cell_1:
```
# ===============================
# Imports & Setup
# ===============================
import os, glob, json, math, random, hashlib, cv2, numpy as np, tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, classification_report
import matplotlib.pyplot as plt
import seaborn as sns

# ==== Config ====
DATASET_ROOT = '/content/drive/MyDrive/arsl_15'        # original videos (for reference)
PROCESSED_ROOT = '/content/drive/MyDrive/processed_arsl_15'  # pre-extracted frames (.npy)
POSE_CACHE_DIR = '/content/drive/MyDrive/arsl_pose_cache_min'        # cached pose (.npy), mirrors dataset stru
os.makedirs(POSE_CACHE_DIR, exist_ok=True)

# Video/image formats
EXTS = ('.mp4', '.mov', '.avi', '.mkv')

# Training parameters
T_FRAMES = 12     # temporal frames per sample
IMG_SIZE = 160    # original preprocessing size (if applicable)
BATCH_SIZE = 4    # initial batch size (can increase if GPU RAM allows)
SHUFFLE_BUFFER = 64  # keep small to avoid RAM blowups
SNAPSHOT_DIR = '/tmp/tf-data-snapshot'  # optional disk snapshot (unused by default)

# Random seeds
RANDOM_SEED = 42
random.seed(RANDOM_SEED)
np.random.seed(RANDOM_SEED)
tf.random.set_seed(RANDOM_SEED)

# ==== GPU setup ====
gpus = tf.config.list_physical_devices('GPU')
if gpus:
try:
for g in gpus:
tf.config.experimental.set_memory_growth(g, True)  # don’t pre-allocate all VRAM
tf.config.set_visible_devices(gpus, 'GPU')
print("GPUs available:", gpus)
except Exception as e:
print("GPU memory growth setting failed:", e)

# ==== Mixed Precision (for T4 GPUs) ====
from tensorflow.keras import mixed_precision
mixed_precision.set_global_policy('mixed_float16')
print("Mixed precision policy:", mixed_precision.global_policy())
```

cell_2: ```# ==============================
# Train/Val Split
# ==============================
from sklearn.model_selection import train_test_split

Stratified split (keeps class balance)
train_paths, val_paths, train_labels, val_labels = train_test_split(
np.array(video_paths),
indices,
test_size=0.3,
stratify=indices,
random_state=RANDOM_SEED
)

# Keep labels as numpy arrays (compact in memory)
train_labels = np.array(train_labels, dtype=np.int32)
val_labels = np.array(val_labels, dtype=np.int32)

