Inference_Colab

December 11, 2023

[ ]:
```
!pip install -q tensorflow-addons
```

```
                                   611.8/611.8
kB 8.5 MB/s eta 0:00:00
```

[ ]:
```
!pip install -q opencv-python mediapipe
```

1 Setup Model Weight Paths

[ ]:
```
use_google_drive = False

transformer_dir = "./transformer/"
lstm_dir = "./lstm/"

if (use_google_drive):
    from google.colab import drive
    drive.mount('/content/drive')
    transformer_dir = '/content/drive/MyDrive/Colab Notebooks/Data/asl-signs/'
    lstm_dir = '/content/drive/MyDrive/Colab Notebooks/Data/asl-signs/'
```

[ ]:
```
import numpy as np
import pandas as pd
import tensorflow as tf
import tensorflow_addons as tfa
import matplotlib.pyplot as plt

import os

import cv2
import numpy as np
import os
from matplotlib import pyplot as plt
import mediapipe as mp
from IPython.display import display, Javascript, Image
from google.colab.output import eval_js
from base64 import b64decode, b64encode
```

```
from google.colab.patches import cv2_imshow
import PIL
import io
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
from tensorflow.keras.callbacks import TensorBoard
from keras.models import load_model
from mediapipe.framework.formats import landmark_pb2
mp_drawing = mp.solutions.drawing_utils
mp_drawing_styles = mp.solutions.drawing_styles
```

2 Model

[ ]:
```
# Code From https://www.kaggle.com/code/markwijkhuizen/
↪gislr-tf-data-processing-transformer-training
# Epsilon value for layer normalisation
LAYER_NORM_EPS = 1e-6

# Dense layer units for landmarks
LIPS_UNITS = 384
HANDS_UNITS = 384
POSE_UNITS = 384
# final embedding and transformer embedding size
UNITS = 512

# Transformer
NUM_BLOCKS = 2
MLP_RATIO = 2

# Dropout
EMBEDDING_DROPOUT = 0.00
MLP_DROPOUT_RATIO = 0.30
CLASSIFIER_DROPOUT_RATIO = 0.10

# Initiailizers
INIT_HE_UNIFORM = tf.keras.initializers.he_uniform
INIT_GLOROT_UNIFORM = tf.keras.initializers.glorot_uniform
INIT_ZEROS = tf.keras.initializers.constant(0.0)
# Activations
GELU = tf.keras.activations.gelu

# If True, processing data from scratch
# If False, loads preprocessed data
PREPROCESS_DATA = False
TRAIN_MODEL = True
# True: use 10% of participants as validation set
```

```
# False: use all data for training -> gives better LB result
USE_VAL = False

N_ROWS = 543
N_DIMS = 3
DIM_NAMES = ['x', 'y', 'z']
SEED = 42
NUM_CLASSES = 250
IS_INTERACTIVE = True
VERBOSE = 1 if IS_INTERACTIVE else 2

INPUT_SIZE = 64

BATCH_ALL_SIGNS_N = 4
BATCH_SIZE = 256
N_EPOCHS = 100
LR_MAX = 1e-3
N_WARMUP_EPOCHS = 0
WD_RATIO = 0.05
MASK_VAL = 4237

USE_TYPES = ['left_hand', 'pose', 'right_hand']
START_IDX = 468
LIPS_IDXS0 = np.array([
        61, 185, 40, 39, 37, 0, 267, 269, 270, 409,
        291, 146, 91, 181, 84, 17, 314, 405, 321, 375,
        78, 191, 80, 81, 82, 13, 312, 311, 310, 415,
        95, 88, 178, 87, 14, 317, 402, 318, 324, 308,
    ])
# Landmark indices in original data
LEFT_HAND_IDXS0 = np.arange(468,489)
RIGHT_HAND_IDXS0 = np.arange(522,543)
LEFT_POSE_IDXS0 = np.array([502, 504, 506, 508, 510])
RIGHT_POSE_IDXS0 = np.array([503, 505, 507, 509, 511])
LANDMARK_IDXS_LEFT_DOMINANT0 = np.concatenate((LIPS_IDXS0, LEFT_HAND_IDXS0,
  ↪LEFT_POSE_IDXS0))
LANDMARK_IDXS_RIGHT_DOMINANT0 = np.concatenate((LIPS_IDXS0, RIGHT_HAND_IDXS0,
  ↪RIGHT_POSE_IDXS0))
HAND_IDXS0 = np.concatenate((LEFT_HAND_IDXS0, RIGHT_HAND_IDXS0), axis=0)
N_COLS = LANDMARK_IDXS_LEFT_DOMINANT0.size
# Landmark indices in processed data
LIPS_IDXS = np.argwhere(np.isin(LANDMARK_IDXS_LEFT_DOMINANT0, LIPS_IDXS0)).
  ↪squeeze()
LEFT_HAND_IDXS = np.argwhere(np.isin(LANDMARK_IDXS_LEFT_DOMINANT0,
  ↪LEFT_HAND_IDXS0)).squeeze()
RIGHT_HAND_IDXS = np.argwhere(np.isin(LANDMARK_IDXS_LEFT_DOMINANT0,
  ↪RIGHT_HAND_IDXS0)).squeeze()
```

```
HAND_IDXS = np.argwhere(np.isin(LANDMARK_IDXS_LEFT_DOMINANT0, HAND_IDXS0)).
↪squeeze()
POSE_IDXS = np.argwhere(np.isin(LANDMARK_IDXS_LEFT_DOMINANT0, LEFT_POSE_IDXS0)).
↪squeeze()

print(f'# HAND_IDXS: {len(HAND_IDXS)}, N_COLS: {N_COLS}')

LIPS_START = 0
LEFT_HAND_START = LIPS_IDXS.size
RIGHT_HAND_START = LEFT_HAND_START + LEFT_HAND_IDXS.size
POSE_START = RIGHT_HAND_START + RIGHT_HAND_IDXS.size

print(f'LIPS_START: {LIPS_START}, LEFT_HAND_START: {LEFT_HAND_START},␣
↪RIGHT_HAND_START: {RIGHT_HAND_START}, POSE_START: {POSE_START}')

LIPS_MEAN = np.load(f'{transformer_dir}/LIPS_MEAN.npy')
LIPS_STD = np.load(f'{transformer_dir}/LIPS_STD.npy')
LEFT_HANDS_MEAN = np.load(f'{transformer_dir}/LEFT_HANDS_MEAN.npy')
LEFT_HANDS_STD = np.load(f'{transformer_dir}/LEFT_HANDS_STD.npy')
POSE_MEAN = np.load(f'{transformer_dir}/POSE_MEAN.npy')
POSE_STD = np.load(f'{transformer_dir}/POSE_STD.npy')
```

