```
import argparse
import os
import random
import time

import numpy as np
import tensorflow as tf
from tensorflow import keras

from AoI_Trade import AoI_Trade
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '1'
parser = argparse.ArgumentParser(description='Hyper_params')

args = parser.parse_args()

os.environ['TF_DETERMINISTIC_OPS'] = 'True'
os.environ["TF_DISABLE_SEGMENT_REDUCTION_OP_DETERMINISM_EXCEPTIONS"] = 'True'
os.environ["CUDA_VISIBLE_DEVICES"] = args.Gpu_Id
gpus = tf.config.experimental.list_physical_devices(device_type='GPU')
for gpu in gpus:
    tf.config.experimental.set_memory_growth(gpu, True)
tf.random.set_seed(args.Seed)
np.random.seed(args.Seed)

# create log file
time_str = time.strftime("%m-%d_%H-%M", time.localtime())
alg = args.Alg
    log_dir_name = time_str + '_alg_' + args.Info + '_n_' + str(args.User) '
fw = tf.summary.create_file_writer(log_dir_name)  # log file writer

# create dir to save model
if not os.path.exists(log_dir_name + '/models'):
    os.makedirs(log_dir_name + '/models')

# save params to a .txt file
prams_file = open(log_dir_name + '/prams_table.txt', 'w')
prams_file.writelines(f'{i:50} {v}\n' for i, v in args.__dict__.items())
prams_file.close()

#  ###################### env ###############################
env = AoI_Trade(user_num=args.User,  seed=args.Seed, request_p=args.Request_P,
                    # ATTENTION
                    slot_Dd=args.slot_Dd)
```

```
Action_Num = env.Actual_Action_Num
print("Action_Num:", Action_Num)
Initial_R = - env.C1
print("Inintial_R:", Initial_R)

#   ###################### others ################################
Optimizer = tf.optimizers.Adam(args.Lr, decay=args.Lr_Decay)
W_Initializer = tf.initializers.he_normal(args.Seed)   # NN initializer
Epsilon_Decay_Rate = (args.Min_Epsilon - args.Max_Epsilon) / args.Memory_Size * args.Epsilon_Decay   # factor of decay
TENSOR_FLOAT_TYPE = tf.dtypes.float32
TENSOR_INT_TYPE = tf.dtypes.int32


# YY 原本的经验池
class OldReplayBuffer:
    def __init__(self, size):
        self.cap = size
        buffer_s_dim = (size, env.N + 1, env.K)

        self.s_buffer = np.empty(buffer_s_dim, dtype=np.float32)
        self.a_buffer = np.random.randint(0, Action_Num, (self.cap, 1), dtype=np.int32)
        self.r_buffer = np.empty((self.cap, 1), dtype=np.float32)
        self.next_s_buffer = np.empty(buffer_s_dim, dtype=np.float32)

        self.cap_index = 0
        self.size = 0

    def store(self, step):
        s, a, r, next_s = step
        self.s_buffer[self.cap_index] = s
        self.a_buffer[self.cap_index][0] = a
        self.r_buffer[self.cap_index][0] = r
        self.next_s_buffer[self.cap_index] = next_s

        self.cap_index = (self.cap_index + 1) % self.cap
        self.size = min(self.size + 1, self.cap)

    def sample(self, batch_size):
        idx = np.random.randint(0, self.size, batch_size)

        batch_s = self.s_buffer[idx]
        batch_a = self.a_buffer[idx]
        batch_r = self.r_buffer[idx]
```

v. batch_next_s = self.next_s_buffer[idx]

vi. return batch_s, batch_a, batch_r, batch_next_s

d) def size(self):
   i. return self.size

43. # YY SumTree 的实现
44. class SumTree:
    a) data_pointer = 0

    b) def __init__(self, capacity):
       i. self.capacity = capacity
       ii. self.tree = np.zeros(2 * capacity - 1, dtype=float)
       iii. self.data = np.zeros(capacity, dtype=object)

    c) # 将一条经验的优先级 p 和数据 data 存储到 SumTree 中
    d) def add(self, p, data):
       i. tree_idx = self.data_pointer + self.capacity - 1
       ii. self.data[self.data_pointer] = data
       iii. self.update(tree_idx, p)
       iv. self.data_pointer += 1
       v. if self.data_pointer >= self.capacity:
          1. self.data_pointer = 0

