04_q_learning_for_trading

September 29, 2021

1 Reinforcement Learning for Trading - Deep Q-learning & the stock market

To train a trading agent, we need to create a market environment that provides price and other information, offers trading-related actions, and keeps track of the portfolio to reward the agent accordingly.

1.1 How to Design an OpenAI trading environment

The OpenAI Gym allows for the design, registration, and utilization of environments that adhere to its architecture, as described in its documentation. The `trading_env.py` file implements an example that illustrates how to create a class that implements the requisite `step()` and `reset()` methods.

The trading environment consists of three classes that interact to facilitate the agent's activities: 1. The `DataSource` class loads a time series, generates a few features, and provides the latest observation to the agent at each time step. 2. `TradingSimulator` tracks the positions, trades and cost, and the performance. It also implements and records the results of a buy-and-hold benchmark strategy. 3. `TradingEnvironment` itself orchestrates the process.

The book chapter explains these elements in more detail.

1.2 A basic trading game

To train the agent, we need to set up a simple game with a limited set of options, a relatively low-dimensional state, and other parameters that can be easily modified and extended.

More specifically, the environment samples a stock price time series for a single ticker using a random start date to simulate a trading period that, by default, contains 252 days, or 1 year. The state contains the (scaled) price and volume, as well as some technical indicators like the percentile ranks of price and volume, a relative strength index (RSI), as well as 5- and 21-day returns. The agent can choose from three actions:

- **Buy**: Invest capital for a long position in the stock
- **Flat**: Hold cash only
- **Sell short**: Take a short position equal to the amount of capital

The environment accounts for trading cost, which is set to 10bps by default. It also deducts a 1bps time cost per period. It tracks the net asset value (NAV) of the agent's portfolio and compares it against the market portfolio (which trades frictionless to raise the bar for the agent).

We use the same DDQN agent and neural network architecture that successfully learned to navigate the Lunar Lander environment. We let exploration continue for 500,000 time steps (~2,000 1yr trading periods) with linear decay of to 0.1 and exponential decay at a factor of 0.9999 thereafter.

1.3 Imports & Settings

1.3.1 Imports

[3]:
```
import warnings
warnings.filterwarnings('ignore')
```

[4]:
```
%matplotlib inline
from pathlib import Path
from time import time
from collections import deque
from random import sample

import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
from matplotlib.ticker import FuncFormatter
import seaborn as sns

import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.regularizers import l2

import gym
from gym.envs.registration import register
```

1.3.2 Settings

[5]:
```
np.random.seed(42)
tf.random.set_seed(42)
```

[6]:
```
sns.set_style('whitegrid')
```

[7]:
```
gpu_devices = tf.config.experimental.list_physical_devices('GPU')
if gpu_devices:
    print('Using GPU')
    tf.config.experimental.set_memory_growth(gpu_devices[0], True)
else:
    print('Using CPU')
```

```
Using CPU
```

[8]:
```
results_path = Path('results', 'trading_bot')
if not results_path.exists():
    results_path.mkdir(parents=True)
```

1.3.3 Helper functions

[9]:
```
def format_time(t):
    m_, s = divmod(t, 60)
    h, m = divmod(m_, 60)
    return '{:02.0f}:{:02.0f}:{:02.0f}'.format(h, m, s)
```

1.4 Set up Gym Environment

Before using the custom environment, just like with the Lunar Lander environment, we need to register it:

[10]:
```
trading_days = 252
```

[11]:
```
register(
    id='trading-v0',
    entry_point='trading_env:TradingEnvironment',
    max_episode_steps=trading_days
)
```

1.4.1 Initialize Trading Environment

We can instantiate the environment by using the desired trading costs and ticker:

[36]:
```
trading_cost_bps = 1e-3
time_cost_bps = 1e-4
```

[37]:
```
f'Trading costs: {trading_cost_bps:.2%} | Time costs: {time_cost_bps:.2%}'
```

[37]:
```
'Trading costs: 0.10% | Time costs: 0.01%'
```

[12]:
```
trading_environment = gym.make('trading-v0')
trading_environment.env.trading_days = trading_days
trading_environment.env.trading_cost_bps = trading_cost_bps
trading_environment.env.time_cost_bps = time_cost_bps
trading_environment.env.ticker = 'AAPL'
trading_environment.seed(42)
```

```
INFO:trading_env:trading_env logger started.
INFO:trading_env:loading data for AAPL…
INFO:trading_env:got data for AAPL…
INFO:trading_env:None
```

```
<class 'pandas.core.frame.DataFrame'>
MultiIndex: 9367 entries, (Timestamp('1981-01-30 00:00:00'), 'AAPL') to
(Timestamp('2018-03-27 00:00:00'), 'AAPL')
Data columns (total 10 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   returns  9367 non-null   float64
 1   ret_2    9367 non-null   float64
 2   ret_5    9367 non-null   float64
 3   ret_10   9367 non-null   float64
 4   ret_21   9367 non-null   float64
 5   rsi      9367 non-null   float64
 6   macd     9367 non-null   float64
 7   atr      9367 non-null   float64
 8   stoch    9367 non-null   float64
 9   ultosc   9367 non-null   float64
dtypes: float64(10)
memory usage: 1.5+ MB
```

