파이썬과 케라스를 이용한

딥러닝/강화학습 주식투자

파이썬과 케라스를 이용한

딥러닝/강화학습 주식투자

지은이 김문권

펴낸이 박찬규 교정/교열 리율 디자인 북누리 표지디자인 아로와 & 아로와나

펴낸곳 위키북스 전화 031-955-3658, 3659 팩스 031-955-3660

주소 경기도 파주시 문발로 115 세종출판벤처타운 #311

가격 25,000 페이지 284 책규격 175 x 235mm

초판 발행 2018년 05월 29일

ISBN 979-11-5839-106-5 (93000)

등록번호 제406-2006-000036호 등록일자 2006년 05월 19일

홈페이지 wikibook.co.kr 전자우편 wikibook@wikibook.co.kr

이 도서의 국립중앙도서관 출판시도서목록(CIP)은

서지정보유통지원시스템 홈페이지(http://seoji.nl.go.kr)와

국가자료공동목록시스템(http://www.nl.go.kr/kolisnet)에서 이용하실 수 있습니다.

CIP제어번호 CIP2018015156

파이썬과
케라스를 이용한

딥러닝/ 강화학습 주식투자

퀀트 투자,
알고리즘 트레이딩을 위한
최첨단 해법 입문

김운권 지음

위키북스

취미 삼아 주식투자를 해 오던 중 시도때도 없이 주가를 확인하는 저의 모습을 보게 되고 '소프트웨어로 사람이 주가를 쳐다보고 있는 시간을 줄일 수 있지 않을까?'라고 생각하게 되었습니다. 단연 전업 투자자가 아닌 이상 주식에 매여 일상 생활에 지장을 주는 것은 바람직하지 않습니다.

이 무렵에 구글은 알파고를 세상에 선보이며 기존에는 불가능하다고 여겨졌던 바둑 인공지능을 실현합니다. 여기서 사용한 기법 중 하나가 강화학습으로, 저자는 '주식투자에도 강화학습을 적용해서 컴퓨터가 스스로 투자를 할 수 있지 않을까?'라고 생각해 보게 되었습니다.

이미 많은 사람이 주식투자 자동화를 시도하고 있었고 논문도 많았습니다. 주식투자에 강화학습을 적용한 기존 연구 성과들을 공부하여 강화학습 주식투자 시뮬레이션 프로그램을 개발하게 되었습니다. 많은 사람들이 자동 주식투자에 관심을 가져서 기술이 더 발전했으면 하는 바람으로 이 책으로 그 과정과 결과를 공유하려 합니다.

이 책은 딥러닝과 강화학습을 이해하고 실제 주식 데이터를 준비해서 직접 파이썬으로 강화학습 주식투자 프로그램을 구현해 볼 수 있도록 구성되어 있습니다. 각자가 더 다양하게 실험해 볼 수 있도록 코드를 수정해 쓰는 방법도 다룹니다. 이 책을 학습하는 과정에서 파이썬을 자연스럽게 학습하고 나아가 딥러닝과 강화학습을 이해하여 주식투자는 물론 다른 분야에도 적용할 수 있을 것입니다.

10년이 넘게 소프트웨어 개발을 해왔습니다. 파이썬, 자바를 포함한 다양한 언어를 익혔으며 안드로이드 앱 개발, 웹 프로그래밍, 데이터 분석, 머신러닝, 딥러닝, 빅데이터 분석을 포함하여 그동안 주목을 받아 온 다양한 기술을 익혀왔습니다.

숭실대학교에서 소프트웨어 공학 전공으로 석사 학위를 취득하고 박사 과정을 수료했습니다. 그 과정에서 국내외 저명 학회와 학술지에 수십 편의 논문을 발표했으며 우수 논문상도 여러 번 수상했습니다. 현재는 네이버에서 검색 기술을 개발하고 있습니다.

퀀트투자 연구소(http://quantylab.com)를 운영하고 있고 GitHub(https://github.com/ quantylab)에서 퀀트투자 관련 프로젝트를 진행하고 있습니다.

