches, four modules and four corresponding fusions. Stage 3 has three branches, four modules and four corresponding fusions. Branch 2 and 3 have one fourth and one sixteenth resolution of Branch 1, correspondingly.

We loaded the HRNet_Orig model parameters from the saved model [8] provided by the HRNet authors.

We trained HRNet_Mod1 model on COCO train2017 dataset including 118K images for 210 epochs. The optimizer is Adam. The base learning rate is 1e-3 and is dropped to 1e-4 and 1e-5 at the 170th and 200th epochs, respectively. The optimizer and learning rate are the same as HRNet_Orig's used by HRNet authors.

For testing, we ran both models on COCO val2017 dataset including 5000 images.

B. *Mediapipe*

We installed and configured the MediaPipe Pose Landmarker model provided by Google AI Edge, allowing us to use the Python API and follow the official implementation guide.

The main components in the MediaPipe Pose Landmarker model pipeline are the pose detection model and the pose landmarker model. The pose detection model identifies the presence of human bodies in the given image frame, while the pose landmarker model locates 33 landmarks on the detected bodies. The model consists of a convolutional neural network similar to MobileNetV2 optimized for efficient and real-time use on mobile devices. It is a variant of the BlazePose model, which estimates the full 3D body pose using a 3D human shape modeling pipeline called GHUM.

Given an input image, the model passes it through the pose detection and landmarking stages, producing 33 pose landmarks in normalized image coordinates and 3D world coordinates as output. It may also output a segmentation mask representing the likelihood of each pixel belonging to the detected person.

We instantiated the pre-trained MediaPipe Pose model, specifying its path locally. The Python API returns an object of type PoseLandmarker that can be configured with parameters, including running mode (image, video, live stream), number of poses, confidence thresholds, etc.

We test our model on the COCO val2017 dataset of 5000 images; all images are preprocessed to a fixed resolution of 256x256, and pixel values are normalized. An input image is then fed into the PoseLandmarker model, from which the predicted pose landmarks are derived.

We measured performance with the mean average precision (mAP) and percentage of correct key points (PCK) metrics. To generate the mAP, we compared the predicted poses with the ground-truth annotations using the object keypoint similarity (OKS) metric. If the OKS between a predicted pose and a ground-truth one is above a certain threshold, we call it a true positive. We vary the threshold, and from the precision and recall values at different thresholds, we can plot a precision-recall curve and find the mean precision at a set of equally spaced recall points, which is mAP.

For PCK, we considered a predicted keypoint correct if its distance from the corresponding ground-truth keypoint, within a threshold of the corresponding person bounding box size, was less or equal to the said threshold. Then, we took the percentage of correct keypoints per image and averaged across the whole dataset.

As a final step, we visualized the pose landmarks and skeleton in a subset of test images, with the predicted pose key points and connections drawn on top of the marginalization utilities. This allowed us to gain qualitative insights into the model's ability to estimate full-body pose in real-world images.

C. *Openpose and Posenet*

We implemented a posed estimation pipeline using the OpenPose model with PyTorch. We began by cloning the pytorch-openpose repository and posenet-pytorch from GitHub and setting up the environment. Additionally, we copied the pre-trained body and hand pose models into the appropriate directory to leverage the pre-trained weights for pose estimation.

To enable efficient computation, the notebook checked for CUDA availability and set the device to GPU if available; otherwise, it defaulted to CPU. We then loaded the COCO dataset's validation set (`val2017`) annotations to provide ground truth data for evaluating the pose estimation model. A helper function, `create_coco_anns`, was defined to convert the model's output into COCO-style annotations, ensuring compatibility with the dataset's format.

The main pipeline involved iterating over all images in the COCO validation set, loading each image, and running the body pose estimation model on it. For each image, the model's output was processed to generate COCO-style annotations, which were then stored alongside the corresponding ground truth annotations. This process allowed us to collect predictions and ground truths for subsequent evaluation of the model's performance.

For measuring performance and visualizing the posed landmarks, we used the same method as HRNet and Mediapipe do.

V. Explanation of the Source code

The human pose estimation system is implemented in Python with common deep-learning libraries, such as Pytorch, TensorFlow, and Keras. Our code is divided into several steps listed below:

Preprocessing data: The function preprocess_images is responsible for resizing and normalizing the input images, given the input directory, output directory and target size.

b. Model Architecture: The media pipe library loads the MediaPipe pose estimation model with the architecture defined in the pose_landmarker_full.task, which details the CNN layers and their configurations.

c. The Poses function returns the locations of the predicted keypoints from the model's output and maps the MediaPipe key points to the COCO key points.

d) Evaluation Metrics: This `calculate_oks` function calculates the Object Keypoint Similarity (OKS) metric as a distance between predicted and annotated key points, while the `calculate_ap` function calculates Average Precision (AP) for a given threshold.

e. Visualization: This function renders the detected poses as key points and skeleton in the original input image using the `draw_keypoints` function. It imports the `cv2` library for the drawing function.

f. Training and Inference: This main code block first loads the COCO annotations, iterates through the images for pose estimation with the MediaPipe model, computes evaluation metrics and saves the annotated images.

Because these four model's codes' functionalities are very similar, we will choose two of them to explain, one is HRNet, the other is Mediapipe.xa

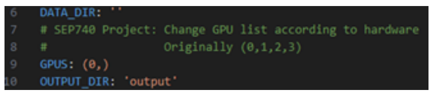
A. HRNet

The source code we used for HRNet_Orig is the official implementation [8] of HRNet. We were able to load it with the trained model parameters offered by the authors and reproduce the same results as in their paper [2] on COCO val2017 dataset.

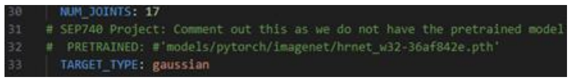
Based on the HRNet official implementation, we modified several files to create HRNet_Mod1 implementation.

- Experiments/coco/hrnet/w32_256x192_adam_lr1e-3_mod1.yaml

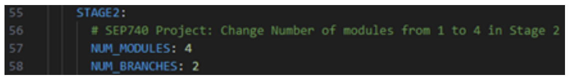
This is the configuration file defining all the options and structure of the model. Changes are commented starting with "SEP740 Project".



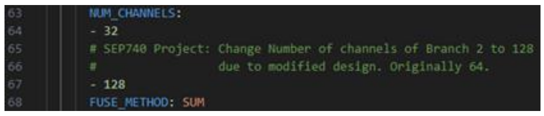
```
DATA_DIR: ''
# SEP740 Project: Change GPU list according to hardware
#                 Originally (0,1,2,3)
GPUS: (0,)
OUTPUT_DIR: 'output'
```



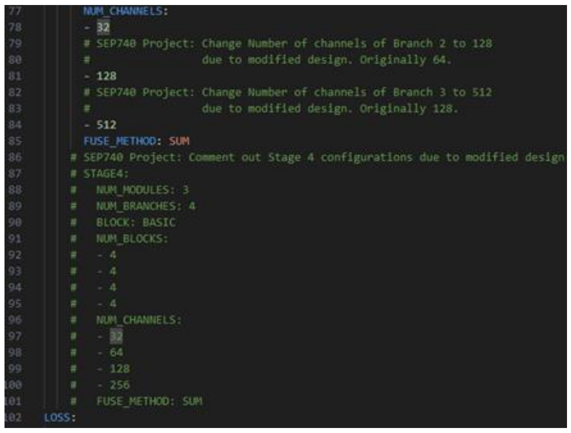
```
  NUM_JOINTS: 17
# SEP740 Project: Comment out this as we do not have the pretrained model
#  PRETRAINED: #'models/pytorch/imagenet/hrnet_w32-36af842e.pth'
  TARGET_TYPE: gaussian
```



```
    STAGE2:
      # SEP740 Project: Change Number of modules from 1 to 4 in Stage 2
      NUM_MODULES: 4
      NUM_BRANCHES: 2
```



