O'REILLY®

Helion

Uczenie przez wzmacnianie w finansach

Wprowadzenie z wykorzystaniem Pythona

Yves J. Hilpisch

Tytuł oryginału: Reinforcement Learning for Finance: A Python-Based Introduction

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-289-2578-6

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
helion.pl/user/opinie/uczfin
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

ROZDZIAŁ 8.

Dynamiczna alokacja w aktywa

Zawodowi hazardziści, którzy muszą mieć jakąś przewagę, mówią o „zarządzaniu pieniędzmi". Dotyczy to trudnej i bardzo ważnej kwestii, jak osiągnąć największy zysk z korzystnej okazji do obstawienia. Możesz być najlepszym na świecie pokerzystą, graczem w backgammona lub typerem, ale jeśli nie potrafisz zarządzać pieniędzmi, skończysz bez grosza. Smutnym faktem jest to, że prawie każdy, kto uprawia hazard, w dłuższej perspektywie bankrutuje.

— Poundstone (2010)

Przez ostatnie dwie dekady światowa gospodarka rozwijała się w przyzwoitym tempie, na poziomie ponad 3% rocznie. Jednak wzrost ilości pieniądza na rynku był zdecydowanie szybszy. W latach 2000 – 2020 łączne zasoby wzrosły ze 160 bilionów dolarów, czyli czterokrotności globalnej produkcji, do 510 bilionów dolarów, czyli sześciokrotności produkcji.

— „The Economist" (2023)

Wyzwania związane z alokacją w aktywa są poważnym problemem w dziedzinie finansów, intensyfikowanym przez ogromne kwoty pieniędzy inwestowanych przez osoby fizyczne i organizacje. Jest to również problem, który zapoczątkował rewolucję ilościową w finansach wraz z przełomową pracą Markowitza (1952) na temat „budowy portfela". W tym artykule Markowitz proponuje czysto statystyczną metodę budowania portfeli w porównaniu na przykład z analizą fundamentalną spółek i ich akcji.

Choć wczesne prace z tej dziedziny koncentrują się na *statycznym* (czyli niepowtarzającym się) problemie alokacji środków w różne aktywa, bardziej realistycznym podejściem jest przyjęcie podejścia *dynamicznego* (czyli z wielokrotnym wyborem). Dynamiczna alokacja w aktywa, podobnie jak handel algorytmiczny i dynamiczny hedging, jest problemem, który dobrze wpisuje się w ogólne ramy programowania dynamicznego przedstawione w rozdziale 3. Dlatego również ten problem można próbować rozwiązać za pomocą algorytmu DQL w celu uzyskania przybliżonych numerycznych rozwiązań. Artykuł Mertona (1969) to jedna z wczesnych prac nad dynamiczną alokacją w aktywa w modelu czasu ciągłego, gdzie niepewność jest określona za pomocą geometrycznych ruchów Browna. Autor wykorzystuje programowanie dynamiczne i zasadę Bellmana, aby wyprowadzić optymalne rozwiązania dla kilku szczególnych scenariuszy, w tym prostego przypadku dwóch aktywów i bardziej realistycznego przypadku wielu aktywów z nieskończonym horyzontem.

W tym rozdziale omawiam trzy scenariusze dynamicznej alokacji w aktywa. W pierwszym, przedstawionym w następnej podrozdziale, dostępne są dwa aktywa, ryzykowne i wolne od ryzyka. W podrozdziale „Scenariusz z dwoma aktywami" opisuję przypadek z dwoma ryzykownymi aktywami. W rozwinięciu w podrozdziale „Scenariusz z trzema aktywami" dodaję do portfela inwestycyjnego trzecie ryzykowne aktywo. Począwszy od trzech aktywów, uogólnianie na cztery instrumenty i większą ich liczbę jest proste. W podrozdziale „Portfel z równymi wagami" porównuję wyniki ze scenariusza z trzema aktywami z wynikami portfela z równymi wagami.

Podział między dwa fundusze

Koncepcja *podziału między dwa fundusze* pochodzi od Markowitza (1952). Zgodnie z nią przy zachowaniu równowagi i przy określonych założeniach inwestorzy na rynku finansowym będą posiadać kombinację aktywów wolnych od ryzyka i ryzykownego portfela rynkowego — i nic więcej. Portfel rynkowy znajduje się na efektywnej granicy zbioru możliwych kombinacji ryzyka i stóp zwrotu. *Efektywna granica* reprezentuje wszystkie portfele, które dają maksymalną oczekiwaną stopę zwrotu dla danego poziomu ryzyka. W praktycznych zastosowaniach portfel rynkowy, w który nie da się bezpośrednio inwestować, jest zazwyczaj przybliżany za pomocą szerokiego indeksu giełdowego, takiego jak S&P 500. Linia prosta łącząca aktywa wolne od ryzyka z portfelem rynkowym w przestrzeni ryzyka i stóp zwrotu jest nazywana *linią rynku kapitałowego* (CML). Więcej szczegółów na ten i powiązane tematy znajdziesz w rozdziale 5. pracy Copelanda, Westona i Shastriego (2005).

W poniższym kodzie w Pythonie na podstawie kilku prostych założeń numerycznych wizualnie zilustrowałem linię rynku kapitałowego. Na początku znajdują się standardowe operacje importu i konfiguracja:

```
In [1]: import math
        import random
        import numpy as np
        import pandas as pd
        from scipy import stats
        from pylab import plt, mpl

In [2]: plt.style.use('seaborn-v0_8')
        mpl.rcParams['figure.dpi'] = 300
        mpl.rcParams['savefig.dpi'] = 300
        mpl.rcParams['font.family'] = 'serif'
        np.set_printoptions(suppress=True)
        pd.set_option('display.float_format', lambda x: '%.3f' % x)
```

Rysunek 8.1 przedstawia ilustrację linii rynku kapitałowego. Bez krótkiej sprzedaży inwestor może osiągnąć dowolną kombinację ryzyka i stóp zwrotu na linii łączącej aktywa wolne od ryzyka (trójkąt) z portfelem rynkowym (gruba kropka). Jeśli krótka sprzedaż jest dozwolona, osiągalne są również kombinacje na prawo od portfela rynkowego. Portfele te reprezentowałyby *pozycje lewarowane* w portfelu rynkowym. Każda taka pozycja byłaby połączeniem krótkiej pozycji w aktywach wolnych od ryzyka i długiej pozycji w portfelu rynkowym większej niż 100% inwestowanego kapitału. Podsumowując, linia rynku kapitałowego obrazuje jedną z podstawowych koncepcji w finansach: inwestor, który jest skłonny ponieść większe ryzyko,

może *oczekiwać* — przy zachowaniu stałych pozostałych czynników — wyższego zwrotu z inwestycji:

```
In [3]: r = 0.025 ❶
        beta = 0.2 ❷
        sigma = 0.375 ❸
        mu = r + beta * sigma ❹
        mu ❹
Out[3]: 0.1

In [4]: vol = np.linspace(0, 0.5) ❺
        ret = r + beta * vol ❺

In [5]: fig, ax = plt.subplots()
        plt.plot(vol, ret, 'b', label='linia rynku kapitałowego')
        plt.plot(0, r, 'g^', label='aktywo wolne od ryzyka')
        plt.plot(sigma, mu, 'ro', label='portfel rynkowy')
        plt.xlabel('zmienność/ryzyko')
        plt.ylabel('oczekiwana stopa zwrotu')
        ax.set_xticks((0, sigma))
        ax.set_xticklabels((0, '$\sigma$',))
        ax.set_yticks((0, r, mu))
        ax.set_yticklabels((0, '$r$', '$\mu$'))
        plt.ylim(0, 0.15)
        plt.legend();
```

❶ Stopa zwrotu dla aktywów wolnych od ryzyka

❷ Nachylenie linii rynku kapitałowego

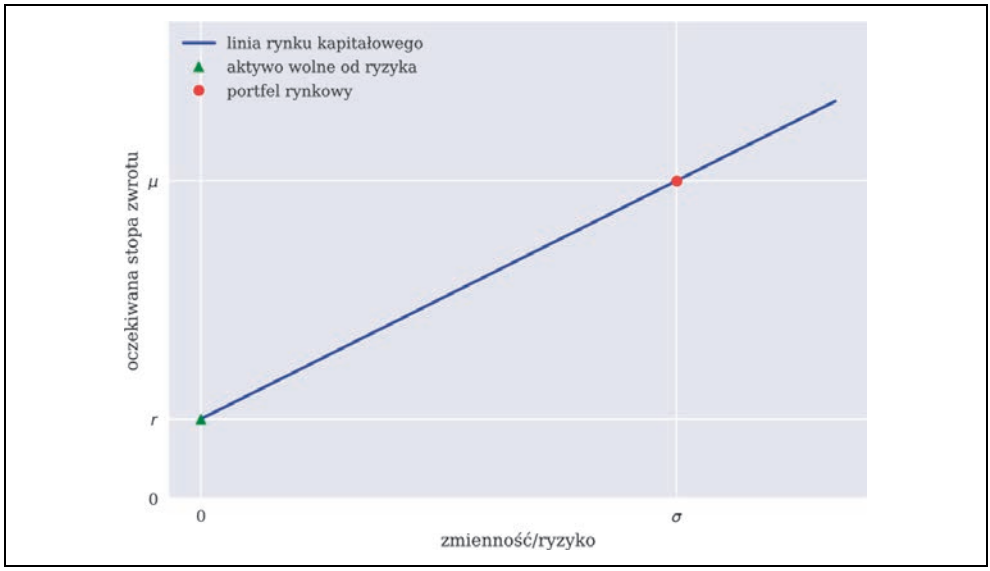
❸ Zmienność portfela rynkowego

❹ Oczekiwana stopa zwrotu dla portfela rynkowego

❺ Kombinacje ryzyka i stopy zwrotu do umieszczenia na wykresie

Portfele 60/40

Popularną strategią inwestycyjną, proponowaną od dziesięcioleci przez branżę zarządzania aktywami i środowiska akademickie, jest tak zwany portfel 60/40 (60% portfela alokowane jest w akcje, a 40% w obligacje). Chociaż obligacje nie są wolne od ryzyka, założenia tej strategii są podobne jak przy podziale na dwa fundusze. Dodanie mniej ryzykownych obligacji do portfela akcyjnego zmniejsza ogólny poziom ryzyka, a jednocześnie pozwala wykorzystać długoterminowy potencjał wzrostowy rynku akcji dzięki większej alokacji w tę część portfela. Często obserwowano również, że ceny obligacji i akcji są ujemnie skorelowane ze sobą, co może dodatkowo zmniejszać ryzyko portfela. Te cechy powinny być szczególnie atrakcyjne dla inwestorów o umiarkowanej akceptacji ryzyka. Jednak na przykład w 2022 roku portfel o takim składzie osiągnął słabe wyniki, głównie z powodu szybko rosnących stóp procentowych. Więcej informacji i szczegółów znajdziesz w pracy Chisholma (2023), który przedstawia również dane dotyczące wyników osiąganych na przestrzeni wielu dziesięcioleci.



