



Creación de un entorno de simulación de mercados financieros (stocks)

Owner	Lucas Bool
Tags	Python
Date	@2 de abril de 2024



La idea es crear un entorno virtual que funcione como la bolsa, para poder entender mejor su funcionamiento (tanto en cómo obtener rentabilidad, o qué son las opciones y como trabajar con ellas), y poder crear modelos de Reinforcement Learning en él. Referencia inicial: <https://gym-trading-env.readthedocs.io/en/latest/>

1. Referencias iniciales

Propuesta de chatGPT

Input

Respuesta

1. Clase `Company`

2. Clase `Portfolio`

3. Clase `Strategy`

4. Clase `Agent`

5. Clase `Environment`

6. Ejemplo de uso

7. Implementación de las Opciones

Clase `Option` :

Clase `Company` (actualizada):

Clase `Portfolio` (actualizada):

8. Más clases necesarias

Clase `Market` :

Clase `Exchange` :

Clase `MarketData` :

Clase `RewardFunction` :

Clase `Simulation` :

1. Referencias iniciales

- Custom crypto trading environment from a scratch - Bitcoin trading bot example #1: https://www.youtube.com/watch?v=QQp1KDIQ3v8&ab_channel=PythonLessons

Propuesta de chatGPT

Input

HOLA! Quiero crear un entorno para simular mercados financieros (bolsa), y sobre él, quiero crear un agente de reinforcement learning, que tenga acceso a un portfolio (que será su capital y las empresas en las que puede invertir), y que siga determinadas estrategias (que deberán ser programadas y testeadas). Tendrá que haber una clase Course: Esta clase será un entorno para una empresa o índice en concreto, y tendrá la estructura típica de un entorno (similar a los entornos gym de openai, con los métodos step, reset y render), y devolverá las recompensas en cada instante segun las acciones posibles (buy, sell, hold). Tiene que funcionar de tal modo que se puedan instanciar varias compañías a la vez, para que el agente pueda tener acceso a varias empresas a la vez. Esto último es el portfolio, que será otra clase, al igual que las estrategias.

Crea la estructura de proyecto y todas las clases y sus métodos básicos en python.

Respuesta

Claro, aquí te proporciono una estructura básica de proyecto con las clases y métodos que mencionaste:

```
pythonCopy code
class Company:
    def __init__(self, name, initial_price):
        self.name = name
        self.price = initial_price
        self.shares = 0 # Number of shares owned by the agent

    def step(self, action):
        # Implement the logic for how the company's price changes based on the action
        # and return the reward for the action
        pass

    def reset(self):
        # Reset the state of the company (e.g., price) to its initial state
        pass

    def render(self):
        # Visualize the current state of the company (optional)
        pass

class Portfolio:
    def __init__(self, capital):
        self.capital = capital
        self.companies = [] # List of companies in the portfolio
```

```

def add_company(self, company):
    # Add a company to the portfolio
    self.companies.append(company)

def remove_company(self, company):
    # Remove a company from the portfolio
    self.companies.remove(company)

def update_capital(self, reward):
    # Update the portfolio's capital based on the reward received
    self.capital += reward

class Strategy:
    def __init__(self):
        pass

    def select_action(self, state):
        # Implement the logic for selecting an action based on the state
        pass

class Agent:
    def __init__(self, portfolio, strategy):
        self.portfolio = portfolio
        self.strategy = strategy

    def step(self, state):
        # Select action based on the strategy and take action in the environment
        action = self.strategy.select_action(state)
        reward = self.portfolio.companies[0].step(action)
        # Assuming only one company for simplicity
        self.portfolio.update_capital(reward)

```

```

class Environment:
    def __init__(self, companies):
        self.companies = companies

    def step(self, actions):
        rewards = []
        for i, company in enumerate(self.companies):
            reward = company.step(actions[i])
            rewards.append(reward)
        return rewards

    def reset(self):
        for company in self.companies:
            company.reset()

    def render(self):
        for company in self.companies:
            company.render()

# Example usage:

company1 = Company("Company1", initial_price=100)
company2 = Company("Company2", initial_price=150)

portfolio = Portfolio(capital=10000)
portfolio.add_company(company1)
portfolio.add_company(company2)

strategy = Strategy()

agent = Agent(portfolio, strategy)

environment = Environment([company1, company2])

# Main loop
for _ in range(NUM_EPISODES):
    state = environment.reset()

