Stock Trading Using Deep Q-Learning

Problem Statement

Prepare an agent by implementing Deep Q-Learning that can perform unsupervised trading in stock trade. The aim of this project is to train an agent that uses Q-learning and neural networks to predict the profit or loss by building a model and implementing it on a dataset that is available for evaluation.

The stock trading index environment provides the agent with a set of actions:

- Buy
- Sell
- Sit

This project has following sections:

- Import libraries
- Create a DQN agent
- Preprocess the data
- Train and build the model
- Evaluate the model and agent

Steps to perform

In the section **create a DQN agent**, create a class called agent where:

- Action size is defined as 3
- Experience replay memory to deque is 1000
- Empty list for stocks that has already been bought
- The agent must possess the following hyperparameters:
  - gamma= 0.95
  - epsilon = 1.0
  - epsilon_final = 0.01
  - epsilon_decay = 0.995

```
Note: It is advised to compare the results using different values in hyperparameters.
```

- Neural network has 3 hidden layers
- Action and experience replay are defined

Solution

Import the libraries

In [ ]:
```python
import keras
from keras.models import Sequential
from keras.models import load_model
from keras.layers import Dense
from keras.optimizers import Adam
import numpy as np
import random
from collections import deque
```

Create a DQN agent

Use the instruction below to prepare an agent

In [ ]:
```python
class Agent():
    # Action space include 3 actions: Buy, Sell, and Sit
    def __init__(self, state_size, is_eval=False, model_name=""):
        #normalize the previous days
        self.state_size = state_size

        #sit, buy, sell
        self.action_size = 3

        #Setting up the experience replay memory to deque with 1000 elements inside it
        self.memory = deque(maxlen=1000)

        #Empty list with inventory is created that contains the stocks that were already bought
        self.inventory=[]
        self.model_name = model_name
        self.is_eval = is_eval

        #Setting up gamma to 0.95, that helps to maximize the current reward over the long-term
        #Epsilon parameter determines whether to use a random action or to use the model for the action.
        #In the beginning random actions are encouraged, hence epsilon is set up to 1.0 when the model is not trained.
        #And over time the epsilon is reduced to 0.01 in order to decrease the random actions and use the trained model
        #We're then set the speed of decreasing epsililon in the epsilon_decay parameter
        self.gamma = 0.95

        self.epsilon = 1.0
        self.epsilon_min = 0.01
        self.epsilon_decay = 0.995

        self.model = load_model(""+model_name) if is_eval else self._model()

#Defining our neural network
#Define the neural network function called _model and it just takes the keyword self
#Define the model with Sequential()
#Define states i.e. the previous n days and stock prices of the days
#Defining 3 hidden layers in this network
#Changing the activation function to relu because mean-squared error is used for the loss
        def _model(self):
            model = Sequential()
            model.add(Dense(units=64, input_dim = self.state_size, activation='relu'))
            model.add(Dense(units=32, activation='relu'))
            model.add(Dense(units=8, activation='relu'))

            model.add(Dense(self.action_size, activation='linear'))

            model.compile(loss = 'mse', optimizer=Adam(lr=0.001))

            return model

        def act(self, state):
            if not self.is_eval and np.random.rand() <= self.epsilon:
                return random.randrange(self.action_size)

            options = self.model.predict(state)
            return np.argmax(options[0])

        def expReplay(self, batch_size):
            mini_batch = []
            l = len(self.memory)

            for i in range(l-batch_size + 1, l):
                mini_batch.append(self.memory[i])

            for state, action, reward, next_state, done in mini_batch:
                target = reward

                if not done:
                    #amax = return the maximum of an array or maximum along an axis
                    target = reward + self.gamma*np.argmax(self.model.predict(next_state[0]))

                target_f = self.model.predict(state)
                target_f[0][action] = target

                self.model.fit(state, target_f, epochs=1, verbose=0)

            if self.epsilon > self.epsilon_min:
                self.epsilon *= self.epsilon_decay
```