Rysunek 8.1. Linia rynku kapitałowego

Teraz pokażę, jak uczyć agenta DQL inwestowania w dwa rodzaje aktywów. Aktywa wolne od ryzyka mają stałą stopę zwrotu. Ryzykowne aktywa są modelowane za pomocą geometrycznych ruchów Browna, jak w pracach Mertona (1969 i 1973) oraz Blacka i Scholesa (1973). Podejście przyjęte w tym podrozdziale jest podobne do zastosowanego w rozdziale 7. Dlatego środowisko `Investing`, rozwijane krok po kroku w tym rozdziale, przypomina środowisko `Hedging`. Tak jak poprzednio używam dwóch klas pomocniczych. Agent może wybrać wielkość pozycji w ryzykownym aktywie z przedziału jednostkowego. Wartość 0 oznacza brak inwestycji w ryzykowne aktywa, a wartość 1 oznacza 100% inwestycji w nie. Różnica między pozycją zainwestowaną w ryzykowne aktywa a 1 (czyli 100%) jest inwestowana w aktywa wolne od ryzyka:

```
In [6]: class observation_space:
            def __init__(self, n):
                self.shape = (n,)

In [7]: class action_space:
            def __init__(self, n):
                self.n = n

            def seed(self, seed):
                random.seed(seed)

            def sample(self):
                return random.random() ❶
```

❶ Losowy wybór działań (poziom inwestycji w akcje) z przedziału jednostkowego

Podobnie jak przy dynamicznym hedgingu środowisko `Investing` przyjmuje wiele parametrów używanych jako dane wejściowe do symulacji geometrycznych ruchów Browna. Zapisuje również saldo początkowe i dwie ostatnie wartości portfela:

```
In [8]: class Investing:
            def __init__(self, S0, T, r_, mu_, sigma_, steps, amount):
                self.initial_value = S0
                self.maturity = T
                self.short_rate_ = r_ ❶
                self.index_drift_ = mu_ ❶
                self.volatility_ = sigma_ ❶
                self.steps = steps
                self.initial_balance = amount ❷
                self.portfolio_value = amount ❸
                self.portfolio_value_new = amount ❹
                self.observation_space = observation_space(4)
                self.osn = self.observation_space.shape[0]
                self.action_space = action_space(1)
                self._generate_data()
                self.portfolios = pd.DataFrame()
                self.episode = 0
```

❶ Parametry można przekazywać jako iterowalne obiekty z wieloma wartościami

❷ Zapisywanie początkowej inwestycji

❸ Inicjalizowanie aktualnej wartości portfela

❹ Inicjalizowanie nowej wartości portfela

Kolejna metoda symuluje sekwencję wartości ryzykownego aktywa (`X`) i oblicza wartości aktywów wolnych od ryzyka (`Y`):

```
In [9]: class Investing(Investing):
            def _generate_data(self):
                s = [self.initial_value]
                self.short_rate = random.choice(self.short_rate_) ❶
                self.index_drift = random.choice(self.index_drift_) ❶
                self.volatility = random.choice(self.volatility_) ❶
                self.dt = self.maturity / self.steps
                for t in range(1, self.steps + 1):
                    st = s[t - 1] * math.exp(((self.index_drift -
                                self.volatility ** 2 / 2) * self.dt +
                                self.volatility * math.sqrt(
                                    self.dt) * random.gauss(0, 1))
                    ) ❷
                    s.append(st)
                self.data = pd.DataFrame(s, columns=['Xt'])
                self.data['Yt'] = self.initial_value * np.exp(
                    self.short_rate * np.arange(len(self.data)) * self.dt) ❸
```

❶ Losowy dobór wartości parametrów

❷ Symulowanie sekwencji wartości ryzykownego aktywa

❸ Obliczanie wartości aktywów wolnych od ryzyka

Następne metody wymagają jedynie niewielkich modyfikacji w porównaniu ze środowiskiem `Hedging`:

```
In [10]: class Investing(Investing):
             def _get_state(self):
                 Xt = self.data['Xt'].iloc[self.bar]
```

```
            Yt = self.data['Yt'].iloc[self.bar]
            return np.array([Xt, Yt, self.xt, self.yt]), {}

        def seed(self, seed=None):
            if seed is not None:
                random.seed(seed)

        def reset(self):
            self.bar = 0
            self.xt = 0
            self.yt = 0
            self.treward = 0
            self.portfolio_value = self.initial_balance
            self.portfolio_value_new = self.initial_balance
            self.episode += 1
            self._generate_data()
            self.state, _ = self._get_state()
            return self.state, _
```

Po dodaniu dwóch kolejnych metod klasa Pythona ze środowiskiem `Investing` będzie kompletna. Metoda `.add_results()` umożliwia zapisywanie potrzebnych punktów danych dla wszystkich epizodów i kroków. Upraszcza to dalsze analizy wyników po fazach uczenia się i testowania:

```
In [11]: class Investing(Investing):
        def add_results(self, pl):
            df = pd.DataFrame({'e': self.episode, 'xt': self.xt,
                       'yt': self.yt, 'pv': self.portfolio_value,
                       'pv_new': self.portfolio_value_new, 'p&l[$]': pl,
                       'p&l[%]': pl / self.portfolio_value_new,
                       'Xt': self.state[0], 'Yt': self.state[1],
                       'Xt_new': self.new_state[0],
                       'Yt_new': self.new_state[1],
                       'r': self.short_rate, 'mu': self.index_drift,
                       'sigma': self.volatility}, index=[0])
            self.portfolios = pd.concat((self.portfolios, df),
                                        ignore_index=True)

    def step(self, action):
        self.bar += 1
        self.new_state, _ = self._get_state()
        if self.bar == 1: ❶
            self.xt = action ❷
            self.yt = (1 - action) ❸
            pl = 0.
            reward = 0.
            self.add_results(pl)
        else:
            self.portfolio_value_new = (
                self.xt * self.portfolio_value *
                self.new_state[0] / self.state[0] +
                self.yt * self.portfolio_value *
                self.new_state[1] / self.state[1]) ❹
            pl = self.portfolio_value_new - self.portfolio_value ❺
            self.xt = action ❻
            self.yt = (1 - action) ❼
            self.add_results(pl) ❽
```

```
            reward = pl ❾
            self.portfolio_value = self.portfolio_value_new ❿
        if self.bar == len(self.data) - 1:
            done = True
        else:
            done = False
        self.state = self.new_state
        return self.state, reward, done, False, {}
```

❶ Początkowe działanie jest traktowane oddzielnie

❷ Określanie pozycji w ryzykownym aktywie

❸ Określanie pozycji dla aktywów wolnych od ryzyka

❹ Obliczanie nowej wartości portfela na podstawie wcześniejszej alokacji w aktywa

❺ Obliczanie zysku lub straty w wartościach bezwzględnych

❻ Aktualizowanie pozycji w ryzykownym aktywie

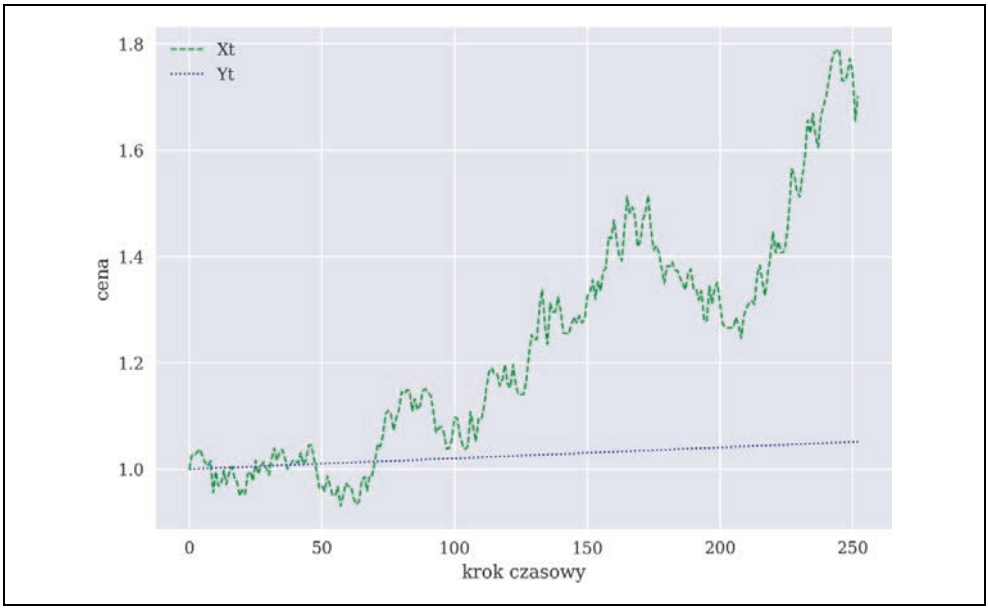
❼ Aktualizowanie pozycji dla aktywów wolnych od ryzyka

❽ Dodawanie wyników do obiektu `DataFrame`

❾ Określanie poziomu nagrody (zysk lub strata)

❿ Aktualizowanie wartości portfela

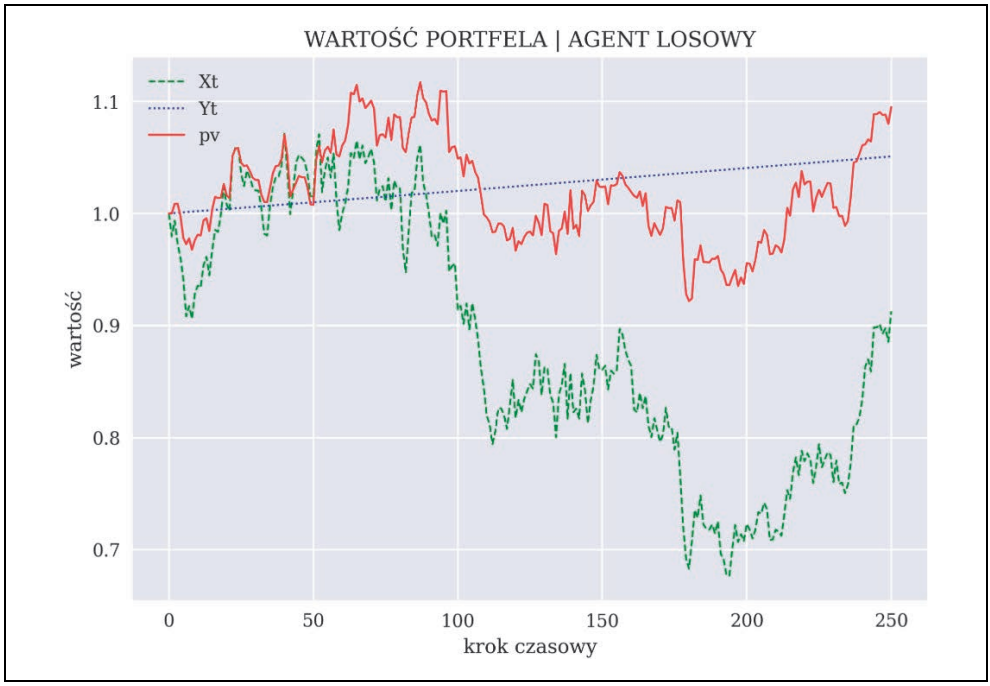
Teraz rozważmy następujące parametry środowiska, w tym stałą wartość ziarna dla generatora liczb losowych. Rysunek 8.2 przedstawia zmiany wartości dwóch aktywów. Tu dla obu aktywów wartość początkowa jest ustawiona na 1:



Rysunek 8.2. Wykres wartości aktywów ryzykownych i wolnych od ryzyka

```
In [12]: S0 = 1.

In [13]: investing = Investing(S0=S0, T=1.0, r_=[0.05], mu_=[0.3],
                               sigma_=[0.35], steps=252, amount=1)

In [14]: investing.seed(750)

In [15]: investing._generate_data()

In [16]: investing.data.plot(style=['g--', 'b:'], lw=1.0)
         plt.xlabel('krok czasowy')
         plt.ylabel('cena');
```

Następny fragment kodu w Pythonie umożliwia losowemu agentowi interakcję ze środowiskiem. Rysunek 8.3 przedstawia wartość portfela w porównaniu z wartościami aktywów wolnych od ryzyka i ryzykownych. Z uwagi na losową alokację stosowaną przez agenta i ogólnie ujemne wyniki dla ryzykownego aktywa strategia losowa w scenariuszu pokazanym na rysunku daje wyniki lepsze zarówno od inwestycji w aktywo wolne od ryzyka, jak i w aktywo ryzykowne:



Rysunek 8.3. Wartość portfela uzyskana przez agenta losowego

```
In [17]: investing.reset()
Out[17]: (array([1., 1., 0., 0.]), {})

In [18]: for _ in range(investing.steps - 1):
             investing.step(investing.action_space.sample())

In [19]: investing.portfolios.head().round(3)
```

```
Out[19]:    e     xt     yt     pv  pv_new  p&l[$]  p&l[%]     Xt     Yt  Xt_new \
         0  1  0.587  0.413  1.000   1.000   0.000   0.000  1.000  1.000   0.979
         1  1  0.001  0.999  1.000   1.009   0.009   0.008  0.979  1.000   0.994
         2  1  0.838  0.162  1.009   1.009   0.000   0.000  0.994  1.000   0.973
         3  1  0.981  0.019  1.009   0.998  -0.011  -0.011  0.973  1.001   0.961
         4  1  0.167  0.833  0.998   0.978  -0.020  -0.020  0.961  1.001   0.941

           Yt_new     r    mu  sigma
         0  1.000 0.050 0.300  0.350
         1  1.000 0.050 0.300  0.350
         2  1.001 0.050 0.300  0.350
         3  1.001 0.050 0.300  0.350
         4  1.001 0.050 0.300  0.350

In [20]: investing.portfolios[['Xt', 'Yt', 'pv']].plot(
             title='WARTOŚĆ PORTFELA | AGENT LOSOWY',
             style=['g--', 'b:', 'r-'], lw=1)
         plt.xlabel('krok czasowy')
         plt.ylabel('wartość');
```

Podobnie jak w poprzednim rozdziale klasa `InvestingAgent` dziedziczy po klasie `DQLAgent` przedstawionej w podrozdziale „Klasa DQLAgent”. Sieć neuronowa przyjmuje jako dane wejściowe cztery wartości reprezentujące stan środowiska i alokację w aktywa: wartość ryzykownego aktywa, wartość aktywa wolnego od ryzyka, pozycję w ryzykownym aktywie i pozycję w aktywie wolnym od ryzyka. Jako wynik zwraca pojedynczą liczbę zmiennoprzecinkową. Wynik ten reprezentuje oczekiwaną nagrodę dla danej kombinacji stanu środowiska i alokacji w aktywa:

```
In [21]: from dqlagent import *

In [22]: opt = keras.optimizers.legacy.Adam

In [23]: class InvestingAgent(DQLAgent):
             def _create_model(self, hu, lr):
                 self.model = Sequential()
                 self.model.add(Dense(hu, input_dim=self.n_features,
                                      activation='relu'))
                 self.model.add(Dense(hu, activation='relu'))
                 self.model.add(Dense(1, activation='linear')) ❶
                 self.model.compile(loss='mse',
                         optimizer=opt(learning_rate=lr))
```

❶ Liniowe dane wyjściowe w postaci liczby zmiennoprzecinkowej

Podobnie jak w hedgingu dynamicznym optymalne działanie jest ustalane na podstawie optymalizacji numerycznej. Metoda `.opt_action()` określa dla ryzykownego aktywa alokację dającą maksymalną oczekiwaną nagrodę. Alokacja w aktywa wolne od ryzyka wynika z definicji:

```
In [24]: from scipy.optimize import minimize

In [25]: class InvestingAgent(InvestingAgent):
             def opt_action(self, state):
                 bnds = [(0, 1)] ❶
                 def f(state, x): ❷
                     s = state.copy()
                     s[0, self.xp] = x ❸
```

```
                    s[0, self.yp] = 1 - x ❹
                    return self.model.predict(s)[0, 0] ❺
                action = minimize(lambda x: -f(state, x), 0.5,
                                  bounds=bnds, method='Nelder-Mead',
                                  )['x'][0] ❻
                return action

    def act(self, state):
        if random.random() <= self.epsilon:
            return self.env.action_space.sample()
        action = self.opt_action(state) ❼
        return action
```

❶ Granice alokacji w ryzykowne aktywa

❷ Funkcja `f()`, której wartość ma zostać zmaksymalizowana

❸ Ustawianie alokacji w ryzykowne aktywa na wartość wejściową `x`

❹ Ustawianie alokacji w aktywa wolne od ryzyka na `1 - x`

❺ Prognozowanie oczekiwanej nagrody za pomocą sieci neuronowej

❻ Maksymalizowanie oczekiwanej nagrody przez minimalizację wartości wyrażenia `-f()`

❼ Wywołanie metody `.opt_action()`

Podobnie metoda `.replay()` prognozuje oczekiwaną przyszłą nagrodę na podstawie alokacji środków w ryzykowne aktywo:

```
In [26]: class InvestingAgent(InvestingAgent):
             def replay(self):
                 batch = random.sample(self.memory, self.batch_size)
                 for state, action, next_state, reward, done in batch:
                     ns = next_state.copy()
                     target = reward
                     if not done:
                         action = self.opt_action(ns) ❶
                         ns[0, self.xp] = action ❷
                         ns[0, self.yp] = 1 - action ❸
                         target += (self.gamma *
                             self.model.predict(ns)[0, 0]) ❹
                     self.model.fit(state, np.array([target]),
                                    epochs=1, verbose=False)
                 if self.epsilon > self.epsilon_min:
                     self.epsilon *= self.epsilon_decay
```

❶ Wybór optymalnego działania (poziomu alokacji w ryzykowne aktywo)

❷ Aktualizowanie alokacji w ryzykowne aktywo

❸ Aktualizowanie alokacji w aktywa wolne od ryzyka

❹ Obliczanie i dodawanie zdyskontowanej odroczonej nagrody

W poniższym kodzie w Pythonie zmodyfikowałem metodę `.test()`, aby uwzględnić nową konfigurację. Główną zmianą jest wywołanie metody `.opt_action()` w celu pobrania optymalnego poziomu alokacji w ryzykowne aktywo:

```
In [27]: class InvestingAgent(InvestingAgent):
             def test(self, episodes, verbose=True):
                 for e in range(1, episodes + 1):
                     state, _ = self.env.reset()
                     state = self._reshape(state)
                     treward = 0
                     for _ in range(1, len(self.env.data) + 1):
                         action = self.opt_action(state)
                         state, reward, done, trunc, _ = self.env.step(action)
                         state = self._reshape(state)
                         treward += reward
                         if done:
                             templ = f'epizod={e} | '
                             templ += f'łączna nagroda={treward:4.2f}'
                             if verbose:
                                 print(templ, end='\r')
                             break
```

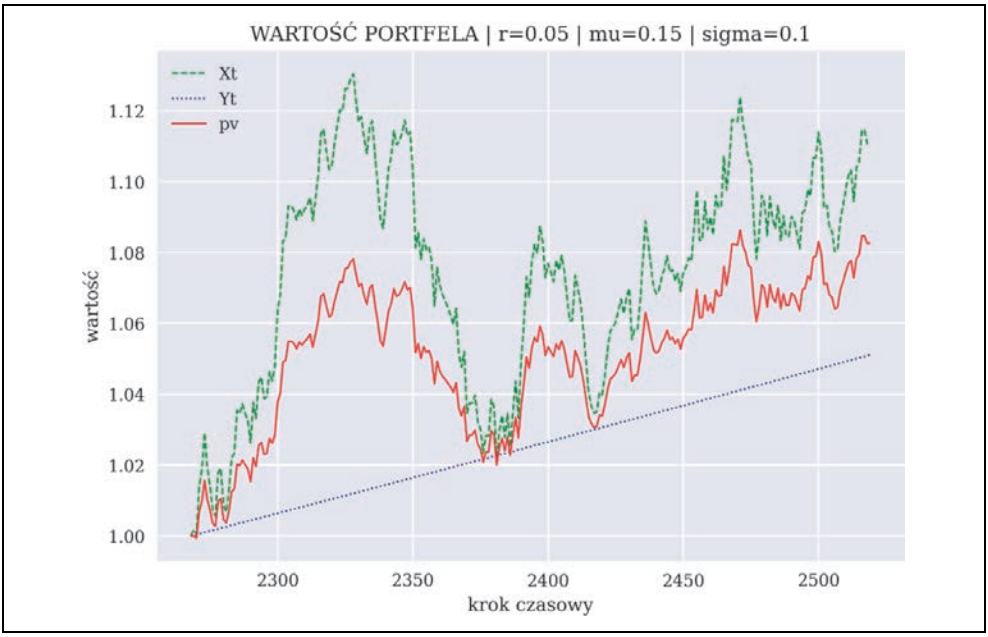
Rozważmy teraz środowisko `Investing` zainicjalizowane kilkoma wartościami: stopą krótkoterminową, oczekiwaną stopą zwrotu (dryf) i zmiennością ryzykownego aktywa. Agent `InvestingAgent` uczy się na podstawie wielu symulacji z losowo wybranymi kombinacjami parametrów:

```
In [28]: def set_seeds(seed=500):
             random.seed(seed)
             np.random.seed(seed)
             tf.random.set_seed(seed)

In [29]: set_seeds()

In [30]: investing = Investing(S0=S0, T=1.0, r_=[0, 0.025, 0.05],
                               mu_=[0.05, 0.1, 0.15],
                               sigma_=[0.1, 0.2, 0.3], steps=252, amount=1)

In [31]: agent = InvestingAgent('2FS', feature=None, n_features=4,
                                env=investing, hu=128, lr=0.00025)

In [32]: agent.xp = 2 ❶
         agent.yp = 3 ❷

In [33]: episodes = 64

In [34]: %time agent.learn(episodes)
         epizod= 64 | łączna nagroda= 0.272 | maks.= 0.326
         CPU times: user 29.9 s, sys: 4.6 s, total: 34.5 s
         Wall time: 29.5 s

In [35]: agent.epsilon
Out[35]: 0.8519730927255319
```

❶ Określanie pozycji w indeksie dla ryzykownego aktywa

❷ Określanie pozycji w indeksie dla aktywów wolnych od ryzyka

Następnie agent jest testowany w kilku przebiegach. Rysunek 8.4 przedstawia zmiany wartości portfela zgodnie z alokacją w aktywa wybraną przez agenta z jednego przebiegu testowego:



Rysunek 8.4. Wartości portfela uzyskane przez agenta InvestingAgent

```
In [36]: agent.env.portfolios = pd.DataFrame()

In [37]: %time agent.test(10)
         CPU times: user 20.3 s, sys: 3.13 s, total: 23.4 s
         Wall time: 19.9 s

In [38]: n = max(agent.env.portfolios['e']) ❶

In [39]: res = agent.env.portfolios[agent.env.portfolios['e'] == n]
         res.head()

Out[39]:        e    xt    yt    pv  pv_new  p&l[$]  p&l[%]    Xt    Yt  Xt_new
         2268  74 0.564 0.436  1.000  1.000   0.000   0.000 1.000 1.000   1.002
         2269  74 0.565 0.435  1.000  0.999  -0.001  -0.001 1.002 1.000   1.001
         2270  74 0.564 0.436  0.999  1.007   0.008   0.007 1.001 1.000   1.014
         2271  74 0.570 0.430  1.007  1.010   0.003   0.003 1.014 1.001   1.019
         2272  74 0.572 0.428  1.010  1.016   0.006   0.006 1.019 1.001   1.029

               Yt_new     r    mu  sigma
         2268   1.000 0.050 0.150  0.100
         2269   1.000 0.050 0.150  0.100
         2270   1.001 0.050 0.150  0.100
         2271   1.001 0.050 0.150  0.100
         2272   1.001 0.050 0.150  0.100

In [40]: p = res.iloc[0][['r', 'mu', 'sigma']]

In [41]: t = f"r={p['r']} | mu={p['mu']} | sigma={p['sigma']}"

In [42]: res[['Xt', 'Yt', 'pv']].plot(
             title='WARTOŚĆ PORTFELA | ' + t,
```

```
            style=['g--', 'b:', 'r-'], lw=1)
        plt.xlabel('krok czasowy')
        plt.ylabel('wartość');
```

❶ Wybieranie ostatecznego przebiegu testowego

Warto przyjrzeć się wybranym statystykom w tym kontekście. W tym konkretnym przebiegu testowym agent prawie dokładnie stosuje dynamiczną strategię 60/40 (rysunek 8.5). Choć stopa zwrotu dla strategii agenta znajduje się pomiędzy stopami zwrotu dla aktywów wolnych od ryzyka i aktywów ryzykownych, współczynnik Sharpe'a uzyskany przez agenta 60/40 jest wyższy niż dla aktywów ryzykownych[1]:

```
In [43]: rets = res[['Xt', 'Yt', 'pv']].pct_change(
            ).mean() / agent.env.dt ❶
         rets
Out[43]: Xt   0.110
         Yt   0.050
         pv   0.081
         dtype: float64

In [44]: stds = res[['Xt', 'Yt', 'pv']].pct_change(
             ).std() / math.sqrt(agent.env.dt) ❷
         stds
Out[44]: Xt   0.102
         Yt   0.000
         pv   0.060
         dtype: float64

In [45]: rets[['Xt', 'pv']] / stds[['Xt', 'pv']] ❸
Out[45]: Xt   1.079
         pv   1.365
         dtype: float64

In [46]: res['xt'].mean() ❹
Out[46]: 0.5845191592261907

In [47]: res['xt'].std() ❺
Out[47]: 0.010688881672664631

In [48]: res['xt'].plot(title='ALOKACJA W RYZYKOWNE AKTYWO | ' + t,
                        lw=1.0, c='b')
         plt.ylim(res['xt'].min() - 0.1, res['xt'].max() + 0.1)
         plt.xlabel('krok czasowy');
```