[ ]:
```
# Code From https://www.kaggle.com/code/markwijkhuizen/
↪gislr-tf-data-processing-transformer-training
class Embedding(tf.keras.Model):
    def __init__(self):
        super(Embedding, self).__init__()

    def get_diffs(self, l):
        S = l.shape[2]
        other = tf.expand_dims(l, 3)
        other = tf.repeat(other, S, axis=3)
        other = tf.transpose(other, [0,1,3,2])
        diffs = tf.expand_dims(l, 3) - other
        diffs = tf.reshape(diffs, [-1, INPUT_SIZE, S*S])
        return diffs

    def build(self, input_shape):
        # Positional Embedding, initialized with zeros
        self.positional_embedding = tf.keras.layers.Embedding(INPUT_SIZE+1,␣
↪UNITS, embeddings_initializer=INIT_ZEROS)
        # Embedding layer for Landmarks
        self.lips_embedding = LandmarkEmbedding(LIPS_UNITS, 'lips')
        self.left_hand_embedding = LandmarkEmbedding(HANDS_UNITS, 'left_hand')
        self.pose_embedding = LandmarkEmbedding(POSE_UNITS, 'pose')
        # Landmark Weights
```

```
        self.landmark_weights = tf.Variable(tf.zeros([3], dtype=tf.float32),␣
 ↪name='landmark_weights')
        # Fully Connected Layers for combined landmarks
        self.fc = tf.keras.Sequential([
            tf.keras.layers.Dense(UNITS, name='fully_connected_1',␣
 ↪use_bias=False, kernel_initializer=INIT_GLOROT_UNIFORM),
            tf.keras.layers.Activation(GELU),
            tf.keras.layers.Dense(UNITS, name='fully_connected_2',␣
 ↪use_bias=False, kernel_initializer=INIT_HE_UNIFORM),
        ], name='fc')


    def call(self, lips0, left_hand0, pose0, non_empty_frame_idxs,␣
 ↪training=False):
        # Lips
        lips_embedding = self.lips_embedding(lips0)
        # Left Hand
        left_hand_embedding = self.left_hand_embedding(left_hand0)
        # Pose
        pose_embedding = self.pose_embedding(pose0)
        # Merge Embeddings of all landmarks with mean pooling
        x = tf.stack((
            lips_embedding, left_hand_embedding, pose_embedding,
        ), axis=3)
        x = x * tf.nn.softmax(self.landmark_weights)
        x = tf.reduce_sum(x, axis=3)
        # Fully Connected Layers
        x = self.fc(x)
        # Add Positional Embedding
        max_frame_idxs = tf.clip_by_value(
                tf.reduce_max(non_empty_frame_idxs, axis=1, keepdims=True),
                1,
                np.PINF,
            )
        normalised_non_empty_frame_idxs = tf.where(
            tf.math.equal(non_empty_frame_idxs, -1.0),
            INPUT_SIZE,
            tf.cast(
                non_empty_frame_idxs / max_frame_idxs * INPUT_SIZE,
                tf.int32,
            ),
        )
        x = x + self.positional_embedding(normalised_non_empty_frame_idxs)

        return x

class LandmarkEmbedding(tf.keras.Model):
```

```python
    def __init__(self, units, name):
        super(LandmarkEmbedding, self).__init__(name=f'{name}_embedding')
        self.units = units

    def build(self, input_shape):
        # Embedding for missing landmark in frame, initizlied with zeros
        self.empty_embedding = self.add_weight(
            name=f'{self.name}_empty_embedding',
            shape=[self.units],
            initializer=INIT_ZEROS,
        )
        # Embedding
        self.dense = tf.keras.Sequential([
            tf.keras.layers.Dense(self.units, name=f'{self.name}_dense_1',␣
 ↪use_bias=False, kernel_initializer=INIT_GLOROT_UNIFORM),
            tf.keras.layers.Activation(GELU),
            tf.keras.layers.Dense(self.units, name=f'{self.name}_dense_2',␣
 ↪use_bias=False, kernel_initializer=INIT_HE_UNIFORM),
        ], name=f'{self.name}_dense')

    def call(self, x):
        return tf.where(
                # Checks whether landmark is missing in frame
                tf.reduce_sum(x, axis=2, keepdims=True) == 0,
                # If so, the empty embedding is used
                self.empty_embedding,
                # Otherwise the landmark data is embedded
                self.dense(x),
            )

# Full Transformer
class Transformer(tf.keras.Model):
    def __init__(self, num_blocks):
        super(Transformer, self).__init__(name='transformer')
        self.num_blocks = num_blocks

    def build(self, input_shape):
        self.ln_1s = []
        self.mhas = []
        self.ln_2s = []
        self.mlps = []
        # Make Transformer Blocks
        for i in range(self.num_blocks):
            # Multi Head Attention
            self.mhas.append(MultiHeadAttention(UNITS, 8))
            # Multi Layer Perception
            self.mlps.append(tf.keras.Sequential([
```

```
                tf.keras.layers.Dense(UNITS * MLP_RATIO, activation=GELU,␣
↪kernel_initializer=INIT_GLOROT_UNIFORM),
                tf.keras.layers.Dropout(MLP_DROPOUT_RATIO),
                tf.keras.layers.Dense(UNITS,␣
↪kernel_initializer=INIT_HE_UNIFORM),
            ]))

    def call(self, x, attention_mask):
        # Iterate input over transformer blocks
        for mha, mlp in zip(self.mhas, self.mlps):