– 김문권

04장
모듈 개발: 강화학습 기반 주식투자 시스템 개발 61

08장
모델 활용: 학습된 정책 신경망 모델을 사용한 투자 시뮬레이션 219

01 장

배경 이론 1: 딥러닝이란?

이번 장에서는 딥러닝(deep learning)의 이론과 적용 사례를 소개합니다. 이 책은 강화학습(reinforcement learning)을 주식투자에 적용하는 것과 파이썬으로 강화학습 기반 주식투자 시스템을 구현하는 것이 목적입니다. 여기서 딥러닝을 다루는 이유는 강화학습을 딥러닝의 일환으로 볼 수 있기 때문입니다. 그래서 전반적인 딥러닝 이론을 먼저 소개하고 강화학습 이론을 이어서 소개합니다.

1.1 딥러닝 개요

이번 절에서는 딥러닝이 무엇이고 딥러닝으로 할 수 있는 게 무엇이며 딥러닝이 최근에 와서야 주목받는 이유에 대해서 다룹니다.

1.1.1 딥러닝의 정의와 역사

딥러닝의 표준 정의는 없지만 일반적으로 딥러닝은 머신러닝(machine learning, ML)의 한 종류로 주로 인공 신경망(artificial neural network, ANN)의 발전된 모델로 볼 수 있습니다. 이 기법들은 인공지능(artifical intelligence, AI)을 실현하기 위한 도구로서 사용됩니다.

사실 딥러닝은 1950년대부터 연구되어 왔습니다. 즉 딥러닝은 새로운 개념이 아닌 오래된 역사를 가진 기술로 볼 수 있습니다. 시대별 주요 특징을 살펴보면 그림 1.1과 같습니다.

1950년대	1960년대	1970년대	1980년대
인공 신경망 유년기	**인공 신경망 황금기**	**인공 신경망 암흑기**	**인공 신경망 부활기**
· 퍼셉트론 등장	· 퍼셉트론 한계 증명		· 오차역전파 적용 · DNN, RNN, CNN 발전

1990년대	2000년대	2010년대
인공 신경망 발전기	**딥러닝 주목기**	**딥러닝 부흥기**
· LSTM 등장 · LeNet-5 등장	· 가트너 10대 전략기술	· 알파고

그림 1.1 딥러닝의 역사

1950년대에 등장한 퍼셉트론(perceptron)은 인공 신경망의 시초라고 할 수 있습니다. 이후 1960년대에 인공 신경망 연구가 활발히 진행되어 왔습니다. 그러나 1969년 ≪Perceptrons≫라는 책이 출간되었는데 이 책은 퍼셉트론의 치명적인 한계점을 밝히고 증명까지 담고 있었습니다. 이후 1970년대에는 많은 학자들로부터 인공 신경망이 외면받게 되는 암흑기에 들어섭니다.

1980년대에는 인공 신경망 연구가 다시 주목을 받기 시작합니다. 이는 1986년에 오차 역전파법(back propagation)을 적용하여 다층의 인공 신경망을 학습하는 방법이 고안된 덕택이라 볼 수 있습니다. 이 시대에 심층 신경망(deep neural network, DNN), 순환 신경망(recurrent neural network, RNN), 합성곱 신경망(convolutional neural network, CNN) 등이 발전되어 왔습니다.

1990년대에는 발전된 형태의 인공 신경망들이 등장했습니다. 고급 순환 신경망인 LSTM이 1997년에, 고급 합성곱 신경망인 LeNet-5가 1998년에 발표되었습니다.

2000년대에는 딥러닝(deep learning)이라는 이름으로 인공 신경망이 각광을 받기 시작합니다. 저명한 정보 기술 연구 및 조사 기관인 가트너(Gartner)가 딥러닝을 10대 전략 기술로 선정합니다.

2010년대에는 구글의 자회사인 딥마인드(Deep Mind)가 익히 알려진 알파고를 공개합니다. 이 이후로 한국에서도 딥러닝이 폭발적인 주목을 받게 됩니다.

1.1.2 딥러닝이 최근에 주목 받는 이유

왜 최근에 와서야 딥러닝이 주목을 받을 수 있었을까요? 여러 이유가 있겠지만 크게 두 가지 이유를 들어 보겠습니다.

