Python les-materialen

# Overview

This Colab helps to create and validate a training set for the k-NN classifier described in the MediaPipe [Pose Classification](https://google.github.io/mediapipe/solutions/pose_classification.html) soultion, test it on an arbitrary video, export to a CSV and then use it in the [ML Kit sample app](https://developers.google.com/ml-kit/vision/pose-detection/classifying-poses#4_integrate_with_the_ml_kit_quickstart_app).

# Step 0: Start Colab

Connect the Colab to hosted Python3 runtime (check top-right corner) and then install required dependencies.

!pip install pillow==8.1.0  
!pip install matplotlib==3.3.4  
!pip install numpy==1.19.3  
!pip install opencv-python==4.5.1.48  
!pip install tqdm==4.56.0  
!pip install requests==2.25.1  
  
!pip install mediapipe==0.8.3

# Codebase

## Commons

from matplotlib import pyplot as plt  
  
  
def show\_image(img, figsize=(10, 10)):  
 """Shows output PIL image."""  
 plt.figure(figsize=figsize)  
 plt.imshow(img)  
 plt.show()

## Pose embedding

class FullBodyPoseEmbedder(object):  
 """Converts 3D pose landmarks into 3D embedding."""  
  
 def \_\_init\_\_(self, torso\_size\_multiplier=2.5):  
 # Multiplier to apply to the torso to get minimal body size.  
 self.\_torso\_size\_multiplier = torso\_size\_multiplier  
  
 # Names of the landmarks as they appear in the prediction.  
 self.\_landmark\_names = [  
 'nose',  
 'left\_eye\_inner', 'left\_eye', 'left\_eye\_outer',  
 'right\_eye\_inner', 'right\_eye', 'right\_eye\_outer',  
 'left\_ear', 'right\_ear',  
 'mouth\_left', 'mouth\_right',  
 'left\_shoulder', 'right\_shoulder',  
 'left\_elbow', 'right\_elbow',  
 'left\_wrist', 'right\_wrist',  
 'left\_pinky\_1', 'right\_pinky\_1',  
 'left\_index\_1', 'right\_index\_1',  
 'left\_thumb\_2', 'right\_thumb\_2',  
 'left\_hip', 'right\_hip',  
 'left\_knee', 'right\_knee',  
 'left\_ankle', 'right\_ankle',  
 'left\_heel', 'right\_heel',  
 'left\_foot\_index', 'right\_foot\_index',  
 ]  
  
 def \_\_call\_\_(self, landmarks):  
 """Normalizes pose landmarks and converts to embedding  
   
 Args:  
 landmarks - NumPy array with 3D landmarks of shape (N, 3).  
  
 Result:  
 Numpy array with pose embedding of shape (M, 3) where `M` is the number of  
 pairwise distances defined in `\_get\_pose\_distance\_embedding`.  
 """  
 assert landmarks.shape[0] == len(self.\_landmark\_names), 'Unexpected number of landmarks: {}'.format(landmarks.shape[0])  
  
 # Get pose landmarks.  
 landmarks = np.copy(landmarks)  
  
 # Normalize landmarks.  
 landmarks = self.\_normalize\_pose\_landmarks(landmarks)  
  
 # Get embedding.  
 embedding = self.\_get\_pose\_distance\_embedding(landmarks)  
  
 return embedding  
  
 def \_normalize\_pose\_landmarks(self, landmarks):  
 """Normalizes landmarks translation and scale."""  
 landmarks = np.copy(landmarks)  
  
 # Normalize translation.  
 pose\_center = self.\_get\_pose\_center(landmarks)  
 landmarks -= pose\_center  
  
 # Normalize scale.  
 pose\_size = self.\_get\_pose\_size(landmarks, self.\_torso\_size\_multiplier)  
 landmarks /= pose\_size  
 # Multiplication by 100 is not required, but makes it eaasier to debug.  
 landmarks \*= 100  
  
 return landmarks  
  
 def \_get\_pose\_center(self, landmarks):  
 """Calculates pose center as point between hips."""  
 left\_hip = landmarks[self.\_landmark\_names.index('left\_hip')]  
 right\_hip = landmarks[self.\_landmark\_names.index('right\_hip')]  
 center = (left\_hip + right\_hip) \* 0.5  
 return center  
  
 def \_get\_pose\_size(self, landmarks, torso\_size\_multiplier):  
 """Calculates pose size.  
   
 It is the maximum of two values:  
 \* Torso size multiplied by `torso\_size\_multiplier`  
 \* Maximum distance from pose center to any pose landmark  
 """  
 # This approach uses only 2D landmarks to compute pose size.  
 landmarks = landmarks[:, :2]  
  
 # Hips center.  
 left\_hip = landmarks[self.\_landmark\_names.index('left\_hip')]  
 right\_hip = landmarks[self.\_landmark\_names.index('right\_hip')]  
 hips = (left\_hip + right\_hip) \* 0.5  
  
 # Shoulders center.  
 left\_shoulder = landmarks[self.\_landmark\_names.index('left\_shoulder')]  
 right\_shoulder = landmarks[self.\_landmark\_names.index('right\_shoulder')]  
 shoulders = (left\_shoulder + right\_shoulder) \* 0.5  
  
 # Torso size as the minimum body size.  
 torso\_size = np.linalg.norm(shoulders - hips)  
  
 # Max dist to pose center.  
 pose\_center = self.\_get\_pose\_center(landmarks)  
 max\_dist = np.max(np.linalg.norm(landmarks - pose\_center, axis=1))  
  
 return max(torso\_size \* torso\_size\_multiplier, max\_dist)  
  
 def \_get\_pose\_distance\_embedding(self, landmarks):  
 """Converts pose landmarks into 3D embedding.  
  
 We use several pairwise 3D distances to form pose embedding. All distances  
 include X and Y components with sign. We differnt types of pairs to cover  
 different pose classes. Feel free to remove some or add new.  
   
 Args:  
 landmarks - NumPy array with 3D landmarks of shape (N, 3).  
  