            x = x + mha(x, attention_mask)
            x = x + mlp(x)

        return x

# based on: https://stackoverflow.com/questions/67342988/
↪verifying-the-implementation-of-multihead-attention-in-transformer
# replaced softmax with softmax layer to support masked softmax
def scaled_dot_product(q,k,v, softmax, attention_mask):
    #calculates Q . K(transpose)
    qkt = tf.matmul(q,k,transpose_b=True)
    #caculates scaling factor
    dk = tf.math.sqrt(tf.cast(q.shape[-1],dtype=tf.float32))
    scaled_qkt = qkt/dk
    softmax = softmax(scaled_qkt, mask=attention_mask)

    z = tf.matmul(softmax,v)
    #shape: (m,Tx,depth), same shape as q,k,v
    return z

class MultiHeadAttention(tf.keras.layers.Layer):
    def __init__(self,d_model,num_of_heads):
        super(MultiHeadAttention,self).__init__()
        self.d_model = d_model
        self.num_of_heads = num_of_heads
        self.depth = d_model//num_of_heads
        self.wq = [tf.keras.layers.Dense(self.depth) for i in␣
↪range(num_of_heads)]
        self.wk = [tf.keras.layers.Dense(self.depth) for i in␣
↪range(num_of_heads)]
        self.wv = [tf.keras.layers.Dense(self.depth) for i in␣
↪range(num_of_heads)]
        self.wo = tf.keras.layers.Dense(d_model)
        self.softmax = tf.keras.layers.Softmax()

    def call(self,x, attention_mask):
```

```
        multi_attn = []
        for i in range(self.num_of_heads):
            Q = self.wq[i](x)
            K = self.wk[i](x)
            V = self.wv[i](x)
            multi_attn.append(scaled_dot_product(Q,K,V, self.softmax,␣
↪attention_mask))

        multi_head = tf.concat(multi_attn,axis=-1)
        multi_head_attention = self.wo(multi_head)
        return multi_head_attention

# source:: https://stackoverflow.com/questions/60689185/
↪label-smoothing-for-sparse-categorical-crossentropy
def scce_with_ls(y_true, y_pred):
    # One Hot Encode Sparsely Encoded Target Sign
    y_true = tf.cast(y_true, tf.int32)
    y_true = tf.one_hot(y_true, NUM_CLASSES, axis=1)
    y_true = tf.squeeze(y_true, axis=2)
    # Categorical Crossentropy with native label smoothing support
    return tf.keras.losses.categorical_crossentropy(y_true, y_pred,␣
↪label_smoothing=0.25)

def get_transformer_model():
    # Inputs
    frames = tf.keras.layers.Input([INPUT_SIZE, N_COLS, N_DIMS], dtype=tf.
↪float32, name='frames')
    non_empty_frame_idxs = tf.keras.layers.Input([INPUT_SIZE], dtype=tf.
↪float32, name='non_empty_frame_idxs')
    # Padding Mask
    mask0 = tf.cast(tf.math.not_equal(non_empty_frame_idxs, -1), tf.float32)
    mask0 = tf.expand_dims(mask0, axis=2)
    # Random Frame Masking
    mask = tf.where(
        (tf.random.uniform(tf.shape(mask0)) > 0.25) & tf.math.not_equal(mask0,␣
↪0.0),
        1.0,
        0.0,
    )
    # Correct Samples Which are all masked now...
    mask = tf.where(
        tf.math.equal(tf.reduce_sum(mask, axis=[1,2], keepdims=True), 0.0),
        mask0,
        mask,
    )
```

```python
"""
    left_hand: 468:489
    pose: 489:522
    right_hand: 522:543
"""
x = frames
x = tf.slice(x, [0,0,0,0], [-1,INPUT_SIZE, N_COLS, 2])
# LIPS
lips = tf.slice(x, [0,0,LIPS_START,0], [-1,INPUT_SIZE, 40, 2])
lips = tf.where(
        tf.math.equal(lips, 0.0),
        0.0,
        (lips - LIPS_MEAN) / LIPS_STD,
    )
# LEFT HAND
left_hand = tf.slice(x, [0,0,40,0], [-1,INPUT_SIZE, 21, 2])
left_hand = tf.where(
        tf.math.equal(left_hand, 0.0),
        0.0,
        (left_hand - LEFT_HANDS_MEAN) / LEFT_HANDS_STD,
    )
# POSE
pose = tf.slice(x, [0,0,61,0], [-1,INPUT_SIZE, 5, 2])
pose = tf.where(
        tf.math.equal(pose, 0.0),
        0.0,
        (pose - POSE_MEAN) / POSE_STD,
    )

# Flatten
lips = tf.reshape(lips, [-1, INPUT_SIZE, 40*2])
left_hand = tf.reshape(left_hand, [-1, INPUT_SIZE, 21*2])
pose = tf.reshape(pose, [-1, INPUT_SIZE, 5*2])

# Embedding
x = Embedding()(lips, left_hand, pose, non_empty_frame_idxs)

# Encoder Transformer Blocks
x = Transformer(NUM_BLOCKS)(x, mask)

# Pooling
x = tf.reduce_sum(x * mask, axis=1) / tf.reduce_sum(mask, axis=1)
# Classifier Dropout
x = tf.keras.layers.Dropout(CLASSIFIER_DROPOUT_RATIO)(x)
# Classification Layer
```

```
    x = tf.keras.layers.Dense(NUM_CLASSES, activation=tf.keras.activations.
↪softmax, kernel_initializer=INIT_GLOROT_UNIFORM)(x)

    outputs = x

    # Create Tensorflow Model
    model = tf.keras.models.Model(inputs=[frames, non_empty_frame_idxs],␣
↪outputs=outputs)

    # Sparse Categorical Cross Entropy With Label Smoothing
    loss = scce_with_ls

    # Adam Optimizer with weight decay
    optimizer = tfa.optimizers.AdamW(learning_rate=1e-3, weight_decay=1e-5,␣
↪clipnorm=1.0)

    # TopK Metrics
    metrics = [
        tf.keras.metrics.SparseCategoricalAccuracy(name='acc'),
        tf.keras.metrics.SparseTopKCategoricalAccuracy(k=5, name='top_5_acc'),
        tf.keras.metrics.SparseTopKCategoricalAccuracy(k=10, name='top_10_acc'),
    ]

    model.compile(loss=loss, optimizer=optimizer, metrics=metrics)

    return model
```