```

```
done = False
while not done:
    actions = agent.step(state)
    state, rewards, done, _ = environment.step(actions)
    environment.render()
```

1. Clase **Company**

- Esta clase representa una empresa en el mercado financiero.
- Atributos:
 - **name** : El nombre de la empresa.
 - **price** : El precio actual de las acciones de la empresa.
 - **shares** : La cantidad de acciones de la empresa que posee el agente.
- Métodos:
 - **step(action)** : Simula un paso en el tiempo y devuelve la recompensa correspondiente a la acción realizada por el agente.
 - **reset()** : Restablece el estado de la empresa a su estado inicial.
 - **render()** : Opcionalmente, visualiza el estado actual de la empresa.

2. Clase **Portfolio**

- Esta clase representa el portafolio del agente, que incluye su capital y las empresas en las que ha invertido.
- Atributos:
 - **capital** : El capital inicial del agente.
 - **companies** : Una lista de objetos **Company** que representan las empresas en las que el agente ha invertido.
- Métodos:
 - **add_company(company)** : Añade una empresa al portafolio.
 - **remove_company(company)** : Elimina una empresa del portafolio.

- `update_capital(reward)` : Actualiza el capital del portafolio basado en la recompensa recibida.

3. Clase **Strategy**

- Esta clase representa la estrategia que seguirá el agente para tomar decisiones de inversión.
- Métodos:
 - `select_action(state)` : Selecciona una acción basada en el estado actual.

4. Clase **Agent**

- Esta clase representa al agente de reinforcement learning.
- Atributos:
 - `portfolio` : El portafolio del agente.
 - `strategy` : La estrategia que sigue el agente para tomar decisiones.
- Métodos:
 - `step(state)` : Toma una acción basada en la estrategia y actualiza el estado del portafolio.

5. Clase **Environment**

- Esta clase representa el entorno en el que opera el agente.
- Atributos:
 - `companies` : Una lista de objetos `Company` que representan las empresas en el mercado financiero.
- Métodos:
 - `step(actions)` : Simula un paso en el tiempo para todas las empresas en el entorno y devuelve las recompensas correspondientes a las acciones realizadas.
 - `reset()` : Restablece el estado de todas las empresas a su estado inicial.

- `render()` : Opcionalmente, visualiza el estado actual de todas las empresas.

6. Ejemplo de uso

- En este ejemplo, se crean objetos de las clases `Company`, se agregan al portafolio, se crea un objeto de la clase `Strategy`, se instancia un agente y un entorno, y se ejecuta un bucle principal para simular el comportamiento del agente en el entorno.

7. Implementación de las Opciones

Para implementar la funcionalidad de compra de opciones, podemos extender la clase `Company` para que tenga un atributo que represente el precio de la opción y el número de opciones disponibles. Luego, modificaremos la clase `Portfolio` para que pueda realizar transacciones de compra y venta de opciones. Aquí tienes la implementación actualizada:

```
pythonCopy code
class Company:
    def __init__(self, name, initial_price, option_price=0,
option_quantity=0):
        self.name = name
        self.price = initial_price
        self.shares = 0
        self.option_price = option_price
        self.option_quantity = option_quantity

    def step(self, action):
        # Implementar la lógica para el cambio de precio de
la empresa y opciones
        pass

    def reset(self):
        # Resetear el estado de la empresa
        pass

    def render(self):
```



```

        # Visualizar el estado actual de la empresa
        pass

class Portfolio:
    def __init__(self, capital):
        self.capital = capital
        self.companies = []

    def add_company(self, company):
        self.companies.append(company)

    def remove_company(self, company):
        self.companies.remove(company)

    def update_capital(self, reward):
        self.capital += reward

    def buy_option(self, company, quantity):
        # Verificar si el agente tiene suficiente capital p
        ara comprar las opciones
        cost = company.option_price * quantity
        if self.capital >= cost and company.option_quantity
        >= quantity:
            company.option_quantity -= quantity
            self.capital -= cost
            return True
        else:
            return False

    def sell_option(self, company, quantity):
        # Vender opciones y aumentar el capital del agente
        revenue = company.option_price * quantity
        company.option_quantity += quantity
        self.capital += revenue
        return revenue

