# vim: expandtab:ts=4:sw=4

import numpy as np

class Detection(object):

    """

    This class represents a bounding box detection in a single image.

    Parameters

    ----------

    tlwh : array\_like

        Bounding box in format `(x, y, w, h)`.

    confidence : float

        Detector confidence score.

    feature : array\_like

        A feature vector that describes the object contained in this image.

    Attributes

    ----------

    tlwh : ndarray

        Bounding box in format `(top left x, top left y, width, height)`.

    confidence : ndarray

        Detector confidence score.

    class\_name : ndarray

        Detector class.

    feature : ndarray | NoneType

        A feature vector that describes the object contained in this image.

    """

    def \_\_init\_\_(self, tlwh, confidence, centroid, feature):

        self.tlwh = np.asarray(tlwh, dtype=np.float32)

        self.confidence = float(confidence)

        self.feature = np.asarray(feature, dtype=np.float32)

        self.centroid = centroid

    def to\_tlbr(self):

        """Convert bounding box to format `(min x, min y, max x, max y)`, i.e.,

        `(top left, bottom right)`.

        """

        ret = self.tlwh.copy()

        ret[2:] += ret[:2]

        return ret

    def to\_xyah(self):

        """Convert bounding box to format `(center x, center y, aspect ratio,

        height)`, where the aspect ratio is `width / height`.

        """

        ret = self.tlwh.copy()

        ret[:2] += ret[2:] / 2

        ret[2] /= ret[3]

        return ret

Detection.py

# vim: expandtab:ts=4:sw=4

import os

import errno

import argparse

import numpy as np

import cv2

import os

os.environ['TF\_CPP\_MIN\_LOG\_LEVEL'] = '2'

import tensorflow.compat.v1 as tf

physical\_devices = tf.config.experimental.list\_physical\_devices('GPU')

if len(physical\_devices) > 0:

    tf.config.experimental.set\_memory\_growth(physical\_devices[0], True)

def \_run\_in\_batches(f, data\_dict, out, batch\_size):

    data\_len = len(out)

    num\_batches = int(data\_len / batch\_size)

    s, e = 0, 0

    for i in range(num\_batches):

        s, e = i \* batch\_size, (i + 1) \* batch\_size

        batch\_data\_dict = {k: v[s:e] for k, v in data\_dict.items()}

        out[s:e] = f(batch\_data\_dict)

    if e < len(out):

        batch\_data\_dict = {k: v[e:] for k, v in data\_dict.items()}

        out[e:] = f(batch\_data\_dict)

def extract\_image\_patch(image, bbox, patch\_shape):

    """Extract image patch from bounding box.

    Parameters

    ----------

    image : ndarray

        The full image.

    bbox : array\_like

        The bounding box in format (x, y, width, height).

    patch\_shape : Optional[array\_like]

        This parameter can be used to enforce a desired patch shape

        (height, width). First, the `bbox` is adapted to the aspect ratio

        of the patch shape, then it is clipped at the image boundaries.

        If None, the shape is computed from :arg:`bbox`.

    Returns

    -------

    ndarray | NoneType

        An image patch showing the :arg:`bbox`, optionally reshaped to

        :arg:`patch\_shape`.

        Returns None if the bounding box is empty or fully outside of the image

        boundaries.

    """

    bbox = np.array(bbox)

    if patch\_shape is not None:

        # correct aspect ratio to patch shape

        target\_aspect = float(patch\_shape[1]) / patch\_shape[0]

        new\_width = target\_aspect \* bbox[3]

        bbox[0] -= (new\_width - bbox[2]) / 2

        bbox[2] = new\_width

    # convert to top left, bottom right

    bbox[2:] += bbox[:2]

    bbox = bbox.astype(np.int\_)

    # clip at image boundaries

    bbox[:2] = np.maximum(0, bbox[:2])

    bbox[2:] = np.minimum(np.asarray(image.shape[:2][::-1]) - 1, bbox[2:])

    if np.any(bbox[:2] >= bbox[2:]):

        return None

    sx, sy, ex, ey = bbox

    image = image[sy:ey, sx:ex]

    image = cv2.resize(image, tuple(patch\_shape[::-1]))

    return image

class ImageEncoder(object):

    def \_\_init\_\_(self, checkpoint\_filename, input\_name="images", output\_name="features"):

        self.session = tf.Session()

        with tf.gfile.GFile(checkpoint\_filename, "rb") as file\_handle:

            graph\_def = tf.GraphDef()

            graph\_def.ParseFromString(file\_handle.read())

        tf.import\_graph\_def(graph\_def)

        try:

            self.input\_var = tf.get\_default\_graph().get\_tensor\_by\_name(input\_name)

            self.output\_var = tf.get\_default\_graph().get\_tensor\_by\_name(output\_name)

        except KeyError:

            layers = [i.name for i in tf.get\_default\_graph().get\_operations()]

            self.input\_var = tf.get\_default\_graph().get\_tensor\_by\_name(layers[0]+':0')

            self.output\_var = tf.get\_default\_graph().get\_tensor\_by\_name(layers[-1]+':0')

        assert len(self.output\_var.get\_shape()) == 2

        assert len(self.input\_var.get\_shape()) == 4

        self.feature\_dim = self.output\_var.get\_shape().as\_list()[-1]

        self.image\_shape = self.input\_var.get\_shape().as\_list()[1:]

    def \_\_call\_\_(self, data\_x, batch\_size=32):

        out = np.zeros((len(data\_x), self.feature\_dim), np.float32)

        \_run\_in\_batches(

            lambda x: self.session.run(self.output\_var, feed\_dict=x),

            {self.input\_var: data\_x}, out, batch\_size)

        return out

def create\_box\_encoder(model\_filename, input\_name="images:0", output\_name="features:0", batch\_size=32):

    image\_encoder = ImageEncoder(model\_filename, input\_name, output\_name)

    image\_shape = image\_encoder.image\_shape

    def encoder(image, boxes):

        image\_patches = []

        for box in boxes:

            patch = extract\_image\_patch(image, box, image\_shape[:2])

            if patch is None:

                print("WARNING: Failed to extract image patch: %s." % str(box))

                patch = np.random.uniform(0., 255., image\_shape).astype(np.uint8)

            image\_patches.append(patch)

        image\_patches = np.asarray(image\_patches)

        return image\_encoder(image\_patches, batch\_size)

    return encoder

def generate\_detections(encoder, mot\_dir, output\_dir, detection\_dir=None):

    """Generate detections with features.

    Parameters

    ----------

    encoder : Callable[image, ndarray] -> ndarray

        The encoder function takes as input a BGR color image and a matrix of

        bounding boxes in format `(x, y, w, h)` and returns a matrix of

        corresponding feature vectors.

    mot\_dir : str

        Path to the MOTChallenge directory (can be either train or test).

    output\_dir

        Path to the output directory. Will be created if it does not exist.

    detection\_dir

        Path to custom detections. The directory structure should be the default

        MOTChallenge structure: `[sequence]/det/det.txt`. If None, uses the

        standard MOTChallenge detections.

    """

    if detection\_dir is None:

        detection\_dir = mot\_dir

    try:

        os.makedirs(output\_dir)

    except OSError as exception:

        if exception.errno == errno.EEXIST and os.path.isdir(output\_dir):

            pass

        else:

            raise ValueError(

                "Failed to created output directory '%s'" % output\_dir)

    for sequence in os.listdir(mot\_dir):

        print("Processing %s" % sequence)

        sequence\_dir = os.path.join(mot\_dir, sequence)

        image\_dir = os.path.join(sequence\_dir, "img1")

        image\_filenames = {

            int(os.path.splitext(f)[0]): os.path.join(image\_dir, f)

            for f in os.listdir(image\_dir)}

        detection\_file = os.path.join(

            detection\_dir, sequence, "det/det.txt")

        detections\_in = np.loadtxt(detection\_file, delimiter=',')

        detections\_out = []

        frame\_indices = detections\_in[:, 0].astype(np.int)

        min\_frame\_idx = frame\_indices.astype(np.int).min()

        max\_frame\_idx = frame\_indices.astype(np.int).max()

        for frame\_idx in range(min\_frame\_idx, max\_frame\_idx + 1):

            print("Frame %05d/%05d" % (frame\_idx, max\_frame\_idx))

            mask = frame\_indices == frame\_idx

            rows = detections\_in[mask]

            if frame\_idx not in image\_filenames:

                print("WARNING could not find image for frame %d" % frame\_idx)

                continue

            bgr\_image = cv2.imread(

                image\_filenames[frame\_idx], cv2.IMREAD\_COLOR)

            features = encoder(bgr\_image, rows[:, 2:6].copy())

            detections\_out += [np.r\_[(row, feature)] for row, feature

                               in zip(rows, features)]

        output\_filename = os.path.join(output\_dir, "%s.npy" % sequence)

        np.save(

            output\_filename, np.asarray(detections\_out), allow\_pickle=False)

def parse\_args():

    """Parse command line arguments.

    """

    parser = argparse.ArgumentParser(description="Re-ID feature extractor")

    parser.add\_argument(

        "--model",

        default="resources/networks/mars-small128.pb",

        help="Path to freezed inference graph protobuf.")

    parser.add\_argument(

        "--mot\_dir", help="Path to MOTChallenge directory (train or test)",

        required=True)

    parser.add\_argument(

        "--detection\_dir", help="Path to custom detections. Defaults to "

        "standard MOT detections Directory structure should be the default "

        "MOTChallenge structure: [sequence]/det/det.txt", default=None)

    parser.add\_argument(

        "--output\_dir", help="Output directory. Will be created if it does not"

        " exist.", default="detections")

    return parser.parse\_args()

def main():

    args = parse\_args()

    encoder = create\_box\_encoder(args.model, batch\_size=32)

    generate\_detections(encoder, args.mot\_dir, args.output\_dir,

                        args.detection\_dir)

if \_\_name\_\_ == "\_\_main\_\_":

    main()

Generate\_detection.py

# vim: expandtab:ts=4:sw=4

from \_\_future\_\_ import absolute\_import

import numpy as np

from . import linear\_assignment

def iou(bbox, candidates):

    """Computer intersection over union.

    Parameters

    ----------

    bbox : ndarray

        A bounding box in format `(top left x, top left y, width, height)`.

    candidates : ndarray

        A matrix of candidate bounding boxes (one per row) in the same format

        as `bbox`.

    Returns

    -------

    ndarray

        The intersection over union in [0, 1] between the `bbox` and each

        candidate. A higher score means a larger fraction of the `bbox` is

        occluded by the candidate.

    """

    bbox\_tl, bbox\_br = bbox[:2], bbox[:2] + bbox[2:]

    candidates\_tl = candidates[:, :2]

    candidates\_br = candidates[:, :2] + candidates[:, 2:]

    tl = np.c\_[np.maximum(bbox\_tl[0], candidates\_tl[:, 0])[:, np.newaxis],

               np.maximum(bbox\_tl[1], candidates\_tl[:, 1])[:, np.newaxis]]

    br = np.c\_[np.minimum(bbox\_br[0], candidates\_br[:, 0])[:, np.newaxis],

               np.minimum(bbox\_br[1], candidates\_br[:, 1])[:, np.newaxis]]

    wh = np.maximum(0., br - tl)

    area\_intersection = wh.prod(axis=1)

    area\_bbox = bbox[2:].prod()

    area\_candidates = candidates[:, 2:].prod(axis=1)

    return area\_intersection / (area\_bbox + area\_candidates - area\_intersection)

def iou\_cost(tracks, detections, track\_indices=None,

             detection\_indices=None):

    """An intersection over union distance metric.