[ ]:
```
tf.keras.backend.clear_session()

model_transformer = get_transformer_model()
```

[ ]:
```
model_transformer.load_weights(f'{transformer_dir}/model.h5')
```

[ ]:
```
import json

signmap_sub_dir = 'sign_to_prediction_index_map.json'
signmap_full_file_path = os.path.join(transformer_dir, signmap_sub_dir)
# Load the sign to index mapping
with open(signmap_full_file_path, 'r') as file:
    sign_to_index = json.load(file)

inverted_mapping = {v: k for k, v in sign_to_index.items()}
# Convert mapping to list
class_names = [inverted_mapping[i] for i in sorted(inverted_mapping)]
```

2.1 Load LSTM

[ ]:
```
h5_file = None
npy_file = None

for file in os.listdir(lstm_dir):
    if file.endswith('.h5') and not h5_file:
        h5_file = file
    elif file.endswith('.npy') and not npy_file:
        npy_file = file
    if h5_file and npy_file:
        break

if h5_file and npy_file:
    model_lstm = load_model(os.path.join(lstm_dir, h5_file))
    actions = np.load(os.path.join(lstm_dir, npy_file), allow_pickle=True)
    print("Model and actions loaded successfully.")
else:
    print("Required files not found in the folder.")
```

3 Inference

[ ]:
```
mp_holistic = mp.solutions.holistic # Holistic model
mp_drawing = mp.solutions.drawing_utils # Drawing utilities

def mediapipe_detection(image, model):
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB) # COLOR CONVERSION BGR 2 RGB
    image.flags.writeable = False                  # Image is no longer␣
 ↪writeable
    results = model.process(image)                 # Make prediction
    image.flags.writeable = True                   # Image is now writeable
    image = cv2.cvtColor(image, cv2.COLOR_RGB2BGR) # COLOR COVERSION RGB 2 BGR
    return image, results

def draw_landmarks(image, results):
    mp_drawing.draw_landmarks(image, results.face_landmarks, mp_holistic.
 ↪FACE_CONTOURS) # Draw face connections
    mp_drawing.draw_landmarks(image, results.pose_landmarks, mp_holistic.
 ↪POSE_CONNECTIONS) # Draw pose connections
    mp_drawing.draw_landmarks(image, results.left_hand_landmarks, mp_holistic.
 ↪HAND_CONNECTIONS) # Draw left hand connections
    mp_drawing.draw_landmarks(image, results.right_hand_landmarks, mp_holistic.
 ↪HAND_CONNECTIONS) # Draw right hand connections

def draw_styled_landmarks(image, results):
    # Draw face connections
```

```
    mp_drawing.draw_landmarks(image, results.face_landmarks, mp_holistic.
↪FACEMESH_CONTOURS,
                             mp_drawing.DrawingSpec(color=(80,110,10),␣
↪thickness=1, circle_radius=1),
                             mp_drawing.DrawingSpec(color=(80,256,121),␣
↪thickness=1, circle_radius=1)
                             )
    # Draw pose connections
    mp_drawing.draw_landmarks(image, results.pose_landmarks, mp_holistic.
↪POSE_CONNECTIONS,
                             mp_drawing.DrawingSpec(color=(80,22,10),␣
↪thickness=2, circle_radius=4),
                             mp_drawing.DrawingSpec(color=(80,44,121),␣
↪thickness=2, circle_radius=2)
                             )
    # Draw left hand connections
    mp_drawing.draw_landmarks(image, results.left_hand_landmarks, mp_holistic.
↪HAND_CONNECTIONS,
                             mp_drawing.DrawingSpec(color=(121,22,76),␣
↪thickness=2, circle_radius=4),
                             mp_drawing.DrawingSpec(color=(121,44,250),␣
↪thickness=2, circle_radius=2)
                             )
    # Draw right hand connections
    mp_drawing.draw_landmarks(image, results.right_hand_landmarks, mp_holistic.
↪HAND_CONNECTIONS,
                             mp_drawing.DrawingSpec(color=(245,117,66),␣
↪thickness=2, circle_radius=4),
                             mp_drawing.DrawingSpec(color=(245,66,230),␣
↪thickness=2, circle_radius=2)
                             )

def␣
↪draw_landmarks(landmarks,image,show_pose=True,show_face_contour=True,show_face_tesselation=
↪
    annotated_image = image.copy()
    results = landmarks
    if show_face_tesselation:
        mp_drawing.draw_landmarks(
            annotated_image,
            results.face_landmarks,
            mp_holistic.FACEMESH_TESSELATION,
            landmark_drawing_spec=None,
            connection_drawing_spec=mp_drawing_styles
            .get_default_face_mesh_tesselation_style())
    if show_face_contour:
```

```
        mp_drawing.draw_landmarks(
            annotated_image,
            results.face_landmarks,
            mp_holistic.FACEMESH_CONTOURS,
            landmark_drawing_spec=None,
            connection_drawing_spec=mp_drawing_styles
            .get_default_face_mesh_contours_style())
    if show_pose:
        mp_drawing.draw_landmarks(
            annotated_image,
            results.pose_landmarks,
            mp_holistic.POSE_CONNECTIONS,
            landmark_drawing_spec=mp_drawing_styles.
            get_default_pose_landmarks_style())
    if show_left_hand:
        mp_drawing.draw_landmarks(
            annotated_image,
            results.left_hand_landmarks,
            mp_holistic.HAND_CONNECTIONS,
            landmark_drawing_spec=mp_drawing_styles
            .get_default_hand_landmarks_style())
    if show_right_hand:
        mp_drawing.draw_landmarks(
            annotated_image,
            results.right_hand_landmarks,
            mp_holistic.HAND_CONNECTIONS,
            landmark_drawing_spec=mp_drawing_styles
            .get_default_hand_landmarks_style())
    return annotated_image

def display_image(img):
    plt.imshow(img)
    plt.axis('off')  # Turn off the axis
    plt.show()
```

