Data visualization

COSC 480B

Reyan Ahmed

rahmed1@colgate.edu

Lecture 23

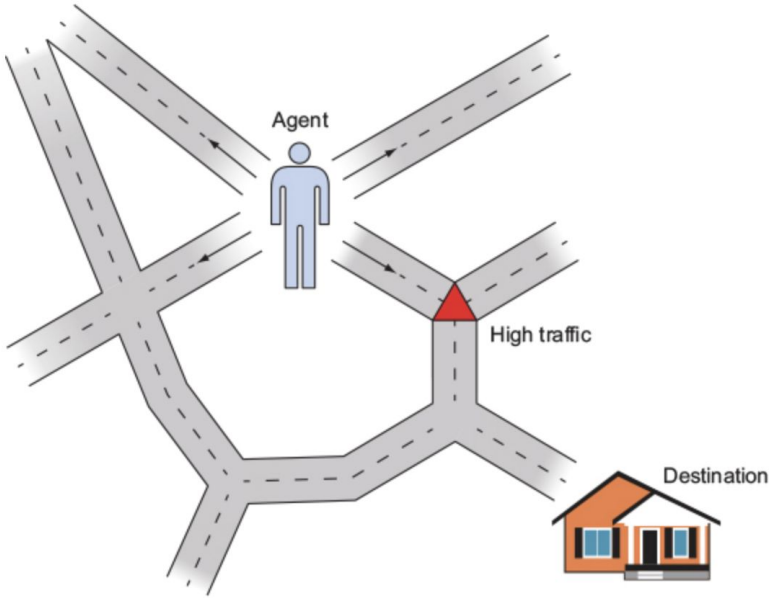
Reinforcement learning

Overview

- Defining reinforcement learning
- Implementing reinforcement learning

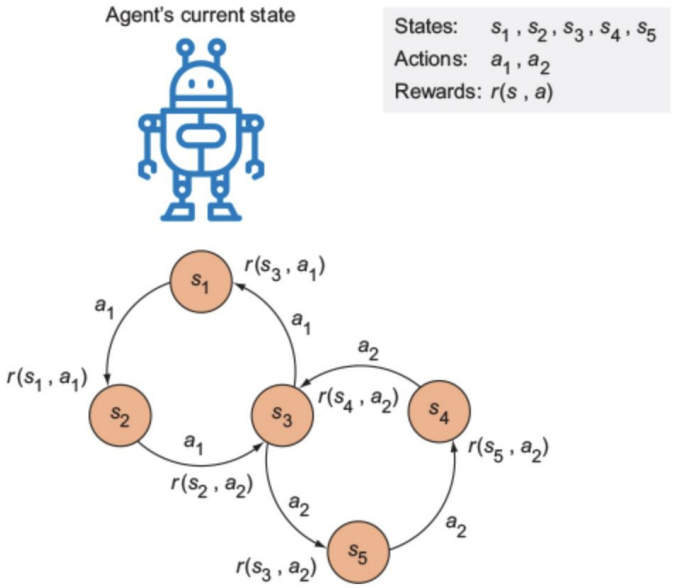
Overview

A person navigating to reach a destination in the midst of traffic and unexpected situations is a problem setup for reinforcement learning.



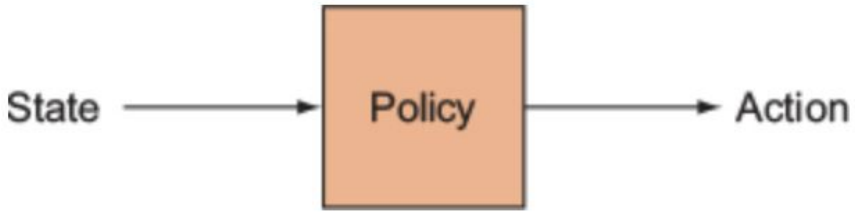
Formal notions

Actions are represented by arrows, and states are represented by circles. Performing an action on a state produces a reward. If you start at state s1, you can perform action a1 to obtain a reward r(s1, a1).



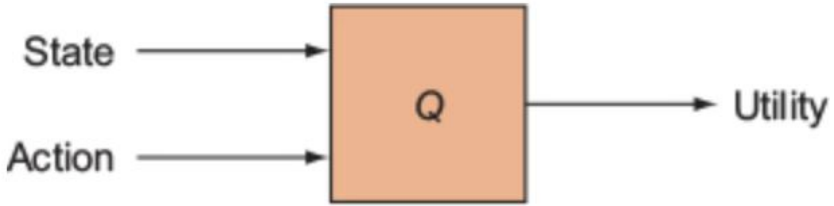
Formal notions

A policy suggests which action to take, given a state.