[12]:
```
[42]
```

1.4.2 Get Environment Params

[13]:
```python
state_dim = trading_environment.observation_space.shape[0]
num_actions = trading_environment.action_space.n
max_episode_steps = trading_environment.spec.max_episode_steps
```

1.5 Define Trading Agent

[14]:
```python
class DDQNAgent:
    def __init__(self, state_dim,
                 num_actions,
                 learning_rate,
                 gamma,
                 epsilon_start,
                 epsilon_end,
                 epsilon_decay_steps,
                 epsilon_exponential_decay,
                 replay_capacity,
                 architecture,
                 l2_reg,
                 tau,
                 batch_size):

        self.state_dim = state_dim
        self.num_actions = num_actions
        self.experience = deque([], maxlen=replay_capacity)
```

```
        self.learning_rate = learning_rate
        self.gamma = gamma
        self.architecture = architecture
        self.l2_reg = l2_reg

        self.online_network = self.build_model()
        self.target_network = self.build_model(trainable=False)
        self.update_target()

        self.epsilon = epsilon_start
        self.epsilon_decay_steps = epsilon_decay_steps
        self.epsilon_decay = (epsilon_start - epsilon_end) / epsilon_decay_steps
        self.epsilon_exponential_decay = epsilon_exponential_decay
        self.epsilon_history = []

        self.total_steps = self.train_steps = 0
        self.episodes = self.episode_length = self.train_episodes = 0
        self.steps_per_episode = []
        self.episode_reward = 0
        self.rewards_history = []

        self.batch_size = batch_size
        self.tau = tau
        self.losses = []
        self.idx = tf.range(batch_size)
        self.train = True

    def build_model(self, trainable=True):
        layers = []
        n = len(self.architecture)
        for i, units in enumerate(self.architecture, 1):
            layers.append(Dense(units=units,
                                input_dim=self.state_dim if i == 1 else None,
                                activation='relu',
                                kernel_regularizer=l2(self.l2_reg),
                                name=f'Dense_{i}',
                                trainable=trainable))
        layers.append(Dropout(.1))
        layers.append(Dense(units=self.num_actions,
                            trainable=trainable,
                            name='Output'))
        model = Sequential(layers)
        model.compile(loss='mean_squared_error',
                      optimizer=Adam(lr=self.learning_rate))
        return model

    def update_target(self):
```

```python
        self.target_network.set_weights(self.online_network.get_weights())

    def epsilon_greedy_policy(self, state):
        self.total_steps += 1
        if np.random.rand() <= self.epsilon:
            return np.random.choice(self.num_actions)
        q = self.online_network.predict(state)
        return np.argmax(q, axis=1).squeeze()

    def memorize_transition(self, s, a, r, s_prime, not_done):
        if not_done:
            self.episode_reward += r
            self.episode_length += 1
        else:
            if self.train:
                if self.episodes < self.epsilon_decay_steps:
                    self.epsilon -= self.epsilon_decay
                else:
                    self.epsilon *= self.epsilon_exponential_decay

            self.episodes += 1
            self.rewards_history.append(self.episode_reward)
            self.steps_per_episode.append(self.episode_length)
            self.episode_reward, self.episode_length = 0, 0

        self.experience.append((s, a, r, s_prime, not_done))

    def experience_replay(self):
        if self.batch_size > len(self.experience):
            return
        minibatch = map(np.array, zip(*sample(self.experience, self.
↪batch_size)))
        states, actions, rewards, next_states, not_done = minibatch

        next_q_values = self.online_network.predict_on_batch(next_states)
        best_actions = tf.argmax(next_q_values, axis=1)

        next_q_values_target = self.target_network.predict_on_batch(next_states)
        target_q_values = tf.gather_nd(next_q_values_target,
                                       tf.stack((self.idx, tf.
↪cast(best_actions, tf.int32)), axis=1))

        targets = rewards + not_done * self.gamma * target_q_values

        q_values = self.online_network.predict_on_batch(states)
        q_values[[self.idx, actions]] = targets
```

```python
        loss = self.online_network.train_on_batch(x=states, y=q_values)
        self.losses.append(loss)

        if self.total_steps % self.tau == 0:
            self.update_target()
```

1.6 Define hyperparameters

[15]:
```python
gamma = .99,  # discount factor
tau = 100  # target network update frequency
```

1.6.1 NN Architecture

[16]:
```python
architecture = (256, 256)  # units per layer
learning_rate = 0.0001  # learning rate
l2_reg = 1e-6  # L2 regularization
```

1.6.2 Experience Replay

[17]:
```python
replay_capacity = int(1e6)
batch_size = 4096
```

1.6.3 ϵ-greedy Policy

[18]:
```python
epsilon_start = 1.0
epsilon_end = .01
epsilon_decay_steps = 250
epsilon_exponential_decay = .99
```

1.7 Create DDQN Agent

We will use TensorFlow to create our Double Deep Q-Network .