3.1 Javascript code taken from a previous Module Lab to display and get frames from a webcam

[ ]:
```
# JavaScript to properly create our live video stream using our webcam as input
def video_stream():
  js = Javascript('''
    var video;
    var div = null;
    var stream;
    var captureCanvas;
    var imgElement;
    var labelElement;
```

```
var pendingResolve = null;
var shutdown = false;

function removeDom() {
   stream.getVideoTracks()[0].stop();
   video.remove();
   div.remove();
   video = null;
   div = null;
   stream = null;
   imgElement = null;
   captureCanvas = null;
   labelElement = null;
}

function onAnimationFrame() {
  if (!shutdown) {
    window.requestAnimationFrame(onAnimationFrame);
  }
  if (pendingResolve) {
    var result = "";
    if (!shutdown) {
      captureCanvas.getContext('2d').drawImage(video, 0, 0, 640, 480);
      result = captureCanvas.toDataURL('image/jpeg', 0.8)
    }
    var lp = pendingResolve;
    pendingResolve = null;
    lp(result);
  }
}

async function createDom() {
  if (div !== null) {
    return stream;
  }

  div = document.createElement('div');
  div.style.border = '2px solid black';
  div.style.padding = '3px';
  div.style.width = '100%';
  div.style.maxWidth = '600px';
  document.body.appendChild(div);

  const modelOut = document.createElement('div');
  modelOut.innerHTML = "<span>Status:</span>";
  labelElement = document.createElement('span');
```

```
    labelElement.innerText = 'No data';
    labelElement.style.fontWeight = 'bold';
    modelOut.appendChild(labelElement);
    div.appendChild(modelOut);

    video = document.createElement('video');
    video.style.display = 'block';
    video.width = div.clientWidth - 6;
    video.setAttribute('playsinline', '');
    video.onclick = () => { shutdown = true; };
    stream = await navigator.mediaDevices.getUserMedia(
        {video: { facingMode: "environment"}});
    div.appendChild(video);

    imgElement = document.createElement('img');
    imgElement.style.position = 'absolute';
    imgElement.style.zIndex = 1;
    imgElement.onclick = () => { shutdown = true; };
    div.appendChild(imgElement);

    const instruction = document.createElement('div');
    instruction.innerHTML =
        '<span style="color: red; font-weight: bold;">' +
        'When finished, click here or on the video to stop this demo</span>';
    div.appendChild(instruction);
    instruction.onclick = () => { shutdown = true; };

    video.srcObject = stream;
    await video.play();

    captureCanvas = document.createElement('canvas');
    captureCanvas.width = 640; //video.videoWidth;
    captureCanvas.height = 480; //video.videoHeight;
    window.requestAnimationFrame(onAnimationFrame);

    return stream;
  }
  async function stream_frame(label, imgData) {
    if (shutdown) {
      removeDom();
      shutdown = false;
      return '';
    }

    var preCreate = Date.now();
    stream = await createDom();
```

```
      var preShow = Date.now();
      if (label != "") {
        labelElement.innerHTML = label;
      }

      if (imgData != "") {
        var videoRect = video.getClientRects()[0];
        imgElement.style.top = videoRect.top + "px";
        imgElement.style.left = videoRect.left + "px";
        imgElement.style.width = videoRect.width + "px";
        imgElement.style.height = videoRect.height + "px";
        imgElement.src = imgData;
      }

      var preCapture = Date.now();
      var result = await new Promise(function(resolve, reject) {
        pendingResolve = resolve;
      });
      shutdown = false;

      return {'create': preShow - preCreate,
              'show': preCapture - preShow,
              'capture': Date.now() - preCapture,
              'img': result};
    }
    ''')

  display(js)

def video_frame(label, bbox):
  data = eval_js('stream_frame("{}", "{}")'.format(label, bbox))
  return data

def js_to_image(js_reply):
  """
  Params:
          js_reply: JavaScript object containing image from webcam
  Returns:
          img: OpenCV BGR image
  """
  # decode base64 image
  image_bytes = b64decode(js_reply.split(',')[1])
  # convert bytes to numpy array
  jpg_as_np = np.frombuffer(image_bytes, dtype=np.uint8)
  # decode numpy array into OpenCV BGR image
  img = cv2.imdecode(jpg_as_np, flags=1)
```

```
  return img

def bbox_to_bytes(bbox_array):
  rgb_image = cv2.cvtColor(bbox_array, cv2.COLOR_BGR2RGB)

  # Convert an RGB array to a PNG byte stream with PIL
  bbox_pil = PIL.Image.fromarray(rgb_image, 'RGB')
  iobuf = io.BytesIO()
  bbox_pil.save(iobuf, format='png')
  # format return string
  bbox_bytes = 'data:image/png;base64,{}'.format((str(b64encode(iobuf.
↪getvalue()), 'utf-8')))

  return bbox_bytes
```