Formal notions

Given a state and the action taken, applying a utility function Q predicts the expected and the total rewards: the immediate reward (next state) plus rewards gained later by following an optimal policy.



Formal notions

Exercise

If you were given the utility function Q(s, a), how could you use it to derive a policy function?

Formal notions

Exercise

If you were given the utility function Q(s, a), how could you use it to derive a policy function?

ANSWER

Policy(s) = argmax_a Q(s, a)

Applying reinforcement learning

Exercise

What are some possible disadvantages of using reinforcement learning for buying and selling stocks?

Applying reinforcement learning

Exercise

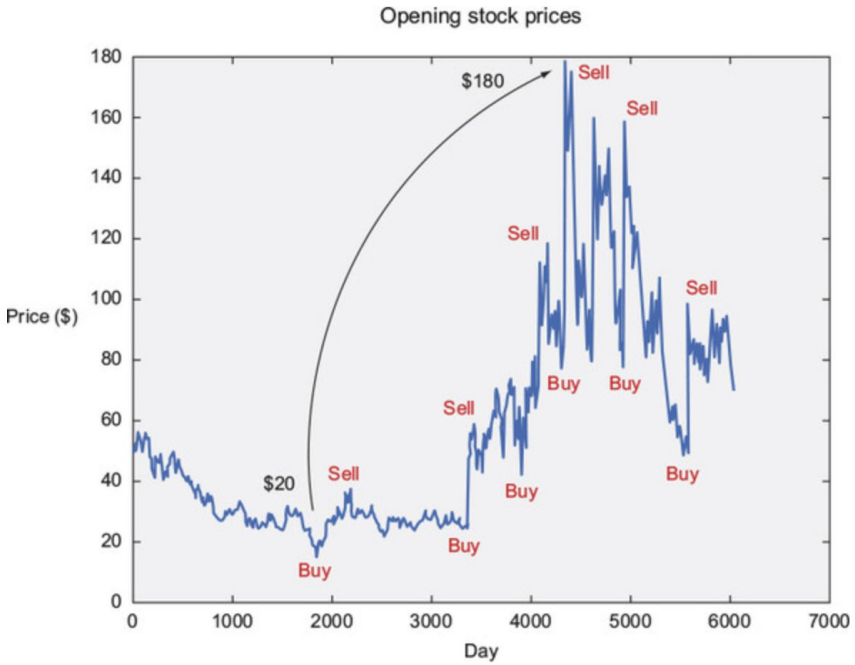
What are some possible disadvantages of using reinforcement learning for buying and selling stocks?

ANSWER

By performing actions on the market, such as buying or selling shares, you could end up influencing the market, causing it to change dramatically from your training data.

Applying reinforcement learning

Ideally, our algorithm should buy low and sell high. Doing so just once, as shown here, might yield a reward of around $160. But the real profit rolls in when you buy and sell more frequently. Ever heard the term high-frequency trading? It's about buying low and selling high as frequently as possible to maximize profits within a period of time.



Implementing reinforcement learning

Importing relevant libraries

```
from yahoo_finance import Share                1
from matplotlib import pyplot as plt           2
import numpy as np                             3
import tensorflow as tf                        3
import random
```

1 For obtaining stock-price raw data
2 For plotting stock prices
3 For numeric manipulation and machine learning

Implementing reinforcement learning

Helper function to get prices

```
def get_prices(share_symbol, start_date, end_date,
               cache_filename='stock_prices.npy'):
    try:                                                    1
        stock_prices = np.load(cache_filename)
    except IOError:
        share = Share(share_symbol)                         2
        stock_hist = share.get_historical(start_date, end_date)
        stock_prices = [stock_price['Open'] for stock_price in stock_hist] 3
        np.save(cache_filename, stock_prices)                4

    return stock_prices.astype(float)
```

1 Tries to load the data from file if it has already been computed
2 Retrieves stock prices from the library
3 Extracts only relevant info from the raw data
4 Caches the result

Implementing reinforcement learning

Helper function to plot the stock prices

```
def plot_prices(prices):
    plt.title('Opening stock prices')
    plt.xlabel('day')
    plt.ylabel('price ($)')
    plt.plot(prices)
    plt.savefig('prices.png')
    plt.show()
```