❶ Obliczanie średnich stóp zwrotu w ujęciu rocznym

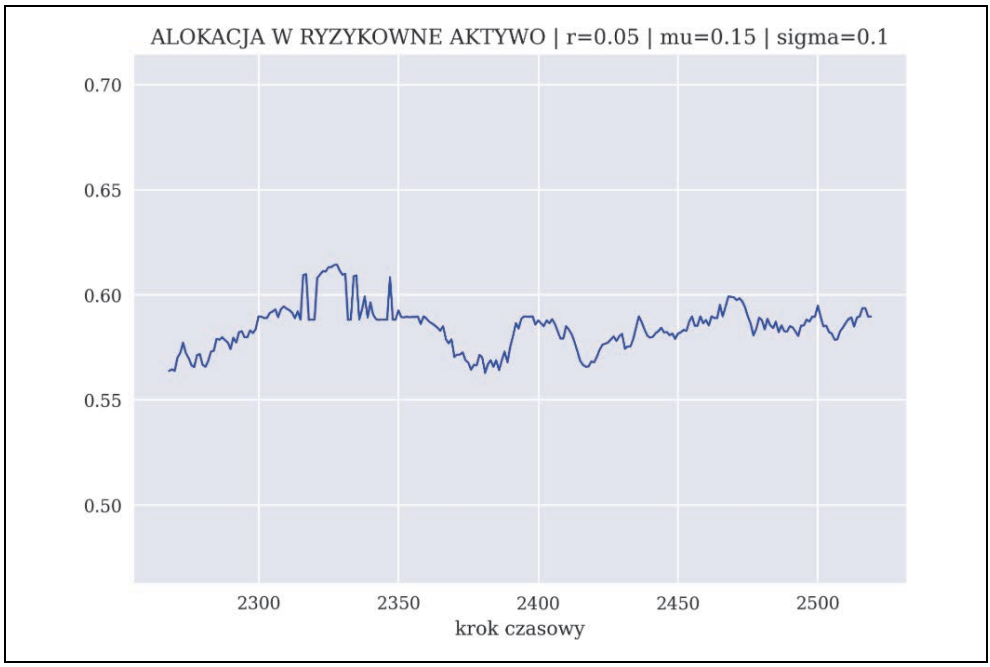
❷ Obliczanie zmienności w ujęciu rocznym

❸ Obliczanie współczynników Sharpe'a

❹ Średnia alokacja w ryzykowne aktywa

❺ Odchylenie standardowe dla tej alokacji

[1] Przy zerowym ryzyku aktywów wolnych od ryzyka ich współczynnik Sharpe'a nie jest zdefiniowany (jest nieskończony). Z tego powodu tego współczynnika nie można uwzględniać jako nagrody, ponieważ agent nauczyłby się przede wszystkim (wyłącznie) inwestować w aktywa wolne od ryzyka, aby uzyskać wysokie (nieskończone) nagrody.



Rysunek 8.5. Dynamiczna alokacja w ryzykowne aktywa (w procentach)

Poniżej przedstawiam kilka statystyk związanych z alokacją w ryzykowne aktywa (`xt`). Niezależnie od parametrów dryfu i ryzyka alokacja w ryzykowne aktywa wynosi średnio około 55%, z maksymalną wartością około 66%:

```
In [49]: agent.env.portfolios.groupby('mu')['xt'].describe()
Out[49]:          count  mean   std   min   25%   50%   75%   max
         mu
         0.050  504.000 0.561 0.040 0.392 0.558 0.565 0.577 0.633
         0.100 1008.000 0.547 0.088 0.394 0.419 0.583 0.615 0.661
         0.150 1008.000 0.561 0.054 0.390 0.555 0.572 0.588 0.635

In [50]: agent.env.portfolios.groupby('sigma')['xt'].describe()
Out[50]:          count  mean   std   min   25%   50%   75%   max
         sigma
         0.100 1260.000 0.593 0.026 0.550 0.574 0.588 0.614 0.659
         0.200  756.000 0.540 0.060 0.390 0.547 0.559 0.570 0.633
         0.300  504.000 0.484 0.083 0.394 0.406 0.419 0.557 0.661
```

Poniższe dane przedstawiają te same statystyki dotyczące wartości portfela na przestrzeni czasu. Poza scenariuszem z najwyższym poziomem ryzyka portfele mają średnią powyżej 1. Na ogólnym poziomie średnie wyniki dla różnych wartości parametrów są dość zbliżone:

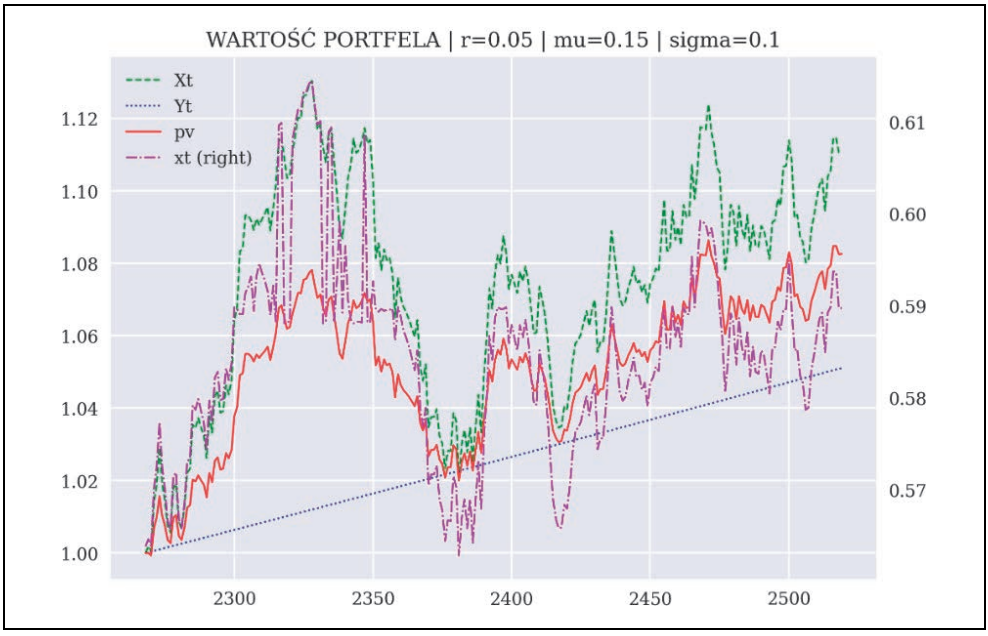
```
In [51]: agent.env.portfolios.groupby('mu')['pv_new'].describe()
Out[51]:          count  mean   std   min   25%   50%   75%   max
         mu
         0.050  504.000 1.016 0.033 0.948 0.994 1.010 1.036 1.114
         0.100 1008.000 1.013 0.087 0.846 0.929 1.025 1.078 1.196
         0.150 1008.000 1.022 0.037 0.926 0.997 1.018 1.055 1.099
```

```
In [52]: agent.env.portfolios.groupby('sigma')['pv_new'].describe()
Out[52]:          count  mean   std   min   25%   50%   75%   max
         sigma
         0.100 1260.000 1.054 0.044 0.986 1.019 1.046 1.076 1.196
         0.200  756.000 1.003 0.034 0.926 0.980 0.999 1.021 1.114
         0.300  504.000 0.947 0.061 0.846 0.904 0.929 0.980 1.138
```

Na zakończenie tego fragmentu przedstawiam kolejne analizy przebiegów testowych, które pomagają zrozumieć działanie agenta. Agent zwiększa ekspozycję na ryzykowne aktywa w sytuacji, gdy ich cena rośnie. Odwrotnie postępuje w momencie, kiedy ich cena spada. Jednak poziom alokacji w ryzykowne aktywa przez cały czas pozostaje między 55% a około 60%. Można to nazwać strategią *dodatniego sprzężenia zwrotnego* (rysunek 8.6). Agent osiąga wynik znacznie powyżej stopy zwrotu wolnej od ryzyka, ale poniżej stopy zwrotu dla ryzykownego aktywa:

```
In [53]: n = max(agent.env.portfolios['e']) ❶

In [54]: res = agent.env.portfolios[agent.env.portfolios['e'] == n]

In [55]: p = res.iloc[0][['r', 'mu', 'sigma']]

In [56]: t = f"r={p['r']} | mu={p['mu']} | sigma={p['sigma']}"

In [57]: ax = res[['Xt', 'Yt', 'pv', 'xt']].plot(
             title='WARTOŚĆ PORTFELA | ' + t,
             style=['g--', 'b:', 'r-', 'm-.'], lw=1,
             secondary_y='xt'
         )
```

❶ Wybór przebiegu testowego



Rysunek 8.6. Wartości portfela i poziom dynamicznej alokacji w ryzykowne aktywa

Scenariusz z dwoma aktywami

Analizy z poprzedniego podrozdziału można łatwo zmodyfikować, aby uwzględnić *dwa ryzykowne aktywa*. W tym podrozdziale wykorzystuję rzeczywiste dane historyczne dla wielu różnych instrumentów finansowych. Analizy dotyczą danych z indeksu giełdowego S&P 500 i indeksu zmienności VIX. Wiadomo, że w szeregach czasowych z poziomami indeksów występują silne ujemne korelacje. Wiadomo również, że strategie inwestycyjne, które utrzymują stały odsetek dwóch aktywów na przestrzeni czasu, przynoszą lepsze zwroty w porównaniu z innymi strategiami inwestycyjnymi obejmującymi te dwa aktywa. Takie strategie nazywane są *strategiami inwestycyjnymi o stałej proporcji*. Wykorzystuje się w nich dynamiczne równoważenie portfela, aby utrzymać proporcje inwestycji w każdy instrument na mniej więcej tym samym poziomie, na przykład 60% w S&P 500 i 40% w VIX[2].

Implementacja strategii

W tym podrozdziale przyjmuję upraszczające założenie, że zarówno S&P 500, jak i VIX są aktywami, którymi można handlować. W praktyce tak nie jest. Potrzebne są inne instrumenty finansowe, bazujące na takich indeksach. Jako zamiennik indeksu giełdowego możesz kupić na przykład jednostki funduszu ETF opartego na indeksie S&P 500. Można również wykorzystać kontrakty futures lub opcje na indeks. Podobnie można handlować kontraktami futures i opcjami na indeks VIX jako zastępnikiem indeksu zmienności. Gdy korzystasz z kontraktów futures i opcji, w implementacji należy uwzględnić szereg kwestii, na przykład rolowanie pozycji w instrumentach pochodnych. W tym podrozdziale nie poruszam tych zagadnień. Ponadto przyjmuję też inne upraszczające założenia, na przykład dotyczące zerowych kosztów transakcyjnych.