    Parameters

    ----------

    tracks : List[deep\_sort.track.Track]

        A list of tracks.

    detections : List[deep\_sort.detection.Detection]

        A list of detections.

    track\_indices : Optional[List[int]]

        A list of indices to tracks that should be matched. Defaults to

        all `tracks`.

    detection\_indices : Optional[List[int]]

        A list of indices to detections that should be matched. Defaults

        to all `detections`.

    Returns

    -------

    ndarray

        Returns a cost matrix of shape

        len(track\_indices), len(detection\_indices) where entry (i, j) is

        `1 - iou(tracks[track\_indices[i]], detections[detection\_indices[j]])`.

    """

    if track\_indices is None:

        track\_indices = np.arange(len(tracks))

    if detection\_indices is None:

        detection\_indices = np.arange(len(detections))

    cost\_matrix = np.zeros((len(track\_indices), len(detection\_indices)))

    for row, track\_idx in enumerate(track\_indices):

        if tracks[track\_idx].time\_since\_update > 1:

            cost\_matrix[row, :] = linear\_assignment.INFTY\_COST

            continue

        bbox = tracks[track\_idx].to\_tlwh()

        candidates = np.asarray([detections[i].tlwh for i in detection\_indices])

        cost\_matrix[row, :] = 1. - iou(bbox, candidates)

    return cost\_matrix

IOU matching.py

# vim: expandtab:ts=4:sw=4

import numpy as np

import scipy.linalg

"""

Table for the 0.95 quantile of the chi-square distribution with N degrees of

freedom (contains values for N=1, ..., 9). Taken from MATLAB/Octave's chi2inv

function and used as Mahalanobis gating threshold.

"""

chi2inv95 = {

    1: 3.8415,

    2: 5.9915,

    3: 7.8147,

    4: 9.4877,

    5: 11.070,

    6: 12.592,

    7: 14.067,

    8: 15.507,

    9: 16.919}

class KalmanFilter(object):

    """

    A simple Kalman filter for tracking bounding boxes in image space.

    The 8-dimensional state space

        x, y, a, h, vx, vy, va, vh

    contains the bounding box center position (x, y), aspect ratio a, height h,

    and their respective velocities.

    Object motion follows a constant velocity model. The bounding box location

    (x, y, a, h) is taken as direct observation of the state space (linear

    observation model).

    """

    def \_\_init\_\_(self):

        ndim, dt = 4, 1.

        # Create Kalman filter model matrices.

        self.\_motion\_mat = np.eye(2 \* ndim, 2 \* ndim)

        for i in range(ndim):

            self.\_motion\_mat[i, ndim + i] = dt

        self.\_update\_mat = np.eye(ndim, 2 \* ndim)

        # Motion and observation uncertainty are chosen relative to the current

        # state estimate. These weights control the amount of uncertainty in

        # the model. This is a bit hacky.

        self.\_std\_weight\_position = 1. / 20

        self.\_std\_weight\_velocity = 1. / 160

    def initiate(self, measurement):

        """Create track from unassociated measurement.

        Parameters

        ----------

        measurement : ndarray

            Bounding box coordinates (x, y, a, h) with center position (x, y),

            aspect ratio a, and height h.

        Returns

        -------

        (ndarray, ndarray)

            Returns the mean vector (8 dimensional) and covariance matrix (8x8

            dimensional) of the new track. Unobserved velocities are initialized

            to 0 mean.

        """

        mean\_pos = measurement

        mean\_vel = np.zeros\_like(mean\_pos)

        mean = np.r\_[mean\_pos, mean\_vel]

        std = [

            2 \* self.\_std\_weight\_position \* measurement[3],

            2 \* self.\_std\_weight\_position \* measurement[3],

            1e-2,

            2 \* self.\_std\_weight\_position \* measurement[3],

            10 \* self.\_std\_weight\_velocity \* measurement[3],

            10 \* self.\_std\_weight\_velocity \* measurement[3],

            1e-5,

            10 \* self.\_std\_weight\_velocity \* measurement[3]]

        covariance = np.diag(np.square(std))

        return mean, covariance

    def predict(self, mean, covariance):

        """Run Kalman filter prediction step.

        Parameters

        ----------

        mean : ndarray

            The 8 dimensional mean vector of the object state at the previous

            time step.

        covariance : ndarray

            The 8x8 dimensional covariance matrix of the object state at the

            previous time step.

        Returns

        -------

        (ndarray, ndarray)

            Returns the mean vector and covariance matrix of the predicted

            state. Unobserved velocities are initialized to 0 mean.

        """

        std\_pos = [

            self.\_std\_weight\_position \* mean[3],

            self.\_std\_weight\_position \* mean[3],

            1e-2,

            self.\_std\_weight\_position \* mean[3]]

        std\_vel = [

            self.\_std\_weight\_velocity \* mean[3],

            self.\_std\_weight\_velocity \* mean[3],

            1e-5,

            self.\_std\_weight\_velocity \* mean[3]]

        motion\_cov = np.diag(np.square(np.r\_[std\_pos, std\_vel]))

        mean = np.dot(self.\_motion\_mat, mean)

        covariance = np.linalg.multi\_dot((

            self.\_motion\_mat, covariance, self.\_motion\_mat.T)) + motion\_cov

        return mean, covariance

    def project(self, mean, covariance):

        """Project state distribution to measurement space.

        Parameters

        ----------

        mean : ndarray

            The state's mean vector (8 dimensional array).

        covariance : ndarray

            The state's covariance matrix (8x8 dimensional).

        Returns

        -------

        (ndarray, ndarray)

            Returns the projected mean and covariance matrix of the given state

            estimate.

        """

        std = [

            self.\_std\_weight\_position \* mean[3],

            self.\_std\_weight\_position \* mean[3],

            1e-1,

            self.\_std\_weight\_position \* mean[3]]

        innovation\_cov = np.diag(np.square(std))

        mean = np.dot(self.\_update\_mat, mean)

        covariance = np.linalg.multi\_dot((

            self.\_update\_mat, covariance, self.\_update\_mat.T))

        return mean, covariance + innovation\_cov

    def update(self, mean, covariance, measurement):

        """Run Kalman filter correction step.

        Parameters

        ----------

        mean : ndarray

            The predicted state's mean vector (8 dimensional).

        covariance : ndarray

            The state's covariance matrix (8x8 dimensional).

        measurement : ndarray

            The 4 dimensional measurement vector (x, y, a, h), where (x, y)

            is the center position, a the aspect ratio, and h the height of the

            bounding box.

        Returns

        -------

        (ndarray, ndarray)

            Returns the measurement-corrected state distribution.

        """

        projected\_mean, projected\_cov = self.project(mean, covariance)

        chol\_factor, lower = scipy.linalg.cho\_factor(

            projected\_cov, lower=True, check\_finite=False)

        kalman\_gain = scipy.linalg.cho\_solve(

            (chol\_factor, lower), np.dot(covariance, self.\_update\_mat.T).T,

            check\_finite=False).T

        innovation = measurement - projected\_mean

        new\_mean = mean + np.dot(innovation, kalman\_gain.T)

        new\_covariance = covariance - np.linalg.multi\_dot((

            kalman\_gain, projected\_cov, kalman\_gain.T))

        return new\_mean, new\_covariance

    def gating\_distance(self, mean, covariance, measurements,

                        only\_position=False):

        """Compute gating distance between state distribution and measurements.

        A suitable distance threshold can be obtained from `chi2inv95`. If

        `only\_position` is False, the chi-square distribution has 4 degrees of

        freedom, otherwise 2.

        Parameters

        ----------

        mean : ndarray

            Mean vector over the state distribution (8 dimensional).

        covariance : ndarray

            Covariance of the state distribution (8x8 dimensional).

        measurements : ndarray

            An Nx4 dimensional matrix of N measurements, each in

            format (x, y, a, h) where (x, y) is the bounding box center

            position, a the aspect ratio, and h the height.

        only\_position : Optional[bool]

            If True, distance computation is done with respect to the bounding

            box center position only.

        Returns

        -------

        ndarray

            Returns an array of length N, where the i-th element contains the

            squared Mahalanobis distance between (mean, covariance) and

            `measurements[i]`.

        """

        mean, covariance = self.project(mean, covariance)

        if only\_position:

            mean, covariance = mean[:2], covariance[:2, :2]

            measurements = measurements[:, :2]

        cholesky\_factor = np.linalg.cholesky(covariance)

        d = measurements - mean

        z = scipy.linalg.solve\_triangular(

            cholesky\_factor, d.T, lower=True, check\_finite=False,

            overwrite\_b=True)

        squared\_maha = np.sum(z \* z, axis=0)

        return squared\_maha

Kalman-filter.py

# vim: expandtab:ts=4:sw=4

from \_\_future\_\_ import absolute\_import

import numpy as np

from scipy.optimize import linear\_sum\_assignment

from . import kalman\_filter

INFTY\_COST = 1e+5

def min\_cost\_matching(

        distance\_metric, max\_distance, tracks, detections, track\_indices=None,

        detection\_indices=None):

    """Solve linear assignment problem.

    Parameters

    ----------

    distance\_metric : Callable[List[Track], List[Detection], List[int], List[int]) -> ndarray

        The distance metric is given a list of tracks and detections as well as

        a list of N track indices and M detection indices. The metric should

        return the NxM dimensional cost matrix, where element (i, j) is the

        association cost between the i-th track in the given track indices and

        the j-th detection in the given detection\_indices.

    max\_distance : float

        Gating threshold. Associations with cost larger than this value are

        disregarded.

    tracks : List[track.Track]

        A list of predicted tracks at the current time step.

    detections : List[detection.Detection]

        A list of detections at the current time step.

    track\_indices : List[int]

        List of track indices that maps rows in `cost\_matrix` to tracks in

        `tracks` (see description above).

    detection\_indices : List[int]

        List of detection indices that maps columns in `cost\_matrix` to

        detections in `detections` (see description above).

    Returns

    -------

    (List[(int, int)], List[int], List[int])

        Returns a tuple with the following three entries:

        \* A list of matched track and detection indices.

        \* A list of unmatched track indices.

        \* A list of unmatched detection indices.

    """

    if track\_indices is None:

        track\_indices = np.arange(len(tracks))

    if detection\_indices is None:

        detection\_indices = np.arange(len(detections))

    if len(detection\_indices) == 0 or len(track\_indices) == 0:

        return [], track\_indices, detection\_indices  # Nothing to match.

    cost\_matrix = distance\_metric(

        tracks, detections, track\_indices, detection\_indices)

    cost\_matrix[cost\_matrix > max\_distance] = max\_distance + 1e-5

    indices = linear\_sum\_assignment(cost\_matrix)

    indices = np.asarray(indices)

    indices = np.transpose(indices)

    matches, unmatched\_tracks, unmatched\_detections = [], [], []

    for col, detection\_idx in enumerate(detection\_indices):

        if col not in indices[:, 1]:

            unmatched\_detections.append(detection\_idx)

    for row, track\_idx in enumerate(track\_indices):

        if row not in indices[:, 0]:

            unmatched\_tracks.append(track\_idx)

    for row, col in indices:

        track\_idx = track\_indices[row]

        detection\_idx = detection\_indices[col]

        if cost\_matrix[row, col] > max\_distance:

            unmatched\_tracks.append(track\_idx)

            unmatched\_detections.append(detection\_idx)

        else:

            matches.append((track\_idx, detection\_idx))

    return matches, unmatched\_tracks, unmatched\_detections

def matching\_cascade(

        distance\_metric, max\_distance, cascade\_depth, tracks, detections,

        track\_indices=None, detection\_indices=None):

    """Run matching cascade.

