```
import glob
import os
import sys
import random
import time
import numpy as np
import cv2
import math
from collections import deque
from keras.applications.xception  import Xception
from keras.layers import Dense, GlobalAveragePooling2D
from keras.optimizers import Adam
from keras.models import Model

import tensorflow as tf
import keras.backend.tensorflow_backend as backend
from threading import Threading

from tqdm import tqdm

try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass

import carla

SHOW_PREVIEW = False
IM_WIDTH = 640
IM_HEIGHT = 480
SECONDS_PER_EPISODE = 10
REPLAY_MEMORY_SIZE =  5000  #5_000
MIN_REPLAY_MEMORY_SIZE = 1000  #1_000
MINIBATCH_SIZE = 16
PREDICTION_BATCH_SIZE = 1
TRAINING_BATCH_SIZE = MINIBATCH_SIZE // 4
UPDATE_TARGET_EVERY = 5
MODEL_NAME = "Xception"

MEMORY_FRACTION = 0.6
MIN_REWARD = -200

EPISODES = 100

DISCOUNT = 0.99
epsilon = 1
EPSILON_DECAY = 0.95 #0.9975 0.99975
```

```
MIN_EPSILON = 0.001

AGGREGATE_STATS_EVERY = 10


class CarEnv:
    SHOW_CAM = SHOW_PREVIEW
    STEER_AMT = 1.0
    im_width = IM_WIDTH
    im_height = IM_HEIGHT
    front_camera = None

    def __init__(self):
        self.client = carla.Client("localhost", 2000)
        self.client.set_timeout(2.0)
        self.world = self.client.get_world()
        self.blueprint_library = self.world.get_blueprint_library()
        self.model_3 = blueprint_library.filter("model3")[0]

    def rest(self):
        self.collision_hist = []
        self.actor_list = []

        self.transform = random.choice(self.world.get_map().get_spawn_points())
        self.vehicle = self.world.spawn_actor(self.model_3,self_transform)
        self.actor_list.append(self.vehicle)

        self.rgb_cam = self.blueprint_library.find('sensor.camera.rgb')
        self.rgb.set_attribute("image_size_x", "{}".format(self.im_width))
        self.rgb.set_attribute("image_size_y", "{}".format(self.im_height))
        self.rgb.set_attribute("fov", "110")

        transform = carla.Transform(carla.Location(x=2.5, z=0.7))
        self.sensor = self.world.spawn_actor(self.rgb.cam, transform,
attach_to=self.vehicle)
        self.actor_list.append(self.sensor)
        self.sensor.llisten(lambda data: self.process_img(data))

        self.vehicle.apply_control(carla.VehicleControl(throttle=0.0, brake=0.0))
        time.sleep(4)

        colsensor = self.blueprint_library.find("sensor.other.collision")
        self.colsensor = self.world.spawn_actor(colsensor, transform,
attach_to=self.vehicle)
        selff.actor_list.append(self.colsensor)
        self.colsensor.listen(lambda event: self.collision_data(event))

        while self.front_camera is None:
            time.sleep(0.01)
```

```
        self.episode_start = time.time()
        self.vehicle.apply_control(carla.VehicleControl(throttle=0.0, brake=0.0))

        return self.front_camera

    def collision_data(self, event):
        self.collision_hist.append(event)

    def process_img(image):
        i = np.array(image.raw_data)
        # print(i.shape)
        i2 = i.reshape((self.im_height, self.im_width, 4))
        i3 = i2[:, :, :3]
        if self.SHOW_CAM:
            cv2.imshow("", i3)
            cv2.waitKey(1)
        self.front_camera = i3

    def step(self, action):
        #LEFT
        if action == 0:
            self.vehicle.apply_control(carla.VehicleControl(throttle=1.0,
steer=-1*self.STEER_AMT))
        #STRAIGHT
        elif action == 1:
            self.vehicle.apply_control(carla.VehicleControl(throttle=1.0, steer=0))
        #RIGHT
        elif action == 2:
            self.vehicle.apply_control(carla.VehicleControl(throttle=1.0, steer=1 *
self.STEER_AMT))

        v = self.vehicle.get_velocity()
        kmh = int(3.6 * math.sqrt(v.x**2 + v.y**2 + v.z**2))

        if len(self.collision_hist) !=0:
            done = True
            reward = -200
        elif kmh <50:
            done = False
            reward = -1
        else:
            done = False
            reward = 1

        if self.episode_start + SECONDS_PER_EPISODE < time.time():
            done = True

        return self.front_camera, reward, done, None

class DQNAgent:
```

```
    def __init__(self):
        self.model = self.create_model()
        self.target_model = self.create_model()
        self.target_model.set_weights(self.model.get_weights())

        self.replay_memory = deque(maxlen=REPLAY_MEMORY_SIZE)

        self.tensorboard = ModifiedTensorBoard(log_dir=
'logs/{}-{}'.format(MODEL_NAME, int(time.time())))
        self.target_update_counter = 0
        self.graph = tf.get_default_graph()

        self.terminate = False
        self.last_logged_episode = 0
        self.training_initialized = False

    def create_model(self):
        base_model = Xception(weights=None, include_top=False,
input_shape=(IM_HEIGHT, IM_WIDTH,3))

        x = base_model.output
        x = GlobalAveragePooling2D()(x)

        predictions = Dense(3, activation="linear")(x)
        model = Model(inputs=base_model.input, outputs=predictions)
        model.compile(loss="mse", optimizer=Adam(lr=0.001), metrics=["accuracy"])
 #mse - mean squared error
        return model

    def update_replay_memory(self, transition):
        # transition = (current_state, action, reward, new_state, done)
        self.replay_memory.append(transition)

    def train(self):
        if len(self.replay_memory) < MIN_REPLAY_MEMORY_SIZE:
            return

        minibatch = random.sample(self.replay_memory, MINIBATCH_SIZE)

        current_states = np.array([transition[0] for transition in minibatch])/255
        with self.graph.as_default():