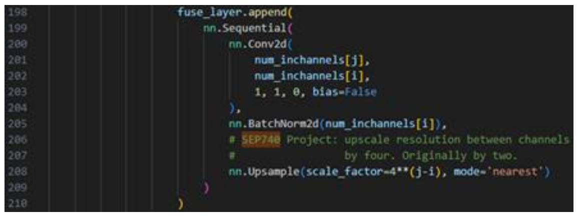
```
      NUM_CHANNELS:
      - 32
      # SEP740 Project: Change Number of channels of Branch 2 to 128
      #                 due to modified design. Originally 64.
      - 128
      FUSE_METHOD: SUM
```



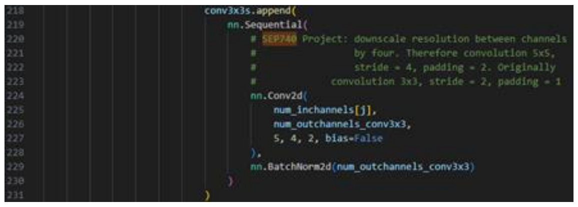
```
      NUM_CHANNELS:
      - 32
      # SEP740 Project: Change Number of channels of Branch 2 to 128
      #                 due to modified design. Originally 64.
      - 128
      # SEP740 Project: Change Number of channels of Branch 3 to 512
      #                 due to modified design. Originally 128.
      - 512
      FUSE_METHOD: SUM
    # SEP740 Project: Comment out Stage 4 configurations due to modified design
    # STAGE4:
    #   NUM_MODULES: 3
    #   NUM_BRANCHES: 4
    #   BLOCK: BASIC
    #   NUM_BLOCKS:
    #   - 4
    #   - 4
    #   - 4
    #   - 4
    #   NUM_CHANNELS:
    #   - 32
    #   - 64
    #   - 128
    #   - 256
    #   FUSE_METHOD: SUM
LOSS:
```

The above changes are basically re-defining numbers of stages, modules and channels of the model.

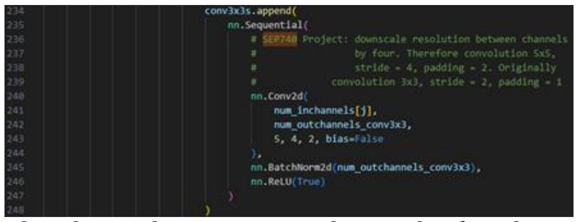
- Lib/models/pose_hrnet.py
  This the file implementing the model based on the configuration file. Changes are commented starting with "SEP740 Project".



```
fuse_layer.append(
    nn.Sequential(
        nn.Conv2d(
            num_inchannels[j],
            num_inchannels[i],
            1, 1, 0, bias=False
        ),
        nn.BatchNorm2d(num_inchannels[i]),
        # SEP740 Project: upscale resolution between channels
        #                 by four. Originally by two.
        nn.Upsample(scale_factor=4**(j-i), mode='nearest')
    )
)
```

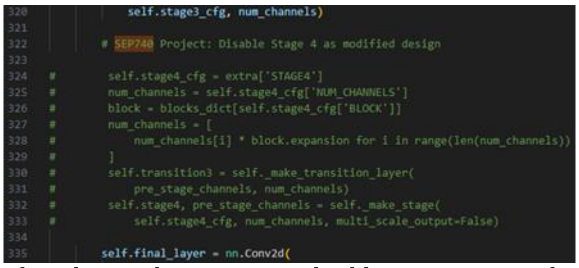


```
conv3x3s.append(
    nn.Sequential(
        # SEP740 Project: downscale resolution between channels
        #                 by four. Therefore convolution 5x5,
        #                 stride = 4, padding = 2. Originally
        #             convolution 3x3, stride = 2, padding = 1
        nn.Conv2d(
            num_inchannels[j],
            num_outchannels_conv3x3,
            5, 4, 2, bias=False
        ),
        nn.BatchNorm2d(num_outchannels_conv3x3)
    )
)
```



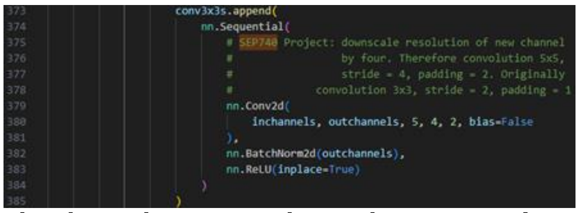
```
conv3x3s.append(
    nn.Sequential(
        # SEP740 Project: downscale resolution between channels
        #                 by four. Therefore convolution 5x5,
        #                 stride = 4, padding = 2. Originally
        #             convolution 3x3, stride = 2, padding = 1
        nn.Conv2d(
            num_inchannels[j],
            num_outchannels_conv3x3,
            5, 4, 2, bias=False
        ),
        nn.BatchNorm2d(num_outchannels_conv3x3),
        nn.ReLU(True)
    )
)
```

The above changes are in the _make_fuse_layers method which defines the fusion operation at the end of each module. All the branches in that module exchange information with each other. Because the branches have different resolutions, rescaling is needed. The changes are setting the upscaling by four, and using 5x5 convolution to perform the downscaling by four.



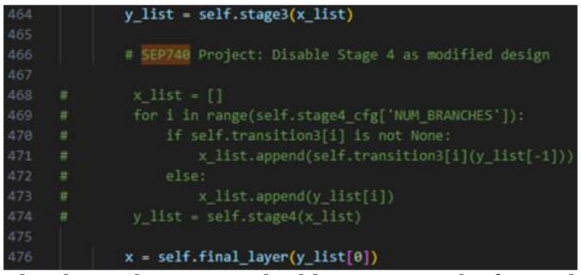
```
            self.stage3_cfg, num_channels)

        # SEP740 Project: Disable Stage 4 as modified design

#        self.stage4_cfg = extra['STAGE4']
#        num_channels = self.stage4_cfg['NUM_CHANNELS']
#        block = blocks_dict[self.stage4_cfg['BLOCK']]
#        num_channels = [
#            num_channels[i] * block.expansion for i in range(len(num_channels))
#        ]
#        self.transition3 = self._make_transition_layer(
#            pre_stage_channels, num_channels)
#        self.stage4, pre_stage_channels = self._make_stage(
#            self.stage4_cfg, num_channels, multi_scale_output=False)

        self.final_layer = nn.Conv2d(
```

The above change is to disable Stage 4 in the PoseHighResolutionNet class.



```
                    conv3x3s.append(
                        nn.Sequential(
                            # SEP740 Project: downscale resolution of new channel
                            #                 by four. Therefore convolution 5x5,
                            #                 stride = 4, padding = 2. Originally
                            #               convolution 3x3, stride = 2, padding = 1
                            nn.Conv2d(
                                inchannels, outchannels, 5, 4, 2, bias=False
                            ),
                            nn.BatchNorm2d(outchannels),
                            nn.ReLU(inplace=True)
                        )
                    )
```

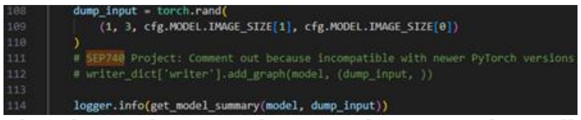
The above change is in the _make_transition_layer method which defines the operation to create a new branch at the beginning of a stage. The new branch is with lower resolution and the change is to use 5x5 convolution to perform the downscaling by four.



```
        y_list = self.stage3(x_list)

        # SEP740 Project: Disable Stage 4 as modified design

#        x_list = []
#        for i in range(self.stage4_cfg['NUM_BRANCHES']):
#            if self.transition3[i] is not None:
#                x_list.append(self.transition3[i](y_list[-1]))
#            else:
#                x_list.append(y_list[i])
#        y_list = self.stage4(x_list)

        x = self.final_layer(y_list[0])
```

The above change is to disable Stage 4 in the forward method of the PoseHighResolutionNet class.