    Parameters

    ----------

    distance\_metric : Callable[List[Track], List[Detection], List[int], List[int]) -> ndarray

        The distance metric is given a list of tracks and detections as well as

        a list of N track indices and M detection indices. The metric should

        return the NxM dimensional cost matrix, where element (i, j) is the

        association cost between the i-th track in the given track indices and

        the j-th detection in the given detection indices.

    max\_distance : float

        Gating threshold. Associations with cost larger than this value are

        disregarded.

    cascade\_depth: int

        The cascade depth, should be se to the maximum track age.

    tracks : List[track.Track]

        A list of predicted tracks at the current time step.

    detections : List[detection.Detection]

        A list of detections at the current time step.

    track\_indices : Optional[List[int]]

        List of track indices that maps rows in `cost\_matrix` to tracks in

        `tracks` (see description above). Defaults to all tracks.

    detection\_indices : Optional[List[int]]

        List of detection indices that maps columns in `cost\_matrix` to

        detections in `detections` (see description above). Defaults to all

        detections.

    Returns

    -------

    (List[(int, int)], List[int], List[int])

        Returns a tuple with the following three entries:

        \* A list of matched track and detection indices.

        \* A list of unmatched track indices.

        \* A list of unmatched detection indices.

    """

    if track\_indices is None:

        track\_indices = list(range(len(tracks)))

    if detection\_indices is None:

        detection\_indices = list(range(len(detections)))

    unmatched\_detections = detection\_indices

    matches = []

    for level in range(cascade\_depth):

        if len(unmatched\_detections) == 0:  # No detections left

            break

        track\_indices\_l = [

            k for k in track\_indices

            if tracks[k].time\_since\_update == 1 + level

        ]

        if len(track\_indices\_l) == 0:  # Nothing to match at this level

            continue

        matches\_l, \_, unmatched\_detections = \

            min\_cost\_matching(

                distance\_metric, max\_distance, tracks, detections,

                track\_indices\_l, unmatched\_detections)

        matches += matches\_l

    unmatched\_tracks = list(set(track\_indices) - set(k for k, \_ in matches))

    return matches, unmatched\_tracks, unmatched\_detections

def gate\_cost\_matrix(

        kf, cost\_matrix, tracks, detections, track\_indices, detection\_indices,

        gated\_cost=INFTY\_COST, only\_position=False):

    """Invalidate infeasible entries in cost matrix based on the state

    distributions obtained by Kalman filtering.

    Parameters

    ----------

    kf : The Kalman filter.

    cost\_matrix : ndarray

        The NxM dimensional cost matrix, where N is the number of track indices

        and M is the number of detection indices, such that entry (i, j) is the

        association cost between `tracks[track\_indices[i]]` and

        `detections[detection\_indices[j]]`.

    tracks : List[track.Track]

        A list of predicted tracks at the current time step.

    detections : List[detection.Detection]

        A list of detections at the current time step.

    track\_indices : List[int]

        List of track indices that maps rows in `cost\_matrix` to tracks in

        `tracks` (see description above).

    detection\_indices : List[int]

        List of detection indices that maps columns in `cost\_matrix` to

        detections in `detections` (see description above).

    gated\_cost : Optional[float]

        Entries in the cost matrix corresponding to infeasible associations are

        set this value. Defaults to a very large value.

    only\_position : Optional[bool]

        If True, only the x, y position of the state distribution is considered

        during gating. Defaults to False.

    Returns

    -------

    ndarray

        Returns the modified cost matrix.

    """

    gating\_dim = 2 if only\_position else 4

    gating\_threshold = kalman\_filter.chi2inv95[gating\_dim]

    measurements = np.asarray(

        [detections[i].to\_xyah() for i in detection\_indices])

    for row, track\_idx in enumerate(track\_indices):

        track = tracks[track\_idx]

        gating\_distance = kf.gating\_distance(

            track.mean, track.covariance, measurements, only\_position)

        cost\_matrix[row, gating\_distance > gating\_threshold] = gated\_cost

    return cost\_matrix

Linear\_assignmen.py

# vim: expandtab:ts=4:sw=4

import numpy as np

def \_pdist(a, b):

    """Compute pair-wise squared distance between points in `a` and `b`.

    Parameters

    ----------

    a : array\_like

        An NxM matrix of N samples of dimensionality M.

    b : array\_like

        An LxM matrix of L samples of dimensionality M.

    Returns

    -------

    ndarray

        Returns a matrix of size len(a), len(b) such that eleement (i, j)

        contains the squared distance between `a[i]` and `b[j]`.

    """

    a, b = np.asarray(a), np.asarray(b)

    if len(a) == 0 or len(b) == 0:

        return np.zeros((len(a), len(b)))

    a2, b2 = np.square(a).sum(axis=1), np.square(b).sum(axis=1)

    r2 = -2. \* np.dot(a, b.T) + a2[:, None] + b2[None, :]

    r2 = np.clip(r2, 0., float(np.inf))

    return r2

def \_cosine\_distance(a, b, data\_is\_normalized=False):

    """Compute pair-wise cosine distance between points in `a` and `b`.

    Parameters

    ----------

    a : array\_like

        An NxM matrix of N samples of dimensionality M.

    b : array\_like

        An LxM matrix of L samples of dimensionality M.

    data\_is\_normalized : Optional[bool]

        If True, assumes rows in a and b are unit length vectors.

        Otherwise, a and b are explicitly normalized to lenght 1.

    Returns

    -------

    ndarray

        Returns a matrix of size len(a), len(b) such that eleement (i, j)

        contains the squared distance between `a[i]` and `b[j]`.

    """

    if not data\_is\_normalized:

        a = np.asarray(a) / np.linalg.norm(a, axis=1, keepdims=True)

        b = np.asarray(b) / np.linalg.norm(b, axis=1, keepdims=True)

    return 1. - np.dot(a, b.T)

def \_nn\_euclidean\_distance(x, y):

    """ Helper function for nearest neighbor distance metric (Euclidean).

    Parameters

    ----------

    x : ndarray

        A matrix of N row-vectors (sample points).

    y : ndarray

        A matrix of M row-vectors (query points).

    Returns

    -------

    ndarray

        A vector of length M that contains for each entry in `y` the

        smallest Euclidean distance to a sample in `x`.

    """

    distances = \_pdist(x, y)

    return np.maximum(0.0, distances.min(axis=0))

def \_nn\_cosine\_distance(x, y):

    """ Helper function for nearest neighbor distance metric (cosine).

    Parameters

    ----------

    x : ndarray

        A matrix of N row-vectors (sample points).

    y : ndarray

        A matrix of M row-vectors (query points).

    Returns

    -------

    ndarray

        A vector of length M that contains for each entry in `y` the

        smallest cosine distance to a sample in `x`.

    """

    distances = \_cosine\_distance(x, y)

    return distances.min(axis=0)

class NearestNeighborDistanceMetric(object):

    """

    A nearest neighbor distance metric that, for each target, returns

    the closest distance to any sample that has been observed so far.

    Parameters

    ----------

    metric : str

        Either "euclidean" or "cosine".

    matching\_threshold: float

        The matching threshold. Samples with larger distance are considered an

        invalid match.

    budget : Optional[int]

        If not None, fix samples per class to at most this number. Removes

        the oldest samples when the budget is reached.

    Attributes

    ----------

    samples : Dict[int -> List[ndarray]]

        A dictionary that maps from target identities to the list of samples

        that have been observed so far.

    """

    def \_\_init\_\_(self, metric, matching\_threshold, budget=None):

        if metric == "euclidean":

            self.\_metric = \_nn\_euclidean\_distance

        elif metric == "cosine":

            self.\_metric = \_nn\_cosine\_distance

        else:

            raise ValueError(

                "Invalid metric; must be either 'euclidean' or 'cosine'")

        self.matching\_threshold = matching\_threshold

        self.budget = budget

        self.samples = {}

    def partial\_fit(self, features, targets, active\_targets):

        """Update the distance metric with new data.

        Parameters

        ----------

        features : ndarray

            An NxM matrix of N features of dimensionality M.

        targets : ndarray

            An integer array of associated target identities.

        active\_targets : List[int]

            A list of targets that are currently present in the scene.

        """

        for feature, target in zip(features, targets):

            self.samples.setdefault(target, []).append(feature)

            if self.budget is not None:

                self.samples[target] = self.samples[target][-self.budget:]

        self.samples = {k: self.samples[k] for k in active\_targets}

    def distance(self, features, targets):

        """Compute distance between features and targets.

        Parameters

        ----------

        features : ndarray

            An NxM matrix of N features of dimensionality M.

        targets : List[int]

            A list of targets to match the given `features` against.

        Returns

        -------

        ndarray

            Returns a cost matrix of shape len(targets), len(features), where

            element (i, j) contains the closest squared distance between

            `targets[i]` and `features[j]`.

        """

        cost\_matrix = np.zeros((len(targets), len(features)))

        for i, target in enumerate(targets):

            cost\_matrix[i, :] = self.\_metric(self.samples[target], features)

        return cost\_matrix

Nn matching.py

# vim: expandtab:ts=4:sw=4

import numpy as np

import cv2

def non\_max\_suppression(boxes, classes, max\_bbox\_overlap, scores=None):

    """Suppress overlapping detections.

    Original code from [1]\_ has been adapted to include confidence score.

    .. [1] http://www.pyimagesearch.com/2015/02/16/

           faster-non-maximum-suppression-python/

    Examples

    --------

        >>> boxes = [d.roi for d in detections]

        >>> classes = [d.classes for d in detections]

        >>> scores = [d.confidence for d in detections]

        >>> indices = non\_max\_suppression(boxes, max\_bbox\_overlap, scores)

        >>> detections = [detections[i] for i in indices]

    Parameters

    ----------

    boxes : ndarray

        Array of ROIs (x, y, width, height).

    max\_bbox\_overlap : float

        ROIs that overlap more than this values are suppressed.

    scores : Optional[array\_like]

        Detector confidence score.

    Returns

    -------

    List[int]

        Returns indices of detections that have survived non-maxima suppression.

    """

    if len(boxes) == 0:

        return []

    boxes = boxes.astype(np.float)

    pick = []

    x1 = boxes[:, 0]

    y1 = boxes[:, 1]

    x2 = boxes[:, 2] + boxes[:, 0]

    y2 = boxes[:, 3] + boxes[:, 1]

    area = (x2 - x1 + 1) \* (y2 - y1 + 1)

    if scores is not None:

        idxs = np.argsort(scores)

    else:

        idxs = np.argsort(y2)

    while len(idxs) > 0:

        last = len(idxs) - 1

        i = idxs[last]

        pick.append(i)

        xx1 = np.maximum(x1[i], x1[idxs[:last]])

        yy1 = np.maximum(y1[i], y1[idxs[:last]])

        xx2 = np.minimum(x2[i], x2[idxs[:last]])

        yy2 = np.minimum(y2[i], y2[idxs[:last]])

        w = np.maximum(0, xx2 - xx1 + 1)

        h = np.maximum(0, yy2 - yy1 + 1)

        overlap = (w \* h) / area[idxs[:last]]

        idxs = np.delete(

            idxs, np.concatenate(

                ([last], np.where(overlap > max\_bbox\_overlap)[0])))

    return pick

Pre processing.py

# vim: expandtab:ts=4:sw=4

class TrackState:

    """

    Enumeration type for the single target track state. Newly created tracks are

    classified as `tentative` until enough evidence has been collected. Then,

    the track state is changed to `confirmed`. Tracks that are no longer alive

    are classified as `deleted` to mark them for removal from the set of active

    tracks.