우선 컴퓨터의 연산 능력이 좋아졌습니다. 인공 신경망을 학습시키는 일은 많은 연산을 필요로 합니다. 복잡한 문제를 풀기 위해서는 인공 신경망의 구조를 매우 복잡하고 정교하게 만들어야 합니다. 이 경우 방대한 연산이 발생하게 되는데 GPU 사용, 분산처리 등으로 이러한 연산이 현실적으로 가능하게 되었습니다.

또 하나의 이유는 빅데이터(big data)의 등장입니다. 인공 신경망을 제대로 학습시키기 위해서는 방대한 양질의 데이터가 필요한데 빅데이터를 학습함으로써 인공 신경망의 정확도를 상당히 향상시킬 수 있었습니다.

딥러닝 이론들을 완벽하게 알지 못해도 목표로 삼은 강화학습을 이용한 주식투자 시스템을 개발하는 데 무리는 없습니다. 인공 신경망을 텐서플로(TensorFlow), 케라스(Keras) 등의 라이브러리를 사용하면 쉽게 구현할 수 있기 때문입니다. 이 책에서는 딥러닝 관련 이론들을 소개하는 정도로만 살펴보고, 대신에 코드로 상세하게 구현해 보는 데 역점을 둡니다.

1.1.3 딥러닝으로 풀고자 하는 문제

딥러닝뿐만 아니라 여타 머신러닝에서 풀고자 하는 문제들은 일반적으로 분류 문제, 군집화 문제, 회귀 문제로 구분할 수 있습니다.

- **분류(classification)**

  데이터의 부류(class)를 알아내기 위한 문제입니다. 이를테면 자동차의 길이, 너비, 높이, 바퀴 크기, 엔진 마력 등의 특징(feature)을 보고 경차, 준중형차, 중형차, 대형차 중 한 가지 부류로 분류하는 문제를 들 수 있습니다.

- **군집화(clustering)**

  데이터 인스턴스(data instance)[1]들을 그룹화하기 위한 문제입니다. 즉, 비슷한 특징을 가지는 데이터 인스턴스들끼리 그룹화하는 것입니다. 예를 들어 자동차의 길이, 너비, 높이, 바퀴 크기, 엔진 마력 등의 특징들을 보고 비슷한 인스턴스끼리 그룹화합니다. 그룹화된 결과를 보고 그룹 1은 경차, 그룹 2는 준준형차 등으로 정해 주기 위해서는 사람의 개입이 필요합니다.

---

1 데이터 인스턴스란 엑셀과 같은 스프레드시트상의 각 행(row), 데이터베이스의 레코드(record)와 같은 개념입니다. 사례(example)라고도 합니다. 각 인스턴스는 여러 특징(feature)으로 구성되는데, 특징을 속성(attribute)이라고도 부릅니다.

- **회귀(regression)**

  불완전한 데이터의 값(value)를 알아내기 위한 문제입니다. 예를 들어 한 데이터 인스턴스 중에 자동차의 너비, 높이, 바퀴의 크기, 엔진 마력 등의 특징을 아는데 길이를 모른다고 할 때, 다른 데이터 인스턴스들의 값을 근거로 불완전한 데이터로 구성된 인스턴스의 길이 특징을 예측할 수 있습니다.

각 문제들을 머신러닝(machine learning) 관점에서 풀 수 있는데, 학습 방법을 지도학습, 비지도학습, 강화학습으로 구분할 수 있습니다.

- **지도학습(supervised learning)**

  레이블(label)이 있는 데이터를 학습하는 방법으로 주로 분류와 회귀 문제를 다룹니다. 레이블이 있다는 것은 정답이 있다는 의미이기 때문에 비교적 학습이 쉽고 효과적입니다. 그러나 레이블을 데이터마다 부여하려면 일반적으로 많은 비용이 요구됩니다. 비용은 돈이 될 수도, 시간이 될 수도, 많은 경우는 둘 다가 될 수도 있습니다. 학습 데이터는 적게는 수백 개에서 많게는 수백만 개나 수천만 개 또는 그 이상으로 방대합니다. 이러한 데이터에 사람이 일일이 레이블을 부여하는 것은 때론 불가능할 수도 있습니다.