    e) # 更新指定节点 tree_idx 处的优先级值
    f) def update(self, tree_idx, p):
       i. # print("tree_idx:",tree_idx)
       ii. change = p - self.tree[tree_idx]
       iii. # print("change:",change)
       iv. # p_scalar = p.item()
       v. self.tree[tree_idx] = p
       vi. self._propagate(tree_idx, change)

    g) # 递归地向上更新树的父节点的优先级。
    h) def _propagate(self, tree_idx, change):
       i. parent = (tree_idx - 1) // 2
       ii. self.tree[parent] += change
       iii. if parent != 0:
          1. self._propagate(parent, change)

    i) # 获取叶子节点的索引、优先级和对应的数据
    j) def get_leaf(self, v):
       i. parent_idx = 0

ii. while True:
    1. left_child_idx = 2 * parent_idx + 1
    2. right_child_idx = left_child_idx + 1
    3. if left_child_idx >= len(self.tree):
        a) leaf_idx = parent_idx
        b) break
    4. else:
        a) if v <= self.tree[left_child_idx]:
            i. parent_idx = left_child_idx
        b) else:
            i. v -= self.tree[left_child_idx]
            ii. parent_idx = right_child_idx
iii. data_idx = leaf_idx - self.capacity + 1
iv. return leaf_idx, self.tree[leaf_idx], self.data[data_idx] # 返回叶子节点的索引、优先级和对应的数据

k) # 返回整个 SumTree 的总优先级和
l) def total(self):
    i. return self.tree[0]

45. # YY 优先级经验池
46. class PrioritizedReplayBuffer:
    a) def __init__(self, capacity, alpha=0.5, beta=0.5, beta_increment_per_sampling=0.001, abs_err_upper=1.):
        i. self.capacity = capacity
        ii. self.alpha = alpha
        iii. self.beta = beta
        iv. self.beta_increment_per_sampling = beta_increment_per_sampling
        v. self.epsilon = 0.01
        vi. self.abs_err_upper = abs_err_upper # 添加 abs_err_upper，限制误差的上限
        vii. self.tree = SumTree(capacity)
        viii. # self.current_size = 0 # 记录当前存储的经验数量

    b) def store(self, transition):
        i. # 检查经验池是否已满，如果已满，则替换最旧的经验
        ii. # if self.current_size < self.capacity:
        iii. # max_p = np.max(self.tree.tree[-self.capacity:])
        iv. # else:
        v. # max_p = np.max(self.tree.tree[-self.capacity: -self.capacity + self.current_size])
        vi. # if self.current_size < self.capacity:

vii. # max_p = np.max(self.tree.tree[-self.capacity:])
viii. # else:
ix. # if self.current_size > 0:
x. # max_p = np.max(self.tree.tree[-self.capacity: -self.capacity + self.current_size])
xi. # else:
xii. # max_p = 1.0 # 或者适当的默认值

xiii. max_p = np.max(self.tree.tree[-self.tree.capacity:])
xiv. if max_p == 0:
    1. max_p = self.abs_err_upper
xv. self.tree.add(max_p, transition)

xvi. # 更新当前存储的经验数量
xvii. # self.current_size = min(self.current_size + 1, self.capacity)

c) def sample(self, batch_size):
    i. b_idx = []
    ii. b_memory = []
    iii. ISWeights = []

    iv. total_priority = self.tree.total()
    v. pri_seg = total_priority / batch_size

    vi. self.beta = np.min([1.0, self.beta + self.beta_increment_per_sampling])

    vii. min_prob = np.min(self.tree.tree[-self.tree.capacity:]) / total_priority
    viii. if min_prob == 0:
        1. min_prob = 0.00001
    ix. for i in range(batch_size):
        1. a, b = pri_seg * i, pri_seg * (i + 1)
        2. sample_value = np.random.uniform(a, b)
        3. idx, p, data = self.tree.get_leaf(sample_value)
        4. prob = p / total_priority
        5. ISWeight = np.power(prob / min_prob, -self.beta)
        6. b_idx.append(idx)
        7. b_memory.append(data)
        8. ISWeights.append(ISWeight)

    x. return b_idx, b_memory, ISWeights

d) # 批量更新存储在经验池中的经验的优先级
e) def batch_update(self, tree_idx, abs_errors):
    i. abs_errors += self.epsilon

ii. # print("abs_errors:",abs_errors)
iii. clipped_errors = np.minimum(abs_errors, self.abs_err_upper)
iv. # print("clipped_errors:",clipped_errors)
v. ps = np.power(clipped_errors, self.alpha)
vi. # print("ps:",ps)
vii. for ti, p in zip(tree_idx, ps):
1. # print("tree_idx:",tree_idx)
2. # print("ti:",ti)
3. # print("p:",p)
4. # p_scalar = p.item()
5. self.tree.update(ti, p)