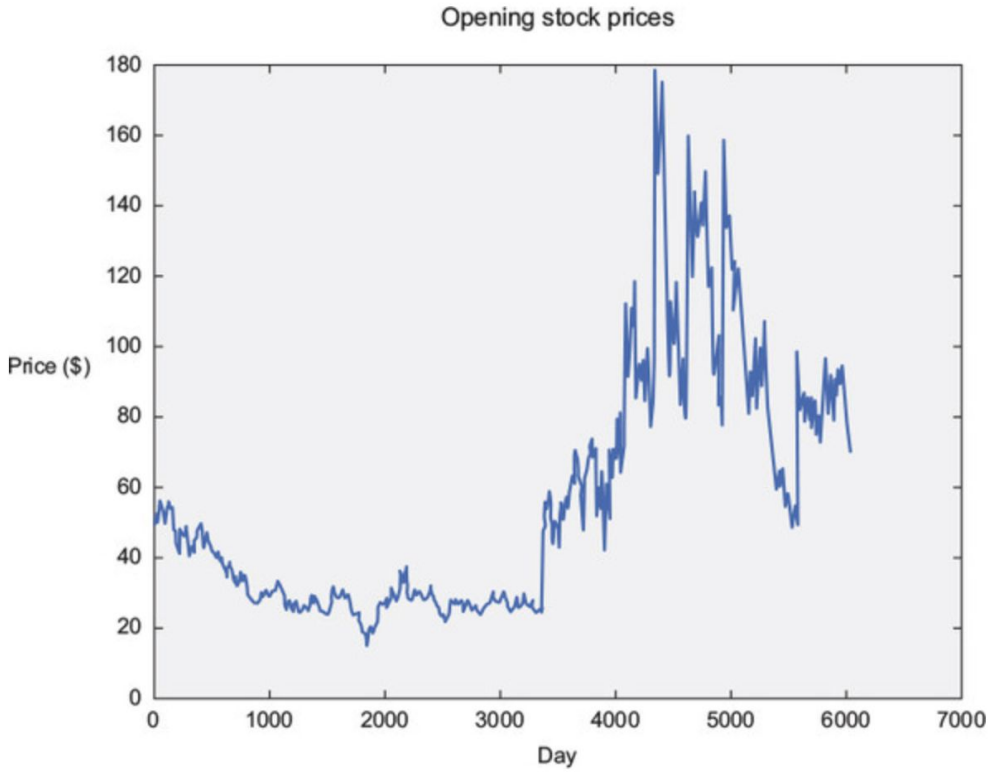
Implementing reinforcement learning

Get data and visualize it

```
if __name__ == '__main__':
  prices = get_prices('MSFT', '1992-07-22',
'2016-07-22')
  plot_prices(prices)
```

Implementing reinforcement learning

This chart summarizes the opening stock prices of Microsoft (MSFT) from 7/22/1992 to 7/22/2016. Wouldn't it have been nice to buy around day 3000 and sell around day 5000? Let's see if our code can learn to buy, sell, and hold to make optimal gain.



Implementing reinforcement learning

Defining a superclass for all decision policies

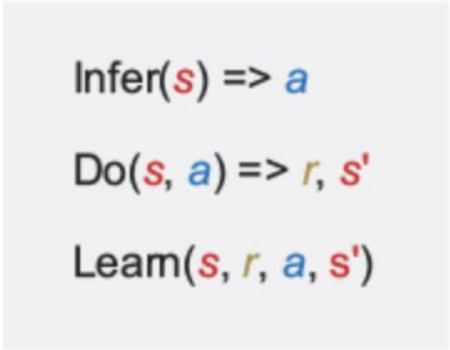
```
class DecisionPolicy:
    def select_action(self, current_state):                 1
        pass

    def update_q(self, state, action, reward, next_state):  2
        pass
```

1 Given a state, the decision policy will calculate the next action to take.
2 Improve the Q-function from a new experience of taking an action.

Implementing reinforcement learning

Most reinforcement-learning algorithms boil down to just three main steps: infer, do, and learn. During the first step, the algorithm selects the best action (a), given a state (s), using the knowledge it has so far. Next, it does the action to find out the reward (r) as well as the next state (s'). Then it improves its understanding of the world by using the newly acquired knowledge (s, r, a, s').