  그림 1.2는 지도학습 학습 데이터의 예로 유명한 Iris 데이터의 일부를 보여줍니다.

1	sepal length	sepal width	petal length	petal width	label
2	5.1	3.5	1.4	0.2	Iris-setosa
3	4.9	3	1.4	0.2	Iris-setosa
4	4.7	3.2	1.3	0.2	Iris-setosa
52	7	3.2	4.7	1.4	Iris-versicolor
53	6.4	3.2	4.5	1.5	Iris-versicolor
54	6.9	3.1	4.9	1.5	Iris-versicolor
102	6.3	3.3	6	2.5	Iris-virginica
103	5.8	2.7	5.1	1.9	Iris-virginica
104	7.1	3	5.9	2.1	Iris-virginica

그림 1.2 지도학습의 학습 데이터 샘플 (출처: https://archive.ics.uci.edu/ml/)

이 데이터는 네 개의 특징과 레이블로 구성됩니다. 각 데이터 인스턴스는 세 가지 레이블 "Iris-setosa", "Iris-versicolor", "Iris-virginica" 중에서 하나의 레이블을 부여받습니다.

비지도학습(unsupervised learning)

레이블이 없는 데이터를 학습하는 방법입니다. 주로 데이터를 그룹화하거나 데이터의 특징을 분석하기 위해서 사용합니다. 비지도학습은 정답이 없는 데이터를 분석하는 것이므로 데이터에 레이블을 부여할 필요가 없습니다. 즉 데이터를 준비하기 위한 비용이 적습니다. 그러나 분석된 결과인 군집들이 어떠한 의미를 가지는지는 사람이 개입하여 확인해야 하는 경우가 많습니다.

Iris 데이터에서 레이블(label)을 제외하면 그림 1.3과 같이 비지도학습으로 분석할 데이터가 됩니다. 이 데이터를 세 개의 그룹으로 (잘) 군집화하면 레이블은 모르지만 각 그룹이 "Iris-setosa", "Iris-versicolor", "Iris-virginica"를 의미하게 됩니다.

sepal length	sepal width	petal length	petal width	
5.1	3.5	1.4	0.2	그룹 1 (Iris-setosa)
4.9	3	1.4	0.2	
4.7	3.2	1.3	0.2	
7	3.2	4.7	1.4	그룹 2 (Iris-versicolor)
6.4	3.2	4.5	1.5	
6.9	3.1	4.9	1.5	
6.3	3.3	6	2.5	그룹 3 (Iris-virginica)
5.8	2.7	5.1	1.9	
7.1	3	5.9	2.1	

그림 1.3 비지도학습의 학습 데이터 샘플 (출처: https://archive.ics.uci.edu/ml/)

강화학습(reinforcement learning)

에이전트가 어떠한 환경에서 행동을 수행했을 때 보상을 함으로써, 에이전트는 그 보상을 최대로 하는 행동을 수행하도록 학습하게 하는 방법입니다. 주로 어떠한 행동을 결정하는 분류 문제나 보상을 예측하는 회귀 문제를 다룹니다. 강화학습은 레이블이 없는 데이터를 학습할 수 있지만 에이전트와 환경을 구성하는 추가적인 비용을 필요로 합니다.

이 책의 범위인 주식투자 시뮬레이션에서 강화학습을 사용하며, 2장과 3장에서 이 강화학습을 더 상세히 다룹니다.

1.2 딥러닝의 발전 과정

최근 주목받는 딥러닝의 기원이 된 기술들은 사실 오랜 역사를 역사를 가지고 있습니다. 이번 장에서는 딥러닝의 시초라 할 수 있는 퍼셉트론부터 발전된 형태의 인공 신경망을 거쳐 딥러닝으로 여겨지는 심층 신경망에 대해서 다룹니다.

1.2.1 퍼셉트론

퍼셉트론(perceptron)은 프랑크 로젠블라트(Frank Rosenblatt)가 1957년에 고안한 기초 형태의 인공 신경망입니다. 퍼셉트론의 구조는 그림 1.4와 같습니다.