47. # YY 自定义 noisy 层
48. class NoisyLayer(tf.keras.layers.Layer):
a) def __init__(self, units, activation='linear', sigma_init=0.5, **kwargs):
i. super(NoisyLayer, self).__init__(**kwargs)
ii. self.units = units
iii. self.activation = tf.keras.activations.get(activation)
iv. self.sigma_init = sigma_init

b) def build(self, input_shape):
i. self.mu_weight = self.add_weight(
1. shape=(input_shape[-1], self.units),
2. initializer='random_normal',
3. trainable=True,
4. name='mu_weight'
ii.)

iii. self.sigma_weight = self.add_weight(
1. shape=(input_shape[-1], self.units),
2. initializer=tf.keras.initializers.Constant(self.sigma_init),
3. trainable=True,
4. name='sigma_weight'
iv.)

v. self.mu_bias = self.add_weight(
1. shape=(self.units,),
2. initializer='random_normal',
3. trainable=True,
4. name='mu_bias'
vi.)

vii. self.sigma_bias = self.add_weight(

1. shape=(self.units,),
2. initializer=tf.keras.initializers.Constant(self.sigma_init),
3. trainable=True,
4. name='sigma_bias'

viii.)

c) def call(self, inputs, noise=True):
   i. e_w, e_b = self.get_noise_params(noise)
   ii. noisy_weights = self.mu_weight + self.sigma_weight * e_w
   iii. noisy_bias = self.mu_bias + self.sigma_bias * e_b
   iv. outputs = tf.matmul(inputs, noisy_weights) + noisy_bias
   v. if self.activation is not None:
      1. outputs = self.activation(outputs)
   vi. return outputs

d) def get_noise_params(self, noise=True):
   i. if noise is True:
      1. e_i = tf.keras.backend.random_normal(shape=tf.shape(self.mu_weight), mean=0.0, stddev=1.0)
      2. e_j = tf.keras.backend.random_normal(shape=tf.shape(self.mu_bias), mean=0.0, stddev=1.0)
      3. e_w = tf.keras.backend.sign(e_i) * tf.keras.backend.sqrt(tf.keras.backend.abs(e_i)) * \
         i. tf.keras.backend.sign(e_j) * tf.keras.backend.sqrt(tf.keras.backend.abs(e_j))
      4. e_b = tf.keras.backend.sign(e_j) * tf.keras.backend.sqrt(tf.keras.backend.abs(e_j))
      5. return e_w, e_b
   ii. else:
      1. return 0, 0

e) def compute_output_shape(self, input_shape):
   i. output_shape = list(input_shape)
   ii. output_shape[-1] = self.units
   iii. return tuple(output_shape)

49. class drn_agent:
a) def __init__(self, max_epsilon, batch_size, memory_size):
   i. # YY 向 dueling 中加入 noisy
   ii. def build_noisy_dueling_net():
      1. inputs = keras.Input(shape=(env.N + 1, env.K))
      2. x = keras.layers.Flatten()(inputs)

```
3.  # v(s)
4.  v_dense = keras.layers.Dense(args.Units / 2, activation='relu', kernel_initializer=W_Initializer)(x)
5.  v_dense = keras.layers.Dense(args.Units / 2, activation='relu', kernel_initializer=W_Initializer)(v_dense)
6.  v_out = keras.layers.Dense(1, kernel_initializer=W_Initializer)(v_dense)

7.  # advantages with Noisy Layer
8.  adv_dense = keras.layers.Dense(args.Units / 2, activation='relu', kernel_initializer=W_Initializer)(x)
9.  adv_dense = keras.layers.Dense(args.Units / 2, activation='relu', kernel_initializer=W_Initializer)(
    a) adv_dense)
10. adv_out = NoisyLayer(Action_Num)(adv_dense)
11. adv_normal = keras.layers.Lambda(lambda x1: x1 - tf.reduce_mean(x1))(adv_out)

12. # q
13. outputs = keras.layers.add([v_out, adv_normal])
14. model = keras.Model(inputs=inputs, outputs=outputs)
15. return model
```