    """

    Tentative = 1

    Confirmed = 2

    Deleted = 3

    Recorded = 4

class Track:

    """

    A single target track with state space `(x, y, a, h)` and associated

    velocities, where `(x, y)` is the center of the bounding box, `a` is the

    aspect ratio and `h` is the height.

    Parameters

    ----------

    mean : ndarray

        Mean vector of the initial state distribution.

    covariance : ndarray

        Covariance matrix of the initial state distribution.

    track\_id : int

        A unique track identifier.

    n\_init : int

        Number of consecutive detections before the track is confirmed. The

        track state is set to `Deleted` if a miss occurs within the first

        `n\_init` frames.

    max\_age : int

        The maximum number of consecutive misses before the track state is

        set to `Deleted`.

    feature : Optional[ndarray]

        Feature vector of the detection this track originates from. If not None,

        this feature is added to the `features` cache.

    Attributes

    ----------

    mean : ndarray

        Mean vector of the initial state distribution.

    covariance : ndarray

        Covariance matrix of the initial state distribution.

    track\_id : int

        A unique track identifier.

    hits : int

        Total number of measurement updates.

    age : int

        Total number of frames since first occurance.

    time\_since\_update : int

        Total number of frames since last measurement update.

    state : TrackState

        The current track state.

    features : List[ndarray]

        A cache of features. On each measurement update, the associated feature

        vector is added to this list.

    """

    def \_\_init\_\_(self, mean, covariance, track\_id, entry, position, n\_init,

        max\_age, feature=None):

        self.mean = mean

        self.covariance = covariance

        self.track\_id = track\_id

        self.hits = 1

        self.age = 1

        self.time\_since\_update = 0

        self.state = TrackState.Tentative

        self.features = []

        if feature is not None:

            self.features.append(feature)

        self.\_n\_init = n\_init

        self.\_max\_age = max\_age

        # Movement trails, recorded by centroids

        self.positions = [position]

        # Initial detection

        self.entry = entry

        self.exit = None

    def to\_tlwh(self):

        """Get current position in bounding box format `(top left x, top left y,

        width, height)`.

        Returns

        -------

        ndarray

            The bounding box.

        """

        ret = self.mean[:4].copy()

        ret[2] \*= ret[3]

        ret[:2] -= ret[2:] / 2

        return ret

    def to\_tlbr(self):

        """Get current position in bounding box format `(min x, miny, max x,

        max y)`.

        Returns

        -------

        ndarray

            The bounding box.

        """

        ret = self.to\_tlwh()

        ret[2:] = ret[:2] + ret[2:]

        return ret

    def predict(self, kf):

        """Propagate the state distribution to the current time step using a

        Kalman filter prediction step.

        Parameters

        ----------

        kf : kalman\_filter.KalmanFilter

            The Kalman filter.

        """

        self.mean, self.covariance = kf.predict(self.mean, self.covariance)

        self.age += 1

        self.time\_since\_update += 1

    def update(self, kf, detection):

        """Perform Kalman filter measurement update step and update the feature

        cache.

        Parameters

        ----------

        kf : kalman\_filter.KalmanFilter

            The Kalman filter.

        detection : Detection

            The associated detection.

        """

        self.mean, self.covariance = kf.update(

            self.mean, self.covariance, detection.to\_xyah())

        self.features.append(detection.feature)

        self.positions.append(detection.centroid)

        self.hits += 1

        self.time\_since\_update = 0

        if self.state == TrackState.Tentative and self.hits >= self.\_n\_init:

            self.state = TrackState.Confirmed

    def mark\_missed(self):

        """Mark this track as missed (no association at the current time step).

        """

        if self.state == TrackState.Tentative:

            self.state = TrackState.Deleted

        elif self.time\_since\_update > self.\_max\_age:

            self.state = TrackState.Recorded

    def is\_tentative(self):

        """Returns True if this track is tentative (unconfirmed).

        """

        return self.state == TrackState.Tentative

    def is\_confirmed(self):

        """Returns True if this track is confirmed."""

        return self.state == TrackState.Confirmed

    def is\_deleted(self):

        """Returns True if this track is dead and should be deleted."""

        return self.state == TrackState.Deleted

    def is\_recorded(self):

        """Returns True if this track is dead and should be recorded."""

        return self.state == TrackState.Recorded

Track.py

# vim: expandtab:ts=4:sw=4

from \_\_future\_\_ import absolute\_import

import numpy as np

from . import kalman\_filter

from . import linear\_assignment

from . import iou\_matching

from .track import Track

class Tracker:

    """

    This is the multi-target tracker.

    Parameters

    ----------

    metric : nn\_matching.NearestNeighborDistanceMetric

        A distance metric for measurement-to-track association.

    max\_age : int

        Maximum number of missed misses before a track is deleted.

    n\_init : int

        Number of consecutive detections before the track is confirmed. The

        track state is set to `Deleted` if a miss occurs within the first

        `n\_init` frames.

    Attributes

    ----------

    metric : nn\_matching.NearestNeighborDistanceMetric

        The distance metric used for measurement to track association.

    max\_age : int

        Maximum number of missed misses before a track is deleted.

    n\_init : int

        Number of frames that a track remains in initialization phase.

    kf : kalman\_filter.KalmanFilter

        A Kalman filter to filter target trajectories in image space.

    tracks : List[Track]

        The list of active tracks at the current time step.

    """

    def \_\_init\_\_(self, metric, max\_iou\_distance=0.7, max\_age=30, n\_init=3):

        self.metric = metric

        self.max\_iou\_distance = max\_iou\_distance

        self.max\_age = max\_age

        self.n\_init = n\_init

        self.kf = kalman\_filter.KalmanFilter()

        self.tracks = []

        self.\_next\_id = 1

    def predict(self):

        """Propagate track state distributions one time step forward.

        This function should be called once every time step, before `update`.

        """

        for track in self.tracks:

            track.predict(self.kf)

    def update(self, detections, time):

        """Perform measurement update and track management.

        Parameters

        ----------

        detections : List[deep\_sort.detection.Detection]

            A list of detections at the current time step.

        """

        # Run matching cascade.

        matches, unmatched\_tracks, unmatched\_detections = self.\_match(detections)

        # Update track set.

        for track\_idx, detection\_idx in matches:

            self.tracks[track\_idx].update(self.kf, detections[detection\_idx])

        for track\_idx in unmatched\_tracks:

            self.tracks[track\_idx].mark\_missed()

        for detection\_idx in unmatched\_detections:

            self.\_initiate\_track(detections[detection\_idx], time)

        expired = []

        for t in self.tracks:

            if t.is\_recorded():

                t.exit = time

                expired.append(t)

        self.tracks = [t for t in self.tracks if not t.is\_deleted() and not t.is\_recorded()]

        # Update distance metric.

        active\_targets = [t.track\_id for t in self.tracks if t.is\_confirmed()]

        features, targets = [], []

        for track in self.tracks:

            if not track.is\_confirmed():

                continue

            features += track.features

            targets += [track.track\_id for \_ in track.features]

            track.features = []

        self.metric.partial\_fit(

            np.asarray(features), np.asarray(targets), active\_targets)

        return expired

    def \_match(self, detections):

        def gated\_metric(tracks, dets, track\_indices, detection\_indices):

            features = np.array([dets[i].feature for i in detection\_indices])

            targets = np.array([tracks[i].track\_id for i in track\_indices])

            cost\_matrix = self.metric.distance(features, targets)

            cost\_matrix = linear\_assignment.gate\_cost\_matrix(

                self.kf, cost\_matrix, tracks, dets, track\_indices,

                detection\_indices)

            return cost\_matrix

        # Split track set into confirmed and unconfirmed tracks.

        confirmed\_tracks = [

            i for i, t in enumerate(self.tracks) if t.is\_confirmed()]

        unconfirmed\_tracks = [

            i for i, t in enumerate(self.tracks) if not t.is\_confirmed()]

        # Associate confirmed tracks using appearance features.

        matches\_a, unmatched\_tracks\_a, unmatched\_detections = \

            linear\_assignment.matching\_cascade(

                gated\_metric, self.metric.matching\_threshold, self.max\_age,

                self.tracks, detections, confirmed\_tracks)

        # Associate remaining tracks together with unconfirmed tracks using IOU.

        iou\_track\_candidates = unconfirmed\_tracks + [

            k for k in unmatched\_tracks\_a if

            self.tracks[k].time\_since\_update == 1]

        unmatched\_tracks\_a = [

            k for k in unmatched\_tracks\_a if

            self.tracks[k].time\_since\_update != 1]

        matches\_b, unmatched\_tracks\_b, unmatched\_detections = \

            linear\_assignment.min\_cost\_matching(

                iou\_matching.iou\_cost, self.max\_iou\_distance, self.tracks,

                detections, iou\_track\_candidates, unmatched\_detections)

        matches = matches\_a + matches\_b

        unmatched\_tracks = list(set(unmatched\_tracks\_a + unmatched\_tracks\_b))

        return matches, unmatched\_tracks, unmatched\_detections

    def \_initiate\_track(self, detection, time):

        mean, covariance = self.kf.initiate(detection.to\_xyah())

        self.tracks.append(Track(

            mean, covariance, self.\_next\_id, time, detection.centroid, self.n\_init,

            self.max\_age, detection.feature))

        self.\_next\_id += 1

Tracker.py

[net]