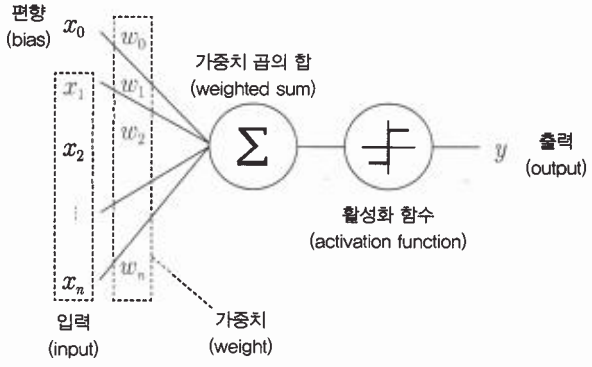


그림 1.4 퍼셉트론의 구조

퍼셉트론은 입력값(input)과 편향 값(bias)에 가중치(weight)를 곱하여 합한 값 즉, '가중치 곱의 합'이 임계치(threshold)를 넘어가면 1, 그렇지 않으면 0을 출력하는 활성화 함수(activation function)를 가집니다. 여기서 주목해야 할 점은 출력 값이 0 또는 1로 둘 중 하나라는 것입니다.

수식으로 표현해 보면 다음과 같습니다.

$$z = \sum_{i=0}^{n} w_i x_i \quad h(z) = \begin{cases} 1, & z > \text{임계치이면} \\ 0, & \text{그 외} \end{cases} = y$$

위에서 설명한 바와 같지만 수식에 쓰인 기호로 다시 설명해 보겠습니다. z는 가중치 곱의 합으로 입력값과 가중치를 곱하여 합한 값입니다. 함수 h는 활성화 함수로 z가 임계치보다 클 경우 1을 출력하고 그렇지 않으면 0을 출력합니다.

x_0은 편향 값을 의미하고 x_1부터는 각 입력값을 나타내는데, 이 둘 사이에 어떠한 차이가 있는지 알아보겠습니다. 입력값은 외부에서 퍼셉트론으로 들어오지만 편향값은 퍼셉트론을 만들 때 엔지니어(우리)가 정하는 값입니다. 편향을 두는 이유는 임계치를 0으로 만들기 위해서입니다. 임계치를 정하는 일이 어렵거나 불가능할 수 있으며 임계치가 0이면 활성화 함수 구현 또한 편해집니다. 편향을 1로 정해 주면 학습 과정에서 w_0이 (임계치 × −1)에 근사해집니다. 학습 과정은 뒤에서 다룰 오차 역전파 방법 부분에서 이어서 설명하겠습니다.

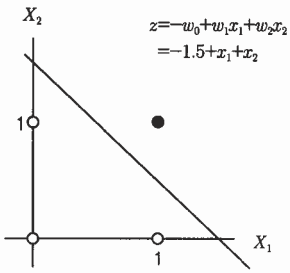
퍼셉트론에서 학습 대상은 w_0을 포함한 전체 가중치입니다. 가중치를 적절히 조절함으로써 입력값을 적절히 선형으로 분리하여 1과 0으로 출력하는 것이 퍼셉트론의 목적입니다.

간단한 예로 AND(논리합) 연산을 위한 퍼셉트론을 생각해 보겠습니다. AND 연산에 쓰이는 두 개의 입력값과 그 출력값은 표 1.1과 같습니다.

표 1.1 AND 연산의 입력과 출력

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

이를 좌표로 표시하고 x_1과 x_2가 모두 1일 때만 양수가 되도록 편향과 가중치를 정해 보면 그림 1.5와 같습니다.



$$h(z)=\begin{cases}1, & z>0\text{이면}\\ 0, & \text{그 외}\end{cases}=y$$

X_1	X_2	Z	Y
0	0	−1.5	0
1	0	−0.5	0
1	1	0.5	1

그림 1.5 AND 연산 퍼셉트론의 편향과 가중치

편향을 −1로 정하고 각 가중치 $\langle w_0, w_1, w_2\rangle$를 $\langle 1.5, 1, 1\rangle$로 정하면 가중치 곱의 합이 X_1과 X_2가 모두 1일 때만 양수이기 때문에 활성화 함수를 거치면 AND 연산의 결과인 Y를 얻을 수 있습니다.