iii. def build_dueling_net():

```
1.  inputs = keras.Input(shape=(env.N + 1, env.K))
2.  x = keras.layers.Flatten()(inputs)

3.  # v(s)
4.  v_dense = keras.layers.Dense(args.Units / 2, activation='relu', kernel_initializer=W_Initializer)(x)
5.  v_dense = keras.layers.Dense(args.Units / 2, activation='relu', kernel_initializer=W_Initializer)(v_dense)
6.  v_out = keras.layers.Dense(1, kernel_initializer=W_Initializer)(v_dense)

7.  # advantages
8.  adv_dense = keras.layers.Dense(args.Units / 2, activation='relu', kernel_initializer=W_Initializer)(x)
9.  adv_dense = keras.layers.Dense(args.Units / 2, activation='relu', kernel_initializer=W_Initializer)(
    a) adv_dense)
10. adv_out = keras.layers.Dense(Action_Num, kernel_initializer=W_Initializer)(adv_dense)
11. adv_normal = keras.layers.Lambda(lambda x1: x1 - tf.reduce_mean(x1))(adv_out)
```

12. # q
13. outputs = keras.layers.add([v_out, adv_normal])
14. model = keras.Model(inputs=inputs, outputs=outputs)
15. return model

iv. # YY 向 DQN 中添加 noisy

v. def build_noisy_net():
1. inputs = keras.Input(shape=(env.N + 1, env.K))
2. x = keras.layers.Flatten()(inputs)

3. # 添加噪声层
4. # x = NoisyLayer(args.Units, activation='relu', sigma_init=0.5)(x)
5. # x = NoisyLayer(args.Units, activation='relu', sigma_init=0.5)(x)
6. #
7. # outputs = keras.layers.Dense(Action_Num, kernel_initializer=W_Initializer)(x)
8. x = keras.layers.Dense(args.Units, activation='relu', kernel_initializer=W_Initializer)(x)
9. x = keras.layers.Dense(args.Units, activation='relu', kernel_initializer=W_Initializer)(x)

10. # Adding NoisyLayer for the final output layer
11. outputs = NoisyLayer(Action_Num, kernel_initializer=W_Initializer)(x)
12. model = keras.Model(inputs=inputs, outputs=outputs)
13. return model

vi. def build_net():
1. inputs = keras.Input(shape=(env.N + 1, env.K))
2. x = keras.layers.Flatten()(inputs)

3. x = keras.layers.Dense(args.Units, activation='relu', kernel_initializer=W_Initializer)(x)
4. x = keras.layers.Dense(args.Units, activation='relu', kernel_initializer=W_Initializer)(x)

5. outputs = keras.layers.Dense(Action_Num, kernel_initializer=W_Initializer)(x)
6. model = keras.Model(inputs=inputs, outputs=outputs)
7. return model

vii. if 'due' in alg: # dueling
1. self.active_qnet = build_noisy_dueling_net() # 评估网络
2. self.lazy_qnet = build_noisy_dueling_net() # target q 目标网络
3. print("dueling net")

viii. elif 'dqn' in alg: # dqn
    1. self.active_qnet = build_net()
    2. self.lazy_qnet = build_net()
    3. print("dqn net")
ix. else:
    1. raise NotImplementedError("alg not implemented")

x. self.active_qnet.compile(optimizer=Optimizer, loss='mse')
xi. self.epsilon = max_epsilon
xii. self.batch_size = batch_size
xiii. # self.buffer = OldReplayBuffer(memory_size)
xiv. self.buffer = PrioritizedReplayBuffer(memory_size) # YY 用于优先经验池的缓冲区
xv. self.alg = alg
xvi. self.R = Initial_R # ATTENTION average reward 平均奖励
xvii. # self.gamma = args.Gamma # YY discount factor 折扣系数(如果不用 R 学习可以打开)