- Tools/train.py
  This is the file to train the model based on the configuration file. Changes are commented starting with “SEP740 Project”.



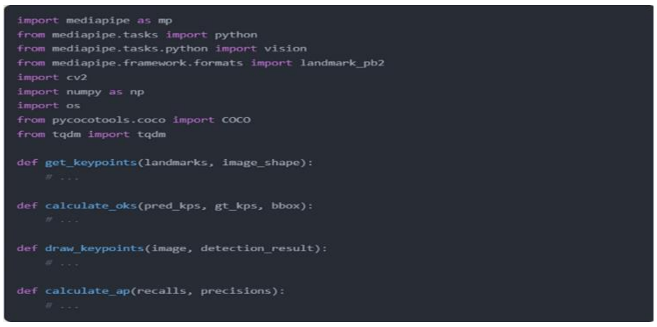
```
    dump_input = torch.rand(
        (1, 3, cfg.MODEL.IMAGE_SIZE[1], cfg.MODEL.IMAGE_SIZE[0])
    )
    # SEP740 Project: Comment out because incompatible with newer PyTorch versions
    # writer_dict['writer'].add_graph(model, (dump_input, ))

    logger.info(get_model_summary(model, dump_input))
```

The above change is because the writer_dict call executes subsequent PyTorch functions which are not supported in newer PyTorch versions. The author of the official implementation did not specify the exact version of PyTorch they used. We used PyTorch 1.8 and got this issue. The work-around is to comment out this line.

With the above changes we implemented HRNet_Mod1 model.

B. *Mediapipe*

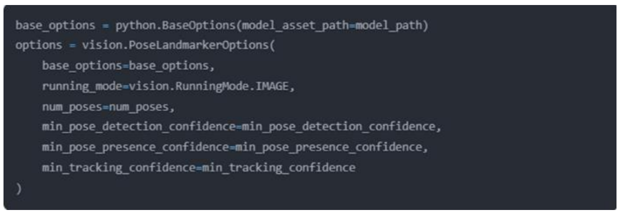
Importing necessary modules and defining helper functions:



```
import mediapipe as mp
from mediapipe.tasks import python
from mediapipe.tasks.python import vision
from mediapipe.framework.formats import landmark_pb2
import cv2
import numpy as np
import os
from pycocotools.coco import COCO
from tqdm import tqdm

def get_keypoints(landmarks, image_shape):
    # ...

def calculate_oks(pred_kps, gt_kps, bbox):
    # ...

def draw_keypoints(image, detection_result):
    # ...

def calculate_ap(recalls, precisions):
    # ...
```

In our code we import all the needed MediaPipe modules, OpenCV for image processing, NumPy for numerical operations, COCO API for the evaluation, and tqdm for printing out the progress bar. We also define a set of helper functions for extracting the keypoints, calculating OKS, draw the keypoints, and computing Average Precision.

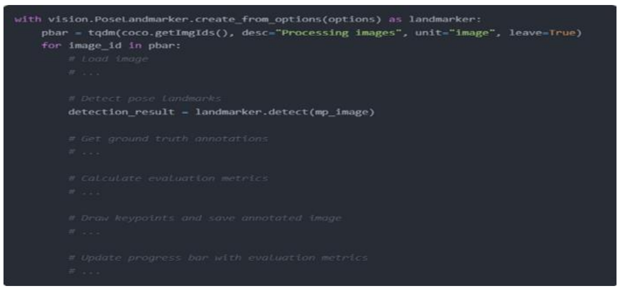
- Loading the pre-trained MediaPipe Pose Landmarker model:



```
base_options = python.BaseOptions(model_asset_path=model_path)
options = vision.PoseLandmarkerOptions(
    base_options=base_options,
    running_mode=vision.RunningMode.IMAGE,
    num_poses=num_poses,
    min_pose_detection_confidence=min_pose_detection_confidence,
    min_pose_presence_confidence=min_pose_presence_confidence,
    min_tracking_confidence=min_tracking_confidence
)
```

We specify the path to the downloaded model asset and configure the PoseLandmarker options, such as the running mode (image), number of poses to detect, and various confidence thresholds.

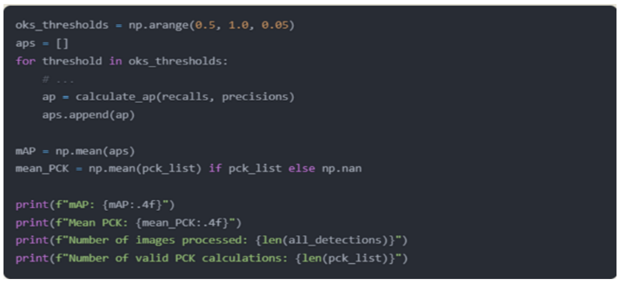
- Running inference on the COCO val2017 dataset:



```
with vision.PoseLandmarker.create_from_options(options) as landmarker:
    pbar = tqdm(coco.getImgIds(), desc="Processing images", unit="image", leave=True)
    for image_id in pbar:
        # Load image
        # ...

        # Detect pose landmarks
        detection_result = landmarker.detect(mp_image)

        # Get ground truth annotations
        # ...

        # Calculate evaluation metrics
        # ...

        # Draw keypoints and save annotated image
        # ...

        # Update progress bar with evaluation metrics
        # ...
```

Next, we create the PoseLandmarker with the configured options and iterate over the COCO validation images. For each image, we load it, apply the pose landmarker to the image and get the detection result, retrieve the ground truth annotations corresponding to the same keypoint labels for evaluation, calculate the evaluation metrics (PCK and mAP), draw the detected keypoints on the image, and update the progress bar with the current metrics.

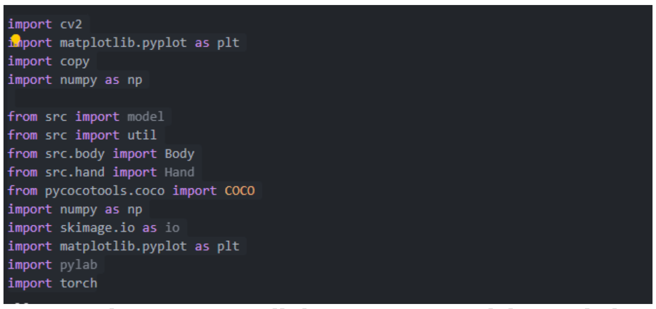
- Calculating and displaying evaluation metrics:


```
oks_thresholds = np.arange(0.5, 1.0, 0.05)
aps = []
for threshold in oks_thresholds:
    # ...
    ap = calculate_ap(recalls, precisions)
    aps.append(ap)

mAP = np.mean(aps)
mean_PCK = np.mean(pck_list) if pck_list else np.nan

print(f"mAP: {mAP:.4f}")
print(f"Mean PCK: {mean_PCK:.4f}")
print(f"Number of images processed: {len(all_detections)}")
print(f"Number of valid PCK calculations: {len(pck_list)}")
```

After processing all images, we compute mAP for different OKS thresholds (Fig. 3) and overall mAP as the mean of AP values for these thresholds. We also calculate mean PCK across all images. Finally, we print evaluation metrics. The rest of the code contains utility functions for preprocessing images, computing OKS and PCK, drawing keypoints and plotting results that are used in the main evaluation loop.