Implementing reinforcement learning

Implementing a random decision policy

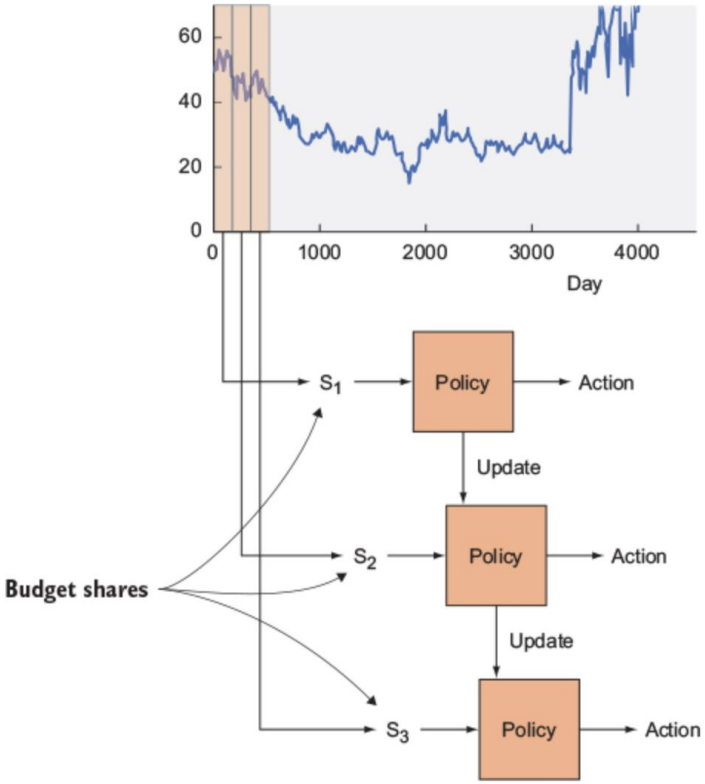
```
class RandomDecisionPolicy(DecisionPolicy):          1
    def __init__(self, actions):
        self.actions = actions

    def select_action(self, current_state):          2
        action = random.choice(self.actions)
        return action
```

1 Inherits from DecisionPolicy to implement its functions
2 Randomly chooses the next action

Implementing reinforcement learning

A rolling window of a certain size iterates through the stock prices, as shown by the chart segmented to form states S1, S2, and S3. The policy suggests an action to take: you may either choose to exploit it or randomly explore another action. As you get rewards for performing an action, you can update the policy function over time.



Implementing reinforcement learning

Using a given policy to make decisions, and returning the performance

```
def run_simulation(policy, initial_budget, initial_num_stocks,
prices, hist):
    budget = initial_budget                                    1
    num_stocks = initial_num_stocks                                1
    share_value = 0                                          1
    transitions = list()
    for i in range(len(prices) - hist - 1):
        if i % 1000 == 0:
            print('progress {:.2f}%'.format(float(100*i) / (len(prices) -
     hist - 1)))
        current_state = np.asmatrix(np.hstack((prices[i:i+hist],
budget,
     num_stocks)))                                             2
```

1 Initializes values that depend on computing the net worth of a portfolio
2 The state is a hist + 2 dimensional vector. You'll force it to be a NumPy matrix.

Implementing reinforcement learning

```
current_portfolio = budget + num_stocks * share_value          3
action = policy.select_action(current_state, i)                4
share_value = float(prices[i + hist])
if action == 'Buy' and budget >= share_value:                  5
    budget -= share_value
    num_stocks += 1
elif action == 'Sell' and num_stocks > 0:                      5
    budget += share_value
    num_stocks -= 1
else:                                                          5
    action = 'Hold'
```

3 Calculates the portfolio value
4 Selects an action from the current policy
5 Updates portfolio values based on action

Implementing reinforcement learning

```
    new_portfolio = budget + num_stocks * share_value                6
    reward = new_portfolio - current_portfolio                       7
    next_state = np.asmatrix(np.hstack((prices[i+1:i+hist+1], budget,
 num_stocks)))
    transitions.append((current_state, action, reward, next_state))
    policy.update_q(current_state, action, reward, next_state)       8

portfolio = budget + num_stocks * share_value                        9
return portfolio
```

6 Computes a new portfolio value after taking action
7 Computes the reward from taking an action at a state
8 Updates the policy after experiencing a new action
9 Computes the final portfolio worth