# Testing

#batch=1

#subdivisions=1

# Training

batch=64

subdivisions=1

width=416

height=416

channels=3

momentum=0.9

decay=0.0005

angle=0

saturation = 1.5

exposure = 1.5

hue=.1

learning\_rate=0.00261

burn\_in=1000

max\_batches = 2000200

policy=steps

steps=1600000,1800000

scales=.1,.1

#weights\_reject\_freq=1001

#ema\_alpha=0.9998

#equidistant\_point=1000

#num\_sigmas\_reject\_badlabels=3

#badlabels\_rejection\_percentage=0.2

[convolutional]

batch\_normalize=1

filters=32

size=3

stride=2

pad=1

activation=leaky

[convolutional]

batch\_normalize=1

filters=64

size=3

stride=2

pad=1

activation=leaky

[convolutional]

batch\_normalize=1

filters=64

size=3

stride=1

pad=1

activation=leaky

[route]

layers=-1

groups=2

group\_id=1

[convolutional]

batch\_normalize=1

filters=32

size=3

stride=1

pad=1

activation=leaky

[convolutional]

batch\_normalize=1

filters=32

size=3

stride=1

pad=1

activation=leaky

[route]

layers = -1,-2

[convolutional]

batch\_normalize=1

filters=64

size=1

stride=1

pad=1

activation=leaky

[route]

layers = -6,-1

[maxpool]

size=2

stride=2

[convolutional]

batch\_normalize=1

filters=128

size=3

stride=1

pad=1

activation=leaky

[route]

layers=-1

groups=2

group\_id=1

[convolutional]

batch\_normalize=1

filters=64

size=3

stride=1

pad=1

activation=leaky

[convolutional]

batch\_normalize=1

filters=64

size=3

stride=1

pad=1

activation=leaky

[route]

layers = -1,-2

[convolutional]

batch\_normalize=1

filters=128

size=1

stride=1

pad=1

activation=leaky

[route]

layers = -6,-1

[maxpool]

size=2

stride=2

[convolutional]

batch\_normalize=1

filters=256

size=3

stride=1

pad=1

activation=leaky

[route]

layers=-1

groups=2

group\_id=1

[convolutional]

batch\_normalize=1

filters=128

size=3

stride=1

pad=1

activation=leaky

[convolutional]

batch\_normalize=1

filters=128

size=3

stride=1

pad=1

activation=leaky

[route]

layers = -1,-2

[convolutional]

batch\_normalize=1

filters=256

size=1

stride=1

pad=1

activation=leaky

[route]

layers = -6,-1

[maxpool]

size=2

stride=2

[convolutional]

batch\_normalize=1

filters=512

size=3

stride=1

pad=1

activation=leaky

##################################

[convolutional]

batch\_normalize=1

filters=256

size=1

stride=1

pad=1

activation=leaky

[convolutional]

batch\_normalize=1

filters=512

size=3

stride=1

pad=1

activation=leaky

[convolutional]

size=1

stride=1

pad=1

filters=255

activation=linear

[yolo]

mask = 3,4,5

anchors = 10,14,  23,27,  37,58,  81,82,  135,169,  344,319

classes=80

num=6

jitter=.3

scale\_x\_y = 1.05

cls\_normalizer=1.0

iou\_normalizer=0.07

iou\_loss=ciou

ignore\_thresh = .7

truth\_thresh = 1

random=0

resize=1.5

nms\_kind=greedynms

beta\_nms=0.6

#new\_coords=1

#scale\_x\_y = 2.0

[route]

layers = -4

[convolutional]

batch\_normalize=1

filters=128

size=1

stride=1

pad=1

activation=leaky

[upsample]

stride=2

[route]

layers = -1, 23

[convolutional]

batch\_normalize=1

filters=256

size=3

stride=1

pad=1

activation=leaky

[convolutional]

size=1

stride=1

pad=1

filters=255

activation=linear

[yolo]

mask = 1,2,3

anchors = 10,14,  23,27,  37,58,  81,82,  135,169,  344,319

classes=80

num=6

jitter=.3

scale\_x\_y = 1.05

cls\_normalizer=1.0

iou\_normalizer=0.07

iou\_loss=ciou

ignore\_thresh = .7

truth\_thresh = 1

random=0

resize=1.5

nms\_kind=greedynms

beta\_nms=0.6

#new\_coords=1

#scale\_x\_y = 2.0

Ylov4-tiny.cfg

# Byte-compiled / optimized / DLL files

\_\_pycache\_\_/

\*.py[cod]

\*$py.class

# C extensions

\*.so

# Distribution / packaging

.Python

build/

develop-eggs/

dist/

downloads/

eggs/

.eggs/

lib/

lib64/

parts/

sdist/

var/

wheels/

pip-wheel-metadata/

share/python-wheels/

\*.egg-info/

.installed.cfg

\*.egg

MANIFEST

# PyInstaller

#  Usually these files are written by a python script from a template

#  before PyInstaller builds the exe, so as to inject date/other infos into it.

\*.manifest

\*.spec

# Installer logs

pip-log.txt

pip-delete-this-directory.txt

# Unit test / coverage reports

htmlcov/

.tox/

.nox/

.coverage

.coverage.\*

.cache

nosetests.xml

coverage.xml

\*.cover

\*.py,cover

.hypothesis/

.pytest\_cache/

# Translations

\*.mo

\*.pot

# Django stuff:

\*.log

local\_settings.py

db.sqlite3

db.sqlite3-journal

# Flask stuff:

instance/

.webassets-cache

# Scrapy stuff:

.scrapy

# Sphinx documentation

docs/\_build/

# PyBuilder

target/

# Jupyter Notebook

.ipynb\_checkpoints

# IPython

profile\_default/

ipython\_config.py

# pyenv

.python-version

# pipenv

#   According to pypa/pipenv#598, it is recommended to include Pipfile.lock in version control.

#   However, in case of collaboration, if having platform-specific dependencies or dependencies

#   having no cross-platform support, pipenv may install dependencies that don't work, or not

#   install all needed dependencies.

#Pipfile.lock

# PEP 582; used by e.g. github.com/David-OConnor/pyflow

\_\_pypackages\_\_/

# Celery stuff

celerybeat-schedule

celerybeat.pid

# SageMath parsed files

\*.sage.py

# Environments

.env

.venv

env/

venv/

ENV/

env.bak/

venv.bak/

# Spyder project settings

.spyderproject

.spyproject

# Rope project settings

.ropeproject

# mkdocs documentation

/site

# mypy

.mypy\_cache/

.dmypy.json

dmypy.json

# Pyre type checker

.pyre/

# venv

myenv/

# Video files

video/

# YOLO config and weights

YOLOv3-tiny/

YOLOv4-tiny/

YOLOv4/

# Processed data files

processed\_data/

movement\_data.csv

crowd\_data.csv

video\_data.json

Gitinore

import matplotlib

matplotlib.use('tkagg')

import matplotlib.pyplot as plt

import csv

import json

import numpy as np

import pandas as pd

from math import ceil

from scipy.spatial.distance import euclidean

with open('processed\_data/video\_data.json', 'r') as file:

    data = json.load(file)

    data\_record\_frame = data["DATA\_RECORD\_FRAME"]

    frame\_size = data["PROCESSED\_FRAME\_SIZE"]

    vid\_fps = data["VID\_FPS"]

    track\_max\_age = data["TRACK\_MAX\_AGE"]

track\_max\_age = 3

time\_steps = data\_record\_frame/vid\_fps

stationary\_time = ceil(track\_max\_age / time\_steps)

stationary\_distance = frame\_size \* 0.01

tracks = []

with open('processed\_data/movement\_data.csv', 'r') as file:

    reader = csv.reader(file, delimiter=',')

    for row in reader:

        if len(row[3:]) > stationary\_time \* 2:

            temp = []

            data = row[3:]

            for i in range(0, len(data), 2):

                temp.append([int(data[i]), int(data[i+1])])

            tracks.append(temp)

print("Tracks recorded: " + str(len(tracks)))

useful\_tracks = []

for movement in tracks:

    check\_index = stationary\_time

    start\_point = 0

    track = movement[:check\_index]

    while check\_index < len(movement):

        for i in movement[check\_index:]:

            if euclidean(movement[start\_point], i) > stationary\_distance:

                track.append(i)

                start\_point += 1

                check\_index += 1

            else:

                start\_point += 1

                check\_index += 1

                break

        useful\_tracks.append(track)

        track = movement[start\_point:check\_index]

energies = []

for movement in useful\_tracks:

    for i in range(len(movement) - 1):

        speed = round(euclidean(movement[i], movement[i+1]) / time\_steps , 2)

        energy = int(0.5 \* speed \*\* 2)

        energies.append(energy)

c = len(energies)

print()

print("Useful movement data: " + str(c))

energies = pd.Series(energies)

x = { 'Energy': energies}

df = pd.DataFrame(x)

print("Kurtosis: " + str(df.kurtosis()[0]))

print("Skew: " + str(df.skew()[0]))

print("Summary of processed data")

print(df.describe())

print("Acceptable energy level (mean value \*\* 1.05) is " + str(int(df.Energy.mean() \*\* 1.05)))

bins = np.linspace(int(min(energies)), int(max(energies)),100)

plt.xlim([min(energies)-5, max(energies)+5])

plt.hist(energies, bins=bins, alpha=0.5)

plt.title('Distribution of energies level')

plt.xlabel('Energy level')

plt.ylabel('Count')

plt.show()

while df.skew()[0] > 7.5:

    print()

    c = len(energies)

    print("Useful movement data: " + str(c))

    energies = energies[abs(energies - np.mean(energies)) < 3 \* np.std(energies)]

    x = { 'Energy': energies}

    df = pd.DataFrame(x)

    print("Outliers removed: " + str(c - df.Energy.count()))

    print("Kurtosis: " + str(df.kurtosis()[0]))

    print("Skew: " + str(df.skew()[0]))

    print("Summary of processed data")

    print(df.describe())

    print("Acceptable energy level (mean value \*\* 1.05) is " + str(int(df.Energy.mean() \*\* 1.05)))

    bins = np.linspace(int(min(energies)), int(max(energies)),100)

    plt.xlim([min(energies)-5, max(energies)+5])

    plt.hist(energies, bins=bins, alpha=0.5)

    plt.title('Distribution of energies level')

    plt.xlabel('Energy level')

    plt.ylabel('Count')

    plt.show()

Abnormal-data-process.py

RGB\_COLORS = {

    "blue": (255, 0, 0),

    "green": (0, 255, 0),

    "red": (0, 0, 255),

    "yellow": (0, 255, 255),

    "white": (0, 0, 0),

    "black": (255, 255, 255)

}

def gradient\_color\_RGB(color1, color2, steps, current):

    step1 = (color2[0] - color1[0])/steps

    step2 = (color2[1] - color1[1])/steps

    step3 = (color2[2] - color1[2])/steps

    color\_1 = int(color1[0] + current\*step1)

    color\_2 = int(color1[1] + current\*step2)

    color\_3 = int(color1[2] + current\*step3)

    return (color\_1, color\_2, color\_3)

Colors.py

import datetime

# Video Path

VIDEO\_CONFIG = {

    "VIDEO\_CAP" : 0,

    "IS\_CAM" : True,

    "CAM\_APPROX\_FPS": 3,

    "HIGH\_CAM": False,

    "START\_TIME": datetime.datetime(2020, 11, 5, 0, 0, 0, 0)

}

# Load YOLOv3-tiny weights and config

YOLO\_CONFIG = {

    "WEIGHTS\_PATH" : "YOLOv4-tiny/yolov4-tiny.weights",

    "CONFIG\_PATH" : "YOLOv4-tiny/yolov4-tiny.cfg"

}

# Show individuals detected

SHOW\_PROCESSING\_OUTPUT = True

# Show individuals detected

SHOW\_DETECT = True

# Data record

DATA\_RECORD = True

# Data record rate (data record per frame)

DATA\_RECORD\_RATE = 5

# Check for restricted entry

RE\_CHECK = False

# Restricted entry time (H:M:S)

RE\_START\_TIME = datetime.time(0,0,0)

RE\_END\_TIME = datetime.time(23,0,0)

# Check for social distance violation

SD\_CHECK = True

# Show violation count

SHOW\_VIOLATION\_COUNT = True

# Show tracking id

SHOW\_TRACKING\_ID = True

# Threshold for distance violation

SOCIAL\_DISTANCE = 250

# Check for abnormal crowd activity

ABNORMAL\_CHECK = True

# Min number of people to check for abnormal

ABNORMAL\_MIN\_PEOPLE = 50

# Abnormal energy level threshold

ABNORMAL\_ENERGY = 1866

# Abnormal activity ratio threhold

ABNORMAL\_THRESH = 0.66

# Threshold for human detection minumun confindence

MIN\_CONF = 0.3

# Threshold for Non-maxima surpression

NMS\_THRESH = 0.2

# Resize frame for processing

FRAME\_SIZE = 1080

# Tracker max missing age before removing (seconds)

TRACK\_MAX\_AGE = 3

#Mailer Alert

MAIL = 'araganigan@gmail.com'