 Result:  
 Numpy array with pose embedding of shape (M, 3) where `M` is the number of  
 pairwise distances.  
 """  
 embedding = np.array([  
 # One joint.  
  
 self.\_get\_distance(  
 self.\_get\_average\_by\_names(landmarks, 'left\_hip', 'right\_hip'),  
 self.\_get\_average\_by\_names(landmarks, 'left\_shoulder', 'right\_shoulder')),  
  
 self.\_get\_distance\_by\_names(landmarks, 'left\_shoulder', 'left\_elbow'),  
 self.\_get\_distance\_by\_names(landmarks, 'right\_shoulder', 'right\_elbow'),  
  
 self.\_get\_distance\_by\_names(landmarks, 'left\_elbow', 'left\_wrist'),  
 self.\_get\_distance\_by\_names(landmarks, 'right\_elbow', 'right\_wrist'),  
  
 self.\_get\_distance\_by\_names(landmarks, 'left\_hip', 'left\_knee'),  
 self.\_get\_distance\_by\_names(landmarks, 'right\_hip', 'right\_knee'),  
  
 self.\_get\_distance\_by\_names(landmarks, 'left\_knee', 'left\_ankle'),  
 self.\_get\_distance\_by\_names(landmarks, 'right\_knee', 'right\_ankle'),  
  
 # Two joints.  
  
 self.\_get\_distance\_by\_names(landmarks, 'left\_shoulder', 'left\_wrist'),  
 self.\_get\_distance\_by\_names(landmarks, 'right\_shoulder', 'right\_wrist'),  
  
 self.\_get\_distance\_by\_names(landmarks, 'left\_hip', 'left\_ankle'),  
 self.\_get\_distance\_by\_names(landmarks, 'right\_hip', 'right\_ankle'),  
  
 # Four joints.  
  
 self.\_get\_distance\_by\_names(landmarks, 'left\_hip', 'left\_wrist'),  
 self.\_get\_distance\_by\_names(landmarks, 'right\_hip', 'right\_wrist'),  
  
 # Five joints.  
  
 self.\_get\_distance\_by\_names(landmarks, 'left\_shoulder', 'left\_ankle'),  
 self.\_get\_distance\_by\_names(landmarks, 'right\_shoulder', 'right\_ankle'),  
   
 self.\_get\_distance\_by\_names(landmarks, 'left\_hip', 'left\_wrist'),  
 self.\_get\_distance\_by\_names(landmarks, 'right\_hip', 'right\_wrist'),  
  
 # Cross body.  
  
 self.\_get\_distance\_by\_names(landmarks, 'left\_elbow', 'right\_elbow'),  
 self.\_get\_distance\_by\_names(landmarks, 'left\_knee', 'right\_knee'),  
  
 self.\_get\_distance\_by\_names(landmarks, 'left\_wrist', 'right\_wrist'),  
 self.\_get\_distance\_by\_names(landmarks, 'left\_ankle', 'right\_ankle'),  
  
 # Body bent direction.  
  
 # self.\_get\_distance(  
 # self.\_get\_average\_by\_names(landmarks, 'left\_wrist', 'left\_ankle'),  
 # landmarks[self.\_landmark\_names.index('left\_hip')]),  
 # self.\_get\_distance(  
 # self.\_get\_average\_by\_names(landmarks, 'right\_wrist', 'right\_ankle'),  
 # landmarks[self.\_landmark\_names.index('right\_hip')]),  
 ])  
  
 return embedding  
  
 def \_get\_average\_by\_names(self, landmarks, name\_from, name\_to):  
 lmk\_from = landmarks[self.\_landmark\_names.index(name\_from)]  
 lmk\_to = landmarks[self.\_landmark\_names.index(name\_to)]  
 return (lmk\_from + lmk\_to) \* 0.5  
  
 def \_get\_distance\_by\_names(self, landmarks, name\_from, name\_to):  
 lmk\_from = landmarks[self.\_landmark\_names.index(name\_from)]  
 lmk\_to = landmarks[self.\_landmark\_names.index(name\_to)]  
 return self.\_get\_distance(lmk\_from, lmk\_to)  
  
 def \_get\_distance(self, lmk\_from, lmk\_to):  
 return lmk\_to - lmk\_from

## Pose classification

class PoseSample(object):  
  
 def \_\_init\_\_(self, name, landmarks, class\_name, embedding):  
 self.name = name  
 self.landmarks = landmarks  
 self.class\_name = class\_name  
   
 self.embedding = embedding  
  
  
class PoseSampleOutlier(object):  
  
 def \_\_init\_\_(self, sample, detected\_class, all\_classes):  
 self.sample = sample  
 self.detected\_class = detected\_class  
 self.all\_classes = all\_classes

import csv  
import numpy as np  
import os  
  
class PoseClassifier(object):  
 """Classifies pose landmarks."""  
  
 def \_\_init\_\_(self,  
 pose\_samples\_folder,  
 pose\_embedder,  
 file\_extension='csv',  
 file\_separator=',',  
 n\_landmarks=33,  
 n\_dimensions=3,  
 top\_n\_by\_max\_distance=30,  
 top\_n\_by\_mean\_distance=10,  
 axes\_weights=(1., 1., 0.2)):  
 self.\_pose\_embedder = pose\_embedder  
 self.\_n\_landmarks = n\_landmarks  
 self.\_n\_dimensions = n\_dimensions  
 self.\_top\_n\_by\_max\_distance = top\_n\_by\_max\_distance  
 self.\_top\_n\_by\_mean\_distance = top\_n\_by\_mean\_distance  
 self.\_axes\_weights = axes\_weights  
  
 self.\_pose\_samples = self.\_load\_pose\_samples(pose\_samples\_folder,  
 file\_extension,  
 file\_separator,  
 n\_landmarks,  
 n\_dimensions,  
 pose\_embedder)  
  
 def \_load\_pose\_samples(self,  
 pose\_samples\_folder,  
 file\_extension,  
 file\_separator,  
 n\_landmarks,  
 n\_dimensions,  
 pose\_embedder):  
 """Loads pose samples from a given folder.  
   