b) def choose_action(self, s, epsilon):
i. # ATTENTION
ii. if args.random is True:
    1. # print("==========random==========")
    2. return np.random.choice(Action_Num)
iii. elif args.max is True:
    1. # print("==========greedy==========")
    2. return tf.argmax(self.active_qnet(s[None, :]), 1)[0].numpy()

iv. else:
    1. # print("==========normal==========")
    2. # if np.random.random() < epsilon:
    3. # return np.random.choice(Action_Num)
    4. # else:
    5. # return tf.argmax(self.active_qnet(s[None, :]), 1)[0].numpy()
    6. # YY 添加了噪声
    7. return tf.argmax(self.active_qnet(s[None, :]), 1)[0].numpy()

c) # YY 原来的 train
d) def train(self, batch_size=args.Batch_Size):
i. # sample from buffer
ii. s, a, r, s_next = self.buffer.sample(batch_size)

iii. # calculate target q
iv. q_next_lazy = self.lazy_qnet(s_next)
v. max_q_next_lazy = tf.reduce_max(q_next_lazy, 1, True)

vi. # ATTENTION 和 DQN 不同 q_target = r + self.gamma * max_q_next_lazy # 乘以折扣系数
vii. q_target = r - self.R + max_q_next_lazy

viii. # calculate loss
ix. with tf.GradientTape() as tape:
1. q_active = self.active_qnet(s)
2. q_chosen_active = tf.gather(q_active, a, batch_dims=-1)

3. td = q_target - q_chosen_active
4. loss = tf.reduce_mean(tf.square(td))

x. # update R
xi. self.R += args.R_Beta * tf.reduce_sum(td).numpy() # YY dqn 不用
xii. grads = tape.gradient(loss, self.active_qnet.trainable_variables) # gradients
xiii. grads = [tf.clip_by_norm(grad, 10.0) for grad in grads]

xiv. self.active_qnet.optimizer.apply_gradients(zip(grads, self.active_qnet.trainable_variables))

e) # YY 优先级经验池 train
f) def train_priorities(self, batch_size=args.Batch_Size):
i. # sample from buffer
ii. b_idx, b_memory, ISWeights = self.buffer.sample(batch_size)
iii. s, a, r, s_next = zip(*b_memory) #

iv. s = np.array(s)
v. # print("s:", s)
vi. a = np.array(a)
vii. # print("a:", a)
viii. r = np.array(r)
ix. # print("r:", r)
x. s_next = np.array(s_next)
xi. # print("s_next:", s_next)

xii. # calculate target q
xiii. q_next_lazy = self.lazy_qnet(s_next) # 二维数组
xiv. # print("q_next_lazy:",q_next_lazy)
xv. max_q_next_lazy = tf.reduce_max(q_next_lazy, 1, True) # 最大 Q 值，二维数组
xvi. # print("max_q_next_lazy:",max_q_next_lazy)
xvii. max_q_next_lazy = tf.squeeze(max_q_next_lazy, axis=1) # 一维数组
xviii. # print("max_q_next_lazy1:",max_q_next_lazy)
xix. q_target = r - self.R + max_q_next_lazy

xx. # print("q_target:",q_target)

xxi. # 调试语句
xxii. # print("Min action index:", np.min(a))
xxiii. # print("Max action index:", np.max(a))

xxiv. # calculate loss
xxv. with tf.GradientTape() as tape:
1. q_active = self.active_qnet(s) # 二维数组
2. # print("q_active:",q_active)
3. # a = tf.clip_by_value(a, 0, 386) # 限制 a 的范围在[0, 386]
4. # q_chosen_active = tf.gather(q_active, a, batch_dims=-1)
5. q_chosen_active = tf.reduce_sum(q_active * tf.one_hot(a, Action_Num), axis=1) # 按动作 a 选择 Q 值 一维数组
6. # q_chosen_active = tf.gather(q_active, a, batch_dims=-1) # YY 另一种二维数组
7. # print("q_chosen_active:",q_chosen_active)