[19]:
```python
tf.keras.backend.clear_session()
```

[20]:
```python
ddqn = DDQNAgent(state_dim=state_dim,
                 num_actions=num_actions,
                 learning_rate=learning_rate,
                 gamma=gamma,
                 epsilon_start=epsilon_start,
                 epsilon_end=epsilon_end,
                 epsilon_decay_steps=epsilon_decay_steps,
                 epsilon_exponential_decay=epsilon_exponential_decay,
                 replay_capacity=replay_capacity,
                 architecture=architecture,
                 l2_reg=l2_reg,
```

```
                 tau=tau,
                 batch_size=batch_size)
```

[21]:
```
ddqn.online_network.summary()
```

```
Model: "sequential"

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Dense_1 (Dense)              (None, 256)               2816

_________________________________________________________________
Dense_2 (Dense)              (None, 256)               65792

_________________________________________________________________
dropout (Dropout)            (None, 256)               0

_________________________________________________________________
Output (Dense)               (None, 3)                 771
=================================================================
Total params: 69,379
Trainable params: 69,379
Non-trainable params: 0

_________________________________________________________________
```

1.8 Run Experiment

1.8.1 Set parameters

[22]:
```
total_steps = 0
max_episodes = 1000
```

1.8.2 Initialize variables

[23]:
```
episode_time, navs, market_navs, diffs, episode_eps = [], [], [], [], []
```

1.9 Visualization

[24]:
```
def track_results(episode, nav_ma_100, nav_ma_10,
                  market_nav_100, market_nav_10,
                  win_ratio, total, epsilon):
    time_ma = np.mean([episode_time[-100:]])
    T = np.sum(episode_time)

    template = '{:>4d} | {} | Agent: {:>6.1%} ({:>6.1%}) | '
    template += 'Market: {:>6.1%} ({:>6.1%}) | '
    template += 'Wins: {:>5.1%} | eps: {:>6.3f}'
    print(template.format(episode, format_time(total),
                          nav_ma_100-1, nav_ma_10-1,
                          market_nav_100-1, market_nav_10-1,
```

```
                        win_ratio, epsilon))
```