 Required folder structure:  
 neutral\_standing.csv  
 pushups\_down.csv  
 pushups\_up.csv  
 squats\_down.csv  
 ...  
  
 Required CSV structure:  
 sample\_00001,x1,y1,z1,x2,y2,z2,....  
 sample\_00002,x1,y1,z1,x2,y2,z2,....  
 ...  
 """  
 # Each file in the folder represents one pose class.  
 file\_names = [name for name in os.listdir(pose\_samples\_folder) if name.endswith(file\_extension)]  
  
 pose\_samples = []  
 for file\_name in file\_names:  
 # Use file name as pose class name.  
 class\_name = file\_name[:-(len(file\_extension) + 1)]  
   
 # Parse CSV.  
 with open(os.path.join(pose\_samples\_folder, file\_name)) as csv\_file:  
 csv\_reader = csv.reader(csv\_file, delimiter=file\_separator)  
 for row in csv\_reader:  
 assert len(row) == n\_landmarks \* n\_dimensions + 1, 'Wrong number of values: {}'.format(len(row))  
 landmarks = np.array(row[1:], np.float32).reshape([n\_landmarks, n\_dimensions])  
 pose\_samples.append(PoseSample(  
 name=row[0],  
 landmarks=landmarks,  
 class\_name=class\_name,  
 embedding=pose\_embedder(landmarks),  
 ))  
  
 return pose\_samples  
  
 def find\_pose\_sample\_outliers(self):  
 """Classifies each sample against the entire database."""  
 # Find outliers in target poses  
 outliers = []  
 for sample in self.\_pose\_samples:  
 # Find nearest poses for the target one.  
 pose\_landmarks = sample.landmarks.copy()  
 pose\_classification = self.\_\_call\_\_(pose\_landmarks)  
 class\_names = [class\_name for class\_name, count in pose\_classification.items() if count == max(pose\_classification.values())]  
  
 # Sample is an outlier if nearest poses have different class or more than  
 # one pose class is detected as nearest.  
 if sample.class\_name not in class\_names or len(class\_names) != 1:  
 outliers.append(PoseSampleOutlier(sample, class\_names, pose\_classification))  
  
 return outliers  
  
 def \_\_call\_\_(self, pose\_landmarks):  
 """Classifies given pose.  
  
 Classification is done in two stages:  
 \* First we pick top-N samples by MAX distance. It allows to remove samples  
 that are almost the same as given pose, but has few joints bent in the  
 other direction.  
 \* Then we pick top-N samples by MEAN distance. After outliers are removed  
 on a previous step, we can pick samples that are closes on average.  
   
 Args:  
 pose\_landmarks: NumPy array with 3D landmarks of shape (N, 3).  
  
 Returns:  
 Dictionary with count of nearest pose samples from the database. Sample:  
 {  
 'pushups\_down': 8,  
 'pushups\_up': 2,  
 }  
 """  
 # Check that provided and target poses have the same shape.  
 assert pose\_landmarks.shape == (self.\_n\_landmarks, self.\_n\_dimensions), 'Unexpected shape: {}'.format(pose\_landmarks.shape)  
  
 # Get given pose embedding.  
 pose\_embedding = self.\_pose\_embedder(pose\_landmarks)  
 flipped\_pose\_embedding = self.\_pose\_embedder(pose\_landmarks \* np.array([-1, 1, 1]))  
  
 # Filter by max distance.  
 #  
 # That helps to remove outliers - poses that are almost the same as the  
 # given one, but has one joint bent into another direction and actually  
 # represnt a different pose class.  
 max\_dist\_heap = []  
 for sample\_idx, sample in enumerate(self.\_pose\_samples):  
 max\_dist = min(  
 np.max(np.abs(sample.embedding - pose\_embedding) \* self.\_axes\_weights),  
 np.max(np.abs(sample.embedding - flipped\_pose\_embedding) \* self.\_axes\_weights),  
 )  
 max\_dist\_heap.append([max\_dist, sample\_idx])  
  
 max\_dist\_heap = sorted(max\_dist\_heap, key=lambda x: x[0])  
 max\_dist\_heap = max\_dist\_heap[:self.\_top\_n\_by\_max\_distance]  
  
 # Filter by mean distance.  
 #  
 # After removing outliers we can find the nearest pose by mean distance.  
 mean\_dist\_heap = []  
 for \_, sample\_idx in max\_dist\_heap:  
 sample = self.\_pose\_samples[sample\_idx]  
 mean\_dist = min(  
 np.mean(np.abs(sample.embedding - pose\_embedding) \* self.\_axes\_weights),  
 np.mean(np.abs(sample.embedding - flipped\_pose\_embedding) \* self.\_axes\_weights),  
 )  
 mean\_dist\_heap.append([mean\_dist, sample\_idx])  
  
 mean\_dist\_heap = sorted(mean\_dist\_heap, key=lambda x: x[0])  
 mean\_dist\_heap = mean\_dist\_heap[:self.\_top\_n\_by\_mean\_distance]  
  
 # Collect results into map: (class\_name -> n\_samples)  
 class\_names = [self.\_pose\_samples[sample\_idx].class\_name for \_, sample\_idx in mean\_dist\_heap]  
 result = {class\_name: class\_names.count(class\_name) for class\_name in set(class\_names)}  
  
 return result

## Classification smoothing

class EMADictSmoothing(object):  
 """Smoothes pose classification."""  
  
 def \_\_init\_\_(self, window\_size=10, alpha=0.2):  
 self.\_window\_size = window\_size  
 self.\_alpha = alpha  
  
 self.\_data\_in\_window = []  
  
 def \_\_call\_\_(self, data):  
 """Smoothes given pose classification.  
  
 Smoothing is done by computing Exponential Moving Average for every pose  
 class observed in the given time window. Missed pose classes arre replaced  
 with 0.  
   