Preprocess the stock market data

In [ ]:
```python
import math

# prints formatted price
def formatPrice(n):
        return ("-$" if n < 0 else "$") + "{0:.2f}".format(abs(n))

# returns the vector containing stock data from a fixed file
def getStockDataVec(key):
        vec = []
        lines = open("" + key + ".csv", "r").read().splitlines()

        for line in lines[1:]:
                vec.append(float(line.split(",")[4]))

        return vec

# returns the sigmoid
def sigmoid(x):
        return 1 / (1 + math.exp(-x))

# returns an an n-day state representation ending at time t
def getState(data, t, n):
        d = t - n + 1
        block = data[d:t + 1] if d >= 0 else -d * [data[0]] + data[0:t + 1] # pad with t0
        res = []
        for i in range(n - 1):
                res.append(sigmoid(block[i + 1] - block[i]))

        return np.array([res])
```

Train and build the model

In [ ]:
```python
import sys

if len(sys.argv) != 4:
        print ("Usage: python train.py [stock] [window] [episodes]")
        exit()


stock_name = input("Enter stock_name, window_size, Episode_count")
#Fill the given information when prompted:
#Enter stock_name = GSPC_Training_Dataset
#window_size = 10
#Episode_count = 100 or it can be 10 or 20 or 30 and so on.

window_size = input()
episode_count = input()
stock_name = str(stock_name)
window_size = int(window_size)
episode_count = int(episode_count)

agent = Agent(window_size)
data = getStockDataVec(stock_name)
l = len(data) - 1
batch_size = 32

for e in range(episode_count + 1):
        print ("Episode " + str(e) + "/" + str(episode_count))
        state = getState(data, 0, window_size + 1)

        total_profit = 0
        agent.inventory = []

        for t in range(l):
                action = agent.act(state)

                # sit
                next_state = getState(data, t + 1, window_size + 1)
                reward = 0

                if action == 1: # buy
                        agent.inventory.append(data[t])
                        print ("Buy: " + formatPrice(data[t]))

                elif action == 2 and len(agent.inventory) > 0: # sell
                        bought_price = agent.inventory.pop(0)
                        reward = max(data[t] - bought_price, 0)
                        total_profit += data[t] - bought_price
                        print ("Sell: " + formatPrice(data[t]) + " | Profit: " + formatPrice(data[t] - bought_price))

                done = True if t == l - 1 else False
                agent.memory.append((state, action, reward, next_state, done))
                state = next_state

                if done:
                        print ("--------------------------------")
                        print ("Total Profit: " + formatPrice(total_profit))


                if len(agent.memory) > batch_size:
                        agent.expReplay(batch_size)

        #if e % 10 == 0:
                agent.model.save("model_ep" + str(e))
```

Evaluate the model and agent

In [ ]:
```python
import sys
from keras.models import load_model


if len(sys.argv) != 3:
        print ("Usage: python evaluate.py [stock] [model]")
        exit()


stock_name = input("Enter Stock_name, Model_name")
model_name = input()
#Note:
#Fill the given information when prompted:
#Enter stock_name = GSPC_Evaluation_Dataset
#Model_name = respective model name

model = load_model("" + model_name)
window_size = model.layers[0].input.shape.as_list()[1]

agent = Agent(window_size, True, model_name)
data = getStockDataVec(stock_name)
l = len(data) - 1
batch_size = 32

state = getState(data, 0, window_size + 1)
total_profit = 0
agent.inventory = []

for t in range(l):
        action = agent.act(state)

        # sit
        next_state = getState(data, t + 1, window_size + 1)
        reward = 0

        if action == 1: # buy
                agent.inventory.append(data[t])
                print ("Buy: " + formatPrice(data[t]))

        elif action == 2 and len(agent.inventory) > 0: # sell
                bought_price = agent.inventory.pop(0)
                reward = max(data[t] - bought_price, 0)
                total_profit += data[t] - bought_price
                print ("Sell: " + formatPrice(data[t]) + " | Profit: " + formatPrice(data[t] - bought_price))

        done = True if t == l - 1 else False
        agent.memory.append((state, action, reward, next_state, done))
        state = next_state

        if done:
                print ("--------------------------------")
                print (stock_name + " Total Profit: " + formatPrice(total_profit))
```

Note: Run the training section for considerable episodes so that while evaluating the model it can generate significant profit.