1.10 Train Agent

[25]:
```
start = time()
results = []
for episode in range(1, max_episodes + 1):
    this_state = trading_environment.reset()
    for episode_step in range(max_episode_steps):
        action = ddqn.epsilon_greedy_policy(this_state.reshape(-1, state_dim))
        next_state, reward, done, _ = trading_environment.step(action)

        ddqn.memorize_transition(this_state,
                                 action,
                                 reward,
                                 next_state,
                                 0.0 if done else 1.0)
        if ddqn.train:
            ddqn.experience_replay()
        if done:
            break
        this_state = next_state

    # get DataFrame with seqence of actions, returns and nav values
    result = trading_environment.env.simulator.result()

    # get results of last step
    final = result.iloc[-1]

    # apply return (net of cost) of last action to last starting nav
    nav = final.nav * (1 + final.strategy_return)
    navs.append(nav)

    # market nav
    market_nav = final.market_nav
    market_navs.append(market_nav)

    # track difference between agent an market NAV results
    diff = nav - market_nav
    diffs.append(diff)

    if episode % 10 == 0:
        track_results(episode,
                      # show mov. average results for 100 (10) periods
                      np.mean(navs[-100:]),
                      np.mean(navs[-10:]),
                      np.mean(market_navs[-100:]),
```

```
                        np.mean(market_navs[-10:]),
                        # share of agent wins, defined as higher ending nav
                        np.sum([s > 0 for s in diffs[-100:]])/min(len(diffs),␣
↪100),
                        time() - start, ddqn.epsilon)
    if len(diffs) > 25 and all([r > 0 for r in diffs[-25:]]):
        print(result.tail())
        break

trading_environment.close()
```

```
 10 | 00:00:01 | Agent: -39.1% (-39.1%) | Market:   4.6% (  4.6%) | Wins: 20.0%
| eps:  0.960
 20 | 00:00:51 | Agent: -34.0% (-28.9%) | Market:  23.2% ( 41.8%) | Wins: 20.0%
| eps:  0.921
 30 | 00:03:02 | Agent: -27.7% (-15.2%) | Market:  20.6% ( 15.4%) | Wins: 16.7%
| eps:  0.881
 40 | 00:05:11 | Agent: -22.8% ( -8.2%) | Market:  21.2% ( 23.0%) | Wins: 20.0%
| eps:  0.842
 50 | 00:07:25 | Agent: -21.8% (-17.5%) | Market:  20.2% ( 16.4%) | Wins: 20.0%
| eps:  0.802
 60 | 00:09:57 | Agent: -22.5% (-26.4%) | Market:  24.5% ( 45.6%) | Wins: 21.7%
| eps:  0.762
 70 | 00:12:38 | Agent: -20.4% ( -7.5%) | Market:  29.4% ( 59.3%) | Wins: 21.4%
| eps:  0.723
 80 | 00:15:26 | Agent: -20.9% (-24.7%) | Market:  27.6% ( 14.7%) | Wins: 22.5%
| eps:  0.683
 90 | 00:18:24 | Agent: -21.1% (-22.2%) | Market:  24.3% ( -2.1%) | Wins: 23.3%
| eps:  0.644
100 | 00:21:15 | Agent: -19.5% ( -5.3%) | Market:  24.0% ( 20.9%) | Wins: 23.0%
| eps:  0.604
110 | 00:24:16 | Agent: -16.0% ( -4.4%) | Market:  26.0% ( 25.0%) | Wins: 24.0%
| eps:  0.564
120 | 00:27:18 | Agent: -10.4% ( 27.0%) | Market:  29.9% ( 80.6%) | Wins: 25.0%
| eps:  0.525
130 | 00:30:25 | Agent:  -9.5% ( -5.8%) | Market:  36.4% ( 80.3%) | Wins: 26.0%
| eps:  0.485
140 | 00:33:52 | Agent:  -7.8% (  8.6%) | Market:  35.4% ( 13.4%) | Wins: 26.0%
| eps:  0.446
150 | 00:37:15 | Agent:  -5.5% (  6.0%) | Market:  37.9% ( 41.0%) | Wins: 28.0%
| eps:  0.406
160 | 00:40:44 | Agent:  -4.4% (-15.8%) | Market:  34.9% ( 15.7%) | Wins: 28.0%
| eps:  0.366
170 | 00:44:21 | Agent:  -4.7% (-10.8%) | Market:  32.7% ( 37.8%) | Wins: 28.0%
| eps:  0.327
180 | 00:47:52 | Agent:  -2.6% ( -2.9%) | Market:  37.6% ( 63.3%) | Wins: 28.0%
| eps:  0.287
```

```
 190 | 00:51:28 | Agent:   0.8% ( 11.2%) | Market:  43.9% ( 61.3%) | Wins: 28.0%
| eps:  0.248
 200 | 00:55:07 | Agent:  -0.2% (-14.5%) | Market:  43.9% ( 20.4%) | Wins: 28.0%
| eps:  0.208