Chociaż istnieje szereg modyfikacji, które należy wprowadzić w środowisku `Investing` z poprzedniego podrozdziału, wszystkie niezbędne zmiany są proste i powinny być łatwe do zrozumienia. Nowa klasa `Investing` pozwala na wybór dwóch ryzykownych aktywów. W tym celu z oryginalnego zestawu danych wybierany jest losowy ciągły podzbiór. Zestaw danych jest tu taki sam jak używany w rozdziale 3. dla klasy `Finance`:

```
In [58]: class Investing(Investing):
             def __init__(self, asset_one='.SPX', asset_two='.VIX',
                          steps=252, amount=1):
                 self.asset_one = asset_one
                 self.asset_two = asset_two
                 self.steps = steps
                 self.initial_balance = amount
                 self.portfolio_value = amount
                 self.portfolio_value_new = amount
                 self.observation_space = observation_space(5)
                 self.osn = self.observation_space.shape[0]
                 self.action_space = action_space(1)
```

[2] Więcej szczegółów i przykładowe obliczenia wykonane w Pythonie znajdziesz w mojej pracy: Hilpisch, 2017, rozdział 4. W zamieszczonych w niej przykładach wykorzystuję indeks EURO STOXX zamiast S&P 500 oraz indeks zmienności VSTOXX zamiast VIX.

```
        self.retrieved = False
        self._generate_data()
        self.portfolios = pd.DataFrame()
        self.episode = 0

    def _generate_data(self):
        if self.retrieved:
            pass
        else:
            url = 'https://certificate.tpq.io/rl4finance.csv' ❶
            self.raw = pd.read_csv(url, index_col=0,
                                   parse_dates=True).dropna() ❶
            self.retrieved = True
        self.data = pd.DataFrame()
        self.data['Xt'] = self.raw[self.asset_one]
        self.data['Yt'] = self.raw[self.asset_two]
        s = random.randint(self.steps, len(self.data)) ❷
        self.data = self.data.iloc[s-self.steps:s] ❸
        self.data = self.data / self.data.iloc[0] ❹
```

❶ Pobieranie danych historycznych z cenami zamknięcia

❷ Generowanie losowej liczby całkowitej w celu wyboru podzbioru danych

❸ Wybieranie losowego podzbioru z oryginalnych danych

❹ Normalizowanie danych przez zastosowanie wartości początkowej 1

Dwie następne metody ilustrują zmiany potrzebne do uwzględnienia *daty* w stanie:

```
In [59]: class Investing(Investing):
             def _get_state(self):
                 Xt = self.data['Xt'].iloc[self.bar]
                 Yt = self.data['Yt'].iloc[self.bar]
                 self.date = self.data.index[self.bar] ❶
                 return np.array([Xt, Yt, Xt - Yt, self.xt, self.yt]), {} ❷

             def add_results(self, pl):
                 df = pd.DataFrame({
                        'e': self.episode, 'date': self.date, ❸
                        'xt': self.xt, 'yt': self.yt,
                        'pv': self.portfolio_value,
                        'pv_new': self.portfolio_value_new, 'p&l[$]': pl,
                        'p&l[%]': pl / self.portfolio_value_new * 100,
                        'Xt': self.state[0], 'Yt': self.state[1],
                        'Xt_new': self.new_state[0],
                        'Yt_new': self.new_state[1],
                              }, index=[0])
                 self.portfolios = pd.concat((self.portfolios, df),
                                             ignore_index=True)
```

❶ Zapisywanie daty dla stanu w atrybucie instancji

❷ Dodawanie różnicy w cenach aktywów do zbioru zmiennych stanu

❸ Zapisywanie daty dla stanu w obiekcie `DataFrame`

Jedna z głównych zmian dotyczy nagrody otrzymywanej przez agenta. Zamiast zwracać bezwzględny zysk, nowe środowisko `Investing` oblicza nagrodę bazującą na **współczynniku Sharpe'a**. Współczynnik ten jest obliczany jako zrealizowany annualizowany zwrot podzielony przez annualizowaną zmienność kroczącą w określonym oknie czasowym. Jeśli nie wprowadzisz dodatkowych zmian, agent wymyśli strategie inwestycyjne z wysoce zmiennym poziomem alokacji środków w dwa ryzykowne aktywa. Zwykle nie jest to pożądane, ponieważ w praktyce prowadzi między innymi do wysokich kosztów transakcyjnych. W związku z tym od uzyskanego współczynnika Sharpe'a odejmowana jest kara za odchylenia od wcześniejszych poziomów alokacji[3]. Ma to motywować agenta do preferowania mniejszych zmian w poziomach alokacji, a także wprowadza formę *regularyzacji* w proces alokacji:

```
In [60]: class Investing(Investing):
             def step(self, action):
                 self.bar += 1
                 self.new_state, info = self._get_state()
                 if self.bar == 1:
                     self.xt = action
                     self.yt = (1 - action)
                     pl = 0.
                     reward = 0.
                     self.add_results(pl)
                 else:
                     self.portfolio_value_new = (
                         self.xt * self.portfolio_value *
                         self.new_state[0] / self.state[0] +
                         self.yt * self.portfolio_value *
                         self.new_state[1] / self.state[1])
                     pl = self.portfolio_value_new - self.portfolio_value
                     pen = (self.xt - action) ** 2 ❶
                     self.xt = action
                     self.yt = (1 - action)
                     self.add_results(pl)
                     ret = self.portfolios['p&l[%]'].iloc[-1] / 100 * 252 ❷
                     vol = self.portfolios['p&l[%]'].rolling(
                         20, min_periods=1).std().iloc[-1] * math.sqrt(252) ❸
                     sharpe = ret / vol ❹
                     reward = sharpe - pen ❺
                     self.portfolio_value = self.portfolio_value_new
                 if self.bar == len(self.data) - 1:
                     done = True
                 else:
                     done = False
                 self.state = self.new_state
                 return self.state, reward, done, False, {}
```

❶ Kara równa kwadratowi różnicy między poprzednią a nową alokacją w pierwsze ryzykowne aktywo

❷ Zrealizowane zannualizowane zyski liczone względem poprzedniego stanu

❸ Zannualizowana zmienność krocząca dla stałego okna czasowego kończącego się datą nowego stanu

[3] W bardziej ogólnym scenariuszu kara może uwzględniać także koszty transakcyjne, wpływ na rynek lub inne komponenty mikrostruktury rynkowej.

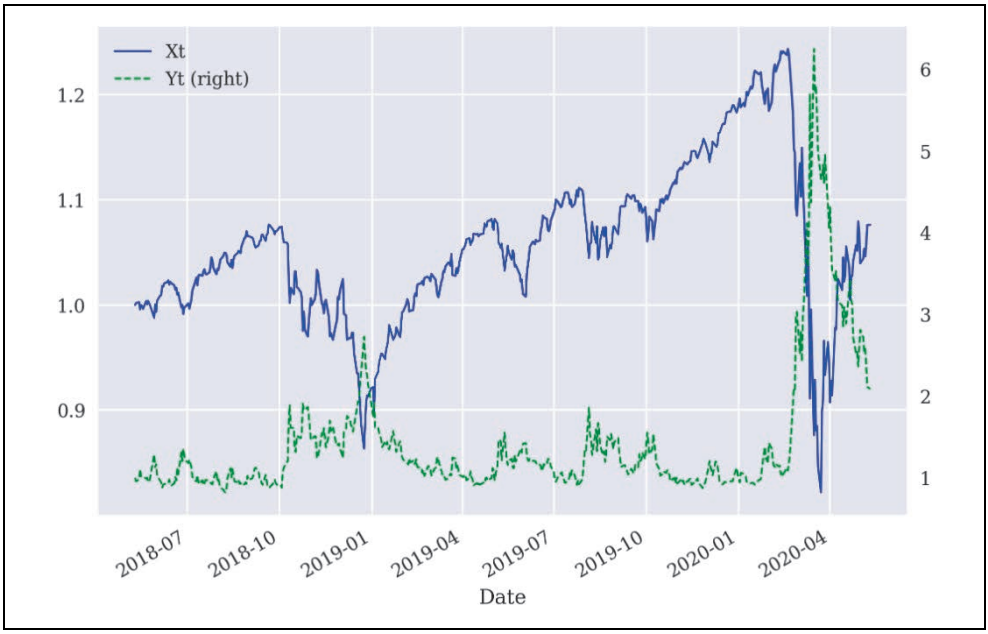
❹ Współczynnik Sharpe'a liczony względem poprzedniego stanu

❺ Nagroda mierzona jako różnica między współczynnikiem Sharpe'a a karą.

Poniższy kod w Pythonie tworzy obiekt środowiska i generuje wykres z losowo wybranym znormalizowanym podzbiorem szeregów czasowych dla indeksów S&P 500 i VIX. Rysunek 8.7 dobrze obrazuje wysoką ujemną korelację między tymi dwoma szeregami:

```
In [61]: days = 2 * 252

In [62]: investing = Investing(steps=days)

In [63]: investing.data.head()
Out[63]:                Xt    Yt
         Date
         2018-05-10 1.000 1.000
         2018-05-11 1.002 0.956
         2018-05-14 1.003 0.977
         2018-05-15 0.996 1.106
         2018-05-16 1.000 1.014

In [64]: investing.data.corr() ❶
Out[64]:        Xt     Yt
         Xt  1.000 -0.457
         Yt -0.457  1.000

In [65]: investing.data.plot(secondary_y='Yt',
                             style=['b', 'g--'], lw=1);
```

❶ Obliczanie korelacji między dwoma używanymi szeregami czasowymi



Rysunek 8.7. Znormalizowane poziomy indeksów S&P 500 i VIX

Klasa `InvestingAgent` nie wymaga żadnych zmian. Poniższy kod uczy agenta z wykorzystaniem nowego środowiska `Investing`:

```
In [66]: set_seeds()

In [67]: investing = Investing(steps=days)

In [68]: agent = InvestingAgent('2AC', feature=None, n_features=5,
                                env=investing, hu=48, lr=0.0005)

In [69]: agent.xp = 3 ❶
         agent.yp = 4 ❷

In [70]: episodes = 250

In [71]: %time agent.learn(episodes)
         epizod= 250 | łączna nagroda=-42.749 | maks.=-38.6463
         CPU times: user 8min 36s, sys: 1min 46s, total: 10min 22s
         Wall time: 9min 27s

In [72]: agent.epsilon
Out[72]: 0.5348427211156283
```

❶ Określanie pozycji w indeksie dla pierwszego ryzykownego aktywa

❷ Określanie pozycji w indeksie dla drugiego ryzykownego aktywa

Następny fragment kodu w Pythonie wykonuje kilka przebiegów testowych. Generuje też ogólne statystyki dotyczące alokacji w pierwsze ryzykowne aktywo:

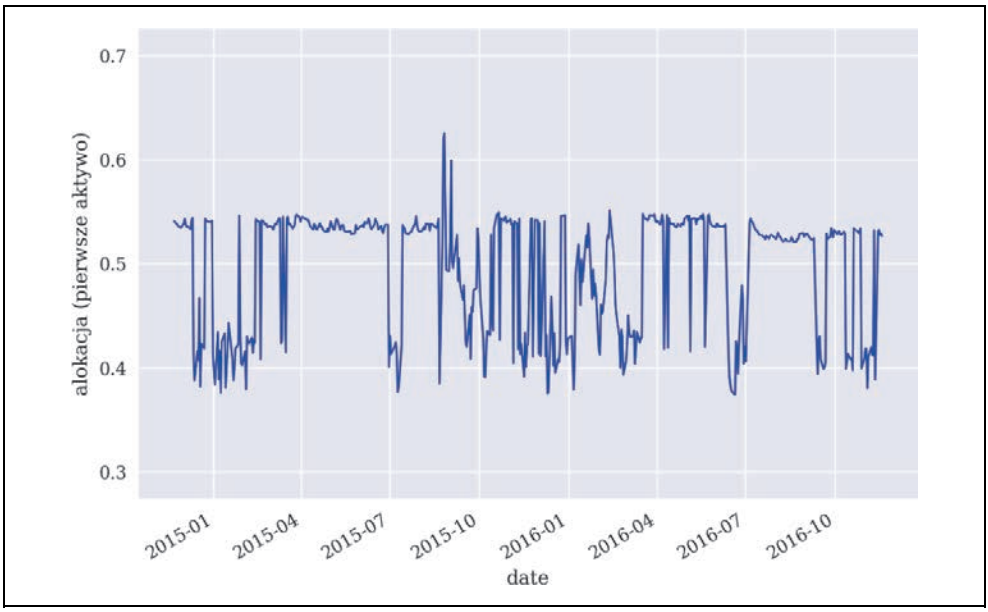
```
In [73]: agent.env.portfolios = pd.DataFrame()

In [74]: %time agent.test(10)
         CPU times: user 42.8 s, sys: 5.84 s, total: 48.7 s
         Wall time: 42 s

In [75]: agent.env.portfolios['xt'].describe()
Out[75]: count 5030.000
         mean     0.433
         std      0.084
         min      0.000
         25%      0.389
         50%      0.428
         75%      0.498
         max      0.676
         Name: xt, dtype: float64
```

Dokładna analiza konkretnego przebiegu testowego pomoże lepiej zrozumieć strategię inwestycyjną agenta. W tym przykładzie agent przez cały czas trwania inwestycji utrzymuje alokację na stosunkowo stałym poziomie, co ilustruje rysunek 8.8.