ALERT = True

Threshold = 10

Config.py

import matplotlib

matplotlib.use('tkagg')

import matplotlib.pyplot as plt

import matplotlib.patches as patches

import matplotlib.dates as mdates

import csv

import json

import datetime

from math import floor

from send\_email import send\_email

human\_count = []

violate\_count = []

restricted\_entry = []

abnormal\_activity = []

with open('processed\_data/crowd\_data.csv', 'r') as file:

    reader = csv.reader(file, delimiter=',')

    next(reader)

    for row in reader:

        human\_count.append(int(row[1]))

        violate\_count.append(int(row[2]))

        restricted\_entry.append(bool(int(row[3])))

        abnormal\_activity.append(bool(int(row[4])))

with open('processed\_data/video\_data.json', 'r') as file:

    data = json.load(file)

    data\_record\_frame = data["DATA\_RECORD\_FRAME"]

    is\_cam = data["IS\_CAM"]

    vid\_fps = data["VID\_FPS"]

    start\_time = data["START\_TIME"]

start\_time = datetime.datetime.strptime(start\_time, "%d/%m/%Y, %H:%M:%S")

time\_steps = data\_record\_frame / vid\_fps

data\_length = len(human\_count)

time\_axis = []

graph\_height = max(human\_count)

fig, ax = plt.subplots()

time = start\_time

for i in range(data\_length):

    time += datetime.timedelta(seconds=time\_steps)

    time\_axis.append(time)

    next\_time = time + datetime.timedelta(seconds=time\_steps)

    rect\_width = mdates.date2num(next\_time) - mdates.date2num(time)

    if restricted\_entry[i]:

        ax.add\_patch(patches.Rectangle((mdates.date2num(time), 0), rect\_width, graph\_height / 10, facecolor='red', fill=True))

    if abnormal\_activity[i]:

        ax.add\_patch(patches.Rectangle((mdates.date2num(time), 0), rect\_width, graph\_height / 20, facecolor='blue', fill=True))

violate\_line, = plt.plot(time\_axis, violate\_count, linewidth=3, label="Violation Count")

crowd\_line, = plt.plot(time\_axis, human\_count, linewidth=3, label="Crowd Count")

plt.title("Crowd Data versus Time")

plt.xlabel("Time")

plt.ylabel("Count")

re\_legend = patches.Patch(color="red", label="Restricted Entry Detected")

an\_legend = patches.Patch(color="blue", label="Abnormal Crowd Activity Detected")

plt.legend(handles=[crowd\_line, violate\_line, re\_legend, an\_legend])

plt.show()

# Send email if abnormal activity detected

if any(abnormal\_activity):

    sender\_email = "davincidecoder878@gmail.com"  # Sender's email address

    receiver\_email = "araganigan@gmail.com"  # Recipient's email address

    password = "muoniwayweihhgyn"  # Sender's email password

    subject = 'Email Alert Student Mood Recognition System!'

    message = 'System detects Fear/Angry from students!.'

    send\_email(sender\_email, password, receiver\_email, subject, message)

Crowd data present.py

import sys

import os

from tkinter import \*

window=Tk()

window.title("Tourist Tracker and Counter Using Computer Vision With Recommender System")

window.geometry('640x410')

window.resizable(0, 0)

def run():

    os.system('python main.py')

x = Label(window,

                  text = "Tourist Tracker & Counter System",

                  font = 'Arial 24 bold').place(x = 65,

                                                y = 120)

btn = Button(window,

             text="Start Crowd Counting and Tracking",

             font = 'Arial 12 bold',

             bg="blue",

             fg="white",

             height= 2,

             width=35,

             command= run)

btn.place(relx=0.5, rely=0.5, anchor=CENTER)

window.mainloop()

Interface.py

import smtplib, ssl

class Mailer:

    """

    This script initiaties the email alert function.

    """

    def \_\_init\_\_(self):

        # Enter your email below. This email will be used to send alerts.

        # E.g., "email@gmail.com"

        self.EMAIL = "carlxavier.c.valdez@isu.edu.ph"

        # Enter the email password below. Note that the password varies if you have secured

        # 2 step verification turned on. You can refer the links below and create an application specific password.

        # Google mail has a guide here: https://myaccount.google.com/lesssecureapps

        # For 2 step verified accounts: https://support.google.com/accounts/answer/185833

        self.PASS = "thorodinson91827364"

        self.PORT = 465

        self.server = smtplib.SMTP\_SSL('smtp.gmail.com', self.PORT)

    def send(self, mail):

        self.server = smtplib.SMTP\_SSL('smtp.gmail.com', self.PORT)

        self.server.login(self.EMAIL, self.PASS)

        # message to be sent

        SUBJECT = 'ALERT!'

        TEXT = 'People limit exceeded in your building!'

        message = 'Subject: {}\n\n{}'.format(SUBJECT, TEXT)

        # sending the mail

        self.server.sendmail(self.EMAIL, mail, message)

        self.server.quit()

Mailer.py

from config import YOLO\_CONFIG, VIDEO\_CONFIG, SHOW\_PROCESSING\_OUTPUT, DATA\_RECORD\_RATE, FRAME\_SIZE, TRACK\_MAX\_AGE

import smtplib

from email.message import EmailMessage

if FRAME\_SIZE > 1920:

    print("Frame size is too large!")

    quit()

elif FRAME\_SIZE < 480:

    print("Frame size is too small! You won't see anything")

    quit()

from datetime import datetime

import time

import numpy as np

import cv2

import os

import csv

import json

from video\_process import video\_process

from deep\_sort import nn\_matching

from deep\_sort.detection import Detection

from deep\_sort.tracker import Tracker

from deep\_sort import generate\_detections as gdet

# Email configuration

sender\_email = "davincidecoder878@gmail.com"  # Sender's email address

receiver\_email = "araganigan@gmail.com"  # Recipient's email address

password = "mouniwayweihhgyn"  # Sender's email password

subject = 'Email Alert: More Than 10 People Detected!'

# Send email alert

def send\_email(sender\_email, sender\_password, receiver\_email, subject, message):

    # Create the email message

    email\_message = EmailMessage()

    email\_message['From'] = sender\_email

    email\_message['To'] = receiver\_email

    email\_message['Subject'] = subject

    email\_message.set\_content(message)

    # Set up the SMTP server

    smtp\_server = 'smtp.gmail.com'

    smtp\_port = 587

    # Connect to the SMTP server

    smtp\_connection = smtplib.SMTP(smtp\_server, smtp\_port)

    smtp\_connection.starttls()

    # Login to the email account

    smtp\_connection.login(sender\_email, sender\_password)

    # Send the email

    smtp\_connection.send\_message(email\_message)

    # Close the connection

    smtp\_connection.quit()

# Read from video

IS\_CAM = VIDEO\_CONFIG["IS\_CAM"]

cap = cv2.VideoCapture(VIDEO\_CONFIG["VIDEO\_CAP"])

# Load YOLOv3-tiny weights and config

WEIGHTS\_PATH = YOLO\_CONFIG["WEIGHTS\_PATH"]

CONFIG\_PATH = YOLO\_CONFIG["CONFIG\_PATH"]

# Load the YOLOv3-tiny pre-trained COCO dataset

net = cv2.dnn.readNetFromDarknet(CONFIG\_PATH, WEIGHTS\_PATH)

# Set the preferable backend to CPU since we are not using GPU

net.setPreferableBackend(cv2.dnn.DNN\_BACKEND\_OPENCV)

net.setPreferableTarget(cv2.dnn.DNN\_TARGET\_CPU)

# Get the names of all the layers in the network

ln = net.getLayerNames()

# Filter out the layer names we don't need for YOLO

ln = [ln[i - 1] for i in net.getUnconnectedOutLayers()]

# Tracker parameters

max\_cosine\_distance = 0.7

nn\_budget = None

# Initialize deep sort object

if IS\_CAM:

    max\_age = VIDEO\_CONFIG["CAM\_APPROX\_FPS"] \* TRACK\_MAX\_AGE

else:

    max\_age = DATA\_RECORD\_RATE \* TRACK\_MAX\_AGE

    if max\_age > 30:

        max\_age = 30

model\_filename = 'model\_data/mars-small128.pb'

encoder = gdet.create\_box\_encoder(model\_filename, batch\_size=1)

metric = nn\_matching.NearestNeighborDistanceMetric(

    "cosine", max\_cosine\_distance, nn\_budget)

tracker = Tracker(metric, max\_age=max\_age)

if not os.path.exists('processed\_data'):

    os.makedirs('processed\_data')

movement\_data\_file = open('processed\_data/movement\_data.csv', 'w')

crowd\_data\_file = open('processed\_data/crowd\_data.csv', 'w')

movement\_data\_writer = csv.writer(movement\_data\_file)

crowd\_data\_writer = csv.writer(crowd\_data\_file)

if os.path.getsize('processed\_data/movement\_data.csv') == 0:

    movement\_data\_writer.writerow(

        ['Track ID', 'Entry time', 'Exit Time', 'Movement Tracks'])

if os.path.getsize('processed\_data/crowd\_data.csv') == 0:

    crowd\_data\_writer.writerow(

        ['Time', 'Human Count', 'Social Distance violate', 'Restricted Entry', 'Abnormal Activity'])

START\_TIME = datetime.now()

processing\_FPS = video\_process(

    cap, FRAME\_SIZE, net, ln, encoder, tracker, movement\_data\_writer, crowd\_data\_writer)

cv2.destroyAllWindows()

movement\_data\_file.close()

crowd\_data\_file.close()

END\_TIME = datetime.now()

PROCESS\_TIME = (END\_TIME - START\_TIME).total\_seconds()

print("Time elapsed: ", PROCESS\_TIME)

if IS\_CAM:

    print("Processed FPS: ", processing\_FPS)

    VID\_FPS = processing\_FPS

    DATA\_RECORD\_FRAME = 1

else:

    print("Processed FPS: ", round(

        cap.get(cv2.CAP\_PROP\_FRAME\_COUNT) / PROCESS\_TIME, 2))

    VID\_FPS = cap.get(cv2.CAP\_PROP\_FPS)

    DATA\_RECORD\_FRAME = int(VID\_FPS / DATA\_RECORD\_RATE)

    START\_TIME = VIDEO\_CONFIG["START\_TIME"]

    time\_elapsed = round(cap.get(cv2.CAP\_PROP\_FRAME\_COUNT) / VID\_FPS)

    END\_TIME = START\_TIME + datetime.timedelta(seconds=time\_elapsed)

cap.release()

# Read crowd data from the CSV file

with open('processed\_data/crowd\_data.csv', 'r') as crowd\_data\_file:

    crowd\_data\_reader = csv.reader(crowd\_data\_file)

    next(crowd\_data\_reader)  # Skip the header row

    for row in crowd\_data\_reader:

        if len(row) >= 2 and int(row[1]) >= 2:  # Check for 5 or more people

            message = "More than 5 people detected!"  # Updated message

            send\_email(sender\_email, password, receiver\_email, subject, message)

        break

video\_data = {

    "IS\_CAM": IS\_CAM,

    "DATA\_RECORD\_FRAME": DATA\_RECORD\_FRAME,

    "VID\_FPS": VID\_FPS,

    "PROCESSED\_FRAME\_SIZE": FRAME\_SIZE,

    "TRACK\_MAX\_AGE": TRACK\_MAX\_AGE,

    "START\_TIME": START\_TIME.strftime("%d/%m/%Y, %H:%M:%S"),

    "END\_TIME": END\_TIME.strftime("%d/%m/%Y, %H:%M:%S")

