```
import numpy as np
import tensorflow as tf
import gym
import matplotlib.pyplot as plt
from datetime import datetime
from buffer import BasicBuffer_a,BasicBuffer_b
from sys import exit



# simple NN Generator
def ANN2(input_shape,layer_sizes, hidden_activation='relu', output_activation=None):
    model = tf.keras.Sequential()
    model.add(tf.keras.layers.Input(shape=input_shape))
    for h in layer_sizes[:-1]:
        x = model.add(tf.keras.layers.Dense(units=h, activation='relu'))
    model.add(tf.keras.layers.Dense(units=layer_sizes[-1], activation=output_activation))
    return model




### Create both the actor and critic networks at once ###
### Q(s, mu(s)) returns the maximum Q for a given state s ###
def ddpg(
    env_fn,
    ac_kwargs=dict(),
    seed=0,
    save_folder=None,
    num_train_episodes=100,
    test_agent_every=25,
    replay_size=int(1e6),
    gamma=0.99,
    decay=0.99,
    mu_lr=1e-3,
    q_lr=1e-3,
    batch_size=32,
    start_steps=1000,
    action_noise=0.0,
    max_episode_length=500):

  tf.random.set_seed(seed)
  np.random.seed(seed)

  env, test_env = env_fn(), env_fn()

  # comment out this line if you don't want to record a video of the agent
  # if save_folder is not None:
  # test_env = gym.wrappers.Monitor(test_env)

  # get size of state space and action space
  num_states = env.observation_space.shape[0]
  num_actions = env.action_space.shape[0]
  action_max = env.action_space.high[0]


  # Network parameters
  X_shape = (num_states)
  QA_shape = (num_states + num_actions)
  hidden_sizes_1=(1000,500,200)
  hidden_sizes_2=(400,200)


  # Main network outputs
  mu = ANN2(X_shape,list(hidden_sizes_1)+[num_actions], hidden_activation='relu',
output_activation='tanh')
```

States and actions both → see later in actor training

list of layer sizes in sequence

Actor network

Note: this stops unbounded outputs

critic network

```
  q_mu = ANN2(QA_shape, list(hidden_sizes_2)+[1], hidden_activation='relu')

  # Target networks
  mu_target = ANN2(X_shape,list(hidden_sizes_1)+[num_actions], hidden_activation='relu',
output_activation='tanh')
  q_mu_target = ANN2(QA_shape, list(hidden_sizes_2)+[1], hidden_activation='relu')

  # Copying weights in,
  mu_target.set_weights(mu.get_weights())
  q_mu_target.set_weights(q_mu.get_weights())
```

as usual, start with identical parameters for online (main) and target networks

```
  # Experience replay memory
  replay_buffer = BasicBuffer_b(size=replay_size,obs_dim=num_states, act_dim=num_actions)


  # Train each network separately
  mu_optimizer =tf.keras.optimizers.Adam(learning_rate=mu_lr)
  q_optimizer = tf.keras.optimizers.Adam(learning_rate=q_lr)

  def get_action(s, noise_scale):
    a = action_max * mu.predict(s.reshape(1,-1))[0]
    a += noise_scale * np.random.randn(num_actions)
    return np.clip(a, -action_max, action_max)
```

since output of mu is tanh, we scale it up to the "true" action

Add noise N_t from paper

```
  test_returns = []
  def test_agent(num_episodes=5):
    t0 = datetime.now()
    n_steps = 0
    for j in range(num_episodes):
      s, episode_return, episode_length, d = test_env.reset(), 0, 0, False
      while not (d or (episode_length == max_episode_length)):
        # Take deterministic actions at test time (noise_scale=0)
        test_env.render()
        s, r, d, _ = test_env.step(get_action(s, 0))
        episode_return += r
        episode_length += 1
        n_steps += 1
      print('test return:', episode_return, 'episode_length:', episode_length)
      test_returns.append(episode_return)


  # Main loop: play episode and train
  returns = []
  q_losses = []
  mu_losses = []
  num_steps = 0
  for i_episode in range(num_train_episodes):

    # reset env
    s, episode_return, episode_length, d = env.reset(), 0, 0, False

    while not (d or (episode_length == max_episode_length)):
      # For the first `start_steps` steps, use randomly sampled actions
      # in order to encourage exploration.

      if num_steps > start_steps:
        a = get_action(s, action_noise)
      else:
        a = env.action_space.sample()

      # Keep track of the number of steps done
      num_steps += 1
      if num_steps == start_steps:
        print("USING AGENT ACTIONS NOW")
```

initial random exploration to populate buffer (there is no ε-greedy here, remember)

```
    # Step the env
    s2, r, d, _ = env.step(a)
    episode_return += r
    episode_length += 1

    # Ignore the "done" signal if it comes from hitting the time
    # horizon (that is, when it's an artificial terminal signal
    # that isn't based on the agent's state)
    d_store = False if episode_length == max_episode_length else d

    # Store experience to replay buffer
    replay_buffer.push(s, a, r, s2, d_store)

    # Assign next state to be the current state on the next round
    s = s2

  # Perform the updates
  for _ in range(episode_length):

    X,A,R,X2,D = replay_buffer.sample(batch_size)
    X = np.asarray(X,dtype=np.float32)
    A = np.asarray(A,dtype=np.float32)
    R = np.asarray(R,dtype=np.float32)
    X2 = np.asarray(X2,dtype=np.float32)
    D = np.asarray(D,dtype=np.float32)
    Xten=tf.convert_to_tensor(X)

    #Actor optimization
    with tf.GradientTape() as tape2:
      Aprime = action_max * mu(X)
      temp = tf.keras.layers.concatenate([Xten,Aprime],axis=1)
      Q = q_mu(temp)
      mu_loss =  -tf.reduce_mean(Q)
      grads_mu = tape2.gradient(mu_loss,mu.trainable_variables)
    mu_losses.append(mu_loss)
    mu_optimizer.apply_gradients(zip(grads_mu, mu.trainable_variables))
```

states + actions

∇Q $\nabla \mu$

Remember this is $\nabla J \approx \frac{1}{N} \Sigma \nabla Q . \nabla \mu$

```
    #Critic Optimization
    with tf.GradientTape() as tape:
      next_a = action_max * mu_target(X2)
      temp = np.concatenate((X2,next_a),axis=1)
      q_target = R + gamma * (1 - D) * q_mu_target(temp)
      temp2 = np.concatenate((X,A),axis=1)
      qvals = q_mu(temp2)
      q_loss = tf.reduce_mean((qvals - q_target)**2)
      grads_q = tape.gradient(q_loss,q_mu.trainable_variables)
    q_optimizer.apply_gradients(zip(grads_q, q_mu.trainable_variables))
    q_losses.append(q_loss)
```

critic training

```
    ## Updating both netwokrs
    ## updating Critic network

    temp1 = np.array(q_mu_target.get_weights())
    temp2 = np.array(q_mu.get_weights())
    temp3 = decay*temp1 + (1-decay)*temp2
    q_mu_target.set_weights(temp3)
```

→ soft update

```
   # updating Actor network
    temp1 = np.array(mu_target.get_weights())
    temp2 = np.array(mu.get_weights())
    temp3 = decay*temp1 + (1-decay)*temp2
    mu_target.set_weights(temp3)
```

```
    print("Episode:", i_episode + 1, "Return:", episode_return, 'episode_length:',
episode_length)
    returns.append(episode_return)

    # Test the agent
    # if i_episode > 0 and i_episode % test_agent_every == 0:
    #   test_agent()
  return (returns,q_losses,mu_losses)
```