퍼셉트론은 선형으로 입력값을 분리하는데, 이러한 특성 때문에 비선형으로 분류해야 하는 문제는 풀지 못합니다. 이러한 경우 퍼셉트론을 여러 층 쌓은 다층 퍼셉트론(multi-layer perceptron)을 사용하여 풀 수 있습니다. 다층 퍼셉트론은 인공 신경망의 구조와 크게 다르지 않기 때문에 여기서는 다루지 않겠습니다.

1.2.2 인공 신경망

인공 신경망(artificial neural network)은 다층 퍼셉트론에서 조금 더 발전된 모델입니다. 신경망은 그림 1.6과 같이 입력층(input layer), 은닉층(hidden layer), 출력층(output layer)으로 구성되어 있습니다.

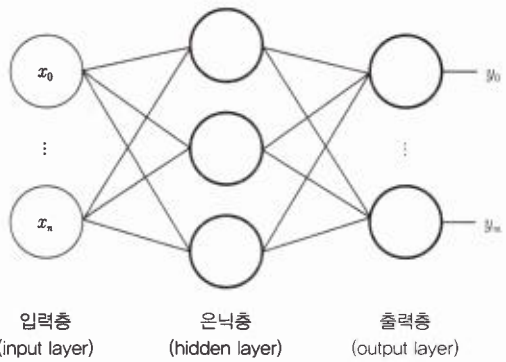


그림 1.6 인공 신경망의 구조

여기서 입력층의 노드는 편향을 포함한 입력값들을 표현합니다. 은닉층과 출력층의 진하게 표시한 노드들은 각기 퍼셉트론입니다.

다양한 활성화 함수 출현

인공 신경망에서는 은닉층에서 다양한 활성화 함수를 사용할 수 있습니다.

활성화 함수란 생체 신경 세포의 시냅스 소포를 모방한 것으로서, 전위가 일정치 이상 되면 소포가 터지면서 시냅스가 서로 화학적으로 연결이 되듯이, 활성화 함수로 들어오는 값이 일정치가 되었을 때 그 값이 다음 퍼셉트론으로 전달되게 하는 역할을 합니다. 다만 활성화 함수에 따라 다음 퍼셉트론으로 전달될 값이 변형될 수 있습니다.

이와 같은 활성화 함수의 종류에 따라서 신경망의 효율이 달라지기도 하는데, 이는 오차 역전파 시의 '기울기 소실' 문제와 관련이 있습니다. 다만, 이는 이 책의 범위를 벗어나므로 관심이 있는 분이라면 인공지능 관련 도서를 참고하시기 바랍니다.

여기서는 대표적인 활성화 함수인 계단 함수(step funtion), 렐루 함수(rectified linear unit function, ReLU), 리키렐루 함수(leaky rectified linear unit function Leaky, ReLU), 시그모이드 함수(sigmoid function), 쌍곡탄젠트 함수(hyperbolic tangent function, tanh function)를 살펴보겠습니다.

계단 함수

계단 함수는 앞서 퍼셉트론에서 다룬 활성화 함수입니다. 퍼셉트론에서의 계단 함수는 가중치 곱의 합이 0보다 작으면 0을, 0보다 크면 1을 반환하였습니다. 계단 함수에서 반환하는 값을 다르게 설정할 수 있습니다. 예를 들어 다음 수식의 계단 함수는 −1 또는 1을 반환합니다.

$$h(z) = \begin{cases} 1, & z > 0\text{이면} \\ -1, & \text{그 외} \end{cases}$$

계단 함수의 모양은 그림 1.7과 같습니다.