}

with open('processed\_data/video\_data.json', 'w') as video\_data\_file:

    json.dump(video\_data, video\_data\_file)

Main.py

import csv

import imutils

import cv2

import json

import math

import numpy as np

from config import VIDEO\_CONFIG

from itertools import zip\_longest

from math import ceil

from scipy.spatial.distance import euclidean

from colors import RGB\_COLORS, gradient\_color\_RGB

tracks = []

with open('processed\_data/movement\_data.csv', 'r') as file:

    reader = csv.reader(file, delimiter=',')

    for row in reader:

        if len(row[3:]) > 4:

            temp = []

            data = row[3:]

            for i in range(0, len(data), 2):

                temp.append([int(data[i]), int(data[i+1])])

            tracks.append(temp)

with open('processed\_data/video\_data.json', 'r') as file:

    data = json.load(file)

    vid\_fps = data["VID\_FPS"]

    data\_record\_frame = data["DATA\_RECORD\_FRAME"]

    frame\_size = data["PROCESSED\_FRAME\_SIZE"]

cap = cv2.VideoCapture(VIDEO\_CONFIG["VIDEO\_CAP"])

cap.set(1, 100)

(ret, tracks\_frame) = cap.read()

tracks\_frame = imutils.resize(tracks\_frame, width=frame\_size)

heatmap\_frame = np.copy(tracks\_frame)

print(tracks\_frame.shape)

stationary\_threshold\_seconds = 2

stationary\_threshold\_frame =  round(vid\_fps \* stationary\_threshold\_seconds / data\_record\_frame)

stationary\_distance = frame\_size \* 0.05

max\_stationary\_time = 120

blob\_layer = 50

max\_blob\_size = frame\_size \* 0.1

layer\_size = max\_blob\_size / blob\_layer

color\_start = 210

color\_end = 0

color\_steps = int((color\_start - color\_end) / blob\_layer)

scale = 1.5

# print(stationary\_distance)

# print(stationary\_threshold\_frame)

stationary\_points = []

movement\_points = []

total = 0

for movement in tracks:

    temp\_movement\_point = [movement[0]]

    stationary = movement[0]

    stationary\_time = 0

    for i in movement[1:]:

        if euclidean(stationary, i) < stationary\_distance:

            stationary\_time += 1

        else:

            temp\_movement\_point.append(i)

            if stationary\_time > stationary\_threshold\_frame:

                stationary\_points.append([stationary, stationary\_time])

            stationary = i

            stationary\_time = 0

    movement\_points.append(temp\_movement\_point)

    total += len(temp\_movement\_point)

# print(total)

# print(movement\_points)

color1 = (255, 96, 0)

color2 = (0, 28, 255)

for track in movement\_points:

    for i in range(len(track) - 1):

        color = gradient\_color\_RGB(color1, color2, len(track) - 1, i)

        cv2.line(tracks\_frame, tuple(track[i]), tuple(track[i+1]), color, 2)

def draw\_blob(frame, coordinates, time):

    if time >= max\_stationary\_time:

        layer = blob\_layer

    else:

        layer = math.ceil(time \* scale / layer\_size)

    for x in reversed(range(layer)):

        color = color\_start - (color\_steps \* x)

        size = x \* layer\_size

        cv2.circle(frame, coordinates, int(size), (color, color, color), -1)

heatmap = np.zeros((heatmap\_frame.shape[0], heatmap\_frame.shape[1]), dtype=np.uint8)

for points in stationary\_points:

    draw\_heatmap = np.zeros((heatmap\_frame.shape[0], heatmap\_frame.shape[1]), dtype=np.uint8)

    draw\_blob(draw\_heatmap, tuple(points[0]), points[1])

    heatmap = cv2.add(heatmap, draw\_heatmap)

lo = np.array([color\_start])

hi = np.array([255])

mask = cv2.inRange(heatmap, lo, hi)

heatmap[mask > 0] = color\_start

heatmap = cv2.applyColorMap(heatmap, cv2.COLORMAP\_JET)

lo = np.array([128,0,0])

hi = np.array([136,0,0])

mask = cv2.inRange(heatmap, lo, hi)

heatmap[mask > 0] = (0, 0, 0)

for row in range(heatmap.shape[0]):

    for col in range(heatmap.shape[1]):

        if (heatmap[row][col] == np.array([0,0,0])).all():

            heatmap[row][col] = heatmap\_frame[row][col]

heatmap\_frame = cv2.addWeighted(heatmap, 0.75, heatmap\_frame, 0.25, 1)

cv2.imshow("Movement Tracks", tracks\_frame)

cv2.imshow("Stationary Location Heatmap", heatmap\_frame)

cv2.waitKey()

cv2.destroyAllWindows()

cap.release()

Movement-data-present.py

import smtplib

from email.message import EmailMessage

def send\_email(sender\_email, sender\_password, receiver\_email, subject, message):

    # Create the email message

    email\_message = EmailMessage()

    email\_message['From'] = sender\_email

    email\_message['To'] = receiver\_email

    email\_message['Subject'] = subject

    email\_message.set\_content(message)

    # Set up the SMTP server

    smtp\_server = 'smtp.gmail.com'

    smtp\_port = 587

    # Connect to the SMTP server

    smtp\_connection = smtplib.SMTP(smtp\_server, smtp\_port)

    smtp\_connection.starttls()

    # Login to the Gmail account

    smtp\_connection.login(sender\_email, sender\_password)

    # Send the email

    smtp\_connection.send\_message(email\_message)

    # Close the connection

    smtp\_connection.quit()

Send-email.py

import smtplib

from email.message import EmailMessage

# Email configuration

sender\_email = "davincidecoder878@gmail.com"  # Sender's email address

receiver\_email = "araganigan@gmail.com"  # Recipient's email address

password = "muoniwayweihhgyn"  # Sender's email password

subject = 'Crowd Detected!'

message = 'Attention! The current crowd density in the Bountiful Garden has reached high levels. Please exercise caution and consider reducing entry to the area.'

# Send email alert

def send\_email(sender\_email, sender\_password, receiver\_email, subject, message):

    # Create the email message

    email\_message = EmailMessage()

    email\_message['From'] = sender\_email

    email\_message['To'] = receiver\_email

    email\_message['Subject'] = subject

    email\_message.set\_content(message)

    # Set up the SMTP server

    smtp\_server = 'smtp.gmail.com'

    smtp\_port = 587

    # Connect to the SMTP server

    smtp\_connection = smtplib.SMTP(smtp\_server, smtp\_port)

    smtp\_connection.starttls()

    # Login to the Gmail account

    smtp\_connection.login(sender\_email, sender\_password)

    # Send the email

    smtp\_connection.send\_message(email\_message)

    # Close the connection

    smtp\_connection.quit()

if \_\_name\_\_ == "\_\_main\_\_":

    send\_email(sender\_email, password, receiver\_email, subject, message)

Test-email.py

import numpy as np

import cv2

from config import MIN\_CONF, NMS\_THRESH

from deep\_sort import nn\_matching

from deep\_sort.detection import Detection

from deep\_sort.tracker import Tracker

from deep\_sort import generate\_detections as gdet

def detect\_human (net, ln, frame, encoder, tracker, time):

# Get the dimension of the frame

    (frame\_height, frame\_width) = frame.shape[:2]

    # Initialize lists needed for detection

    boxes = []

    centroids = []

    confidences = []

    # Construct a blob from the input frame

    blob = cv2.dnn.blobFromImage(frame, 1 / 255.0, (416, 416),

        swapRB=True, crop=False)

    # Perform forward pass of YOLOv3, output are the boxes and probabilities

    net.setInput(blob)

    layer\_outputs = net.forward(ln)

    # For each output

    for output in layer\_outputs:

        # For each detection in output

        for detection in output:

            # Extract the class ID and confidence

            scores = detection[5:]

            class\_id = np.argmax(scores)

            confidence = scores[class\_id]

            # Class ID for person is 0, check if the confidence meet threshold

            if class\_id == 0 and confidence > MIN\_CONF:

                # Scale the bounding box coordinates back to the size of the image

                box = detection[0:4] \* np.array([frame\_width, frame\_height, frame\_width, frame\_height])

                (center\_x, center\_y, width, height) = box.astype("int")

                # Derive the coordinates for the top left corner of the bounding box

                x = int(center\_x - (width / 2))

                y = int(center\_y - (height / 2))

                # Add processed results to respective list

                boxes.append([x, y, int(width), int(height)])

                centroids.append((center\_x, center\_y))

                confidences.append(float(confidence))

    # Perform Non-maxima suppression to suppress weak and overlapping boxes

    # It will filter out unnecessary boxes, i.e. box within box

    # Output will be indexs of useful boxes

    idxs = cv2.dnn.NMSBoxes(boxes, confidences, MIN\_CONF, NMS\_THRESH)

    tracked\_bboxes = []

    expired = []

    if len(idxs) > 0:

        del\_idxs = []

        for i in range(len(boxes)):

            if i not in idxs:

                del\_idxs.append(i)

        for i in sorted(del\_idxs, reverse=True):

            del boxes[i]

            del centroids[i]

            del confidences[i]

        boxes = np.array(boxes)

        centroids = np.array(centroids)

        confidences = np.array(confidences)

        features = np.array(encoder(frame, boxes))

        detections = [Detection(bbox, score, centroid, feature) for bbox, score, centroid, feature in zip(boxes, scores, centroids, features)]

        tracker.predict()

        expired = tracker.update(detections, time)

        # Obtain info from the tracks

        for track in tracker.tracks:

                if not track.is\_confirmed() or track.time\_since\_update > 5:

                        continue

                tracked\_bboxes.append(track)

    return [tracked\_bboxes, expired]

Tracking.py

from scipy.spatial.distance import euclidean

# Calculate shortest distance between two rectangle

def rect\_distance(rect1, rect2):

    (x1, y1, x1b, y1b) = rect1

    (x2, y2, x2b, y2b) = rect2

    # Rect 2 is at the left of rect 1

    left = x2b < x1

    # Rect 2 is at the right of rect 1

    right = x1b < x2

    # Rect 2 is at the bottom of rect 1

    bottom = y2b < y1

    # Rect 2 is at the top of rect 1

    top = y1b < y2

    if top and left:

        return euclidean((x1, y1b), (x2b, y2))

    elif left and bottom:

        return euclidean((x1, y1), (x2b, y2b))

    elif bottom and right:

        return euclidean((x1b, y1), (x2, y2b))

    elif right and top:

        return euclidean((x1b, y1b), (x2, y2))

    elif left:

        return x1 - x2b

    elif right:

        return x2 - x1b

    elif bottom:

        return y1 - y2b

    elif top:

        return y2 - y1b

    else:

        # Rect 1 & 2 intersects

        return  0

def progress(frame\_count):

    import sys

    sys.stdout.write('\r')

    if frame\_count % 2 == 0:

        sys.stdout.write("Processing .. ")

    else:

        sys.stdout.write("Processing .  ")

    sys.stdout.flush()

def kinetic\_energy(point1, point2, time\_step):

    speed = euclidean(point1, point2) / time\_step

    return int(0.5 \* speed \*\* 2)