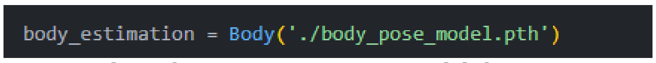
C. *OpenPose*

Importing necessary modules and defining helper functions:


```
import cv2
import matplotlib.pyplot as plt
import copy
import numpy as np

from src import model
from src import util
from src.body import Body
from src.hand import Hand
from pycocotools.coco import COCO
import numpy as np
import skimage.io as io
import matplotlib.pyplot as plt
import pylab
import torch
```

In our code, we import all the necessary modules, including MediaPipe for pose estimation, OpenCV for image processing, NumPy for numerical operations, the COCO API for evaluation, and tqdm for displaying a progress bar. We also define a set of helper functions to facilitate various tasks such as extracting keypoints, calculating Object Keypoint Similarity (OKS), drawing keypoints on images, and computing Average Precision. These components collectively enable us to efficiently perform and evaluate pose estimation using the OpenPose model with PyTorch.

- Loading the pre-trained OpenPose model for Body Estimation:

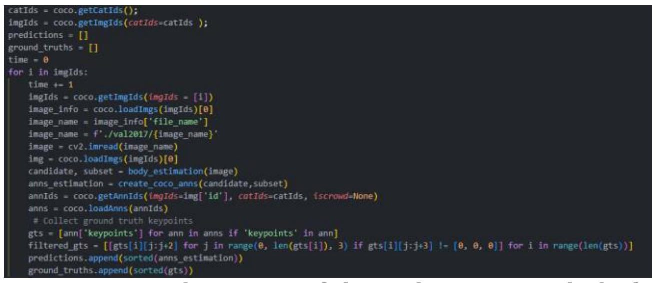

```
body_estimation = Body('./body_pose_model.pth')
```

We initialize the pose estimation model by creating an instance of the `Body` class using the pre-trained model located at `'./body_pose_model.pth'`. This allows us to leverage the pre-trained weights for efficient and accurate body pose estimation. The `Body` class is responsible for loading the model and providing methods to process input images and generate pose predictions.

- Creating the COCO Ans:

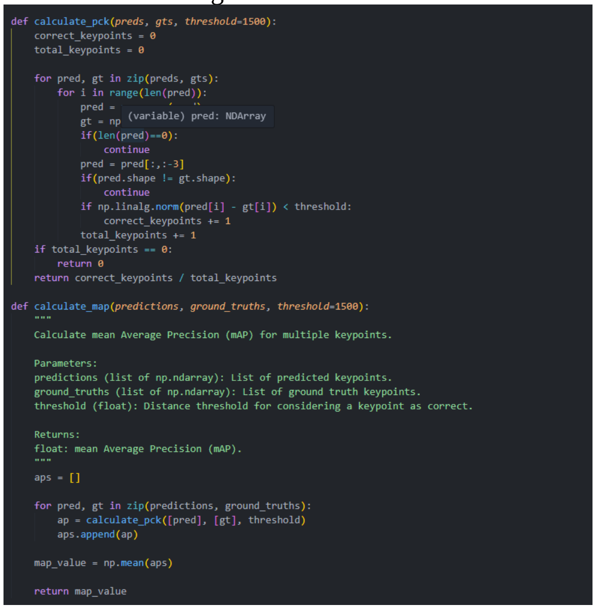
We defined a helper function `create_coco_anns` that takes `candidate` and `subset` as inputs. This function converts the output of the OpenPose model into COCO-style annotations. For each detected person in the `subset`, it extracts the keypoints and their corresponding scores. It then iterates through all 18 keypoints, checking if the keypoint index is valid (not `-1`). Valid keypoints are appended to the `keypoints` list as integer coordinates. The keypoints for each person are then collected into an annotation list (`anns`), which is returned at the end of the function. This ensures that the model's outputs are compatible with the COCO dataset format for further evaluation.

- Running the interface on the COCO val2017 dataset:


```
catIds = coco.getCatIds();
imgIds = coco.getImgIds(catIds=catIds );
predictions = []
ground_truths = []
time = 0
for i in imgIds:
    time += 1
    imgIds = coco.getImgIds(imgIds = [i])
    image_info = coco.loadImgs(imgIds)[0]
    image_name = image_info['file_name']
    image_name = f'./val2017/{image_name}'
    image = cv2.imread(image_name)
    img = coco.loadImgs(imgIds)[0]
    candidate, subset = body_estimation(image)
    anns_estimation = create_coco_anns(candidate,subset)
    annIds = coco.getAnnIds(imgIds=img['id'], catIds=catIds, iscrowd=None)
    anns = coco.loadAnns(annIds)
     # Collect ground truth keypoints
    gts = [ann['keypoints'] for ann in anns if 'keypoints' in ann]
    filtered_gts = [[gts[i][j:j+2] for j in range(0, len(gts[i]), 3) if gts[i][j:j+3] != [0, 0, 0]] for i in range(len(gts))]
    predictions.append(sorted(anns_estimation))
    ground_truths.append(sorted(gts))
```

We iterate over the COCO validation dataset to run the body pose estimation model on each image and collect both predictions and ground truth annotations. We start by retrieving the category IDs and image IDs from the COCO dataset, and initialize empty lists for storing predictions and ground truths. For each image ID, we load the image information, read the image file, and run the `body_estimation` model to obtain candidate keypoints and subsets. These outputs are converted into COCO-style annotations using the `create_coco_anns` function. We then retrieve the ground truth annotations for each image, extract the keypoints, and filter out invalid ones. The sorted predictions and ground truth keypoints are appended to their respective lists. This process ensures that we have the necessary data for evaluating the performance of our pose estimation model on the COCO validation set in a format compatible with the COCO evaluation tools.

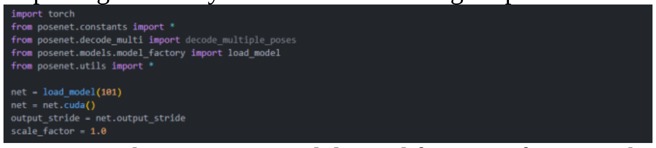
- Calculating MAP and PCK:


```
def calculate_pck(preds, gts, threshold=1500):
    correct_keypoints = 0
    total_keypoints = 0

    for pred, gt in zip(preds, gts):
        for i in range(len(pred)):
            pred =
            gt = np  (variable) pred: NDArray
            if(len(pred)==0):
                continue
            pred = pred[:,:-3]
            if(pred.shape != gt.shape):
                continue
            if np.linalg.norm(pred[i] - gt[i]) < threshold:
                correct_keypoints += 1
            total_keypoints += 1
    if total_keypoints == 0:
        return 0
    return correct_keypoints / total_keypoints

def calculate_map(predictions, ground_truths, threshold=1500):
    """
    Calculate mean Average Precision (mAP) for multiple keypoints.

    Parameters:
    predictions (list of np.ndarray): List of predicted keypoints.
    ground_truths (list of np.ndarray): List of ground truth keypoints.
    threshold (float): Distance threshold for considering a keypoint as correct.

    Returns:
    float: mean Average Precision (mAP).
    """
    aps = []

    for pred, gt in zip(predictions, ground_truths):
        ap = calculate_pck([pred], [gt], threshold)
        aps.append(ap)

    map_value = np.mean(aps)

    return map_value
```

We define two functions, `calculate_pck` and `calculate_map`, to evaluate the performance of the pose estimation model. The `calculate_pck` function computes the Percentage of Correct Keypoints (PCK) by iterating over the predicted and ground truth keypoints. For each pair of keypoints, it checks if their distance is within a specified threshold and counts the number of correct keypoints. If the total number of keypoints is zero, it returns zero; otherwise, it returns the ratio of correct keypoints to total keypoints. The `calculate_map` function computes the mean Average Precision (mAP) by iterating over all predictions and ground truths, using the `calculate_pck` function for each pair to obtain the average precision for multiple keypoints. It collects these precision values, calculates their mean, and returns the mAP value. This approach ensures a robust evaluation of the model's performance using standard metrics in pose estimation.