3.2 Inference Transformer Processing

[ ]:

```
SEQUENCE = []
TenDataFrame = []

def transform(results, frame_number):
  frame = []
  type_ = []
  index = []
  x = []
  y = []
  z = []
  #image.flags.writeable = False
  #image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
  #results = holistic.process(image)
  #face
  if(results.face_landmarks is None):
    for ind in range(468):
      frame.append(frame_number)
      type_.append("face")
      index.append(ind)
      x.append(np.nan)
      y.append(np.nan)
      z.append(np.nan)
  else:
    for ind,val in enumerate(results.face_landmarks.landmark):
      frame.append(frame_number)
      type_.append("face")
      index.append(ind)
      x.append(val.x)
      y.append(val.y)
      z.append(val.z)
```

```python
#left hand
if(results.left_hand_landmarks is None):
  for ind in range(21):
    frame.append(frame_number)
    type_.append("left_hand")
    index.append(ind)
    x.append(np.nan)
    y.append(np.nan)
    z.append(np.nan)
else:
  for ind,val in enumerate(results.left_hand_landmarks.landmark):
    frame.append(frame_number)
    type_.append("left_hand")
    index.append(ind)
    x.append(val.x)
    y.append(val.y)
    z.append(val.z)

#pose
if(results.pose_landmarks is None):
  for ind in range(33):
    frame.append(frame_number)
    type_.append("pose")
    index.append(ind)
    x.append(np.nan)
    y.append(np.nan)
    z.append(np.nan)
else:
  for ind,val in enumerate(results.pose_landmarks.landmark):
    frame.append(frame_number)
    type_.append("pose")
    index.append(ind)
    x.append(val.x)
    y.append(val.y)
    z.append(val.z)

#right hand
if(results.right_hand_landmarks is None):
  for ind in range(21):
    frame.append(frame_number)
    type_.append("right_hand")
    index.append(ind)
    x.append(np.nan)
    y.append(np.nan)
    z.append(np.nan)
else:
```

```
    for ind,val in enumerate(results.right_hand_landmarks.landmark):
      frame.append(frame_number)
      type_.append("right_hand")
      index.append(ind)
      x.append(val.x)
      y.append(val.y)
      z.append(val.z)
  #data = np.array([frame, type_, index, x, y, z])
  return pd.DataFrame({"frame" : frame,"type"  : type_,"landmark_index" :␣
↪index,"x" : x,"y" : y,"z" : z})

# Code From https://www.kaggle.com/code/markwijkhuizen/
↪gislr-tf-data-processing-transformer-training
"""
    Tensorflow layer to process data in TFLite
    Data needs to be processed in the model itself, so we can not use Python
"""
class PreprocessLayer(tf.keras.layers.Layer):
    def __init__(self):
        super(PreprocessLayer, self).__init__()
        normalisation_correction = tf.constant([
                        # Add 0.50 to left hand (original right hand) and substract␣
↪0.50 of right hand (original left hand)
                        [0] * len(LIPS_IDXS) + [0.50] * len(LEFT_HAND_IDXS) + [0.
↪50] * len(POSE_IDXS),
                        # Y coordinates stay intact
                        [0] * len(LANDMARK_IDXS_LEFT_DOMINANT0),
                        # Z coordinates stay intact
                        [0] * len(LANDMARK_IDXS_LEFT_DOMINANT0),
                    ],
                    dtype=tf.float32,
                )
        self.normalisation_correction = tf.transpose(normalisation_correction,␣
↪[1,0])

    def pad_edge(self, t, repeats, side):
        if side == 'LEFT':
            return tf.concat((tf.repeat(t[:1], repeats=repeats, axis=0), t),␣
↪axis=0)
        elif side == 'RIGHT':
            return tf.concat((t, tf.repeat(t[-1:], repeats=repeats, axis=0)),␣
↪axis=0)

    @tf.function(
        input_signature=(tf.TensorSpec(shape=[None,N_ROWS,N_DIMS], dtype=tf.
↪float32),),
```

```
    )
    def call(self, data0):
        # Number of Frames in Video
        N_FRAMES0 = tf.shape(data0)[0]

        # Find dominant hand by comparing summed absolute coordinates
        left_hand_sum = tf.math.reduce_sum(tf.where(tf.math.is_nan(tf.
↪gather(data0, LEFT_HAND_IDXS0, axis=1)), 0, 1))
        right_hand_sum = tf.math.reduce_sum(tf.where(tf.math.is_nan(tf.
↪gather(data0, RIGHT_HAND_IDXS0, axis=1)), 0, 1))
        left_dominant = left_hand_sum >= right_hand_sum

        # Count non NaN Hand values in each frame for the dominant hand
        if left_dominant:
            frames_hands_non_nan_sum = tf.math.reduce_sum(
                    tf.where(tf.math.is_nan(tf.gather(data0, LEFT_HAND_IDXS0,␣