Jednak zdarzają się też duże operacje rebalancingu, zależne od względnych zwrotów dla dwóch ryzykownych aktywów:

```
In [76]: n = max(agent.env.portfolios['e']) - 3

In [77]: res = agent.env.portfolios[
                agent.env.portfolios['e'] == n].set_index('date')
```

```
In [78]: res['xt'].plot(lw=1, c='b')
         plt.ylim(res['xt'].min() - 0.1, res['xt'].max() + 0.1)
         plt.ylabel('alokacja (pierwsze aktywo)');
```

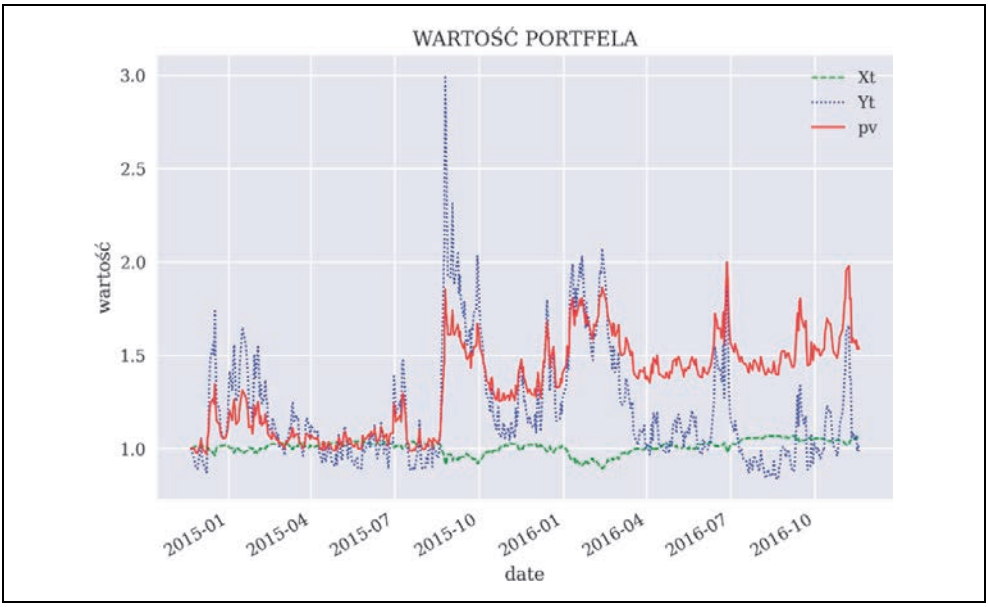


Rysunek 8.8. Alokacja w pierwsze ryzykowne aktywo

W tym konkretnym scenariuszu strategia agenta nie tylko daje wynik znacznie lepszy niż inwestycja w poszczególne ryzykowne aktywa, ale także osiąga najwyższy współczynnik Sharpe'a. Rysunek 8.9 przedstawia wyniki strategii agenta w porównaniu z inwestycją w poszczególne ryzykowne aktywa:

```
In [79]: res[['Xt', 'Yt', 'pv']].iloc[-1]
Out[79]: Xt   1.065
         Yt   0.983
         pv   2.022
         Name: 2016-11-18 00:00:00, dtype: float64

In [80]: r = np.log(res[['Xt', 'Yt', 'pv']] /
                    res[['Xt', 'Yt', 'pv']].shift(1))

In [81]: rets = np.exp(r.mean() * 252) - 1
         rets
Out[81]: Xt    0.032
         Yt   -0.009
         pv    0.424
         dtype: float64

In [82]: stds = r.std() * math.sqrt(252)
         stds
Out[82]: Xt    0.146
         Yt    1.338
         pv    0.670
         dtype: float64
```

```
In [83]: rets / stds
Out[83]: Xt    0.221
         Yt   -0.006
         pv    0.633
         dtype: float64

In [84]: res[['Xt', 'Yt', 'pv']].plot(
             title='WARTOŚĆ PORTFELA',
             style=['g--', 'b:', 'r-'],
             lw=1, grid=True)
         plt.ylabel('wartość');
```



Rysunek 8.9. Ceny aktywów i wartość portfela na przestrzeni czasu

We wszystkich przebiegach testowych strategia agenta daje w horyzoncie inwestycyjnym wyniki lepsze niż zakup poszczególnych aktywów:

```
In [85]: values = agent.env.portfolios.groupby('e')[
                  ['Xt', 'Yt', 'pv_new']].last()
         values.tail()

Out[85]:        Xt    Yt  pv_new
         e
         256 1.285 1.067   1.998
         257 1.065 0.983   1.971
         258 1.301 1.138   2.558
         259 1.196 1.103   2.175
         260 1.389 1.373   2.672

In [86]: values.mean()
Out[86]: Xt       1.233
         Yt       1.077
         pv_new   2.187
         dtype: float64
```

```
In [87]: ((values['pv_new'] > values['Xt']) &
          (values['pv_new'] > values['Yt'])).value_counts()
Out[87]: True 10
         Name: count, dtype: int64
```

Scenariusz z trzema aktywami

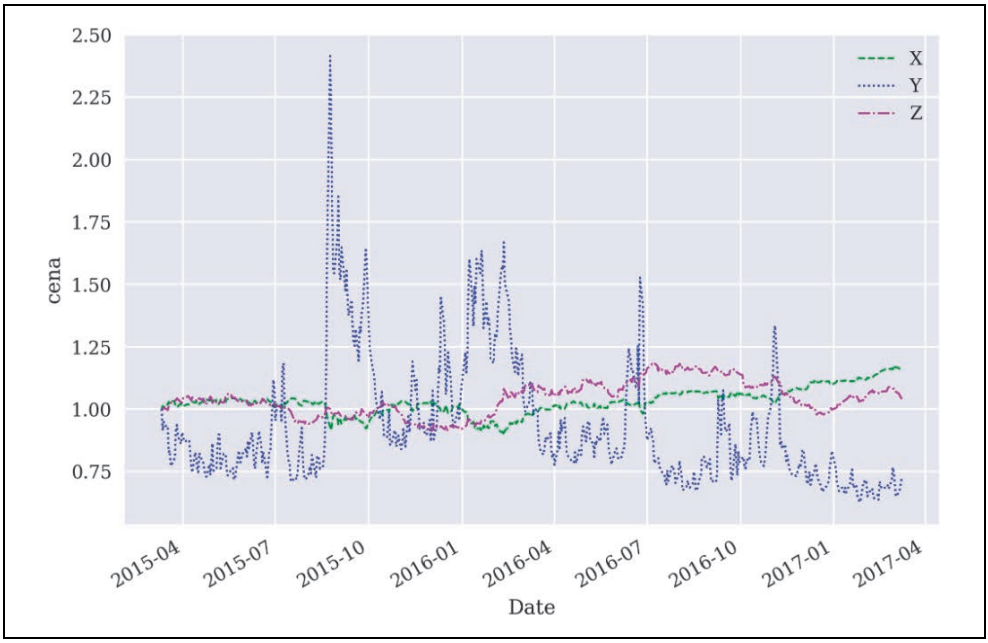
W tym podrozdziale omawiam inwestycję z *trzema ryzykownymi aktywami*. Jest to przykład, który został już przeanalizowany przez Markowitza (1952) w warunkach statycznych, czyli z tylko dwoma punktami w czasie. Podobnie jak wcześniej w tym podrozdziale stosuję podejście dynamiczne bazujące na danych historycznych. W każdym epizodzie na etapie uczenia i testowania wybierana jest losowa ciągła próbka tych danych.

Kod z tego podrozdziału jest przedstawiony w formie skryptu Pythona w podrozdziale „Kod dla scenariusza z trzema aktywami" na końcu tego rozdziału. W pewnym sensie kod ten stanowi podsumowanie fragmentów z poprzednich dwóch podrozdziałów. Oczywiście zawiera również modyfikacje niezbędne w celu uwzględnienia dodatkowego aktywa. Na podstawie tego kodu dalsze uogólnienie dla $n > 3$ aktywów jest stosunkowo proste.

Gdy dostępny jest już kod w Pythonie z podrozdziału „Kod dla scenariusza z trzema aktywami", uruchomienie go jest proste. Wystarczy uruchomić skrypt:

```
In [1]: %run assetallocation.py
```

Aby utworzyć instancję środowiska `Investing`, trzeba podać symbole trzech aktywów. Rysunek 8.10 przedstawia losowo wybrany podzbiór danych z szeregów czasowych dla użytych symboli:



Rysunek 8.10. Losowe ciągłe próbki z cenami trzech ryzykownych aktywów

```
In [2]: days = 2 * 252

In [3]: random.seed(100)

In [4]: # 1 = X, 2 = Y, 3 = Z
        investing = Investing('.SPX', '.VIX', 'XAU=', steps=days)

In [5]: investing.data.plot(lw=1, style=['g--', 'b:', 'm-.'])
        plt.ylabel('cena');
```

Następny fragment kodu w Pythonie przeprowadza etap uczenia agenta `InvestingAgent`:

```
In [6]: random.seed(100)
        np.random.seed(100)
        tf.random.set_seed(100)

In [7]: agent = InvestingAgent('3AC', feature=None, n_features=6,
                               env=investing, hu=128, lr=0.00025)

In [8]: episodes = 64

In [9]: %time agent.learn(episodes)
        epizod= 64 | łączna nagroda= 2.201 | maks.= 7.745
        CPU times: user 1min 7s, sys: 9.85 s, total: 1min 17s
        Wall time: 1min 19s

In [10]: agent.epsilon
Out[10]: 0.8519730927255319
```

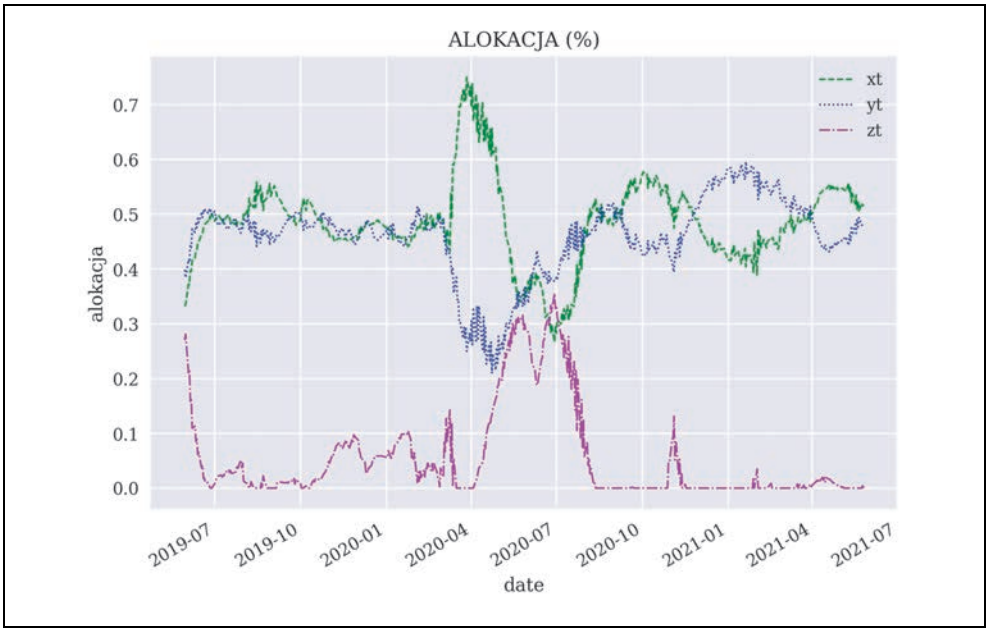
W przebiegach testowych agent `InvestingAgent` uzyskuje średnią ostateczną wartość portfela znacznie przewyższającą wartość końcową któregokolwiek z trzech ryzykownych aktywów. Osiąga to dzięki alokacji średnio największej części środków w pierwsze aktywo i średnio najmniejszej części w trzecie aktywo:

```
In [11]: agent.env.portfolios = pd.DataFrame()

In [12]: %time agent.test(10)
         epizod=10 | łączna nagroda=8.24
         CPU times: user 52.9 s, sys: 7.34 s, total: 1min
         Wall time: 53.1 s

In [13]: agent.env.portfolios.groupby('e')[
             ['xt', 'yt', 'zt']].mean().mean()
Out[13]: xt    0.572418
         yt    0.341007
         zt    0.086576
         dtype: float64

In [14]: agent.env.portfolios.groupby('e')[
             ['Xt', 'Yt', 'Zt', 'pv']].last().mean()
Out[14]: Xt    1.184271
         Yt    1.303997
         Zt    1.219622
         pv    2.927294
         dtype: float64
```