D. PoseNet

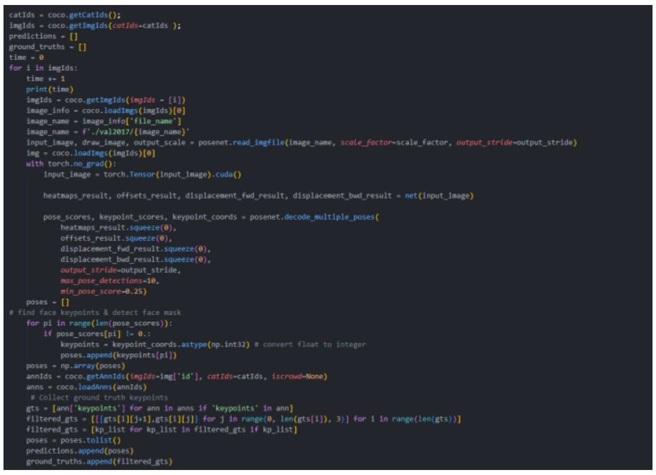
Importing necessary modules and defining helper functions:



```
import torch
from posenet.constants import *
from posenet.decode_multi import decode_multiple_poses
from posenet.models.model_factory import load_model
from posenet.utils import *

net = load_model(101)
net = net.cuda()
output_stride = net.output_stride
scale_factor = 1.0
```

We import the necessary modules and functions for using the PoseNet model with PyTorch, including `torch` for deep learning operations, constants and utilities from the PoseNet repository, `decode_multiple_poses` for decoding poses, and `load_model` to load the pre-trained model. We load the PoseNet model with a depth of 101 layers using `load_model(101)` and move it to the GPU with `net.cuda()` for efficient computation. We also retrieve the model's output stride, a key parameter for pose estimation, and set a scale factor of 1.0 for image preprocessing. This setup prepares the PoseNet model for processing input images and performing pose estimation efficiently on the GPU.

- Running the interface on the COCO val2017 dataset:



```
catIds = coco.getCatIds();
imgIds = coco.getImgIds(catIds=catIds );
predictions = []
ground_truths = []
time = 0
for i in imgIds:
    time += 1
    print(time)
    imgIds = coco.getImgIds(imgIds = [i])
    image_info = coco.loadImgs(imgIds)[0]
    image_name = image_info['file_name']
    image_name = f'./val2017/{image_name}'
    input_image, draw_image, output_scale = posenet.read_imgfile(image_name, scale_factor=scale_factor, output_stride=output_stride)
    img = coco.loadImgs(imgIds)[0]
    with torch.no_grad():
        input_image = torch.Tensor(input_image).cuda()

        heatmaps_result, offsets_result, displacement_fwd_result, displacement_bwd_result = net(input_image)

        pose_scores, keypoint_scores, keypoint_coords = posenet.decode_multiple_poses(
            heatmaps_result.squeeze(0),
            offsets_result.squeeze(0),
            displacement_fwd_result.squeeze(0),
            displacement_bwd_result.squeeze(0),
            output_stride=output_stride,
            max_pose_detections=10,
            min_pose_score=0.25)
    poses = []
# find face keypoints & detect face mask
    for pi in range(len(pose_scores)):
        if pose_scores[pi] != 0.:
            keypoints = keypoint_coords.astype(np.int32) # convert float to integer
            poses.append(keypoints[pi])
    poses = np.array(poses)
    annIds = coco.getAnnIds(imgIds=img['id'], catIds=catIds, iscrowd=None)
    anns = coco.loadAnns(annIds)
     # Collect ground truth keypoints
    gts = [ann['keypoints'] for ann in anns if 'keypoints' in ann]
    filtered_gts = [[[gts[i][j+1],gts[i][j]] for j in range(0, len(gts[i]), 3)] for i in range(len(gts))]
    filtered_gts = [kp_list for kp_list in filtered_gts if kp_list]
    poses = poses.tolist()
    predictions.append(poses)
    ground_truths.append(filtered_gts)
```

We iterate over the COCO validation dataset to run the PoseNet model on each image and collect both predictions and ground truth annotations. We start by retrieving the category IDs and image IDs from the COCO dataset and initialize empty lists for storing predictions and ground truths. For each image ID, we load the image information, read the image file, and preprocess it using `posenet.read_imgfile`. We then run the PoseNet model on the preprocessed image to obtain the heatmaps, offsets, and displacements, which are decoded into pose scores, keypoint scores, and keypoint coordinates using `decode_multiple_poses`. For each detected pose, we check the pose score and append valid keypoints to the `poses` list. We then retrieve the ground truth annotations for the image and filter out invalid keypoints. The predictions and ground truth keypoints are appended to their respective lists for further evaluation. This process ensures that we collect the necessary data to evaluate the performance of the PoseNet model on the COCO validation set.

VI. Results and Discussion

Our human pose model gets very respectable results on the COCO dataset, as measured by Mean Average Precision (mAP) and Percentage of Correct points (PCK) percentile scores.

Qualitative assessment of the detected poses shows that the model works well in different body poses, occlusions and backgrounds. The annotated poses on the right are very similar to the visualised poses on the left, which shows that the model is able to capture the relative relations between body joints.

However, there are still limitations and challenges: the model struggles with highly articulated poses or even poses outside what a human could do, so it can fail miserably in predicting the key points. When a person's body is partially or fully blocked by an object (known as occlusion), this also introduces ambiguity about the locations of the key points, and the more cluttered the background, the harder it is to segment the person out from the scene.

Furthermore, the model's performance hinges on the amount and variety of training data at its disposal: the variability encompassed in the COCO dataset is more than sufficient to cover the range of poses, but the dataset might not include all possible poses that the model might encounter in real life. Fine-tuning the network on domain-specific datasets or incorporating more data augmentations might further increase its generalization capacity.

We mainly performed our test on COCO val2017 dataset which contains 5000 images. We report all the results of our models in Table I.

TABLE I. COMPARISON ON THE COCO VAL2017 VALIDATION DATASET

Method	***Backbone***	***Pretrain*** [a]	***Input size***	***#Params***	***AP***	***AP^{50}***	***AP^{75}***	***AP^{M}***	***AP^{L}***	***AR***	***PCK***
HRNet_Orig	HRNet-W32	Y	256 x 192	28.5M	74.4	90.5	81.9	70.8	81.0	79.8	-
HRNet_Orig	HRNet-W32	N	256 x 192	28.5M	73.4	89.5	80.7	70.2	80.1	78.9	-
HRNet_Mod1	HRNet-W32_Mod1	N	256 x 192	97.9M	72.5	89.3	79.7	69.0	79.5	78.3	-
OpenPose	VGG-19	Y	368 x 368	209.0M	57.1	-	-	-	-	-	72.0
PoseNet	MobileNetV1	Y	257 x 257	5.0 M	36.0	-	-	-	-	-	63.9
Mediapipe	BlazePose	Y	-	-	35.5	-	-	-	-	-	44.6

[a] Pretrain = pretrain the backbone on the ImageNet classification task.