```

```

class Strategy:
    def __init__(self):
        pass

    def select_action(self, state):
        pass

class Agent:
    def __init__(self, portfolio, strategy):
        self.portfolio = portfolio
        self.strategy = strategy

    def step(self, state):
        action = self.strategy.select_action(state)
        # Implementar la lógica para la compra y venta de o
pciones
        pass

class Environment:
    def __init__(self, companies):
        self.companies = companies

    def step(self, actions):
        pass

    def reset(self):
        pass

    def render(self):
        pass

# Ejemplo de uso:

company1 = Company("Company1", initial_price=100, option_pr
ice=5, option_quantity=10)

```

```

company2 = Company("Company2", initial_price=150, option_price=7, option_quantity=15)

portfolio = Portfolio(capital=10000)
portfolio.add_company(company1)
portfolio.add_company(company2)

strategy = Strategy()

agent = Agent(portfolio, strategy)

environment = Environment([company1, company2])

for _ in range(NUM_EPISODES):
    state = environment.reset()
    done = False
    while not done:
        actions = agent.step(state)
        state, rewards, done, _ = environment.step(actions)
        environment.render()

```

En este código, he agregado los atributos `option_price` y `option_quantity` a la clase `Company` para representar el precio y la cantidad de opciones disponibles para esa empresa. Luego, en la clase `Portfolio`, he agregado los métodos `buy_option` y `sell_option` para permitir al agente comprar y vender opciones, respectivamente. Estos métodos verifican si el agente tiene suficiente capital y si hay suficientes opciones disponibles antes de realizar la transacción. La lógica real de compra y venta de opciones se implementará en el método `step` de la clase `Agent`.

Clase `Option`:

- La clase `Option` representa una opción específica asociada a una empresa en el mercado financiero.
- Atributos:
 - `price`: El precio de la opción.
 - `quantity`: La cantidad de opciones disponibles para esa empresa.

- Métodos:
 - `buy(quantity)` : Permite al agente comprar un número específico de opciones de esa empresa si tiene suficiente capital y hay suficientes opciones disponibles.
 - `sell(quantity)` : Permite al agente vender un número específico de opciones de esa empresa, aumentando su capital.

Clase `Company` (actualizada):

- Se añaden los atributos `option_price` y `option_quantity` para representar el precio y cantidad de opciones asociadas a esa empresa.
- Se implementan métodos para la compra y venta de opciones en función de los atributos `option_price` y `option_quantity`.

Clase `Portfolio` (actualizada):

- Se añaden los métodos `buy_option` y `sell_option` para permitir al agente comprar y vender opciones en su portafolio.

8. Más clases necesarias

Clase `Market` :

- Esta clase puede representar el mercado financiero en su conjunto, que incluye múltiples empresas, índices, commodities, etc.
- Atributos:
 - Una lista de objetos `Company` y `Option` que representan las diferentes empresas e opciones disponibles en el mercado.
- Métodos:
 - Funciones para calcular el rendimiento general del mercado, como el valor del índice.
 - Métodos para agregar nuevas empresas al mercado o eliminarlas.

Clase `Exchange` :

- Esta clase puede representar una bolsa de valores donde se llevan a cabo las transacciones.
- Métodos:
 - Funciones para realizar transacciones de compra y venta de acciones y opciones.
 - Lógica para el manejo de órdenes de compra y venta, como el emparejamiento de órdenes y el cálculo del precio de ejecución.
 - Gestión del libro de órdenes.

Clase `MarketData` :

- Esta clase puede proporcionar datos históricos y en tiempo real sobre los precios de las acciones y opciones en el mercado.
- Métodos:
 - Funciones para obtener precios históricos y en tiempo real de las acciones y opciones.
 - Métodos para el análisis técnico y la generación de características (features) para el agente de reinforcement learning.

Clase `RewardFunction` :

- Esta clase puede definir la función de recompensa que el agente busca maximizar durante el proceso de aprendizaje.
- Métodos:
 - Funciones para calcular la recompensa en función de las acciones del agente y el estado del mercado.

Clase `Simulation` :

- Esta clase puede coordinar la simulación general del mercado financiero, incluyendo la interacción del agente, el entorno del mercado y el intercambio de valores.
- Métodos:
 - Funciones para iniciar, ejecutar y detener la simulación.
 - Lógica para registrar y analizar los resultados de la simulación.