Implementing reinforcement learning

Running multiple simulations to calculate an average performance

```
def run_simulations(policy, budget, num_stocks, prices, hist):
    num_tries = 10                                                  1
    final_portfolios = list()                                       2
    for i in range(num_tries):
        final_portfolio = run_simulation(policy, budget, num_stocks,
prices,
     hist)                                                          3
        final_portfolios.append(final_portfolio)
        print('Final portfolio: ${}'.format(final_portfolio))
    plt.title('Final Portfolio Value')
    plt.xlabel('Simulation #')
    plt.ylabel('Net worth')
    plt.plot(final_portfolios)
    plt.show()
```

1 Decides the number of times to rerun the simulations
2 Stores the portfolio worth of each run in this array
3 Runs this simulation

Implementing reinforcement learning

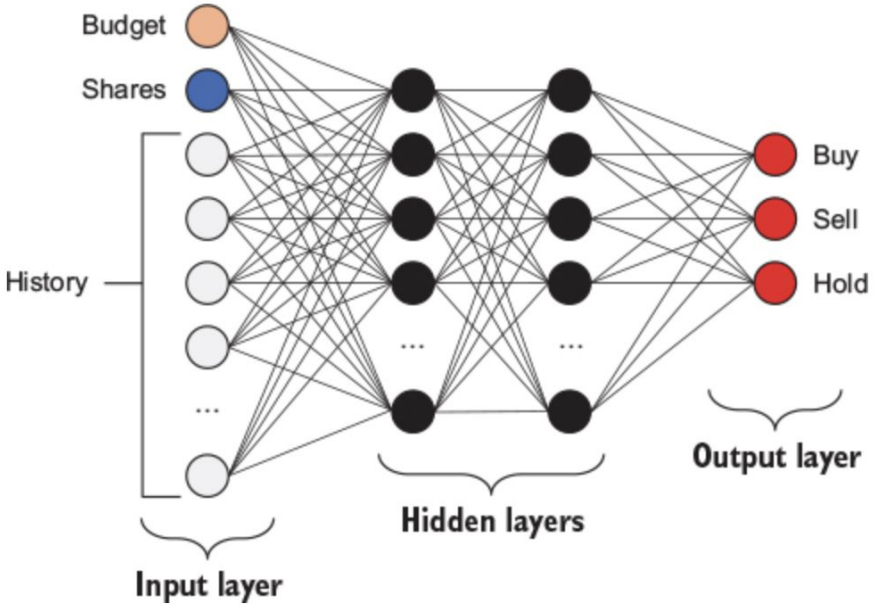
Defining the decision policy

```
if __name__ == '__main__':
    prices = get_prices('MSFT', '1992-07-22', '2016-07-22')
    plot_prices(prices)
    actions = ['Buy', 'Sell', 'Hold']                              1
    hist = 3
    policy = RandomDecisionPolicy(actions)                         2
    budget = 100000.0                                              3
    num_stocks = 0                                                 4
    run_simulations(policy, budget, num_stocks, prices, hist)      5
```

1 Defines the list of actions the agent can take
2 Initializes a random decision policy
3 Sets the initial amount of money available to use
4 Sets the number of stocks already owned
5 Runs simulations multiple times to compute the expected value of your final net worth

Implementing reinforcement learning

The input is the state space vector, with three outputs: one for each output's Q-value.



Implementing reinforcement learning

Exercise

What are other possible factors that your state-space representation ignores that can affect the stock prices? How could you factor them into the simulation?

Implementing reinforcement learning

Exercise

What are other possible factors that your state-space representation ignores that can affect the stock prices? How could you factor them into the simulation?

ANSWER

Stock prices depend on a variety of factors, including general market trends, breaking news, and specific industry trends. Each of these, once quantified, could be applied as additional dimensions to the model.

Implementing reinforcement learning

Implementing a more intelligent decision policy

```
class QLearningDecisionPolicy(DecisionPolicy):
    def __init__(self, actions, input_dim):
        self.epsilon = 0.95                                          1
        self.gamma = 0.3                                             1
        self.actions = actions
        output_dim = len(actions)
        h1_dim = 20                                                  2

        self.x = tf.placeholder(tf.float32, [None, input_dim])       3
        self.y = tf.placeholder(tf.float32, [output_dim])            3
        W1 = tf.Variable(tf.random_normal([input_dim, h1_dim]))      4
        b1 = tf.Variable(tf.constant(0.1, shape=[h1_dim]))           4
        h1 = tf.nn.relu(tf.matmul(self.x, W1) + b1)                  4
        W2 = tf.Variable(tf.random_normal([h1_dim, output_dim]))     4
        b2 = tf.Variable(tf.constant(0.1, shape=[output_dim]))       4
        self.q = tf.nn.relu(tf.matmul(h1, W2) + b2)                  5

        loss = tf.square(self.y - self.q)                            6
        self.train_op = tf.train.AdagradOptimizer(0.01).minimize(loss)   7
        self.sess = tf.Session()                                     7
        self.sess.run(tf.global_variables_initializer())             8
```