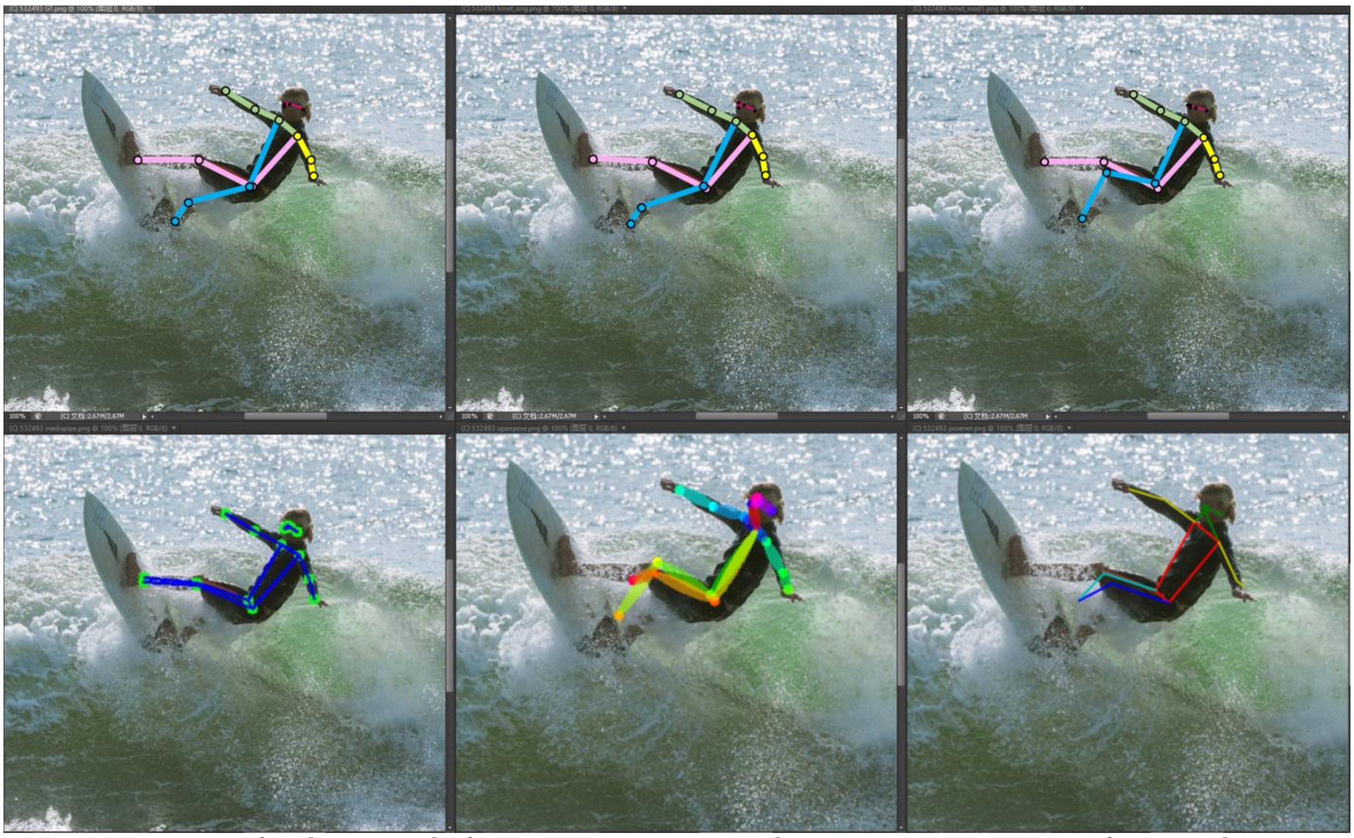


Figure 3. Comparison of qualitative results from our pose estimation implements on Image ID 532493 of COCO val2017 dataset. On the top row from the left to the right: Ground Truth, HRNet_Orig and HRNet_Mod1; on the bottom row from the left to the right: Mediapipe, OpenPose and PoseNet.

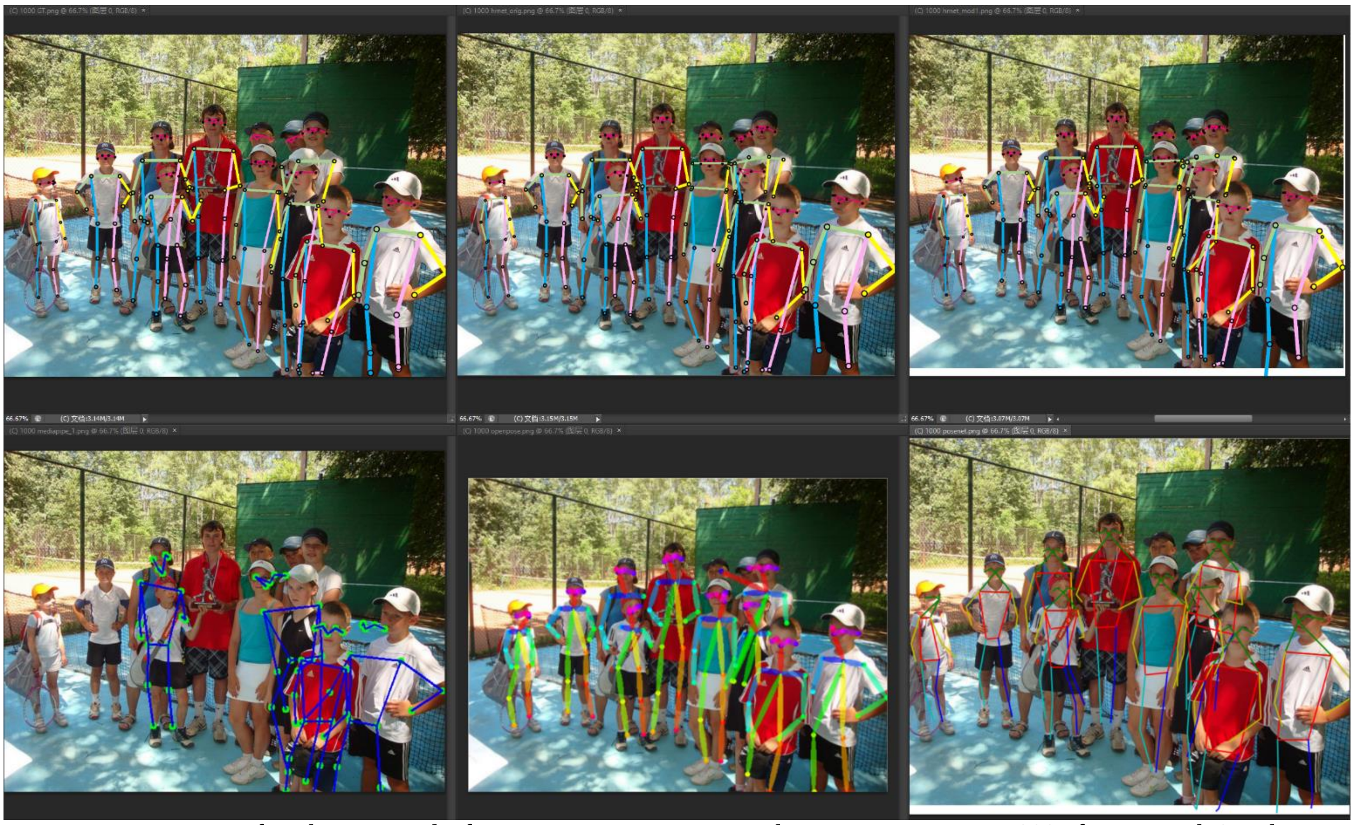


Figure 4. Comparison of qualitative results from our pose estimation implements on Image ID 1000 of COCO val2017 dataset. On the top row from the left to the right: Ground Truth, HRNet_Orig and HRNet_Mod1; on the bottom row from the left to the right: Mediapipe, OpenPose and PoseNet.