 Args:  
 data: Dictionary with pose classification. Sample:  
 {  
 'pushups\_down': 8,  
 'pushups\_up': 2,  
 }  
  
 Result:  
 Dictionary in the same format but with smoothed and float instead of  
 integer values. Sample:  
 {  
 'pushups\_down': 8.3,  
 'pushups\_up': 1.7,  
 }  
 """  
 # Add new data to the beginning of the window for simpler code.  
 self.\_data\_in\_window.insert(0, data)  
 self.\_data\_in\_window = self.\_data\_in\_window[:self.\_window\_size]  
  
 # Get all keys.  
 keys = set([key for data in self.\_data\_in\_window for key, \_ in data.items()])  
  
 # Get smoothed values.  
 smoothed\_data = dict()  
 for key in keys:  
 factor = 1.0  
 top\_sum = 0.0  
 bottom\_sum = 0.0  
 for data in self.\_data\_in\_window:  
 value = data[key] if key in data else 0.0  
  
 top\_sum += factor \* value  
 bottom\_sum += factor  
  
 # Update factor.  
 factor \*= (1.0 - self.\_alpha)  
  
 smoothed\_data[key] = top\_sum / bottom\_sum  
  
 return smoothed\_data

## Repetition counter

class RepetitionCounter(object):  
 """Counts number of repetitions of given target pose class."""  
  
 def \_\_init\_\_(self, class\_name, enter\_threshold=6, exit\_threshold=4):  
 self.\_class\_name = class\_name  
  
 # If pose counter passes given threshold, then we enter the pose.  
 self.\_enter\_threshold = enter\_threshold  
 self.\_exit\_threshold = exit\_threshold  
  
 # Either we are in given pose or not.  
 self.\_pose\_entered = False  
  
 # Number of times we exited the pose.  
 self.\_n\_repeats = 0  
  
 @property  
 def n\_repeats(self):  
 return self.\_n\_repeats  
  
 def \_\_call\_\_(self, pose\_classification):  
 """Counts number of repetitions happend until given frame.  
  
 We use two thresholds. First you need to go above the higher one to enter  
 the pose, and then you need to go below the lower one to exit it. Difference  
 between the thresholds makes it stable to prediction jittering (which will  
 cause wrong counts in case of having only one threshold).  
   
 Args:  
 pose\_classification: Pose classification dictionary on current frame.  
 Sample:  
 {  
 'pushups\_down': 8.3,  
 'pushups\_up': 1.7,  
 }  
  
 Returns:  
 Integer counter of repetitions.  
 """  
 # Get pose confidence.  
 pose\_confidence = 0.0  
 if self.\_class\_name in pose\_classification:  
 pose\_confidence = pose\_classification[self.\_class\_name]  
  
 # On the very first frame or if we were out of the pose, just check if we  
 # entered it on this frame and update the state.  
 if not self.\_pose\_entered:  
 self.\_pose\_entered = pose\_confidence > self.\_enter\_threshold  
 return self.\_n\_repeats  
  
 # If we were in the pose and are exiting it, then increase the counter and  
 # update the state.  
 if pose\_confidence < self.\_exit\_threshold:  
 self.\_n\_repeats += 1  
 self.\_pose\_entered = False  
  
 return self.\_n\_repeats

## Classification visualizer

import io  
from PIL import Image  
from PIL import ImageFont  
from PIL import ImageDraw  
import requests  
  
class PoseClassificationVisualizer(object):  
 """Keeps track of claassifcations for every frame and renders them."""  
  
 def \_\_init\_\_(self,  
 class\_name,  
 plot\_location\_x=0.05,  
 plot\_location\_y=0.05,  
 plot\_max\_width=0.4,  
 plot\_max\_height=0.4,  
 plot\_figsize=(9, 4),  
 plot\_x\_max=None,  
 plot\_y\_max=None,  
 counter\_location\_x=0.85,  
 counter\_location\_y=0.05,  
 counter\_font\_path='https://github.com/googlefonts/roboto/blob/main/src/hinted/Roboto-Regular.ttf?raw=true',  
 counter\_font\_color='red',  
 counter\_font\_size=0.15):  
 self.\_class\_name = class\_name  
 self.\_plot\_location\_x = plot\_location\_x  
 self.\_plot\_location\_y = plot\_location\_y  
 self.\_plot\_max\_width = plot\_max\_width  
 self.\_plot\_max\_height = plot\_max\_height  
 self.\_plot\_figsize = plot\_figsize  
 self.\_plot\_x\_max = plot\_x\_max  
 self.\_plot\_y\_max = plot\_y\_max  
 self.\_counter\_location\_x = counter\_location\_x  
 self.\_counter\_location\_y = counter\_location\_y  
 self.\_counter\_font\_path = counter\_font\_path  
 self.\_counter\_font\_color = counter\_font\_color  
 self.\_counter\_font\_size = counter\_font\_size  
  
 self.\_counter\_font = None  
  
 self.\_pose\_classification\_history = []  
 self.\_pose\_classification\_filtered\_history = []  
  
 def \_\_call\_\_(self,  
 frame,  
 pose\_classification,  
 pose\_classification\_filtered,  
 repetitions\_count):  
 """Renders pose classifcation and counter until given frame."""  
 # Extend classification history.  
 self.\_pose\_classification\_history.append(pose\_classification)  
 self.\_pose\_classification\_filtered\_history.append(pose\_classification\_filtered)  
  
 # Output frame with classification plot and counter.  
 output\_img = Image.fromarray(frame)  
  
 output\_width = output\_img.size[0]  
 output\_height = output\_img.size[1]  
  
 # Draw the plot.  
 img = self.\_plot\_classification\_history(output\_width, output\_height)  
 img.thumbnail((int(output\_width \* self.\_plot\_max\_width),  
 int(output\_height \* self.\_plot\_max\_height)),  
 Image.ANTIALIAS)  
 output\_img.paste(img,  
 (int(output\_width \* self.\_plot\_location\_x),  
 int(output\_height \* self.\_plot\_location\_y)))  
  