W procesie określania optymalnego działania klasy `InvestingAgent` uwzględniana jest kara za duże rozbieżności w porównaniu z pozycjami z wcześniejszego stanu portfela. Pozwala to uniknąć stosunkowo dużych dynamicznych zmian pozycji. Rysunek 8.11 przedstawia poziom

zmian w konkretnym przebiegu testowym. Jednak choć agent zaczyna od prawie zrównoważonego portfela, szybko dostosowuje poziom alokacji na podstawie zmian cen aktywów:

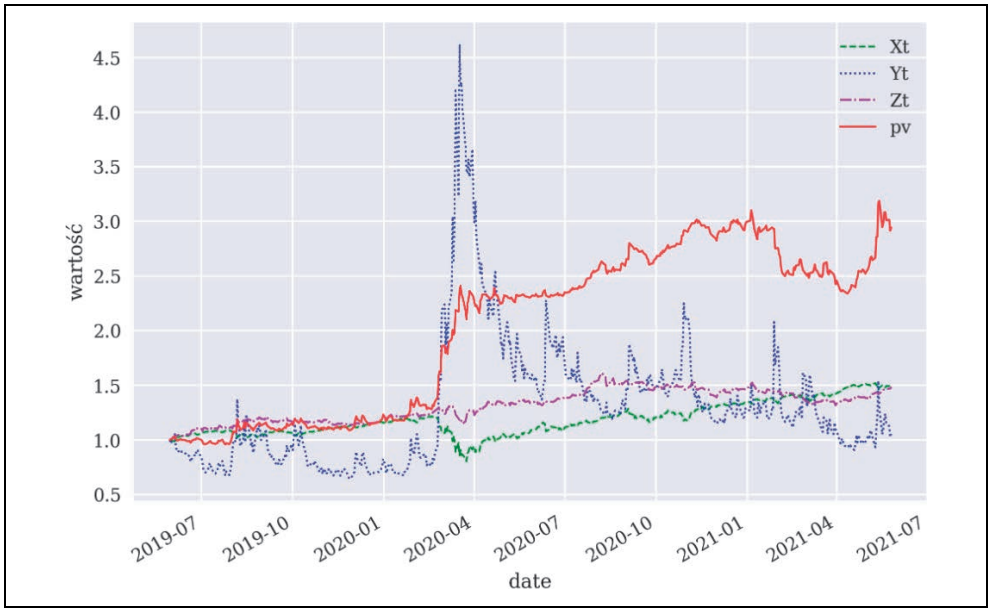
```
In [15]: def get_r(n):
             r = agent.env.portfolios[
                 agent.env.portfolios['e'] == n
                 ].set_index('date')
             return r

In [16]: n = min(agent.env.portfolios['e']) + 1
         n
Out[16]: 66

In [17]: r = get_r(n)

In [18]: r[['xt', 'yt', 'zt']].mean()
Out[18]: xt    0.518429
         yt    0.375992
         zt    0.105579
         dtype: float64

In [19]: r[['xt', 'yt', 'zt']].std()
Out[19]: xt    0.089908
         yt    0.127021
         zt    0.147788
         dtype: float64

In [20]: r[['xt', 'yt', 'zt']].plot(
             title='ALOKACJA (%)',
             style=['g--', 'b:', 'm-.'],
             lw=1, grid=True)
         plt.ylabel('alokacja');
```



Rysunek 8.11. Dynamiczna alokacja w trzy ryzykowne aktywa

Rysunek 8.12 przedstawia wyniki uzyskane przez agenta w porównaniu z trzema ryzykownymi aktywami. W tym przykładzie dynamiczna strategia inwestycyjna agenta nie tylko daje najwyższy zwrot, ale także zdecydowanie najwyższy współczynnik Sharpe'a:

```
In [21]: cols = ['Xt', 'Yt', 'Zt', 'pv']

In [22]: sub = r[cols]

In [23]: rets = sub.iloc[-1] / sub.iloc[0] - 1
         rets
Out[23]: Xt    0.504887
         Yt    0.052514
         Zt    0.484728
         pv    2.670451
         dtype: float64

In [24]: stds = sub.pct_change().std() * math.sqrt(252)
         stds
Out[24]: Xt    0.261492
         Yt    1.475499
         Zt    0.167226
         pv    0.529418
         dtype: float64

In [25]: rets / stds
Out[25]: Xt    1.930792
         Yt    0.035591
         Zt    2.898632
         pv    5.044123
         dtype: float64

In [26]: sub.plot(style=['g--', 'b:', 'm-.', 'r-'], lw=1)
         plt.ylabel('wartość');
```



Rysunek 8.12. Wartość portfela agenta w porównaniu z poszczególnymi aktywami

Nagroda otrzymywana przez agenta bazuje na współczynniku Sharpe'a obliczanym w kolejnych krokach. W tym podejściu nagradzane są wyższe zwroty, a karane jest wyższe ryzyko. Warto również przyjrzeć się współczynnikom Sharpe'a osiągniętym podczas wszystkich przebiegów testowych w porównaniu z inwestowaniem w poszczególne ryzykowne aktywa. Wyniki są oczywiste — agent osiąga średnio znacznie wyższy współczynnik Sharpe'a niż strategia alokacji w poszczególne aktywa:

```
In [27]: sharpe = pd.DataFrame()

In [28]: def calculate_sr():
             for n in set(investing.portfolios['e']):
                 r = get_r(n)
                 sub = r[cols]
                 rets = sub.iloc[-1] / sub.iloc[0] - 1
                 stds = sub.pct_change().std() * math.sqrt(252)
                 sharpe[n] = rets / stds

In [29]: calculate_sr()

In [30]: sharpe.round(2)
Out[30]:       65    66    67    68    69    70    71    72    73    74
         Xt  1.69  1.93 -0.01  0.41  0.16  1.34  0.30  1.31  1.52  0.53
         Yt  0.29  0.04 -0.13 -0.05 -0.14  0.31  0.76 -0.11  0.21  0.80
         Zt  2.78  2.90  0.86 -0.21  0.51  0.71  2.13  1.12  1.19  3.24
         pv  6.55  5.04  2.08  1.11  2.32  3.67  7.09  2.80  3.76  7.84

In [31]: sharpe.mean(axis=1)
Out[31]: Xt    0.917560
         Yt    0.197753
         Zt    1.523657
         pv    4.225037
         dtype: float64
```

Oprócz lepszych *uśrednionych* wyników agent uzyskuje też lepsze rezultaty w każdym pojedynczym przebiegu testowym. Agent w każdym przebiegu osiąga wyższy współczynnik Sharpe'a niż strategia inwestowania w którekolwiek z trzech ryzykownych aktywów:

```
In [32]: ((sharpe.loc['pv'] > sharpe.loc['Xt']) &
          (sharpe.loc['pv'] > sharpe.loc['Yt']) &
          (sharpe.loc['pv'] > sharpe.loc['Zt'])).value_counts()
Out[32]: True    10
         Name: count, dtype: int64
```

Uproszczone modelowanie

Metody i implementacje przedstawione w tym rozdziale są dość uproszczone. Na przykład stan środowiska obejmuje tylko bieżące ceny aktywów uwzględnianych w procesie inwestycji, czasem różnice cenowe i bieżący poziom alokacji. Tak więc model działa zgodnie z założeniem, że zmiany cen ryzykownych aktywów odbywają się zgodnie z procesami Markowa — dla przyszłych zmian istotna jest tylko bieżąca cena, a nie ceny historyczne.

Inna kwestia dotyczy tego, że uwzględnianie tylko dwóch lub trzech aktywów to za mało dla praktycznych zastosowań. Jednak przedstawione scenariusze inwestycyjne są kanonicznymi i ważnymi przykładami w literaturze finansowej dotyczącej teorii portfelowej.

Ponadto w analizach w tym rozdziale zakładam zerowe koszty transakcyjne. Jak ilustrują to niektóre rysunki z tego rozdziału, dynamiczna zmiana alokacji przez agenta odbywa się prawie każdego dnia, co prowadziłoby do dość wysokich kosztów transakcyjnych. Tego typu założenie jest jednak zgodne z analizami przedstawionymi w rozdziale 7.

Wszystkie te aspekty można naturalnie zmodyfikować, wzbogacić i ulepszyć w stosunkowo prosty sposób.

Portfel o równych wagach

Powszechnie wiadomo, że portfel o równych wagach jest trudnym do pokonania punktem odniesienia dla większości aktywnych i dynamicznych strategii alokacji aktywów. Dotyczy to również podejść z poprzednich podrozdziałów. W poniższym kodzie w Pythonie zastąpiłem metodę `.opt_action()` uproszczoną wersją, która zawsze zwraca wektor równych wag ($^1/_3$, $^1/_3$, $^1/_3$). Średnie wyniki mierzone współczynnikiem Sharpe'a są bardzo dobre w porównaniu z inwestycją w poszczególne aktywa. W 10 przebiegach testowych portfel z równymi wagami sześciokrotnie dał wyniki lepsze od najlepszego ryzykownego aktywa. Wydaje się, że ten najprostszy rodzaj dywersyfikacji rzeczywiście jest skuteczny i nie wymaga korzystania z jakichkolwiek informacji lub analiz:

```
In [33]: agent.opt_action = lambda state: np.ones(3) / 3

In [34]: agent.env.portfolios = pd.DataFrame()

In [35]: %time agent.test(10)
         epizod=10 | łączna nagroda=4.75
         CPU times: user 1.98 s, sys: 47.7 ms, total: 2.03 s
         Wall time: 3.53 s

In [36]: sharpe = pd.DataFrame()

In [37]: calculate_sr()

In [38]: sharpe.round(2)
Out[38]:       75    76    77    78    79    80    81    82    83    84
         Xt  1.35  0.41  2.73  1.10  0.38  3.46  1.35  0.81  0.61  1.84
         Yt  0.06  0.20 -0.08  0.62 -0.02 -0.18  0.06 -0.05  0.75 -0.16
         Zt  1.23 -0.44  0.37  1.52 -0.16 -0.87  1.23 -0.72  4.86  1.30
         pv  1.67  1.52  1.32  2.52  1.25  0.96  1.67  1.27  3.77  1.76

In [39]: sharpe.mean(axis=1)
Out[39]: Xt    1.402960
         Yt    0.121449
         Zt    0.830933
         pv    1.769955
         dtype: float64

In [40]: ((sharpe.loc['pv'] > sharpe.loc['Xt']) &
          (sharpe.loc['pv'] > sharpe.loc['Yt']) &
          (sharpe.loc['pv'] > sharpe.loc['Zt'])).value_counts(
Out[40]: True     6
         False    4
         Name: count, dtype: int64
```

Podsumowanie

Dynamiczna alokacja aktywów to kolejny problem finansowy, który można rozwiązać za pomocą metod uczenia przez wzmacnianie i podejścia DQL. W tym rozdziale omówiłem trzy różne kanoniczne scenariusze:

- z jednym aktywem ryzykownym i jednym wolnym od ryzyka,
- z dwoma aktywami ryzykownymi,
- z trzema aktywami ryzykownymi.