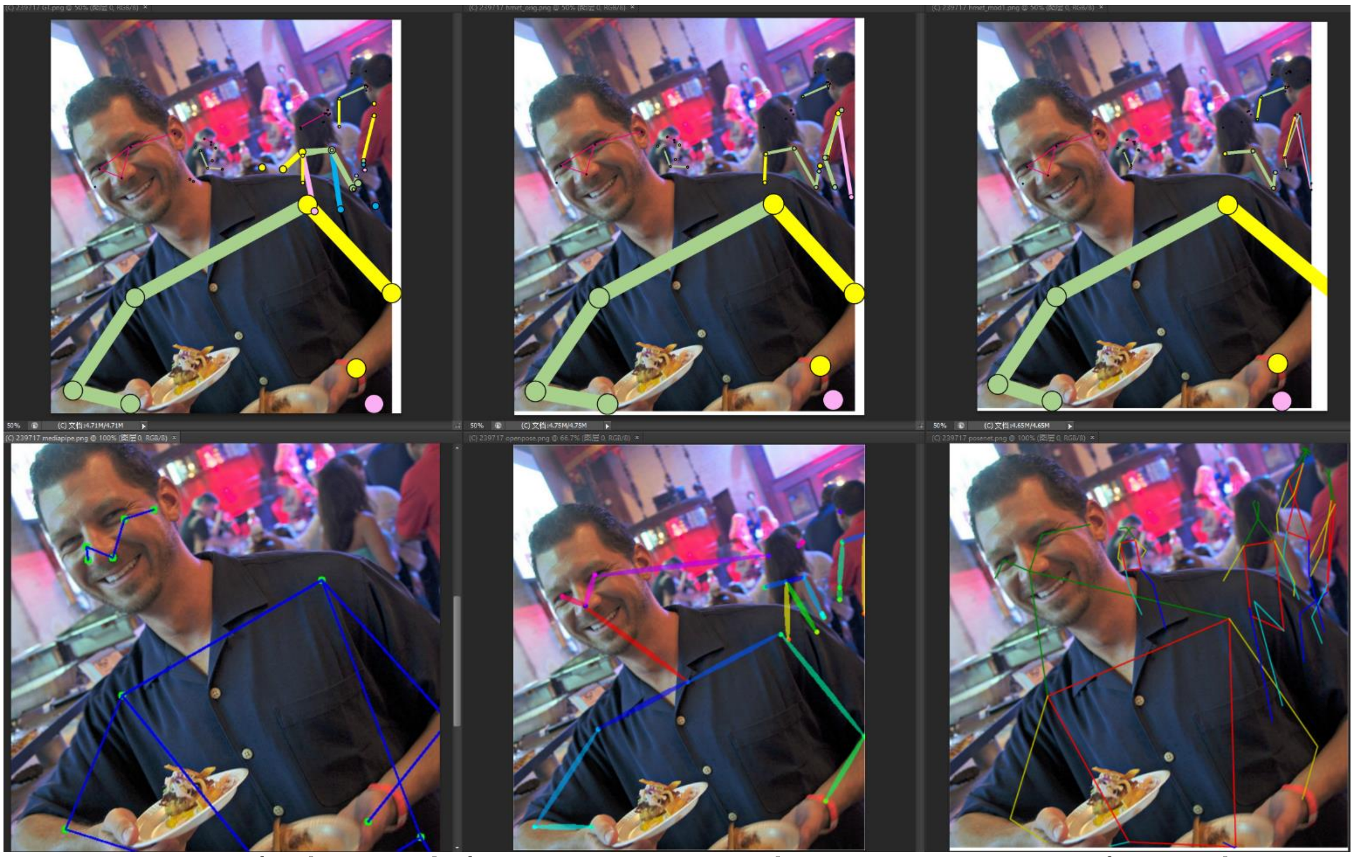


Figure 5. Comparison of qualitative results from our pose estimation implements on Image ID 239717 of COCO val2017 dataset. On the top row from the left to the right: Ground Truth, HRNet_Orig and HRNet_Mod1; on the bottom row from the left to the right: Mediapipe, OpenPose and PoseNet.

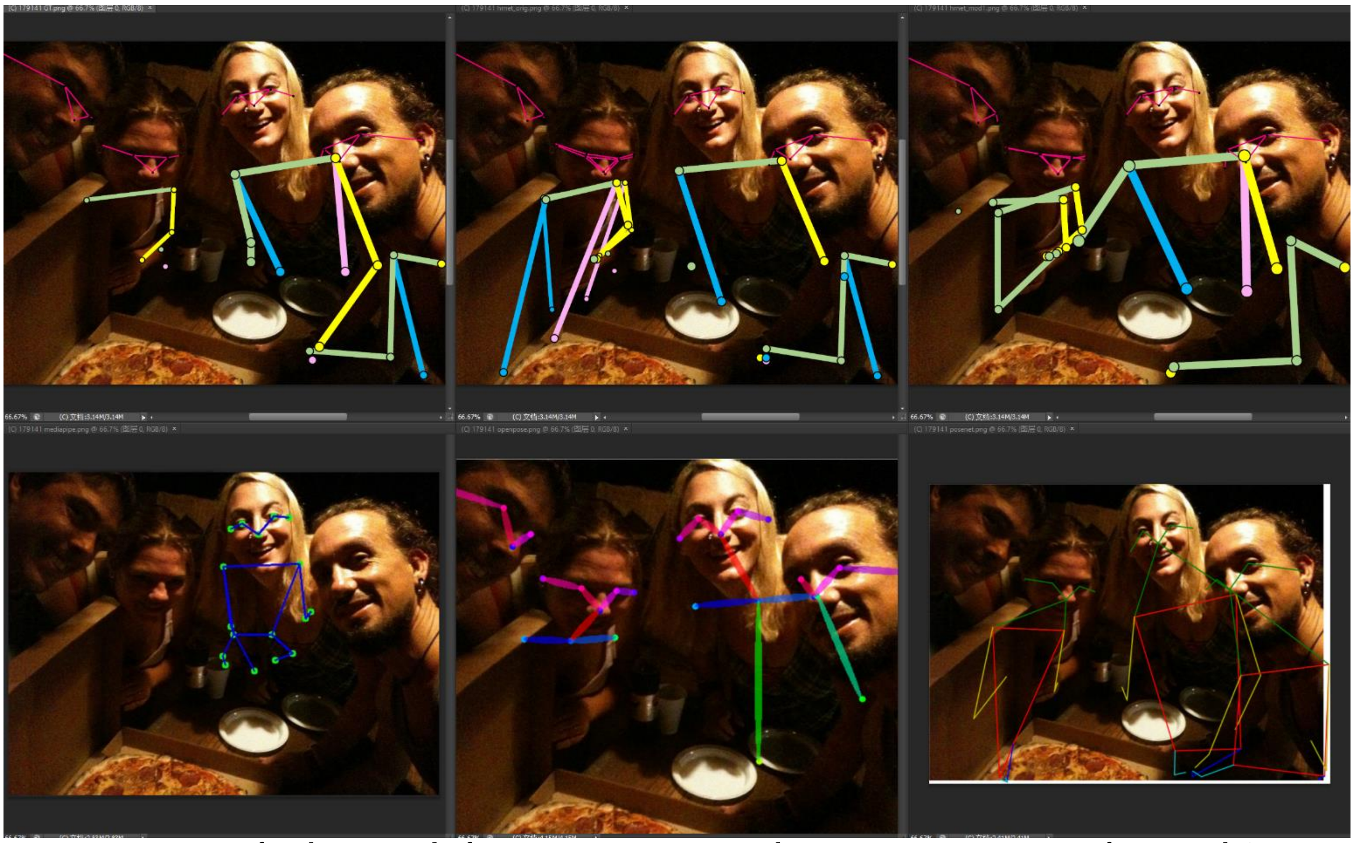


Figure 6. Comparison of qualitative results from our pose estimation implements on Image ID 179141 of COCO val2017 dataset. On the top row from the left to the right: Ground Truth, HRNet_Orig and HRNet_Mod1; on the bottom row from the left to the right: Mediapipe, OpenPose and PoseNet.