 # Draw the count.  
 output\_img\_draw = ImageDraw.Draw(output\_img)  
 if self.\_counter\_font is None:  
 font\_size = int(output\_height \* self.\_counter\_font\_size)  
 font\_request = requests.get(self.\_counter\_font\_path, allow\_redirects=True)  
 self.\_counter\_font = ImageFont.truetype(io.BytesIO(font\_request.content), size=font\_size)  
 output\_img\_draw.text((output\_width \* self.\_counter\_location\_x,  
 output\_height \* self.\_counter\_location\_y),  
 str(repetitions\_count),  
 font=self.\_counter\_font,  
 fill=self.\_counter\_font\_color)  
  
 return output\_img  
  
 def \_plot\_classification\_history(self, output\_width, output\_height):  
 fig = plt.figure(figsize=self.\_plot\_figsize)  
  
 for classification\_history in [self.\_pose\_classification\_history,  
 self.\_pose\_classification\_filtered\_history]:  
 y = []  
 for classification in classification\_history:  
 if classification is None:  
 y.append(None)  
 elif self.\_class\_name in classification:  
 y.append(classification[self.\_class\_name])  
 else:  
 y.append(0)  
 plt.plot(y, linewidth=7)  
  
 plt.grid(axis='y', alpha=0.75)  
 plt.xlabel('Frame')  
 plt.ylabel('Confidence')  
 plt.title('Classification history for `{}`'.format(self.\_class\_name))  
 plt.legend(loc='upper right')  
  
 if self.\_plot\_y\_max is not None:  
 plt.ylim(top=self.\_plot\_y\_max)  
 if self.\_plot\_x\_max is not None:  
 plt.xlim(right=self.\_plot\_x\_max)  
  
 # Convert plot to image.  
 buf = io.BytesIO()  
 dpi = min(  
 output\_width \* self.\_plot\_max\_width / float(self.\_plot\_figsize[0]),  
 output\_height \* self.\_plot\_max\_height / float(self.\_plot\_figsize[1]))  
 fig.savefig(buf, dpi=dpi)  
 buf.seek(0)  
 img = Image.open(buf)  
 plt.close()  
  
 return img

## Bootstrap helper

import cv2  
from matplotlib import pyplot as plt  
import numpy as np  
import os  
from PIL import Image  
import sys  
import tqdm  
  
from mediapipe.python.solutions import drawing\_utils as mp\_drawing  
from mediapipe.python.solutions import pose as mp\_pose  
  
  
class BootstrapHelper(object):  
 """Helps to bootstrap images and filter pose samples for classification."""  
  
 def \_\_init\_\_(self,  
 images\_in\_folder,  
 images\_out\_folder,  
 csvs\_out\_folder):  
 self.\_images\_in\_folder = images\_in\_folder  
 self.\_images\_out\_folder = images\_out\_folder  
 self.\_csvs\_out\_folder = csvs\_out\_folder  
  
 # Get list of pose classes and print image statistics.  
 self.\_pose\_class\_names = sorted([n for n in os.listdir(self.\_images\_in\_folder) if not n.startswith('.')])  
   
 def bootstrap(self, per\_pose\_class\_limit=None):  
 """Bootstraps images in a given folder.  
   
 Required image in folder (same use for image out folder):  
 pushups\_up/  
 image\_001.jpg  
 image\_002.jpg  
 ...  
 pushups\_down/  
 image\_001.jpg  
 image\_002.jpg  
 ...  
 ...  
  
 Produced CSVs out folder:  
 pushups\_up.csv  
 pushups\_down.csv  
  
 Produced CSV structure with pose 3D landmarks:  
 sample\_00001,x1,y1,z1,x2,y2,z2,....  
 sample\_00002,x1,y1,z1,x2,y2,z2,....  
 """  
 # Create output folder for CVSs.  
 if not os.path.exists(self.\_csvs\_out\_folder):  
 os.makedirs(self.\_csvs\_out\_folder)  
  
 for pose\_class\_name in self.\_pose\_class\_names:  
 print('Bootstrapping ', pose\_class\_name, file=sys.stderr)  
  
 # Paths for the pose class.  
 images\_in\_folder = os.path.join(self.\_images\_in\_folder, pose\_class\_name)  
 images\_out\_folder = os.path.join(self.\_images\_out\_folder, pose\_class\_name)  
 csv\_out\_path = os.path.join(self.\_csvs\_out\_folder, pose\_class\_name + '.csv')  
 if not os.path.exists(images\_out\_folder):  
 os.makedirs(images\_out\_folder)  
  
 with open(csv\_out\_path, 'w') as csv\_out\_file:  
 csv\_out\_writer = csv.writer(csv\_out\_file, delimiter=',', quoting=csv.QUOTE\_MINIMAL)  
 # Get list of images.  
 image\_names = sorted([n for n in os.listdir(images\_in\_folder) if not n.startswith('.')])  
 if per\_pose\_class\_limit is not None:  
 image\_names = image\_names[:per\_pose\_class\_limit]  
  
 # Bootstrap every image.  
 for image\_name in tqdm.tqdm(image\_names):  
 # Load image.  
 input\_frame = cv2.imread(os.path.join(images\_in\_folder, image\_name))  
 input\_frame = cv2.cvtColor(input\_frame, cv2.COLOR\_BGR2RGB)  
  
 # Initialize fresh pose tracker and run it.  
 with mp\_pose.Pose(upper\_body\_only=False) as pose\_tracker:  
 result = pose\_tracker.process(image=input\_frame)  
 pose\_landmarks = result.pose\_landmarks  
  
 # Save image with pose prediction (if pose was detected).  
 output\_frame = input\_frame.copy()  
 if pose\_landmarks is not None:  
 mp\_drawing.draw\_landmarks(  
 image=output\_frame,  
 landmark\_list=pose\_landmarks,  
 connections=mp\_pose.POSE\_CONNECTIONS)  
 output\_frame = cv2.cvtColor(output\_frame, cv2.COLOR\_RGB2BGR)  
 cv2.imwrite(os.path.join(images\_out\_folder, image\_name), output\_frame)  
   
