Homework #2

Abstract: This code is publicly available from GitHub uploaded by Singla et. al, "https://github.com/deependersingla/deep_trader/blob/master/tensor-reinforcement/pg_stock_model.py", the purpose of this code is implementing a Policy Gradient model for stock trading based on a reinforcement learning framework, it utilized TensorFlow to manage and train the model. This code utilized PG model to predict optimal stocking trading strategy by learning from market data, aiming to maximize trading performance through reinforcement learning. The model takes market state inputs and outputs actions to guide trading decisions. This code includes a neural network for policy estimation, an experience replay buffer for efficient training, and a learning loop that updates the policy based on rewards.

Core Sections:

```
# Hyper Parameters for PG
GAMMA = 0.9 # discount factor for target Q (Reward Optimization)
INITIAL_EPSILON = 1 # starting value of epsilon (Exploration Probability)
FINAL_EPSILON = 0.1 # final value of epsilon (Exploration Probability)
BATCH_SIZE = 32 # size of minibatch
LEARNING_RATE = 1e-4
```

Initialization:

```
def __init__(self, data_dictionary):
        # initialize replay buffer, exploration rate, network input and output 
dimensions.
        self.replay_buffer = []
        self.time_step = 0
        self.epsilon = INITIAL_EPSILON
        self.state_dim = data_dictionary["input"]
        self.action_dim = data_dictionary["action"]
        self.n_input = self.state_dim
        self.state_input = tf.placeholder("float", [None, self.n_input])
        self.y_input = tf.placeholder("float",[None, self.action_dim])
        self.create_pg_network(data_dictionary)
        self.create_training_method()
        self.create_supervised_accuracy()
```

Define the architecture of the neural network:

```
def create_pg_network(self, data_dictionary):
        # Define network Layers and weights
        W0 = self.weight_variable([self.state_dim,80])  #defining the weights for 
the first layer
        b0 = self.bias_variable([80])   #defining the bias for the first layer
        W1 = 
self.weight_variable([80,data_dictionary["hidden_layer_1_size"]])  #defining the 
weights for the second layer
```

```
        b1 =
self.bias_variable([data_dictionary["hidden_layer_1_size"]])   #defining the bias
for the second layer
        W2 =
self.weight_variable([data_dictionary["hidden_layer_1_size"],data_dictionary["hid
den_layer_2_size"]])     #defining the weights for the third layer
        b2 =
self.bias_variable([data_dictionary["hidden_layer_2_size"]])   #defining the bias
for the third layer
        W3 =
self.weight_variable([data_dictionary["hidden_layer_2_size"],self.action_dim])
 #defining the weights for the fourth layer
        b3 = self.bias_variable([self.action_dim])  #defining the bias for the
fourth layer
        variable_summaries(b3, "layer2/bias")   #summarizing the bias for the
second layer
        h_1_layer = tf.nn.relu(tf.matmul(self.state_input,W0) + b0) #defining the
first layer
        h_2_layer = tf.nn.relu(tf.matmul(h_1_layer,W1) + b1)    #defining the
second layer
        h_layer = tf.nn.relu(tf.matmul(h_2_layer,W2) + b2)  #defining the third
layer
        self.PG_value = tf.nn.softmax(tf.matmul(h_layer,W3) + b3)   #defining the
fourth layer
```

Define Training method:

```
def create_training_method(self):
        #this needs to be updated to use softmax
        #P_action = tf.reduce_sum(self.PG_value,reduction_indices = 1)
        #self.cost = tf.reduce_mean(tf.square(self.y_input - P_action))
        self.cost =
tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(self.PG_value,
self.y_input))    #defining the cost function
        #self.cost = tf.reduce_mean(-tf.reduce_sum(self.y_input *
tf.log(self.PG_value), reduction_indices=[1]))
        tf.scalar_summary("loss",self.cost) #summarizing the loss
        global merged_summary_op    # merging the summaries
        merged_summary_op = tf.merge_all_summaries()
        self.optimizer =
tf.train.AdamOptimizer(LEARNING_RATE).minimize(self.cost)  #defining the
optimizer
```

