```
import time
from collections import deque

import numpy as np
import tensorflow as tf
import tensorflow.keras as keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import InputLayer, Dense

class DQN:
    '''
    Baseline Deep Q-Network with experience replay
    '''

    def __init__(self, state_size, action_size, policy, learning_delay, loss_fn, epsilon,
 gamma,
        learning_rate, n_units, buffer_size, l2=0, learning_freq=1, verbose=False, **kwar
gs):
        '''
        Initialize necessary fields
        '''
        self.type = "DQN"

        self.state_size = state_size
        self.action_size = action_size
        self.policy = policy
        self.learning_delay = learning_delay
        self.learning_freq = learning_freq
        self.loss_fn = loss_fn
        self.epsilon = epsilon
        self.gamma = gamma
        self.optimizer = keras.optimizers.Adam(lr=learning_rate)
        self.setup_model(n_units, l2=l2)

        self.epsilon_log = []
        self.reward_log = []
        self.loss_log = []
        self.deque_log = []
        self.verbose = verbose
        self.replay_buffer = {
                    "states": deque(maxlen=buffer_size),
                    "actions": deque(maxlen=buffer_size),
                    "rewards": deque(maxlen=buffer_size),
                    "next_states": deque(maxlen=buffer_size),
                    "dones": deque(maxlen=buffer_size)
        }
        self.episode = 0

    def build_model(self, n_units, activation="elu", l2=0):
        '''
        Build a simple sequential model.
        '''
        print("L2: {}".format(l2))

        model = Sequential()
        i = 0

        # Input layer
        model.add(InputLayer(input_shape=(self.state_size,)))

        # Loop over hidden layers
        for n in n_units:
            model.add(Dense(n,
                        activation=activation,
                        kernel_regularizer=keras.regularizers.l2(l2),
                        name = "D"+str(i)))
            i=i+1

        # model.add(BatchNormalization())
```

```
        # Output layer
        model.add(Dense(self.action_size,
                        activation=None,
                        name = "D"+str(i)))

        return model

    def setup_model(self, n_units, l2=0):
        '''
        Compile a simple sequential model
        '''

        model = self.build_model(n_units=n_units, l2=l2)
        self.model = model
        model.summary()

    def get_epsilon(self):
        try:
            return self.epsilon(self.episode)
        except TypeError as e:
            return self.epsilon

    def play_one_step(self, env, state):
        '''
        Take one step in the environment based on the agent parameters
        '''

        action = self.policy(state, self.model, self.get_epsilon()) # Query policy

        next_state, reward, done, info = env.step(action) # Query environment
        self.memorize(state, action, reward, next_state, done) # Log

        return next_state, reward, done, info

    def memorize(self, state, action, reward, next_state, done):
        '''
        Log the experience from one step into the replay buffer as a dictionary
        '''

        state =  np.array(state, ndmin=2)
        next_state =  np.array(next_state, ndmin=2)

        self.replay_buffer["states"].append(state)
        self.replay_buffer["actions"].append(action)
        self.replay_buffer["rewards"].append(reward)
        self.replay_buffer["next_states"].append(next_state)
        self.replay_buffer["dones"].append(done)

    def sample_experience_inds(self, batch_size):
        '''
        Sample batch_size number of experience indices from the replay buffer
        '''

        # If batch size greater than current length of buffer, give all indices for buffe
r.
        # Otherwise, get random sampling of batch_size indices.
        choice_range = len(self.replay_buffer["states"])
        if batch_size is None or batch_size > choice_range:
            indices = np.random.choice(choice_range, size=choice_range, replace=False)
        else:
            indices = np.random.choice(choice_range, size=batch_size, replace=False)

        return indices

    def sample_experience_inds_old(self, batch_size):
        '''
        Sample batch_size number of experience indices from the replay buffer
        '''

        # If batch size greater than current length of buffer, give all indices for buffe
```

```
r.