print(f"□ Train videos: {len(train_paths)}")
print(f"□ Val videos: {len(val_paths)}")
```
cell_3: ```def sample_indices(total, t):
if total <= t:
return list(range(total)) + [total-1]*(t-total) if total>0 else [0]*t
step = total / float(t)
return [int(i*step) for i in range(t)]```

cell_4: ```
# ==============================
# Augmentation + Efficient tf.data Pipeline
# ==============================
import tensorflow as tf
import numpy as np

AUTOTUNE = tf.data.AUTOTUNE
BATCH_SIZE = 4
TARGET_FRAMES = 12
SHUFFLE_BUFFER = 64
AUG_PROB = 0.6

def _np_load(path_str):
path = path_str.decode('utf-8') if isinstance(path_str, (bytes, bytearray)) else path_str
arr = np.load(path)
return arr.astype(np.float32)

def np_augment_sequence(
seq,
```

```
target_frames=TARGET_FRAMES,
aug_prob=AUG_PROB,
noise_scale=0.02,
max_shift_frames=5,
scale_range=(0.9, 1.1),
rotate_deg=10,
# new params
P_HFLIP=0.5,
BRIGHT_DELTA=0.2,
CONTRAST_LO=0.8,
CONTRAST_HI=1.2,
SAT_LO=0.8,
SAT_HI=1.2,
CROP_FRACTION=0.9,
NOISE_STD=0.02,
):
T = seq.shape[0]

# --- Temporal crop/pad ---
if T >= target_frames:
start = np.random.randint(0, T - target_frames + 1)
seq = seq[start:start + target_frames]
else:
pad_len = target_frames - T
pad_frames = np.repeat(seq[-1:,...], pad_len, axis=0)
seq = np.concatenate([seq, pad_frames], axis=0)

# --- Apply augmentations ---
if np.random.rand() < aug_prob:

# Time jitter
shift = np.random.randint(-max_shift_frames, max_shift_frames + 1)
if shift > 0:
seq = np.concatenate([seq[shift:], np.repeat(seq[-1:,...], shift, axis=0)], axis=0)
elif shift < 0:
shift = -shift
seq = np.concatenate([np.repeat(seq[:1,...], shift, axis=0), seq[:-shift]], axis=0)

# Gaussian noise
seq = seq + np.random.normal(scale=NOISE_STD, size=seq.shape).astype(np.float32)

# Horizontal flip (for x-coordinates if pose, or image flip if RGB)
if np.random.rand() < P_HFLIP:
if seq.ndim >= 3 and seq.shape[-1] >= 2:
seq[..., 0] = -seq[..., 0]  # flip x-axis

# Spatial scale & rotation
```

```
if seq.ndim >= 3 and seq.shape[-1] >= 2:
angle = np.deg2rad(np.random.uniform(-rotate_deg, rotate_deg))
s = np.random.uniform(scale_range[0], scale_range[1])
cosA, sinA = np.cos(angle), np.sin(angle)
coords = seq.reshape((seq.shape[0], -1, seq.shape[-1]))
center = coords.reshape(-1, seq.shape[-1]).mean(axis=0)
coords = coords - center
R = np.array([[cosA, -sinA], [sinA, cosA]], dtype=np.float32)
coords[..., :2] = coords[..., :2].dot(R.T) * s
coords = coords + center
seq = coords.reshape(seq.shape)

# Random crop (simulate zoom)
if np.random.rand() < 0.5 and seq.ndim >= 3:
frac = np.random.uniform(CROP_FRACTION, 1.0)
crop_len = int(seq.shape[1] * frac)
start = np.random.randint(0, seq.shape[1] - crop_len + 1)
seq = seq[:, start:start+crop_len, ...]
# pad back to original width
pad_len = seq.shape[1] - crop_len
if pad_len > 0:
pad = np.repeat(seq[:, -1:, ...], pad_len, axis=1)
seq = np.concatenate([seq, pad], axis=1)

# Brightness / Contrast / Saturation (if RGB data)
if seq.ndim == 4 and seq.shape[-1] == 3:
# Brightness
delta = np.random.uniform(-BRIGHT_DELTA, BRIGHT_DELTA)
seq = np.clip(seq + delta, 0, 1)

# Contrast
factor = np.random.uniform(CONTRAST_LO, CONTRAST_HI)
mean = seq.mean(axis=(1, 2), keepdims=True)
seq = np.clip((seq - mean) * factor + mean, 0, 1)

# Saturation
factor = np.random.uniform(SAT_LO, SAT_HI)
gray = seq.mean(axis=-1, keepdims=True)
seq = np.clip((seq - gray) * factor + gray, 0, 1)

return seq.astype(np.float32)


def load_and_preprocess(path, label):
seq = tf.numpy_function(_np_load, [path], tf.float32)
seq.set_shape([None, None])
seq_aug = tf.numpy_function(np_augment_sequence, [seq, TARGET_FRAMES, AUG_PROB], tf.float32)
```

```
def _fix_shape(x):
x = tf.ensure_shape(x, [TARGET_FRAMES, None])
x = tf.reshape(x, (TARGET_FRAMES, -1))
return x
seq_aug = _fix_shape(seq_aug)
mean = tf.reduce_mean(seq_aug, axis=(0,1), keepdims=True)
std = tf.math.reduce_std(seq_aug, axis=(0,1), keepdims=True) + 1e-6
seq_aug = (seq_aug - mean) / std
return seq_aug, tf.cast(label, tf.int32)

def make_dataset(paths, labels, batch_size=BATCH_SIZE, is_training=True):
ds = tf.data.Dataset.from_tensor_slices((paths, labels))
if is_training:
ds = ds.shuffle(SHUFFLE_BUFFER, reshuffle_each_iteration=True)
ds = ds.map(load_and_preprocess, num_parallel_calls=AUTOTUNE)
ds = ds.batch(batch_size, drop_remainder=False)
ds = ds.prefetch(AUTOTUNE)
return ds
```