↪axis=1)), 0, 1),
                    axis=[1, 2],
                )
        else:
            frames_hands_non_nan_sum = tf.math.reduce_sum(
                    tf.where(tf.math.is_nan(tf.gather(data0, RIGHT_HAND_IDXS0,␣
↪axis=1)), 0, 1),
                    axis=[1, 2],
                )

        # Find frames indices with coordinates of dominant hand
        non_empty_frames_idxs = tf.where(frames_hands_non_nan_sum > 0)
        non_empty_frames_idxs = tf.squeeze(non_empty_frames_idxs, axis=1)
        # Filter frames
        data = tf.gather(data0, non_empty_frames_idxs, axis=0)

        # Cast Indices in float32 to be compatible with Tensorflow Lite
        non_empty_frames_idxs = tf.cast(non_empty_frames_idxs, tf.float32)
        # Normalize to start with 0
        non_empty_frames_idxs -= tf.reduce_min(non_empty_frames_idxs)

        # Number of Frames in Filtered Video
        N_FRAMES = tf.shape(data)[0]

        # Gather Relevant Landmark Columns
        if left_dominant:
            data = tf.gather(data, LANDMARK_IDXS_LEFT_DOMINANT0, axis=1)
        else:
            data = tf.gather(data, LANDMARK_IDXS_RIGHT_DOMINANT0, axis=1)
            data = (
                    self.normalisation_correction + (
```

```
                        (data - self.normalisation_correction) * tf.where(self.
↪normalisation_correction != 0, -1.0, 1.0))
                )

        # Video fits in INPUT_SIZE
        if N_FRAMES < INPUT_SIZE:
            # Pad With -1 to indicate padding
            non_empty_frames_idxs = tf.pad(non_empty_frames_idxs, [[0,␣
↪INPUT_SIZE-N_FRAMES]], constant_values=-1)
            # Pad Data With Zeros
            data = tf.pad(data, [[0, INPUT_SIZE-N_FRAMES], [0,0], [0,0]],␣
↪constant_values=0)
            # Fill NaN Values With 0
            data = tf.where(tf.math.is_nan(data), 0.0, data)
            return data, non_empty_frames_idxs
        # Video needs to be downsampled to INPUT_SIZE
        else:
            # Repeat
            if N_FRAMES < INPUT_SIZE**2:
                repeats = tf.math.floordiv(INPUT_SIZE * INPUT_SIZE, N_FRAMES0)
                data = tf.repeat(data, repeats=repeats, axis=0)
                non_empty_frames_idxs = tf.repeat(non_empty_frames_idxs,␣
↪repeats=repeats, axis=0)

            # Pad To Multiple Of Input Size
            pool_size = tf.math.floordiv(len(data), INPUT_SIZE)
            if tf.math.mod(len(data), INPUT_SIZE) > 0:
                pool_size += 1

            if pool_size == 1:
                pad_size = (pool_size * INPUT_SIZE) - len(data)
            else:
                pad_size = (pool_size * INPUT_SIZE) % len(data)

            # Pad Start/End with Start/End value
            pad_left = tf.math.floordiv(pad_size, 2) + tf.math.
↪floordiv(INPUT_SIZE, 2)
            pad_right = tf.math.floordiv(pad_size, 2) + tf.math.
↪floordiv(INPUT_SIZE, 2)
            if tf.math.mod(pad_size, 2) > 0:
                pad_right += 1

            # Pad By Concatenating Left/Right Edge Values
            data = self.pad_edge(data, pad_left, 'LEFT')
            data = self.pad_edge(data, pad_right, 'RIGHT')
```

```
            # Pad Non Empty Frame Indices
            non_empty_frames_idxs = self.pad_edge(non_empty_frames_idxs,␣
↪pad_left, 'LEFT')
            non_empty_frames_idxs = self.pad_edge(non_empty_frames_idxs,␣
↪pad_right, 'RIGHT')

            # Reshape to Mean Pool
            data = tf.reshape(data, [INPUT_SIZE, -1, N_COLS, N_DIMS])
            non_empty_frames_idxs = tf.reshape(non_empty_frames_idxs,␣
↪[INPUT_SIZE, -1])

            # Mean Pool
            data = tf.experimental.numpy.nanmean(data, axis=1)
            non_empty_frames_idxs = tf.experimental.numpy.
↪nanmean(non_empty_frames_idxs, axis=1)

            # Fill NaN Values With 0
            data = tf.where(tf.math.is_nan(data), 0.0, data)

            return data, non_empty_frames_idxs

preprocess_layer = PreprocessLayer()

ROWS_PER_FRAME = 543  # number of landmarks per frame
def load_and_preprocess_data(data, preprocess_layer):
    # Load data
    data_columns = ['x', 'y', 'z']
    data = data[data_columns]
    n_frames = int(len(data) / ROWS_PER_FRAME)
    data = data.values.reshape(n_frames, ROWS_PER_FRAME, len(data_columns))
    # Apply preprocessing using the PreprocessLayer
    processed_data = preprocess_layer(data.astype(np.float32))

    return processed_data
```