 210 | 00:57:39 | Agent:   1.7% ( 14.5%) | Market:  45.1% ( 37.2%) | Wins: 28.0%
| eps:  0.168
 220 | 01:00:23 | Agent:   0.6% ( 15.3%) | Market:  41.2% ( 41.3%) | Wins: 30.0%
| eps:  0.129
 230 | 01:03:12 | Agent:   1.4% (  2.2%) | Market:  35.5% ( 24.2%) | Wins: 33.0%
| eps:  0.089
 240 | 01:06:03 | Agent:   1.5% (  9.4%) | Market:  32.7% (-14.7%) | Wins: 35.0%
| eps:  0.050
 250 | 01:08:51 | Agent:   0.3% ( -6.2%) | Market:  31.5% ( 28.6%) | Wins: 32.0%
| eps:  0.010
 260 | 01:11:38 | Agent:   3.6% ( 17.9%) | Market:  33.1% ( 31.6%) | Wins: 34.0%
| eps:  0.009
 270 | 01:14:24 | Agent:   6.4% ( 17.4%) | Market:  31.5% ( 21.8%) | Wins: 36.0%
| eps:  0.008
 280 | 01:17:19 | Agent:   8.6% ( 18.6%) | Market:  33.6% ( 84.3%) | Wins: 35.0%
| eps:  0.007
 290 | 01:20:21 | Agent:   6.9% ( -5.2%) | Market:  28.4% (  9.6%) | Wins: 35.0%
| eps:  0.007
 300 | 01:23:18 | Agent:  10.1% ( 16.6%) | Market:  31.0% ( 46.1%) | Wins: 36.0%
| eps:  0.006
 310 | 01:26:13 | Agent:   8.1% ( -4.9%) | Market:  31.0% ( 36.7%) | Wins: 36.0%
| eps:  0.005
 320 | 01:29:12 | Agent:   7.3% (  7.4%) | Market:  30.9% ( 40.5%) | Wins: 34.0%
| eps:  0.005
 330 | 01:32:03 | Agent:  10.9% ( 37.6%) | Market:  32.3% ( 38.6%) | Wins: 34.0%
| eps:  0.004
 340 | 01:34:48 | Agent:  15.3% ( 54.2%) | Market:  41.8% ( 79.7%) | Wins: 33.0%
| eps:  0.004
 350 | 01:38:07 | Agent:  18.1% ( 21.0%) | Market:  42.4% ( 35.0%) | Wins: 38.0%
| eps:  0.004
 360 | 01:41:09 | Agent:  20.5% ( 41.9%) | Market:  39.9% (  6.4%) | Wins: 39.0%
| eps:  0.003
 370 | 01:44:11 | Agent:  21.6% ( 28.7%) | Market:  40.8% ( 30.6%) | Wins: 39.0%
| eps:  0.003
 380 | 01:47:17 | Agent:  19.6% ( -1.4%) | Market:  39.3% ( 69.3%) | Wins: 40.0%
| eps:  0.003
 390 | 01:50:20 | Agent:  22.5% ( 23.7%) | Market:  42.6% ( 42.7%) | Wins: 41.0%
| eps:  0.002
 400 | 01:53:19 | Agent:  22.2% ( 13.9%) | Market:  40.2% ( 22.7%) | Wins: 44.0%
| eps:  0.002
 410 | 01:57:03 | Agent:  24.5% ( 18.5%) | Market:  42.3% ( 57.8%) | Wins: 44.0%
| eps:  0.002
 420 | 02:00:58 | Agent:  28.4% ( 45.9%) | Market:  38.6% (  2.7%) | Wins: 48.0%
| eps:  0.002
```

```
 430 | 02:04:21 | Agent:  25.8% ( 11.1%) | Market:  36.7% ( 19.6%) | Wins: 48.0%
| eps:  0.002
 440 | 02:07:56 | Agent:  25.8% ( 54.4%) | Market:  30.4% ( 17.3%) | Wins: 49.0%
| eps:  0.001
 450 | 02:11:14 | Agent:  27.4% ( 37.6%) | Market:  31.8% ( 48.6%) | Wins: 46.0%
| eps:  0.001
 460 | 02:14:15 | Agent:  26.1% ( 28.9%) | Market:  38.0% ( 68.8%) | Wins: 44.0%
| eps:  0.001
 470 | 02:17:15 | Agent:  27.0% ( 37.6%) | Market:  37.1% ( 21.5%) | Wins: 45.0%
| eps:  0.001
 480 | 02:20:20 | Agent:  34.1% ( 69.4%) | Market:  35.9% ( 56.9%) | Wins: 48.0%
| eps:  0.001
 490 | 02:23:17 | Agent:  34.8% ( 30.6%) | Market:  31.6% ( -0.1%) | Wins: 51.0%
| eps:  0.001
 500 | 02:26:40 | Agent:  36.1% ( 27.3%) | Market:  32.3% ( 29.9%) | Wins: 49.0%
| eps:  0.001
 510 | 02:30:11 | Agent:  35.4% ( 11.0%) | Market:  30.2% ( 36.8%) | Wins: 49.0%
| eps:  0.001
 520 | 02:33:49 | Agent:  31.6% (  7.5%) | Market:  32.8% ( 29.0%) | Wins: 46.0%
| eps:  0.001
 530 | 02:37:32 | Agent:  34.1% ( 36.8%) | Market:  33.1% ( 22.4%) | Wins: 46.0%
| eps:  0.001
 540 | 02:41:29 | Agent:  35.4% ( 67.0%) | Market:  33.1% ( 17.4%) | Wins: 50.0%
| eps:  0.001
 550 | 02:45:14 | Agent:  35.4% ( 38.0%) | Market:  33.2% ( 49.0%) | Wins: 51.0%
| eps:  0.000
 560 | 02:48:50 | Agent:  36.0% ( 35.0%) | Market:  29.9% ( 35.9%) | Wins: 54.0%
| eps:  0.000
 570 | 02:52:27 | Agent:  36.1% ( 38.4%) | Market:  31.1% ( 34.2%) | Wins: 53.0%
| eps:  0.000
 580 | 02:56:06 | Agent:  33.4% ( 41.9%) | Market:  33.4% ( 79.9%) | Wins: 50.0%
| eps:  0.000
 590 | 02:59:44 | Agent:  31.5% ( 12.6%) | Market:  37.0% ( 35.8%) | Wins: 46.0%
| eps:  0.000
 600 | 03:03:31 | Agent:  31.7% ( 29.3%) | Market:  39.2% ( 51.3%) | Wins: 45.0%
| eps:  0.000