1 Sets the hyperparameters from the Q-function
2 Sets the number of hidden nodes in the neural networks
3 Defines the input and output tensors
4 Designs the neural network architecture
5 Defines the op to compute the utility
6 Sets the loss as the square error
7 Uses an optimizer to update model parameters to minimize the loss
8 Sets up the session, and initializes variables

Implementing reinforcement learning

```
    def select_action(self, current_state, step):
        threshold = min(self.epsilon, step / 1000.)
        if random.random() < threshold:                                    9
                # Exploit best option with probability epsilon
            action_q_vals = self.sess.run(self.q, feed_dict={self.x:
current_state})
            action_idx = np.argmax(action_q_vals)
            action = self.actions[action_idx]
        else:                                                             10
            # Explore random option with probability 1 - epsilon
            action = self.actions[random.randint(0,
len(self.actions) - 1)]
        return action
```

9 Exploits the best option with probability epsilon
10 Explores a random option with probability 1 - epsilon

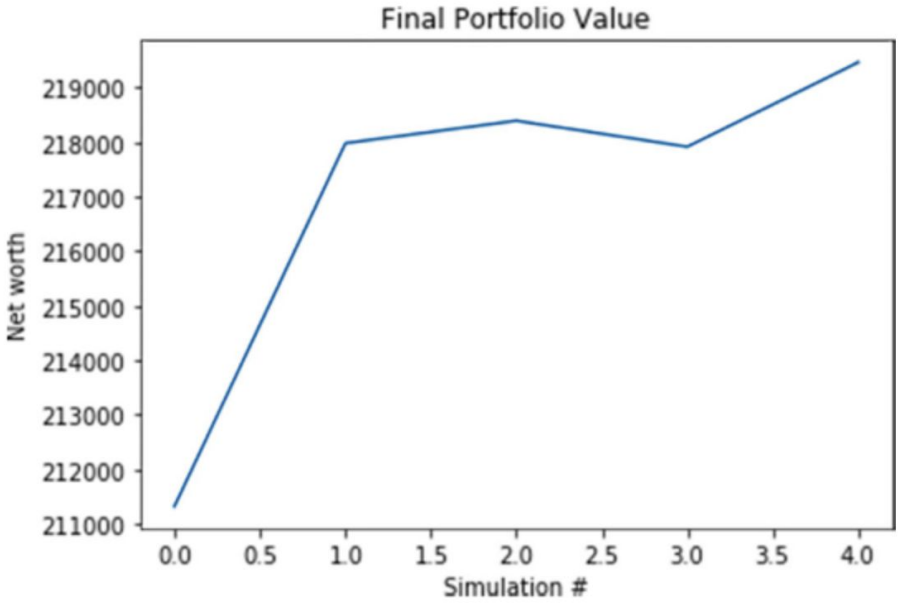
Implementing reinforcement learning

```
def update_q(self, state, action, reward, next_state):                    11
    action_q_vals = self.sess.run(self.q, feed_dict={self.x: state})
    next_action_q_vals = self.sess.run(self.q, feed_dict={self.x:
next_state})
    next_action_idx = np.argmax(next_action_q_vals)
    current_action_idx = self.actions.index(action)
    action_q_vals[0, current_action_idx] = reward + self.gamma
* next_action_q_vals[0, next_action_idx]
    action_q_vals = np.squeeze(np.asarray(action_q_vals))
    self.sess.run(self.train_op, feed_dict={self.x: state, self.y:
action_q_vals})
```

11 Updates the Q-function by updating its model parameters

Implementing reinforcement learning

The algorithm learns a good policy to trade Microsoft stocks.



Exploring other applications of reinforcement learning

- Game playing—In February 2015, Google developed a reinforcement-learning system called Deep RL to learn how to play arcade video games from the Atari 2600 console. Unlike most RL solutions, this algorithm had a high-dimensional input: it perceived the raw frame-by-frame images of the video game. That way, the same algorithm could work with any video game without much reprogramming or reconfiguring.
- More game playing—In January 2016, Google released a paper about an AI agent capable of winning the board game Go.
- Robotics and control