Popularną strategią inwestycyjną jest tworzenia portfela z wagami 60/40, w którym 60% inwestowane jest w aktywa ryzykowne, takie jak indeksy akcji, a 40% w aktywa mniej ryzykowne, na przykład obligacje skarbowe lub korporacyjne. Przykłady w podrozdziale „Podział między dwa fundusze" niemal dokładnie odzwierciedlają tę strategię, ponieważ alokacja agenta `InvestingAgent` w ryzykowne aktywa często oscyluje w okolicach 60%.

W podrozdziale „Scenariusz z dwoma aktywami" aktywa wolne od ryzyka zostały zastąpione innym aktywem ryzykownym. Wiadomo, że aktywa wybrane w przykładzie, indeks giełdowy S&P 500 i indeks VIX, są silnie ujemnie skorelowane. Na ogólnym poziomie okazuje się, że dywersyfikacja jest wysoce opłacalna. Agent stosujący strategię dynamicznej alokacji aktywów zwykle osiąga wyższy bezwzględny zwrot i wyższy współczynnik Sharpe'a w porównaniu z inwestowaniem w poszczególne aktywa.

Podejście przedstawione w podrozdziale „Scenariusz z trzema aktywami" jest uogólnieniem wersji z dwoma aktywami. Ten scenariusz inwestycyjny w formie statycznej został przeanalizowany w przełomowym artykule Markowitza (1952) na temat nowoczesnej teorii portfelowej. Agent stosujący strategię dynamicznej alokacji uzyskał wyniki lepsze niż inwestowanie w każde z trzech pojedynczych aktywów (mierzone na podstawie współczynnika Sharpe'a) w 10 przebiegach testowych na 10.

Literatura

- Black Fischer i Scholes Myron, *The Pricing of Options and Corporate Liabilities*, „Journal of Political Economy" 81, nr 3, maj – czerwiec 1973, s. 637 – 654.
- Chisholm Denise, *Three Key Catalysts for the 60/40 Strategy* (*https://oreil.ly/0oAyA*), Komentarz, Fidelity Investments, 2023.
- Copeland Thomas E., Weston J. Fred i Shastri Kuldeep, *Financial Theory and Corporate Policy*, wydanie czwarte, Pearson Addison Wesley, Reading (Massachusetts) 2005.
- „Economist", *The $100trn Battle for the World's Wealthiest People*, 5 września 2023.
- Markowitz Harry, *Portfolio Selection*, „Journal of Finance" 7, nr 1, marzec 1952, s. 77 – 91.
- Merton Robert C., *Lifetime Portfolio Selection Under Uncertainty: The Continuous-Time Case*, „The Review of Economics and Statistics" 51, nr 3, sierpień 1969, s. 247 – 257.

- Merton Robert C., *Theory of Rational Option Pricing*, „Bell Journal of Economics and Management Science" 4, nr 1, wiosna 1973, s. 141 – 183.
- Poundstone William, *Fortune's Formula: The Untold Story of the Scientific Betting System That Beat the Casinos and Wall Street*, Hill and Wang, Nowy Jork 2006.

Kod dla scenariusza z trzema aktywami

W poniższym kodzie w Pythonie przedstawione są dwie główne klasy, `Investing` i `Investing Agent`, do scenariusza z trzema aktywami:

```
#
# Środowisko Investing i agent
# Scenariusz z trzema aktywami
#
# (c) Dr Yves J. Hilpisch
# Uczenie przez wzmacnianie w branży finansowej
#

import os
import math
import random
import numpy as np
import pandas as pd
from scipy import stats
from pylab import plt, mpl
from scipy.optimize import minimize

from dqlagent import *

plt.style.use('seaborn-v0_8')
mpl.rcParams['figure.dpi'] = 300
mpl.rcParams['savefig.dpi'] = 300
mpl.rcParams['font.family'] = 'serif'
np.set_printoptions(suppress=True)

opt = keras.optimizers.legacy.Adam

os.environ['PYTHONHASHSEED'] = '0'
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'

class observation_space:
    def __init__(self, n):
        self.shape = (n,)

class action_space:
    def __init__(self, n):
        self.n = n
    def seed(self, seed):
        random.seed(seed)
    def sample(self):
        rn = np.random.random(3)
        return rn / rn.sum()

class Investing:
```

```
def __init__(self, asset_one, asset_two, asset_three,
             steps=252, amount=1):
    self.asset_one = asset_one
    self.asset_two = asset_two
    self.asset_three = asset_three
    self.steps = steps
    self.initial_balance = amount
    self.portfolio_value = amount
    self.portfolio_value_new = amount
    self.observation_space = observation_space(4)
    self.osn = self.observation_space.shape[0]
    self.action_space = action_space(3)
    self.retrieved = 0
    self._generate_data()
    self.portfolios = pd.DataFrame()
    self.episode = 0

def _generate_data(self):
    if self.retrieved:
        pass
    else:
        url = 'https://certificate.tpq.io/rl4finance.csv'
        self.raw = pd.read_csv(url, index_col=0, parse_dates=True).dropna()
        self.retrieved
    self.data = pd.DataFrame()
    self.data['X'] = self.raw[self.asset_one]
    self.data['Y'] = self.raw[self.asset_two]
    self.data['Z'] = self.raw[self.asset_three]
    s = random.randint(self.steps, len(self.data))
    self.data = self.data.iloc[s-self.steps:s]
    self.data = self.data / self.data.iloc[0]

def _get_state(self):
    Xt = self.data['X'].iloc[self.bar]
    Yt = self.data['Y'].iloc[self.bar]
    Zt = self.data['Z'].iloc[self.bar]
    date = self.data.index[self.bar]
    return np.array(
        [Xt, Yt, Zt, self.xt, self.yt, self.zt]
        ), {'date': date}

def seed(self, seed=None):
    if seed is not None:
        random.seed(seed)

def reset(self):
    self.xt = 0
    self.yt = 0
    self.zt = 0
    self.bar = 0
    self.treward = 0
    self.portfolio_value = self.initial_balance
    self.portfolio_value_new = self.initial_balance
    self.episode += 1
    self._generate_data()
    self.state, info = self._get_state()
    return self.state, info
```

```
    def add_results(self, pl):
        df = pd.DataFrame({
                    'e': self.episode, 'date': self.date,
                    'xt': self.xt, 'yt': self.yt, 'zt': self.zt,
                    'pv': self.portfolio_value,
                    'pv_new': self.portfolio_value_new, 'p&l[$]': pl,
                    'p&l[%]': pl / self.portfolio_value_new * 100,
                    'Xt': self.state[0], 'Yt': self.state[1],
                    'Zt': self.state[2], 'Xt_new': self.new_state[0],
                    'Yt_new': self.new_state[1],
                    'Zt_new': self.new_state[2],
                           }, index=[0])
        self.portfolios = pd.concat((self.portfolios, df), ignore_index=True)

    def step(self, action):
        self.bar += 1
        self.new_state, info = self._get_state()
        self.date = info['date']
        if self.bar == 1:
            self.xt = action[0]
            self.yt = action[1]
            self.zt = action[2]
            pl = 0.
            reward = 0.
            self.add_results(pl)
        else:
            self.portfolio_value_new = (
                self.xt * self.portfolio_value *
                    self.new_state[0] / self.state[0] +
                self.yt * self.portfolio_value *
                    self.new_state[1] / self.state[1] +
                self.zt * self.portfolio_value *
                    self.new_state[2] / self.state[2]
            )
            pl = self.portfolio_value_new - self.portfolio_value
            self.xt = action[0]
            self.yt = action[1]
            self.zt = action[2]
            self.add_results(pl)
            ret = self.portfolios['p&l[%]'].iloc[-1] / 100 * 252
            vol = self.portfolios['p&l[%]'].rolling(
                20, min_periods=1).std().iloc[-1] * math.sqrt(252)
            sharpe = ret / vol
            reward = sharpe
            self.portfolio_value = self.portfolio_value_new
        if self.bar == len(self.data) - 1:
            done = True
        else:
            done = False
        self.state = self.new_state
        return self.state, reward, done, False, {}

class InvestingAgent(DQLAgent):
    def _create_model(self, hu, lr):
        self.model = Sequential()
        self.model.add(Dense(hu, input_dim=self.n_features,
                             activation='relu'))
        self.model.add(Dense(hu, activation='relu'))
```

```
        self.model.add(Dense(1, activation='linear'))
        self.model.compile(loss='mse',
                optimizer=opt(learning_rate=lr))

    def opt_action(self, state):
        bnds = 3 * [(0, 1)]
        cons = [{'type': 'eq', 'fun': lambda x: x.sum() - 1}]
        def f(state, x):
            s = state.copy()
            s[0, 3] = x[0]
            s[0, 4] = x[1]
            s[0, 5] = x[2]
            pen = np.mean((state[0, 3:] - x) ** 2)
            return self.model.predict(s)[0, 0] - pen
        try:
            state = self._reshape(state)
            self.action = minimize(lambda x: -f(state, x),
                                   3 * [1 / 3],
                                   bounds=bnds,
                                   constraints=cons,
                                   options={
                                       'eps': 1e-4,
                                        },
                                   method='SLSQP'
                                  )['x']
        except:
            print(state)
        return self.action

    def act(self, state):
        if random.random() <= self.epsilon:
            return self.env.action_space.sample()
        action = self.opt_action(state)
        return action

    def replay(self):
        batch = random.sample(self.memory, self.batch_size)
        for state, action, next_state, reward, done in batch:
            target = reward
            if not done:
                ns = next_state.copy()
                action = self.opt_action(ns)
                ns[0, 3:] = action
                target += self.gamma * self.model.predict(ns)[0, 0]
            self.model.fit(state, np.array([target]), epochs=1,
                           verbose=False)
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay

    def test(self, episodes, verbose=True):
        for e in range(1, episodes + 1):
            state, _ = self.env.reset()
            state = self._reshape(state)
            treward = 0
            for _ in range(1, len(self.env.data) + 1):
                action = self.opt_action(state)
                state, reward, done, trunc, _ = self.env.step(action)
                state = self._reshape(state)
```

```
            treward += reward
            if done:
                templ = f'epizod={e} | '
                templ += f'łączna nagroda={treward:4.2f}'
                if verbose:
                    print(templ, end='\r')
                break
    print()
```

Książka ta doskonale wypełnia lukę między teorią a praktyką dzięki jasnym objaśnieniom i szczegółowemu kodowi w Pythonie!

Ivilina Popova, Texas State University

Uczenie przez wzmacnianie okazało się przełomowym rozwiązaniem. Jednym z najciekawszych algorytmów jest Deep Q-Learning (DQL), który może być stosowany do zmieniających się warunków decyzyjnych. DQL w wielu przypadkach wykazuje skuteczność nieosiągalną dla człowieka. Nic dziwnego, że użycie tego rodzaju algorytmów w branży finansowej wydaje się wyjątkowo atrakcyjną opcją.

Ta książka jest zwięzłym wprowadzeniem do głównych zagadnień i aspektów uczenia przez wzmacnianie i algorytmów DQL. Docenią ją zarówno naukowcy, jak i praktycy poszukujący skutecznych algorytmów, przydatnych w pracy z finansami. Znajdziesz tu wiele interesujących przykładów w języku Python, zaprezentowanych w formie najciekawszych algorytmów gotowych do samodzielnego modyfikowania i testowania.

W książce między innymi:

- uczenie przez wzmacnianie
- algorytm DQL
- algorytm aktor-krytyk
- implementacja powyższych algorytmów w Pythonie
- rozwiązywanie problemów handlu algorytmicznego, hedgingu dynamicznego i dynamicznej alokacji środków w aktywa

Dr Yves J. Hilpisch kieruje programem Certificate in Python for Finance, pionierską inicjatywą edukacyjną. Jest też twórcą DX Analytics, biblioteki służącej do analityki finansowej. Regularnie organizuje w globalnych centrach finansowych konferencje i bootcampy dotyczące zastosowań Pythona, a także sztucznej inteligencji w obszarze finansów ilościowych i handlu algorytmicznego.

Helion
helion.pl
HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-2578-6
9 788328 925786
Cena: 79,00 zł