 610 | 03:07:22 | Agent:  35.1% ( 44.4%) | Market:  38.5% ( 29.6%) | Wins: 50.0%
| eps:  0.000
 620 | 03:11:15 | Agent:  38.5% ( 42.1%) | Market:  39.6% ( 40.6%) | Wins: 52.0%
| eps:  0.000
 630 | 03:15:03 | Agent:  36.5% ( 16.7%) | Market:  40.7% ( 33.5%) | Wins: 51.0%
| eps:  0.000
 640 | 03:18:45 | Agent:  31.8% ( 20.2%) | Market:  44.5% ( 54.8%) | Wins: 44.0%
| eps:  0.000
 650 | 03:22:24 | Agent:  32.7% ( 46.9%) | Market:  40.2% (  6.4%) | Wins: 47.0%
| eps:  0.000
 660 | 03:26:06 | Agent:  32.1% ( 28.6%) | Market:  38.9% ( 22.9%) | Wins: 46.0%
| eps:  0.000
```

```
670 | 03:29:49 | Agent:  29.5% ( 12.3%) | Market:  37.5% ( 20.2%) | Wins: 46.0%
| eps:  0.000
680 | 03:33:33 | Agent:  28.3% ( 29.7%) | Market:  31.2% ( 16.4%) | Wins: 50.0%
| eps:  0.000
690 | 03:37:19 | Agent:  32.8% ( 57.9%) | Market:  30.5% ( 29.1%) | Wins: 53.0%
| eps:  0.000
700 | 03:41:07 | Agent:  32.2% ( 23.2%) | Market:  27.8% ( 24.7%) | Wins: 53.0%
| eps:  0.000
710 | 03:44:55 | Agent:  32.2% ( 44.6%) | Market:  25.4% (  5.0%) | Wins: 51.0%
| eps:  0.000
720 | 03:48:46 | Agent:  33.6% ( 55.8%) | Market:  25.5% ( 41.7%) | Wins: 49.0%
| eps:  0.000
730 | 03:52:39 | Agent:  32.4% (  4.3%) | Market:  26.5% ( 44.2%) | Wins: 51.0%
| eps:  0.000
740 | 03:56:31 | Agent:  32.2% ( 18.9%) | Market:  25.7% ( 46.5%) | Wins: 54.0%
| eps:  0.000
750 | 04:00:26 | Agent:  28.4% (  8.5%) | Market:  28.4% ( 33.3%) | Wins: 51.0%
| eps:  0.000
760 | 04:04:22 | Agent:  26.1% (  5.5%) | Market:  26.9% (  8.3%) | Wins: 52.0%
| eps:  0.000
770 | 04:08:19 | Agent:  25.8% (  9.4%) | Market:  28.1% ( 31.9%) | Wins: 51.0%
| eps:  0.000
780 | 04:12:19 | Agent:  27.2% ( 44.1%) | Market:  28.0% ( 15.5%) | Wins: 52.0%
| eps:  0.000
790 | 04:16:20 | Agent:  24.1% ( 26.4%) | Market:  29.0% ( 38.5%) | Wins: 52.0%
| eps:  0.000
800 | 04:20:23 | Agent:  23.8% ( 20.3%) | Market:  29.3% ( 28.1%) | Wins: 53.0%
| eps:  0.000
810 | 04:24:27 | Agent:  27.1% ( 77.7%) | Market:  35.6% ( 67.6%) | Wins: 51.0%
| eps:  0.000
820 | 04:28:34 | Agent:  24.5% ( 30.3%) | Market:  34.7% ( 33.0%) | Wins: 51.0%
| eps:  0.000
830 | 04:32:43 | Agent:  26.3% ( 21.6%) | Market:  30.3% (  0.7%) | Wins: 52.0%
| eps:  0.000
840 | 04:36:53 | Agent:  30.3% ( 59.3%) | Market:  26.7% (  9.7%) | Wins: 54.0%
| eps:  0.000
850 | 04:41:05 | Agent:  33.6% ( 41.2%) | Market:  26.2% ( 28.5%) | Wins: 54.0%
| eps:  0.000
860 | 04:45:20 | Agent:  38.5% ( 55.0%) | Market:  27.7% ( 23.3%) | Wins: 52.0%
| eps:  0.000
870 | 04:49:36 | Agent:  42.9% ( 53.4%) | Market:  24.0% ( -4.8%) | Wins: 56.0%
| eps:  0.000
880 | 04:53:54 | Agent:  48.2% ( 96.8%) | Market:  23.8% ( 13.1%) | Wins: 56.0%
| eps:  0.000
890 | 04:58:14 | Agent:  50.0% ( 44.1%) | Market:  22.2% ( 22.3%) | Wins: 56.0%
| eps:  0.000
900 | 05:02:35 | Agent:  52.9% ( 49.5%) | Market:  22.0% ( 26.5%) | Wins: 58.0%
| eps:  0.000
```

```
 910 | 05:06:58 | Agent:  52.3% ( 72.1%) | Market:  15.5% (  2.9%) | Wins: 60.0%
| eps:  0.000
 920 | 05:11:23 | Agent:  53.9% ( 45.7%) | Market:  16.9% ( 46.4%) | Wins: 60.0%
| eps:  0.000
 930 | 05:15:49 | Agent:  60.2% ( 85.2%) | Market:  15.2% (-16.4%) | Wins: 60.0%
| eps:  0.000
 940 | 05:20:18 | Agent:  56.8% ( 25.5%) | Market:  15.8% ( 15.7%) | Wins: 60.0%
| eps:  0.000
 950 | 05:24:51 | Agent:  58.4% ( 56.6%) | Market:  17.2% ( 43.3%) | Wins: 61.0%
| eps:  0.000
 960 | 05:29:24 | Agent:  57.4% ( 45.0%) | Market:  21.4% ( 65.3%) | Wins: 59.0%
| eps:  0.000
 970 | 05:33:58 | Agent:  54.2% ( 21.4%) | Market:  22.2% (  2.4%) | Wins: 59.0%
| eps:  0.000
 980 | 05:38:33 | Agent:  49.7% ( 52.2%) | Market:  22.9% ( 20.5%) | Wins: 57.0%
| eps:  0.000
 990 | 05:43:11 | Agent:  47.6% ( 22.9%) | Market:  19.9% ( -7.9%) | Wins: 57.0%
| eps:  0.000
1000 | 05:47:51 | Agent:  46.8% ( 41.5%) | Market:  17.6% (  3.5%) | Wins: 57.0%
| eps:  0.000
```