3.3 Inference LSTM Processing

[ ]:
```
def extract_keypoints(results):
    pose = np.array([[res.x, res.y, res.z, res.visibility] for res in results.
↪pose_landmarks.landmark]).flatten() if results.pose_landmarks else np.
↪zeros(33*4)
    face = np.array([[res.x, res.y, res.z] for res in results.face_landmarks.
↪landmark]).flatten() if results.face_landmarks else np.zeros(468*3)
    lh = np.array([[res.x, res.y, res.z] for res in results.left_hand_landmarks.
↪landmark]).flatten() if results.left_hand_landmarks else np.zeros(21*3)
```

```
    rh = np.array([[res.x, res.y, res.z] for res in results.
 ↪right_hand_landmarks.landmark]).flatten() if results.right_hand_landmarks␣
 ↪else np.zeros(21*3)
    return np.concatenate([pose, face, lh, rh])
```

[ ]:
```
video_stream()
# label for video
label_html = 'Waiting for the first 24 frames...'

holistic = mp_holistic.Holistic(min_detection_confidence=0.5,␣
 ↪min_tracking_confidence=0.5, model_complexity=0)

bbox = ''
count = 0
frame_time = 0
frame_count = 0
sequence = []
printed = False
vframe_number = 0
combine_df = pd.DataFrame()
fps = 0
while True:
    js_reply = video_frame(label_html, bbox)

    if not js_reply:
        break

    vframe_number += 1
    frame = js_to_image(js_reply["img"])

    # Transformer pre processing
    image, results = mediapipe_detection(frame, holistic)
    testdf =  transform(results, vframe_number)
    combine_df = pd.concat([combine_df, testdf])

    # LSTM pre processing
    keypoints = extract_keypoints(results)
    sequence.append(keypoints)
    sequence = sequence[-10:]

    if vframe_number == 24:

        if not printed:
          print('predicting...')
          printed = True

        # LSTM Prediction
```

```python
            predictions = model_lstm.predict(np.expand_dims(sequence, axis=0),␣
↪verbose=0)[0]
            max_confidence = np.max(predictions)
            lstm_predicted_action = actions[np.argmax(predictions)]

            # Transformer Inference
            processed_data, non_empty_frame_idxs =␣
↪load_and_preprocess_data(combine_df, preprocess_layer)
            X = np.zeros([1, INPUT_SIZE, N_COLS, N_DIMS], dtype=np.float32)
            NON_EMPTY_FRAME_IDXS = np.full([1, INPUT_SIZE], -1, dtype=np.float32)
            X[0] = processed_data
            NON_EMPTY_FRAME_IDXS[0] = non_empty_frame_idxs
            predicted = model_transformer.predict({ 'frames': X,␣
↪'non_empty_frame_idxs': NON_EMPTY_FRAME_IDXS }, verbose=0).argmax(axis=1)
            transformer_predicted_action = class_names[predicted[0]]
            combine_df = pd.DataFrame()
            label_html = f"Predicted Action: Transformer:␣
↪{transformer_predicted_action}, LSTM: {lstm_predicted_action}"
            vframe_number = 0

        draw_styled_landmarks(image, results)
        bbox_bytes = bbox_to_bytes(image)
        bbox = bbox_bytes
```