```python
import time
import numpy as np
from collections import deque

from .linear_assignment import min_cost_matching, matching_cascade
from .kalman_filter import KalmanFilter
from .iou_matching import iou_cost


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


class Detection(object):
    """This class represents a bounding box, keypoints, score of person detected
    in a single image.

    Args:
        tlbr: (float array) Of shape [top, left, bottom, right].,
        keypoints: (float array) Of shape [node, pts].,
        confidence: (float) Confidence score of detection.
    """
    def __init__(self, tlbr, keypoints, confidence):
        self.tlbr = tlbr
        self.keypoints = keypoints
        self.confidence = confidence

    def to_tlwh(self):
        """Get (top, left, width, height).
        """
        ret = self.tlbr.copy()
        ret[2:] = ret[2:] - ret[:2]
        return ret

    def to_xyah(self):
        """Get (x_center, y_center, aspect ratio, height).
        """
        ret = self.to_tlwh()
        ret[:2] += (ret[2:] / 2)
        ret[2] /= ret[3]
        return ret


class Track:
    def __init__(self, mean, covariance, track_id, n_init, max_age=30, buffer=30):
        self.mean = mean
        self.covariance = covariance
        self.track_id = track_id
        self.hist = 1
        self.age = 1
        self.time_since_update = 0
        self.n_init = n_init
        self.max_age = max_age

        # keypoints list for use in Actions prediction.
        self.keypoints_list = deque(maxlen=buffer)

        self.state = TrackState.Tentative
```

```python
    def to_tlwh(self):
        ret = self.mean[:4].copy()
        ret[2] *= ret[3]
        ret[:2] -= ret[2:] / 2
        return ret

    def to_tlbr(self):
        ret = self.to_tlwh()
        ret[2:] = ret[:2] + ret[2:]
        return ret

    def get_center(self):
        return self.mean[:2].copy()    중심 반환

    def predict(self, kf):                                    칼만 필터
        """Propagate the state distribution to the current time step using a
        Kalman filter prediction step.
        """
        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.
        """
        self.mean, self.covariance = kf.update(self.mean, self.covariance,
                                               detection.to_xyah())
        self.keypoints_list.append(detection.keypoints)

        self.hist += 1
        self.time_since_update = 0
        if self.state == TrackState.Tentative and self.hist >= 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.Deleted

    def is_tentative(self):
        return self.state == TrackState.Tentative

    def is_confirmed(self):
        return self.state == TrackState.Confirmed

    def is_deleted(self):
        return self.state == TrackState.Deleted


class Tracker:
    def __init__(self, max_iou_distance=0.7, max_age=30, n_init=5):
        self.max_iou_dist = max_iou_distance
        self.max_age = max_age
        self.n_init = n_init

        self.kf = 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):
        """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 matched tracks set.
        for track_idx, detection_idx in matches:
            self.tracks[track_idx].update(self.kf, detections[detection_idx])
        # Update tracks that missing.
        for track_idx in unmatched_tracks:
            self.tracks[track_idx].mark_missed()
        # Create new detections track.
        for detection_idx in unmatched_detections:
            self._initiate_track(detections[detection_idx])

        # Remove deleted tracks.
        self.tracks = [t for t in self.tracks if not t.is_deleted()]

    def _match(self, detections):
        confirmed_tracks, unconfirmed_tracks = [], []
        for i, t in enumerate(self.tracks):
            if t.is_confirmed():
                confirmed_tracks.append(i)
            else:
                unconfirmed_tracks.append(i)

        matches_a, unmatched_tracks_a, unmatched_detections = matching_cascade(
            iou_cost, self.max_iou_dist, self.max_age, self.tracks, detections,
confirmed_tracks
        )

        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 = min_cost_matching(
            iou_cost, self.max_iou_dist, self.tracks, detections, 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):
        if detection.confidence < 0.4:
            return
        mean, covariance = self.kf.initiate(detection.to_xyah())
        self.tracks.append(Track(mean, covariance, self._next_id, self.n_init, self.max_age))
        self._next_id += 1
```