        # Otherwise, get random sampling of batch_size indices.
        if batch_size > len(self.replay_buffer["states"]):
            indices = list(range(len(self.replay_buffer["states"])))
        else:
            indices = np.random.randint(len(self.replay_buffer["states"]), size=batch_siz
e)

        return indices

    def sample_experience(self, inds):
        '''
        Sample experiences with indices from replay buffer
        '''

        batch = {}
        for key in self.replay_buffer.keys():
            batch[key] = [self.replay_buffer[key][index] for index in inds]


        batch["states"] = np.concatenate(batch["states"], axis=0)
        batch["next_states"] = np.concatenate(batch["next_states"], axis=0)

        return batch

    def get_current_Q_values(self, states):
        return self.model(states)

    def get_next_Q_values(self, next_states):
        return self.model.predict(next_states)

    def learning_step(self, batch_size=100):
        '''
        Train the model with one batch by sampling from replay buffer
        Use the gradient tape method
        '''

        batch_time_start = time.time()
        # Fetch batch
        batch_inds = self.sample_experience_inds(batch_size)
        batch = self.sample_experience(batch_inds)

        # Create target q values, with mask to disregard irrelevant actions
        next_Q_values = self.get_next_Q_values(batch["next_states"]) # Get subsequent Q v
alues
        max_next_Q_values = np.max(next_Q_values, axis=1) # Get max of subsequent Q value
s
        target_Q_values = (batch["rewards"] + (1 - np.asarray(batch["dones"])) * self.gam
ma * max_next_Q_values) # Define Q targets
        mask = tf.one_hot(batch["actions"], self.action_size) # Create mask to mask actio
ns not taken

        # Use optimizer to apply gradient to model
        with tf.GradientTape() as tape:
            all_Q_values = self.get_current_Q_values(batch["states"]) # Get all possible
q values from the states
            masked_Q_values = all_Q_values * mask # Mask the actions which were not taken
            Q_values = tf.reduce_sum(masked_Q_values, axis=1) # Get the sum to reduce to
action taken
            loss = tf.reduce_mean(self.loss_fn(target_Q_values, Q_values)) # Compute the
losses
            self.loss_log.append(loss) # Append to log
            grads = tape.gradient(loss, self.model.trainable_variables) # Compute the gra
dients
            self.optimizer.apply_gradients(zip(grads, self.model.trainable_variables)) #
Apply the gradients to the model

    def execute_episode(self, env, n_steps=None, render_flag=False, batch_size=100, verbo
se=False,
        train=True):
```

```
        '''
        Execute one episode, which terminates when done if flagged or step limit is reach
ed
        '''

        # Initialize vars
        reward_total = 0
        step = 0
        done = False
        state = env.reset()
        while (n_steps is None or step < n_steps) and not done: # Continue till step coun
t, or until done

            if render_flag: # Create visualization for environment
                env.render()

            state, reward, done, info = self.play_one_step(env, state) # Custom step func
tion
            reward_total += reward
            step += 1
            if done:
                break

        # If train flag and episode above some threshold (to fill buffer), train
        if train and self.episode >= self.learning_delay and self.episode % self.learning
_freq == 0:
            print("\tLearning")
            self.learning_step(batch_size=batch_size)
        else:
            print("\tCollecting")

        self.reward_log.append(reward_total)
        self.epsilon_log.append(self.get_epsilon())
        self.deque_log.append(len(self.replay_buffer["states"]))
        self.episode += 1

        if verbose:
            print("\tReward: {}".format(reward_total))

    def execute_episodes(self, env, n_episodes, n_steps, render_flag=False, batch_size=10
0, verbose=False,
        train=True):
        '''
        Execute multiple episodes
        '''

        for episode in range(n_episodes):
            if verbose:
                print("Episode: {}".format(self.episode))

            self.execute_episode(
                env=env,
                n_steps=n_steps,
                render_flag=render_flag,
                batch_size=batch_size,
                verbose=verbose,
                train=train)

            if render_flag:
                env.close()
```