1.10.1 Store Results

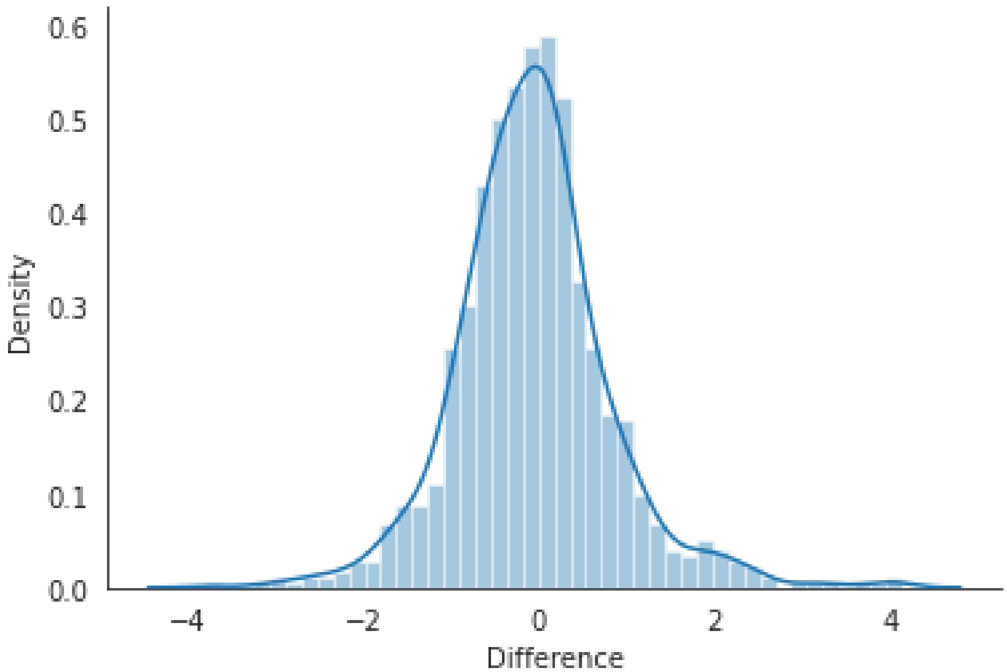
[26]:
```
results = pd.DataFrame({'Episode': list(range(1, episode+1)),
                        'Agent': navs,
                        'Market': market_navs,
                        'Difference': diffs}).set_index('Episode')

results['Strategy Wins (%)'] = (results.Difference > 0).rolling(100).sum()
results.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1000 entries, 1 to 1000
Data columns (total 4 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   Agent              1000 non-null   float64
 1   Market             1000 non-null   float64
 2   Difference         1000 non-null   float64
 3   Strategy Wins (%)  901 non-null    float64
dtypes: float64(4)
memory usage: 39.1 KB
```