Accuracy Measurement and Replay Buffer & Network training:

```
def create_supervised_accuracy(self):
        correct_prediction = tf.equal(tf.argmax(self.PG_value,1),
tf.argmax(self.y_input,1))    #defining the correct prediction
```

```
        self.accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
#defining the accuracy

    def perceive(self,states,epd):
        temp = []   #temporary list
        for index, value in enumerate(states):  #iterating through the states
            temp.append([states[index], epd[index]])    #appending the states and
epd to the temporary list
        self.replay_buffer += temp  #appending the temporary list to the replay
buffer

    def train_pg_network(self):
        minibatch = random.sample(self.replay_buffer,BATCH_SIZE*5)  #sampling the
replay buffer
        state_batch = [data[0] for data in minibatch]   #defining the state batch
        y_batch = [data[1] for data in minibatch]   #defining the y batch
        #pdb.set_trace();
        self.optimizer.run(feed_dict={self.y_input:y_batch,self.state_input:state
_batch})   #running the optimizer
        summary_str = self.session.run(merged_summary_op,feed_dict={
            self.y_input:y_batch,       #feeding the y batch
            self.state_input:state_batch    #feeding the state batch
            })
        summary_writer.add_summary(summary_str,self.time_step)  #adding the
summary
        self.replay_buffer = [] #emptying the replay buffer
```

Define the action function:

```
def action(self,state):
        prob = self.PG_value.eval(feed_dict =
{self.state_input:[state]})[0]    #evaluating the policy gradient value
        action = np.argmax(prob)    #taking the argmax of the probability
        y = np.zeros([self.action_dim]) #defining the y
        y[action] = 1   #setting the y[action] to 1
        return y, action    #returning the y and action
```

Define the discounted rewards function:

```
def discounted_rewards(self,rewards):
        reward_discounted = np.zeros_like(rewards)  #defining the discounted
rewards
        track = 0   #defining the track
        for index in reversed(xrange(len(rewards))):    #iterating through the
reversed range of rewards
            track = track * GAMMA + rewards[index]  #updating the track
            reward_discounted[index] = track    #updating the discounted rewards
        return reward_discounted
```

Supervised Seeding and Accuracy Measurement:

```
def supervised_seeding(agent, data_dictionary):
    # Train the agent initially using supervised learning
    for iter in xrange(ITERATION):
        print("Iteration:")
        print(iter)
        iteration_accuracy = [] #defining the iteration accuracy
        train_iteration_accuracy = []   #defining the train iteration accuracy
        data = data_dictionary["x_train"]   #defining the data
        y_label_data = data_dictionary["y_train"]   #defining the y label data
        for episode in xrange(len(data)):   #iterating through the range of data
            state_batch, y_batch = make_supervised_input_vector(episode, data,
y_label_data)    #making the supervised input vector
            #print(episode)
            agent.train_supervised(state_batch, y_batch)    #training the agent
            accuracy = agent.supervised_accuracy(state_batch,
y_batch)  #calculating the accuracy
            train_iteration_accuracy.append(accuracy)   #appending the accuracy
        avg_accuracy = sum(train_iteration_accuracy)/
float(len(train_iteration_accuracy))  #calculating the average accuracy
        print("Train Average accuracy")
        print(avg_accuracy)

        data = data_dictionary["x_test"]
        y_label_data = data_dictionary["y_test"]
        for episode in xrange(len(data)):
            #pdb.set_trace();
            state_batch, y_batch = make_supervised_input_vector(episode, data,
y_label_data)    #making the supervised input vector
            accuracy = agent.supervised_accuracy(state_batch,
y_batch)  #calculating the accuracy
            iteration_accuracy.append(accuracy) #appending the accuracy
        avg_accuracy = sum(iteration_accuracy) / float(len(iteration_accuracy))
#calculating the average accuracy
        print("Test Average accuracy")
        print(avg_accuracy)
```