Util.py

import time

import datetime

import numpy as np

import imutils

import cv2

import time

from math import ceil

from scipy.spatial.distance import euclidean

from tracking import detect\_human

from util import rect\_distance, progress, kinetic\_energy

from colors import RGB\_COLORS

from config import SHOW\_DETECT, DATA\_RECORD, RE\_CHECK, RE\_START\_TIME, RE\_END\_TIME, SD\_CHECK, SHOW\_VIOLATION\_COUNT, SHOW\_TRACKING\_ID, SOCIAL\_DISTANCE,\

    SHOW\_PROCESSING\_OUTPUT, YOLO\_CONFIG, VIDEO\_CONFIG, DATA\_RECORD\_RATE, ABNORMAL\_CHECK, ABNORMAL\_ENERGY, ABNORMAL\_THRESH, ABNORMAL\_MIN\_PEOPLE

from deep\_sort import nn\_matching

from deep\_sort.detection import Detection

from deep\_sort.tracker import Tracker

from deep\_sort import generate\_detections as gdet

IS\_CAM = VIDEO\_CONFIG["IS\_CAM"]

HIGH\_CAM = VIDEO\_CONFIG["HIGH\_CAM"]

def \_record\_movement\_data(movement\_data\_writer, movement):

    track\_id = movement.track\_id

    entry\_time = movement.entry

    exit\_time = movement.exit

    positions = movement.positions

    positions = np.array(positions).flatten()

    positions = list(positions)

    data = [track\_id] + [entry\_time] + [exit\_time] + positions

    movement\_data\_writer.writerow(data)

def \_record\_crowd\_data(time, human\_count, violate\_count, restricted\_entry, abnormal\_activity, crowd\_data\_writer):

    data = [time, human\_count, violate\_count, int(restricted\_entry), int(abnormal\_activity)]

    crowd\_data\_writer.writerow(data)

def \_end\_video(tracker, frame\_count, movement\_data\_writer):

    for t in tracker.tracks:

        if t.is\_confirmed():

            t.exit = frame\_count

            \_record\_movement\_data(movement\_data\_writer, t)

def video\_process(cap, frame\_size, net, ln, encoder, tracker, movement\_data\_writer, crowd\_data\_writer):

    def \_calculate\_FPS():

        t1 = time.time() - t0

        VID\_FPS = frame\_count / t1

    if IS\_CAM:

        VID\_FPS = 1

        DATA\_RECORD\_FRAME = 1

        TIME\_STEP = 1

        t0 = time.time()

    else:

        VID\_FPS = cap.get(cv2.CAP\_PROP\_FPS)

        DATA\_RECORD\_FRAME = int(VID\_FPS / DATA\_RECORD\_RATE)

        TIME\_STEP = DATA\_RECORD\_FRAME/VID\_FPS

    frame\_count = 0

    display\_frame\_count = 0

    re\_warning\_timeout = 0

    sd\_warning\_timeout = 0

    ab\_warning\_timeout = 0

    RE = False

    ABNORMAL = False

    while True:

        (ret, frame) = cap.read()

        # Stop the loop when video ends

        if not ret:

            \_end\_video(tracker, frame\_count, movement\_data\_writer)

            if not VID\_FPS:

                \_calculate\_FPS()

            break

        # Update frame count

        if frame\_count > 1000000:

            if not VID\_FPS:

                \_calculate\_FPS()

            frame\_count = 0

            display\_frame\_count = 0

        frame\_count += 1

        # Skip frames according to given rate

        if frame\_count % DATA\_RECORD\_FRAME != 0:

            continue

        display\_frame\_count += 1

        # Resize Frame to given size

        frame = imutils.resize(frame, width=frame\_size)

        # Get current time

        current\_datetime = datetime.datetime.now()

        # Run detection algorithm

        if IS\_CAM:

            record\_time = current\_datetime

        else:

            record\_time = frame\_count

        # Run tracking algorithm

        [humans\_detected, expired] = detect\_human(net, ln, frame, encoder, tracker, record\_time)

        # Record movement data

        for movement in expired:

            \_record\_movement\_data(movement\_data\_writer, movement)

        # Check for restricted entry

        if RE\_CHECK:

            RE = False

            if (current\_datetime.time() > RE\_START\_TIME) and (current\_datetime.time() < RE\_END\_TIME) :

                if len(humans\_detected) > 0:

                    RE = True

        # Initiate video process loop

        if SHOW\_PROCESSING\_OUTPUT or SHOW\_DETECT or SD\_CHECK or RE\_CHECK or ABNORMAL\_CHECK:

            # Initialize set for violate so an individual will be recorded only once

            violate\_set = set()

            # Initialize list to record violation count for each individual detected

            violate\_count = np.zeros(len(humans\_detected))

            # Initialize list to record id of individual with abnormal energy level

            abnormal\_individual = []

            ABNORMAL = False

            for i, track in enumerate(humans\_detected):

                # Get object bounding box

                [x, y, w, h] = list(map(int, track.to\_tlbr().tolist()))

                # Get object centroid

                [cx, cy] = list(map(int, track.positions[-1]))

                # Get object id

                idx = track.track\_id

                # Check for social distance violation

                if SD\_CHECK:

                    if len(humans\_detected) >= 2:

                        # Check the distance between current loop object with the rest of the object in the list

                        for j, track\_2 in enumerate(humans\_detected[i+1:], start=i+1):

                            if HIGH\_CAM:

                                [cx\_2, cy\_2] = list(map(int, track\_2.positions[-1]))

                                distance = euclidean((cx, cy), (cx\_2, cy\_2))

                            else:

                                [x\_2, y\_2, w\_2, h\_2] = list(map(int, track\_2.to\_tlbr().tolist()))

                                distance = rect\_distance((x, y, w, h), (x\_2, y\_2, w\_2, h\_2))

                            if distance < SOCIAL\_DISTANCE:

                                # Distance between detection less than minimum social distance

                                violate\_set.add(i)

                                violate\_count[i] += 1

                                violate\_set.add(j)

                                violate\_count[j] += 1

                # Compute energy level for each detection

                if ABNORMAL\_CHECK:

                    ke = kinetic\_energy(track.positions[-1], track.positions[-2], TIME\_STEP)

                    if ke > ABNORMAL\_ENERGY:

                        abnormal\_individual.append(track.track\_id)

                # If restrited entry is on, draw red boxes around each detection

                if RE:

                    cv2.rectangle(frame, (x + 5 , y + 5 ), (w - 5, h - 5), RGB\_COLORS["red"], 5)

                # Draw yellow boxes for detection with social distance violation, green boxes for no violation

                # Place a number of violation count on top of the box

                if i in violate\_set:

                    cv2.rectangle(frame, (x, y), (w, h), RGB\_COLORS["yellow"], 2)

                    if SHOW\_VIOLATION\_COUNT:

                        cv2.putText(frame, str(int(violate\_count[i])), (x, y - 10), cv2.FONT\_HERSHEY\_SIMPLEX, 0.8, RGB\_COLORS["yellow"], 2)

                elif SHOW\_DETECT and not RE:

                    cv2.rectangle(frame, (x, y), (w, h), RGB\_COLORS["green"], 2)

                    if SHOW\_VIOLATION\_COUNT:

                        cv2.putText(frame, str(int(violate\_count[i])), (x, y - 10), cv2.FONT\_HERSHEY\_SIMPLEX, 0.8, RGB\_COLORS["green"], 2)

                if SHOW\_TRACKING\_ID:

                    cv2.putText(frame, str(int(idx)), (x, y - 10), cv2.FONT\_HERSHEY\_SIMPLEX, 0.8, RGB\_COLORS["green"], 2)

            # Check for overall abnormal level, trigger notification if exceeds threshold

            if len(humans\_detected)  > ABNORMAL\_MIN\_PEOPLE:

                if len(abnormal\_individual) / len(humans\_detected) > ABNORMAL\_THRESH:

                    ABNORMAL = True

        # Place violation count on frames

        if SD\_CHECK:

            # Warning stays on screen for 10 frames

            if (len(violate\_set) > 0):

                sd\_warning\_timeout = 10

            else:

                sd\_warning\_timeout -= 1

            # Display violation warning and count on screen

            if sd\_warning\_timeout > 0:

                text = "Violation count: {}".format(len(violate\_set))

                cv2.putText(frame, text, (200, frame.shape[0] - 30),

                    cv2.FONT\_HERSHEY\_SIMPLEX, 1, (0, 0, 255), 3)

        # Place restricted entry warning

        if RE\_CHECK:

            # Warning stays on screen for 10 frames

            if RE:

                re\_warning\_timeout = 10

            else:

                re\_warning\_timeout -= 1

            # Display restricted entry warning and count on screen

            if re\_warning\_timeout > 0:

                if display\_frame\_count % 3 != 0 :

                    cv2.putText(frame, "RESTRICTED ENTRY", (200, 100),

                        cv2.FONT\_HERSHEY\_SIMPLEX, 1, RGB\_COLORS["red"], 3)

        # Place abnormal activity warning

        if ABNORMAL\_CHECK:

            if ABNORMAL:

                # Warning stays on screen for 10 frames

                ab\_warning\_timeout = 10

                # Draw blue boxes over the the abnormally behave detection if abnormal activity detected

                for track in humans\_detected:

                    if track.track\_id in abnormal\_individual:

                        [x, y, w, h] = list(map(int, track.to\_tlbr().tolist()))

                        cv2.rectangle(frame, (x , y ), (w, h), RGB\_COLORS["blue"], 5)

            else:

                ab\_warning\_timeout -= 1

            if ab\_warning\_timeout > 0:

                if display\_frame\_count % 3 != 0:

                    cv2.putText(frame, "ABNORMAL ACTIVITY", (130, 250),

                        cv2.FONT\_HERSHEY\_SIMPLEX, 1.5, RGB\_COLORS["blue"], 5)

        # Display crowd count on screen

        if SHOW\_DETECT:

            text = "Crowd count: {}".format(len(humans\_detected))

            cv2.putText(frame, text, (10, 30),

                cv2.FONT\_HERSHEY\_SIMPLEX, 1, (255, 255, 255), 3)

        # Display current time on screen

        # current\_date = str(current\_datetime.strftime("%b-%d-%Y"))

        # current\_time = str(current\_datetime.strftime("%I:%M:%S %p"))

        # cv2.putText(frame, (current\_date), (500, 30), cv2.FONT\_HERSHEY\_SIMPLEX, 1, (0, 0, 0), 3)

        # cv2.putText(frame, (current\_time), (500, 60), cv2.FONT\_HERSHEY\_SIMPLEX, 1, (0, 0, 0), 3)

        # Record crowd data to file

        if DATA\_RECORD:

            \_record\_crowd\_data(record\_time, len(humans\_detected), len(violate\_set), RE, ABNORMAL, crowd\_data\_writer)

        # Display video output or processing indicator

        if SHOW\_PROCESSING\_OUTPUT:

            cv2.imshow("Processed Output", frame)

        else:

            progress(display\_frame\_count)

        # Press 'Q' to stop the video display

        if cv2.waitKey(1) & 0xFF == ord('q'):

            # Record the movement when video ends

            \_end\_video(tracker, frame\_count, movement\_data\_writer)

            # Compute the processing speed

            if not VID\_FPS:

                \_calculate\_FPS()

            break

    cv2.destroyAllWindows()

    return VID\_FPS

Video-process.py