[27]:
```
results.to_csv(results_path / 'results.csv', index=False)
```

[28]:
```
with sns.axes_style('white'):
    sns.distplot(results.Difference)
    sns.despine()
```



1.10.2 Evaluate Results

[29]:
```
results.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1000 entries, 1 to 1000
Data columns (total 4 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   Agent             1000 non-null   float64
 1   Market            1000 non-null   float64
 2   Difference        1000 non-null   float64
 3   Strategy Wins (%) 901 non-null    float64
dtypes: float64(4)
memory usage: 39.1 KB
```

The following diagram shows the rolling average of agent and market returns over 100 periods on the left, and the share of the last 100 periods the agent outperformed the market on the right. It uses AAPL stock data with some 9,000 daily price and volume observations, corresponding to ~35 years of data.

It shows how the agent's performance improves significantly while exploring at a higher rate over the first ~600 periods (that is, years) and approaches a level where it outperforms the market around 40 percent of the time, despite transaction costs. In an increasing number of instances, it beats the market over half the time out of 100 periods:

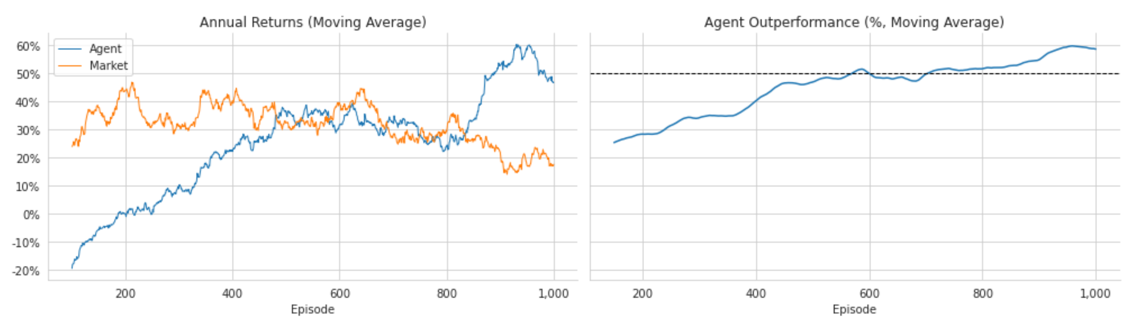
[30]:
```
fig, axes = plt.subplots(ncols=2, figsize=(14, 4), sharey=True)

df1 = (results[['Agent', 'Market']]
       .sub(1)
       .rolling(100)
       .mean())
df1.plot(ax=axes[0],
         title='Annual Returns (Moving Average)',
         lw=1)

df2 = results['Strategy Wins (%)'].div(100).rolling(50).mean()
df2.plot(ax=axes[1],
         title='Agent Outperformance (%, Moving Average)')

for ax in axes:
    ax.yaxis.set_major_formatter(
        FuncFormatter(lambda y, _: '{:.0%}'.format(y)))
    ax.xaxis.set_major_formatter(
        FuncFormatter(lambda x, _: '{:,.0f}'.format(x)))
axes[1].axhline(.5, ls='--', c='k', lw=1)

sns.despine()
fig.tight_layout()
fig.savefig(results_path / 'performance', dpi=300)
```



1.11 Summary

This relatively simple agent uses **no information beyond the latest market data and the reward signal** compared to the machine learning models we covered elsewhere in this book. Nonetheless, it learns to make a profit and achieve performance similar to that of the market (after training on <1,000 years' worth of data, which takes only a fraction of the time on a GPU).

Keep in mind that using a single stock also increases the **risk of overfitting** to the data — by a lot. You can test your trained agent on new data using the saved model (see the notebook for Lunar Lander).

In summary, we have demonstrated the **mechanics of setting up an RL trading environment** and experimented with a basic agent that uses a small number of technical indicators. You should **try to extend both the environment and the agent** - for example, by allowing it to choose from several assets, size the positions, and manage risks.

Reinforcement learning is often considered **one of the most promising approaches to algorithmic trading** because it most accurately models the task an investor is facing. However, our dramatically simplified examples illustrate that creating a **realistic environment poses a considerable challenge**. Moreover, deep reinforcement learning that has achieved impressive breakthroughs in other domains may face greater obstacles given the noisy nature of financial data, which makes it even harder to learn a value function based on delayed rewards.

Nonetheless, the substantial interest in this subject makes it likely that institutional investors are working on larger-scale experiments that may yield tangible results. An interesting complementary approach beyond the scope of this book is **Inverse Reinforcement Learning**, which aims to identify the reward function of an agent(for example, a human trader) given its observed behavior; see Arora and Doshi (2019) for a survey and Roa-Vicens et al. (2019) for an application on trading in the limit-order book context.

[ ]:

```

```