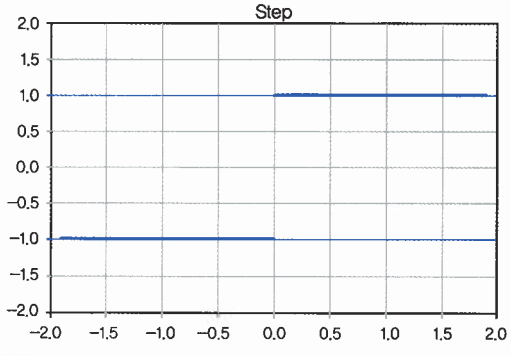


그림 1.7 계단 함수

렐루 함수

렐루 함수는 가중치 곱의 합이 0보다 크면 그 값을 그대로 반환하고 0보다 작으면 0을 반환하는 활성화 함수입니다. 이를 수식으로 표현하면 다음과 같습니다.

$$h(z) = \begin{cases} z, & z > 0\text{이면} \\ 0, & \text{그 외} \end{cases}$$

이 함수는 인공 신경망의 은닉층에서 많이 쓰이는 활성화 함수입니다. 그 모양은 그림 1.8과 같습니다.

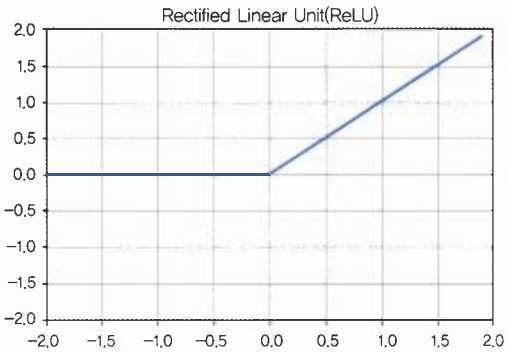


그림 1.8 렐루 함수

리키렐루 함수

렐루의 변형으로 리키렐루가 있습니다. 렐루와 거의 유사하지만 한가지 차이점은 가중치 곱의 합이 0보다 작을 때의 값도 약간 고려한다는 것입니다. 리키렐루의 수식은 다음과 같습니다.

$$h(z) = \begin{cases} z, z > 0\text{이면} \\ z \times \alpha, \text{ 그 외} \end{cases}$$

이때 α는 정하기 나름이지만 0.3 정도를 줄 수 있습니다. 그럼 그 모양이 그림 1.9와 같아집니다.

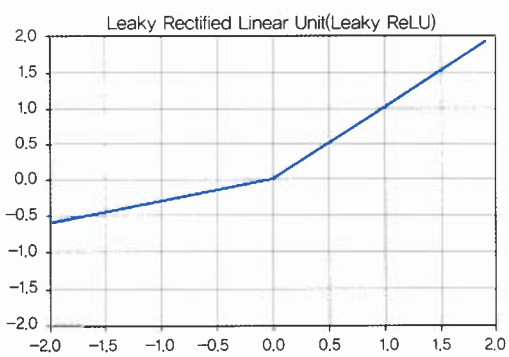


그림 1.9 Leaky ReLU 함수

시그모이드 함수

시그모이드 함수는 가중치 곱의 합을 0과 1 사이의 값으로 조정하여 반환하는 활성화 함수이고 그 모양이 S자와 닮았습니다. 시그모이드 함수는 로지스틱(logistic) 함수라고도 부르고 수식은 아래와 같습니다.

$$h(z) = \frac{1}{1+e^{-z}} = \frac{e^{z}}{e^{z}+1}$$

시그모이드 함수는 가중치 곱의 합이 0에서 양이나 음으로 커지면 반환하는 값이 급격하게 변하고 절대값이 2.5 이상일 때부터는 크게 변하지 않는 특징이 있습니다. 이 S자 모양을 변경할 수도 있으나 딥러닝에서는 일반적으로 있는 그대로 사용하므로 여기서는 다루지 않겠습니다.

시그모이드 함수의 모양은 그림 1.10과 같습니다.

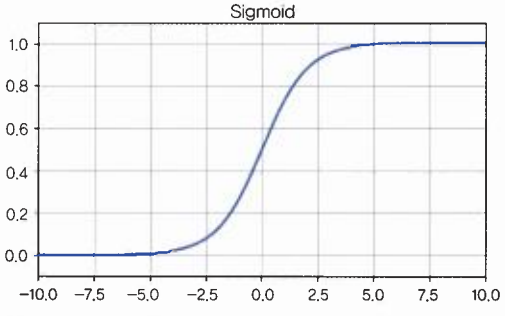


그림 1.10 시그모이드 함수

쌍곡탄젠트 함수

쌍곡탄젠트 함수(tanh 함수)는 시그모이드 함수와 유사한 모양을 가집니다. 쌍곡탄젠트 함수가 반환하는 값의 범위는 −1에서 1 사이의 값이며 가중치 곱의 합이 0에서 양이나 음으로 커질 때 시그모이드 함수보다 더욱 급격하게 변합니다. 쌍곡탄젠트 함수의 수식은 아래와 같습니다.