8. td = q_target - q_chosen_active
9. # print("td:",td)
10. loss = tf.reduce_mean(ISWeights * tf.square(td)) # Apply importance sampling weights

xxvi. # update R
xxvii. self.R += args.R_Beta * tf.reduce_sum(td).numpy()
xxviii. grads = tape.gradient(loss, self.active_qnet.trainable_variables)
xxix. grads = [tf.clip_by_norm(grad, 10.0) for grad in grads]

xxx. self.active_qnet.optimizer.apply_gradients(zip(grads, self.active_qnet.trainable_variables))
xxxi. # 更新优先级
xxxii. abs_td = np.abs(td)
xxxiii. # print("abs_td:",abs_td)
xxxiv. self.buffer.batch_update(b_idx, abs_td)

g) def update_lazy_q(self):
i. # update target q
ii. for lazy, active in zip(self.lazy_qnet.trainable_variables, self.active_qnet.trainable_variables):
1. lazy.assign(active)

h) def save_model(self, dir_=log_dir_name + '/models'):
i. self.lazy_qnet.save_weights(dir_ + '/' + self.alg + '_lazy_qnet.h5')
ii. self.active_qnet.save_weights(dir_ + '/' + self.alg + '_active_qnet.h5')

50. def train(points=args.Points): # points == 150
    a) agent = drn_agent(args.Max_Epsilon, args.Batch_Size, args.Memory_Size)
    b) print("============" + agent.alg + "============")

    c) st = env.reset() / env.AoI_Max # 初始化状态 St
    d) step = 0 # 每一轮次的步数
    e) summary_step = 0 # 迭代次数

    f) while summary_step < points:
        i. # ATTENTION
        ii. env.update_para()

        iii. env.simulation_user_request()

        iv. a = agent.choose_action(st, agent.epsilon)
        v. stp1, r = env.step(a) # St+1, reward
        vi. stp1 = stp1 / env.AoI_Max # normalize state 归一化
        vii. agent.buffer.store((st, a, r[0], stp1))
        viii. # YY 更新经验
        ix. # q_lazy = agent.lazy_qnet(st)
        x. # max_q_lazy = tf.reduce_max(q_lazy, 1, True)
        xi. # q_target = r - agent.R + max_q_lazy
        xii. # q_active = agent .active_qnet(st)
        xiii. # q_chosen_active = tf.gather(q_active, a, batch_dims=-1)
        xiv. # td = q_target - q_chosen_active
        xv. # abs_td = np.abs(td)
        xvi. # agent.buffer.batch_update(b_idx, abs_td)
        xvii. # transition = np.hstack((st, a, r[0], stp1))
        xviii. # agent.buffer.store(transition) # have high priority for newly arrived transition
        xix. st = stp1

        xx. # 原来的 train
        xxi. # if step > args.Start_Size: # 50000
        xxii. # agent.train(args.Batch_Size)

        xxiii. # YY 使用优先级经验池进行更新
        xxiv. if step > args.Start_Size:
            1. # 使用采样数据进行训练
            2. agent.train_priorities(args.Batch_Size)

        xxv. # 原来的 update target q

xxvi. # if step % args.Update_Lazy_Step == 0: # 2000
xxvii. # agent.update_lazy_q()

xxviii. # YY 在训练代理时，使用 优先级经验池进行更新
xxix. if step > args.Start_Size and step % args.Update_Lazy_Step == 0:
    1. agent.update_lazy_q()

xxx. # evaluate
xxxi. if step > args.Start_Size and step % args.Evaluate_Interval == 0: # Evaluate_Interval = 2000
    1. feedbacks = evaluate(agent, eps=args.Test_Epsilon)
    2. greedy_test_mean_r = feedbacks[0]
    3. aoi_cost = feedbacks[1]
    4. energy_cost = feedbacks[2]
    5. # ATTENTION
    6. extra_cost = feedbacks[3]
    7. extra_update_times = feedbacks[4]
    8. print("extra_update_times:", feedbacks[4])

    9. agent.save_model()

    10. # log
    11. with fw.as_default():
        a) print('\nData stored')
        b) tf.summary.scalar('greedy005_test_mean_r', greedy_test_mean_r, step=summary_step)
        c) tf.summary.scalar('R', agent.R, step=summary_step)
        d) tf.summary.scalar('epsilon', agent.epsilon, step=summary_step)
        e) summary_step += 1
    12. print("summary_step:",summary_step)

xxxii. # epsilon decay
xxxiii. agent.epsilon = max(Epsilon_Decay_Rate * step + args.Max_Epsilon, args.Min_Epsilon)
xxxiv. step += 1

51. if __name__ == "__main__":
    a) train(args.Points)