 # Save landmarks if pose was detected.  
 if pose\_landmarks is not None:  
 # Get landmarks.  
 frame\_height, frame\_width = output\_frame.shape[0], output\_frame.shape[1]  
 pose\_landmarks = np.array(  
 [[lmk.x \* frame\_width, lmk.y \* frame\_height, lmk.z \* frame\_width]  
 for lmk in pose\_landmarks.landmark],  
 dtype=np.float32)  
 assert pose\_landmarks.shape == (33, 3), 'Unexpected landmarks shape: {}'.format(pose\_landmarks.shape)  
 csv\_out\_writer.writerow([image\_name] + pose\_landmarks.flatten().astype(np.str).tolist())  
  
 # Draw XZ projection and concatenate with the image.  
 projection\_xz = self.\_draw\_xz\_projection(  
 output\_frame=output\_frame, pose\_landmarks=pose\_landmarks)  
 output\_frame = np.concatenate((output\_frame, projection\_xz), axis=1)  
  
 def \_draw\_xz\_projection(self, output\_frame, pose\_landmarks, r=0.5, color='red'):  
 frame\_height, frame\_width = output\_frame.shape[0], output\_frame.shape[1]  
 img = Image.new('RGB', (frame\_width, frame\_height), color='white')  
  
 if pose\_landmarks is None:  
 return np.asarray(img)  
  
 # Scale radius according to the image width.  
 r \*= frame\_width \* 0.01  
  
 draw = ImageDraw.Draw(img)  
 for idx\_1, idx\_2 in mp\_pose.POSE\_CONNECTIONS:  
 # Flip Z and move hips center to the center of the image.  
 x1, y1, z1 = pose\_landmarks[idx\_1] \* [1, 1, -1] + [0, 0, frame\_height \* 0.5]  
 x2, y2, z2 = pose\_landmarks[idx\_2] \* [1, 1, -1] + [0, 0, frame\_height \* 0.5]  
  
 draw.ellipse([x1 - r, z1 - r, x1 + r, z1 + r], fill=color)  
 draw.ellipse([x2 - r, z2 - r, x2 + r, z2 + r], fill=color)  
 draw.line([x1, z1, x2, z2], width=int(r), fill=color)  
  
 return np.asarray(img)  
  
 def align\_images\_and\_csvs(self, print\_removed\_items=False):  
 """Makes sure that image folders and CSVs have the same sample.  
   
 Leaves only intersetion of samples in both image folders and CSVs.  
 """  
 for pose\_class\_name in self.\_pose\_class\_names:  
 # Paths for the pose class.  
 images\_out\_folder = os.path.join(self.\_images\_out\_folder, pose\_class\_name)  
 csv\_out\_path = os.path.join(self.\_csvs\_out\_folder, pose\_class\_name + '.csv')  
  
 # Read CSV into memory.  
 rows = []  
 with open(csv\_out\_path) as csv\_out\_file:  
 csv\_out\_reader = csv.reader(csv\_out\_file, delimiter=',')  
 for row in csv\_out\_reader:  
 rows.append(row)  
  
 # Image names left in CSV.  
 image\_names\_in\_csv = []  
  
 # Re-write the CSV removing lines without corresponding images.  
 with open(csv\_out\_path, 'w') as csv\_out\_file:  
 csv\_out\_writer = csv.writer(csv\_out\_file, delimiter=',', quoting=csv.QUOTE\_MINIMAL)  
 for row in rows:  
 image\_name = row[0]  
 image\_path = os.path.join(images\_out\_folder, image\_name)  
 if os.path.exists(image\_path):  
 image\_names\_in\_csv.append(image\_name)  
 csv\_out\_writer.writerow(row)  
 elif print\_removed\_items:  
 print('Removed image from CSV: ', image\_path)  
  
 # Remove images without corresponding line in CSV.  
 for image\_name in os.listdir(images\_out\_folder):  
 if image\_name not in image\_names\_in\_csv:  
 image\_path = os.path.join(images\_out\_folder, image\_name)  
 os.remove(image\_path)  
 if print\_removed\_items:  
 print('Removed image from folder: ', image\_path)  
  
 def analyze\_outliers(self, outliers):  
 """Classifies each sample agains all other to find outliers.  
   
 If sample is classified differrrently than the original class - it sould  
 either be deleted or more similar samples should be aadded.  
 """  
 for outlier in outliers:  
 image\_path = os.path.join(self.\_images\_out\_folder, outlier.sample.class\_name, outlier.sample.name)  
  
 print('Outlier')  
 print(' sample path = ', image\_path)  
 print(' sample class = ', outlier.sample.class\_name)  
 print(' detected class = ', outlier.detected\_class)  
 print(' all classes = ', outlier.all\_classes)  
  
 img = cv2.imread(image\_path)  
 img = cv2.cvtColor(img, cv2.COLOR\_BGR2RGB)  
 show\_image(img, figsize=(20, 20))  
  
 def remove\_outliers(self, outliers):  
 """Removes outliers from the image folders."""  
 for outlier in outliers:  
 image\_path = os.path.join(self.\_images\_out\_folder, outlier.sample.class\_name, outlier.sample.name)  
 os.remove(image\_path)  
  
 def print\_images\_in\_statistics(self):  
 """Prints statistics from the input image folder."""  
 self.\_print\_images\_statistics(self.\_images\_in\_folder, self.\_pose\_class\_names)  
  
 def print\_images\_out\_statistics(self):  
 """Prints statistics from the output image folder."""  
 self.\_print\_images\_statistics(self.\_images\_out\_folder, self.\_pose\_class\_names)  
  
 def \_print\_images\_statistics(self, images\_folder, pose\_class\_names):  
 print('Number of images per pose class:')  
 for pose\_class\_name in pose\_class\_names:  
 n\_images = len([  
 n for n in os.listdir(os.path.join(images\_folder, pose\_class\_name))  
 if not n.startswith('.')])  
 print(' {}: {}'.format(pose\_class\_name, n\_images))

# Step 1: Build classifier

## Upload image samples

Locally create a folder named fitness\_poses\_images\_in with image samples.

Images should repesent terminal states of desired pose classes. I.e. if you want to classify push-up provide iamges for two classes: when person is up, and when person is down.