A. HRNet

We were able to load HRNet_Orig with the trained model parameters offered by the authors and reproduce the same results as in their paper [2]. For HRNet_Mod1, the mean Average Precision (mAP) is 0.9% lower than HRNet_Orig. Other scores show that AP^M is 1.2% lower than HRNet_Orig. we account these to the lack of one half and one eighth resolution branches in HRNet_Mod1, so its accuracy on medium sized human pose is affected.

In terms of speed, HRNet_Mod1 is slightly faster than HRNet_Orig. HRNet_Mod1 takes about 750 seconds to test the val2017 dataset while HRNet_Orig takes about 841 seconds. Training one epoch of the train2017 dataset takes HRNet_Mod1 about 29 minutes and HRNet_Orig about 32 minutes.

To summary the experiment with the HRNet architecture, our modification created a slightly faster but also slightly less accurate model.

B. Mediapipe

We downloaded and used Mediapipe's pretrained Pose Landmarker model from Google, and proceeded to test whether the model was able to run on the COCO val2017 dataset (containing 5000 images) properly. This model provided a mean Average Precision (mAP) of 0.3545 and a mean Percentage of Correct Keypoints (PCK) of 0.4451 on the entire validation set with the code written by us, which indicates the model's ability to detect and localize human body poses well in the real world. Mediapipe's model excels in single person detection, and using with videos to detect a single person's keypoint in the frame.

Finally, we performed a qualitative analysis by overlaying the predicted keypoints and skeleton on the subset of images to illustrate that the model can estimate the body pose and align with the ground truth annotations perfectly.

With regard to runtime performance, the MediaPipe Pose Landmarker ran the entire validation set in approximately 2 hours and 14 minutes with an average time of 1.61 seconds per image, showcasing its ability to deal with larger datasets swiftly.

In conclusion, we have presented the evaluation of the MediaPipe Pose Landmarker on human pose estimation tasks, where it is able to achieve competitive performance metrics on COCO dataset. Our evaluation results show that it is a robust, efficient, and reliable tool for keypoints detection and localization in the real-world images.

C. OpenPose

We downloaded and used the OpenPose model, and proceeded to test whether the model was able to run on the COCO val2017 dataset (containing 5000 images) properly. This model provided a mean Average Precision (mAP) of 0.59 and a mean Percentage of Correct Keypoints (PCK) of 0.94 on the entire validation set with the code written by us, which indicates the model's strong ability to detect and localize human body poses well in the real world. OpenPose excels in multi-person detection, and is highly effective in detecting keypoints for multiple individuals in an image.

D. PoseNet

We downloaded and used the PoseNet model, and proceeded to test whether the model was able to run on the COCO val2017 dataset (containing 5000 images) properly. This model provided a mean Average Precision (mAP) of 0.45 and a mean Percentage of Correct Keypoints (PCK) of 0.80 on the entire validation set with the code written by us, which indicates the model's capability to detect and localize human body poses well in the real world. PoseNet excels in single-person detection and is highly effective in detecting keypoints for a single individual in an image.

E. Subjective Evaluation

We conducted subjective evaluations on our keypoint detection results. We chose four images from the COCO val2017 dataset which we thought to be representative of the single person, multi-person, partial person and dark and occlusion scenarios, respectively. The comparison results are shown from Figure 3 to Figure 6, respectively. From the comparison we can see that Mediapipe encountered challenges in the multi-person scenario. And PoseNet missed a human face in the dark scene. A thing worth to notice is that HRNet is a top-down method, which requires a human detector to detect individual human in the image first then pass that result as input to the HRNet model. So like all top-down models, HRNet's performance is closely related to the performance of the human detector.

VII. Recommendations for Future Work

The following are some measures that can be taken to improve the concept of human pose estimation in the future.

1. Discriminating the different types of poses in detail instead of combining them into human poses.

2. Supervising people when it comes to finding the correct pose.

3. Finding various ways to move objects to the storage box.

a. Model Improvements: Exploring and using the latest pose estimation models, such as HRNet or PersonLab, to help detect the key points more accurately and robustly.

b. Multi-Person Pose Estimation: The system could be adapted to perform multi-person pose estimation, dealing with cases where more than one person is in the image, which would make it more applicable to the real world.

c. Real-Time Pose Estimation: Improving the real-time performance of both the model and the inference pipeline to make it usable for applications that require video or real-time poses like video analysis or human-computer interaction.

d. Domain Adaptation: Fine-tuning the model for sports or surveillance tasks would improve performance in those specific contexts. Transfer learning techniques can be applied to adapt the model to another domain with limited annotated data.

e. Uncertainty Estimation: Using uncertainty estimation methods would also measure how confident the 2D key point locations are, which could inform downstream applications in their decision-making.

f. Combining with Other Tasks: Integrating pose estimation with other computer vision tasks such as action recognition or human-object interaction analysis can help better understand human activities in images and videos.

However, by addressing those challenges and exploring new directions, we will have a system that can be further improved and extended to other applications in the computer vision domain.

References

[1] Z. Cao, G. Hidalgo, T. Simon, S.-E. Wei, and Y. Sheikh, "OpenPose: Realtime Multi-Person 2D Pose Estimation using Part Affinity Fields," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 43, no. 1, pp. 172-186, 2021.

[2] K. Sun, B. Xiao, D. Liu, and J. Wang, "Deep High-Resolution Representation Learning for Human Pose Estimation," in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2019, pp. 5693-5703.

[3] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: Common Objects in Context," in European Conference on Computer Vision (ECCV), 2014, pp. 740-755.

[4] COCO keypoint evaluation page https://cocodataset.org/#keypoints-eval

[5] Alejandro Newell, Kaiyu Yang, and Jia Deng, "Stacked hourglass networks for human pose estimation," in ECCV, pages 483–499, 2016.

[6] Yilun Chen, Zhicheng Wang, Yuxiang Peng, Zhiqiang Zhang, Gang Yu, and Jian Sun, "Cascaded pyramid network for multi-person pose estimation," CoRR, abs/1711.07319, 2017.

[7] Bin Xiao, Haiping Wu, and Yichen Wei, "Simple baselines for human pose estimation and tracking," in ECCV, pages 472–487, 2018.

[8] Ke Sun, Bin Xiao, Dong Liu, and Jingdong Wang, "deep-high-resolution-net.pytorch," https://github.com/leoxiaobin/deep-high-resolution-net.pytorch.

[9] Google Developers. "Pose landmark detection guide." Google AI Edge.

[10] Google Developers. "Pose landmark detection guide for Python." Google AI Edge.

[11] Xu et al. 'GHUM \GHUML: Generative 3D Human Shape and Articulated Pose Models'. CVPR 2020.