cell_5:

```
import pathlib


def path_to_cache_npy(video_path, dataset_root=DATASET_ROOT, cache_root=POSE_CACHE_DIR):
# Map dataset file path to cache path by relative path (replace extension with .npy)
rel = os.path.relpath(video_path, dataset_root)
rel_no_ext = os.path.splitext(rel)[0]
npy_path = os.path.join(cache_root, rel_no_ext + '.npy')
return npy_path


def load_frames_and_pose(path):
# Load uniformly sampled frames and the precomputed pose from cache
cap = cv2.VideoCapture(path)
total = int(cap.get(cv2.CAP_PROP_FRAME_COUNT)) or 1
idxs = sample_indices(total, T_FRAMES)
frames = []
for i in idxs:
cap.set(cv2.CAP_PROP_POS_FRAMES, i)
ok, frame = cap.read()
if not ok:
frame = np.zeros((IMG_SIZE, IMG_SIZE, 3), dtype=np.uint8)
frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
frame = cv2.resize(frame, (IMG_SIZE, IMG_SIZE))
frames.append(frame)
cap.release()
frames = np.stack(frames, axis=0).astype(np.float32) / 255.0
```

npy = path_to_cache_npy(path)
pose = np.load(npy).astype(np.float32) # (T,75,4)
return frames, pose```

cell_6: ```def py_loader(path, label):
"""Load frames (uint8, possibly memmapped) + pose for a single video path."""
path_str = path.decode('utf-8')
f, p = load_frames_and_pose(path_str)
Return frames as uint8 to reduce memory footprint (4x smaller); cast later in a tf.map
return f.astype(np.uint8), p.astype(np.float32), label

def to_model_dtypes(inputs, label):
# Cast frames to float32 [0,1] just before the model; keep pose as float32
f = tf.cast(inputs['frames_rgb'], tf.float32) / 255.0
return {'frames_rgb': f, 'frames_pose': inputs['frames_pose']}, label

def tf_loader(path, label):
"""Wrap py_loader for use in tf.data pipeline."""
f, p, l = tf.numpy_function(
py_loader, [path, label], Tout=[tf.uint8, tf.float32, tf.int32]
)
# Ensure shapes
f = tf.ensure_shape(f, (T_FRAMES, None, None, 3))
p = tf.ensure_shape(p, (T_FRAMES, 75, 4))
l = tf.reshape(l, [])

If frames are not the expected spatial size, resize down to NEW_SIZE.
NEW_SIZE = 112
f = tf.image.resize(f, (NEW_SIZE, NEW_SIZE), method='bilinear', antialias=True)
Keep dtype as uint8 for now; will cast in a separate lightweight map

return {'frames_rgb': f, 'frames_pose': p}, l

AUTOTUNE = tf.data.AUTOTUNE

# Training dataset (streamed from disk, low RAM)
train_ds = (
tf.data.Dataset.from_tensor_slices((train_paths, train_labels))
.shuffle(buffer_size=SHUFFLE_BUFFER, seed=RANDOM_SEED)
.map(tf_loader, num_parallel_calls=2)   # limited parallelism
.map(to_model_dtypes, num_parallel_calls=2) # cast to float32 at the end
.batch(BATCH_SIZE, drop_remainder=True)
.prefetch(2)

```
)

# Validation dataset (no shuffle)
val_ds = (
tf.data.Dataset.from_tensor_slices((val_paths, val_labels))
.map(tf_loader, num_parallel_calls=2)
.map(to_model_dtypes, num_parallel_calls=2)
.batch(BATCH_SIZE, drop_remainder=False)
.prefetch(2)
)

steps_per_epoch = math.ceil(len(train_paths) / BATCH_SIZE)
val_steps = math.ceil(len(val_paths) / BATCH_SIZE)

print(f"□ Dataset ready | Train steps: {steps_per_epoch}, Val steps: {val_steps}")
```

cell_7: ``````