There should be about a few hundred samples per class covering different camera angles, environment conditions, body shapes, and exercise variations to build a good classifier.

Required structure of the images\_in\_folder:

fitness\_poses\_images\_in/  
 pushups\_up/  
 image\_001.jpg  
 image\_002.jpg  
 ...  
 pushups\_down/  
 image\_001.jpg  
 image\_002.jpg  
 ...  
 ...

Zip the fitness\_poses\_images\_in folder:

zip -r fitness\_poses\_images\_in.zip fitness\_poses\_images\_in

And run the code below to upload it to the Colab runtime

from google.colab import files  
import os  
  
uploaded = files.upload()  
os.listdir('.')

Unzip the archive:

import zipfile  
import io  
  
zf = zipfile.ZipFile(io.BytesIO(uploaded['fitness\_poses\_images\_in.zip']), "r")  
zf.extractall()  
os.listdir('.')

## Bootstrap images

# Required structure of the images\_in\_folder:  
#  
# fitness\_poses\_images\_in/  
# pushups\_up/  
# image\_001.jpg  
# image\_002.jpg  
# ...  
# pushups\_down/  
# image\_001.jpg  
# image\_002.jpg  
# ...  
# ...  
bootstrap\_images\_in\_folder = 'fitness\_poses\_images\_in'  
  
# Output folders for bootstrapped images and CSVs.  
bootstrap\_images\_out\_folder = 'fitness\_poses\_images\_out'  
bootstrap\_csvs\_out\_folder = 'fitness\_poses\_csvs\_out'

# Initialize helper.  
bootstrap\_helper = BootstrapHelper(  
 images\_in\_folder=bootstrap\_images\_in\_folder,  
 images\_out\_folder=bootstrap\_images\_out\_folder,  
 csvs\_out\_folder=bootstrap\_csvs\_out\_folder,  
)

# Check how many pose classes and images for them are available.  
bootstrap\_helper.print\_images\_in\_statistics()

# Bootstrap all images.  
# Set limit to some small number for debug.  
bootstrap\_helper.bootstrap(per\_pose\_class\_limit=None)

# Check how many images were bootstrapped.  
bootstrap\_helper.print\_images\_out\_statistics()

# After initial bootstrapping images without detected poses were still saved in  
# the folderd (but not in the CSVs) for debug purpose. Let's remove them.  
bootstrap\_helper.align\_images\_and\_csvs(print\_removed\_items=False)  
bootstrap\_helper.print\_images\_out\_statistics()

## Manual filtration

Please manually verify predictions and remove samples (images) that has wrong pose prediction. Check as if you were asked to classify pose just from predicted landmarks. If you can’t - remove it.

Align CSVs and image folders once you are done.

# Align CSVs with filtered images.  
bootstrap\_helper.align\_images\_and\_csvs(print\_removed\_items=False)  
bootstrap\_helper.print\_images\_out\_statistics()

## Automatic filtration

Classify each sample against database of all other samples and check if it gets in the same class as annotated after classification.

There can be two reasons for the outliers:

* **Wrong pose prediction**: In this case remove such outliers.
* **Wrong classification** (i.e. pose is predicted correctly and you aggree with original pose class assigned to the sample): In this case sample is from the underrepresented group (e.g. unusual angle or just very few samples). Add more similar samples and run bootstrapping from the very beginning.

Even if you just removed some samples it makes sence to re-run automatic filtration one more time as database of poses has changed.

**Important!!** Check that you are using the same parameters when classifying whole videos later.

# Find outliers.  
  
# Transforms pose landmarks into embedding.  
pose\_embedder = FullBodyPoseEmbedder()  
  
# Classifies give pose against database of poses.  
pose\_classifier = PoseClassifier(  
 pose\_samples\_folder=bootstrap\_csvs\_out\_folder,  
 pose\_embedder=pose\_embedder,  
 top\_n\_by\_max\_distance=30,  
 top\_n\_by\_mean\_distance=10)  
  
outliers = pose\_classifier.find\_pose\_sample\_outliers()  
print('Number of outliers: ', len(outliers))

# Analyze outliers.  
bootstrap\_helper.analyze\_outliers(outliers)

# Remove all outliers (if you don't want to manually pick).  
bootstrap\_helper.remove\_outliers(outliers)

# Align CSVs with images after removing outliers.  
bootstrap\_helper.align\_images\_and\_csvs(print\_removed\_items=False)  
bootstrap\_helper.print\_images\_out\_statistics()

## Dump for the App

Dump filtered poses to CSV and download it.

Please check this [guide](https://developers.google.com/ml-kit/vision/pose-detection/classifying-poses#4_integrate_with_the_ml_kit_quickstart_app) on how to use this CSV in the ML Kit sample app.

import csv  
import os  
import numpy as np  
  
  
def dump\_for\_the\_app():  
 pose\_samples\_folder = 'fitness\_poses\_csvs\_out'  
 pose\_samples\_csv\_path = 'fitness\_poses\_csvs\_out.csv'  
 file\_extension = 'csv'  
 file\_separator = ','  
  
 # Each file in the folder represents one pose class.  
 file\_names = [name for name in os.listdir(pose\_samples\_folder) if name.endswith(file\_extension)]  
  
 with open(pose\_samples\_csv\_path, 'w') as csv\_out:  
 csv\_out\_writer = csv.writer(csv\_out, delimiter=file\_separator, quoting=csv.QUOTE\_MINIMAL)  
 for file\_name in file\_names:  
 # Use file name as pose class name.  
 class\_name = file\_name[:-(len(file\_extension) + 1)]  
  
 # One file line: `sample\_00001,x1,y1,x2,y2,....`.  
 with open(os.path.join(pose\_samples\_folder, file\_name)) as csv\_in:  
 csv\_in\_reader = csv.reader(csv\_in, delimiter=file\_separator)  
 for row in csv\_in\_reader:  
 row.insert(1, class\_name)  
 csv\_out\_writer.writerow(row)  
  
 files.download(pose\_samples\_csv\_path)  
  
  
dump\_for\_the\_app()