            current_qs_list = self.model.predict(current_states,
PREDICTION_BATCH_SIZE)

        new_current_states = np.array([transition[3] for transition in minibatch]) /
255
        with self.graph.as_default():
            future_qs_list = self.target_model.predict(new_current_states,
PREDICTION_BATCH_SIZE)
```

```
        X = []
        y = []

        for index, (current_state, action, reward, new_state, done) in
enumerate(minibatch):
            if not done:
                max_future_q = np.max(future_qs_list[index])
                new_q = reward + DISCOUNT * max_future_q
            else:
                new_q = reward

            current_qs = current_qs_list[index]
            current_qs[action] = new_q

            X.append(current_state)
            y.append(current_qs)

        log_this_step = False
        if self.tensorboard.step > self.last_logged_episode:
            log_this_step = True
            self.last_log_episode = self.tensorboard.step

        with self.graph.as_default():
            self.model.fit(np.array(X)/255, np.array(y),
batch_size=TRAINING_BATCH_SIZE, verbose=0, shuffle=False,
callbacks=[self.tensorboard] if log_this_step else None)

        if log_this_step:
            self.target_update_counter += 1

        if self.target_update_counter > UPDATE_TARGET_EVERY:
            self.target_model.set_weights(self.model.get_weights())
            self.target_update_counter = 0

    def get_qs(self, state):
        return self.model.predict(np.array(state).reshape(-1, *state.shape)/255)[0]

    def train_in_loop(self):
        X = np.random.uniform(size=(1, IM_HEIGHT, IM_WIDTH, 3)).astype(np.float32)
        y = np.random.uniform(size=(1, 3)).astype(np.float32)
        with self.graph.as_default():
            self.model.fit(X, y, verbose=False, batch_size=1)

        self.training_initialized = True

        while True:
            if self.terminate:
                return
            self.train()
            time.sleep(0.01)
```

```
if __name__ == "__main__":
    FPS = 60
    #For stats
    ep_rewards = [-200]

    #For more repetitive results
    random.seed(1)
    np.random.seed(1)
    tf.set_random_seed(1)

    #Memory fraction, used mostly when training multiple agents
    gpu_options = tf.GPUOptions(per_process_gpu_memory_fraction=MEMORY_FRACTION)
    backend.set_session(tf.Session(config=tf.ConfigProto(gpu_options=gpu_options)))

    #Create models folder
    if not os.path.isdir('models'):
        os.makedirs('models')

    #Create agent and environment
    agent = DQNAgent()
    env = CarEnv()

    #Start training thread and wait for training to be initialized
    trainer_thread = Thread(target=agent.train_in_loop, daemon=True)
    trainer_thread.start()

    while not agent.training_initialized:
        time.sleep(0.01)

    # Initialize predictions - forst prediction takes longer as of initialization
that has to be done
    #It's better to do a first prediction then before we start iterating over
episode steps
    agent.get_qs(np.ones((env.im_height, env.im_width,3)))

    #Iterate over episodes
    for episode in tqdm(range(1, EPISODES+1), unit="episodes"):
        env.collision_hist = []
        agent.tensorboard.step = episode   #Update tensorboard step every episode
        episode_reward = 0   #Restarting episode - reset episode reward and step
number
        step = 1
        current_state = env.reset()  #Reset environment and get initial state
        done = False            #Reset flag and start iterating until episode ends
        episode_start = time.time()

        while True:
            if np.random.random() > epsilon:
                action = np.argmax(agent.get_qs(current_state))
```

```
            else:
                action = np.random.randint(0,3)
                time.sleep(1/FPS)

            new_state, reward, done, _ = env.step(action)
            episode_reward += reward
            agent.update_replay_memory((current_state, action, reward, new_state,
done))
            step += 1

            if done:
                break

        for actor in env.actor_list:
            actor.destroy()

        # Append episode reward to a list and log stats (every given number of
episodes)
        ep_rewards.append(episode_reward)
        if not episode % AGGREGATE_STATS_EVERY or episode == 1:
            average_reward = sum(ep_rewards[-AGGREGATE_STATS_EVERY:]) /
len(ep_rewards[-AGGREGATE_STATS_EVERY:])
            min_reward = min(ep_rewards[-AGGREGATE_STATS_EVERY:])
            max_reward = max(ep_rewards[-AGGREGATE_STATS_EVERY:])
            agent.tensorboard.update_stats(reward_avg=average_reward,
reward_min=min_reward, reward_max=max_reward,
                                           epsilon=epsilon)
            # Save model, but only when min reward is greater or equal a set value
            if min_reward >= MIN_REWARD:

#agent.model.save('{}'.format(models/(MODEL_NAME)__(max_reward:_>7.2f)max_(average_r
eward:_>7.2f)avg_(min_reward:_>7.2f)min__(int(time.time())).model))

agent.model.save(f'models/{MODEL_NAME}__{max_reward:_>7.2f}max_{average_reward:_>7.2
f}avg_{min_reward:_>7.2f}min__{int(time.time())}.model')


        # Decay epsilon
        if epsilon > MIN_EPSILON:
            epsilon *= EPSILON_DECAY
            epsilon = max(MIN_EPSILON, epsilon)


    # Set termination flag for training thread and wait for it to finish
    agent.terminate = True
    trainer_thread.join()

agent.model.save(f'models/{MODEL_NAME}__{max_reward:_>7.2f}max_{average_reward:_>7.2
f}avg_{min_reward:_>7.2f}min__{int(time.time())}.model')
```