# Step 2: Classification

**Important!!** Check that you are using the same classification parameters as while building classifier.

# Upload your video.  
uploaded = files.upload()  
os.listdir('.')

# Specify your video name and target pose class to count the repetitions.  
video\_path = 'pushups-sample.mov'  
class\_name='pushups\_down'  
out\_video\_path = 'pushups-sample-out.mov'

# Open the video.  
import cv2  
  
video\_cap = cv2.VideoCapture(video\_path)  
  
# Get some video parameters to generate output video with classificaiton.  
video\_n\_frames = video\_cap.get(cv2.CAP\_PROP\_FRAME\_COUNT)  
video\_fps = video\_cap.get(cv2.CAP\_PROP\_FPS)  
video\_width = int(video\_cap.get(cv2.CAP\_PROP\_FRAME\_WIDTH))  
video\_height = int(video\_cap.get(cv2.CAP\_PROP\_FRAME\_HEIGHT))

# Initilize tracker, classifier and counter.  
# Do that before every video as all of them have state.  
from mediapipe.python.solutions import pose as mp\_pose  
  
  
# Folder with pose class CSVs. That should be the same folder you using while  
# building classifier to output CSVs.  
pose\_samples\_folder = 'fitness\_poses\_csvs\_out'  
  
# Initialize tracker.  
pose\_tracker = mp\_pose.Pose(upper\_body\_only=False)  
  
# Initialize embedder.  
pose\_embedder = FullBodyPoseEmbedder()  
  
# Initialize classifier.  
# Ceck that you are using the same parameters as during bootstrapping.  
pose\_classifier = PoseClassifier(  
 pose\_samples\_folder=pose\_samples\_folder,  
 pose\_embedder=pose\_embedder,  
 top\_n\_by\_max\_distance=30,  
 top\_n\_by\_mean\_distance=10)  
  
# # Uncomment to validate target poses used by classifier and find outliers.  
# outliers = pose\_classifier.find\_pose\_sample\_outliers()  
# print('Number of pose sample outliers (consider removing them): ', len(outliers))  
  
# Initialize EMA smoothing.  
pose\_classification\_filter = EMADictSmoothing(  
 window\_size=10,  
 alpha=0.2)  
  
# Initialize counter.  
repetition\_counter = RepetitionCounter(  
 class\_name=class\_name,  
 enter\_threshold=6,  
 exit\_threshold=4)  
  
# Initialize renderer.  
pose\_classification\_visualizer = PoseClassificationVisualizer(  
 class\_name=class\_name,  
 plot\_x\_max=video\_n\_frames,  
 # Graphic looks nicer if it's the same as `top\_n\_by\_mean\_distance`.  
 plot\_y\_max=10)

# Run classification on a video.  
import os  
import tqdm  
  
from mediapipe.python.solutions import drawing\_utils as mp\_drawing  
  
  
# Open output video.  
out\_video = cv2.VideoWriter(out\_video\_path, cv2.VideoWriter\_fourcc(\*'mp4v'), video\_fps, (video\_width, video\_height))  
  
frame\_idx = 0  
output\_frame = None  
with tqdm.tqdm(total=video\_n\_frames, position=0, leave=True) as pbar:  
 while True:  
 # Get next frame of the video.  
 success, input\_frame = video\_cap.read()  
 if not success:  
 break  
  
 # Run pose tracker.  
 input\_frame = cv2.cvtColor(input\_frame, cv2.COLOR\_BGR2RGB)  
 result = pose\_tracker.process(image=input\_frame)  
 pose\_landmarks = result.pose\_landmarks  
  
 # Draw pose prediction.  
 output\_frame = input\_frame.copy()  
 if pose\_landmarks is not None:  
 mp\_drawing.draw\_landmarks(  
 image=output\_frame,  
 landmark\_list=pose\_landmarks,  
 connections=mp\_pose.POSE\_CONNECTIONS)  
   
 if pose\_landmarks is not None:  
 # Get landmarks.  
 frame\_height, frame\_width = output\_frame.shape[0], output\_frame.shape[1]  
 pose\_landmarks = np.array([[lmk.x \* frame\_width, lmk.y \* frame\_height, lmk.z \* frame\_width]  
 for lmk in pose\_landmarks.landmark], dtype=np.float32)  
 assert pose\_landmarks.shape == (33, 3), 'Unexpected landmarks shape: {}'.format(pose\_landmarks.shape)  
  
 # Classify the pose on the current frame.  
 pose\_classification = pose\_classifier(pose\_landmarks)  
  
 # Smooth classification using EMA.  
 pose\_classification\_filtered = pose\_classification\_filter(pose\_classification)  
  
 # Count repetitions.  
 repetitions\_count = repetition\_counter(pose\_classification\_filtered)  
 else:  
 # No pose => no classification on current frame.  
 pose\_classification = None  
  
 # Still add empty classification to the filter to maintaing correct  
 # smoothing for future frames.  
 pose\_classification\_filtered = pose\_classification\_filter(dict())  
 pose\_classification\_filtered = None  
  
 # Don't update the counter presuming that person is 'frozen'. Just  
 # take the latest repetitions count.  
 repetitions\_count = repetition\_counter.n\_repeats  
  
 # Draw classification plot and repetition counter.  
 output\_frame = pose\_classification\_visualizer(  
 frame=output\_frame,  
 pose\_classification=pose\_classification,  
 pose\_classification\_filtered=pose\_classification\_filtered,  
 repetitions\_count=repetitions\_count)  
  
 # Save the output frame.  
 out\_video.write(cv2.cvtColor(np.array(output\_frame), cv2.COLOR\_RGB2BGR))  
  
 # Show intermediate frames of the video to track progress.  
 if frame\_idx % 50 == 0:  
 show\_image(output\_frame)  
  
 frame\_idx += 1  
 pbar.update()  
  
# Close output video.  
out\_video.release()  
  
# Release MediaPipe resources.  
pose\_tracker.close()  
  
# Show the last frame of the video.  
if output\_frame is not None:  
 show\_image(output\_frame)

# Download generated video  
files.download(out\_video\_path)