$$h(z) = \frac{\sinh z}{\cosh z} = \frac{\frac{1-e^{-2z}}{2e^{-z}}}{\frac{1+e^{-2z}}{2e^{-z}}} = \frac{1-e^{-2z}}{1+e^{-2z}}$$

쌍곡탄젠트 함수의 모양은 그림 1.11과 같습니다.

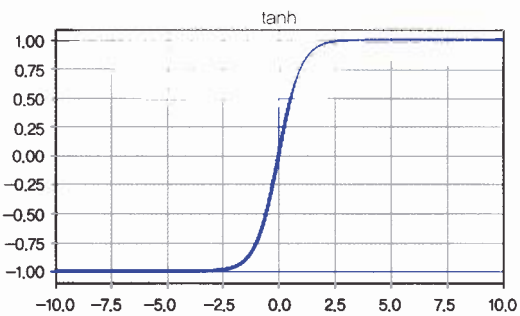


그림 1.11 tanh 함수

출력층에서 활성화 함수를 사용

출력층에서도 활성화 함수를 사용하는데, 일반적으로 시그모이드 함수 또는 소프트맥스(softmax) 함수를 사용합니다. 시그모이드 함수는 은닉층 활성화 함수에서 사용되는데 출력층에서 시그모이드 함수를 사용하는 이유는 이항 분류 문제에서 출력 값을 확률화시키기 위해서입니다. 시그모이드 함수가 0에서 1 사이의 값을 반환하기 때문에 여기에 100을 곱하면 확률로 사용할 수 있습니다.

소프트맥스 함수는 다항 분류 문제에서 출력 값을 확률화시키기 위해서 사용합니다. 소프트맥스 함수는 분류될 수 있는 각 결과, 즉 레이블(label)에 대한 입력값에 지수(exponential)를 취해 합하여 분모로 취함으로써 정규화(normalization)합니다. 수식은 다음과 같습니다.

$$h(z_x) = \frac{e^{z_x}}{\sum_{i=1}^{n} e^{z_i}}$$

예를 들어 사진을 입력으로 받아서 이 사진을 개, 고양이, 소, 말로 분류하는 문제에서 개, 고양이, 소, 말이 각각 레이블이 되고 이들에 대한 지수화 출력 값의 합을 분모로 취하여 나누어 줍니다. 만약 각 레이블에 대한 지수화 출력 값이 1.1, 1.1, 1.7, 1.2였을 때 1.7에 대한 소프트맥스 값은 $\frac{1.7}{1.1+1.1+1.7+1.2} \approx 0.33$이 됩니다.

1.2.3 심층 신경망

심층 신경망(deep neural network)은 은닉층이 2개 이상인 인공 신경망입니다. 그림 1.12는 은닉층이 2개인 심층 신경망의 구조를 보여줍니다. 은닉층을 더 많이 쌓을 수도 있습니다.

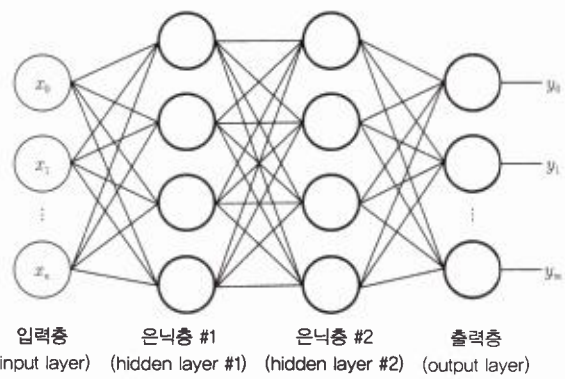


그림 1.12 심층 신경망의 구조

은닉층의 수를 제외하고는 인공 신경망과 같습니다. 은닉층 수를 많이 쌓음으로써 인공 신경망이 비선형 문제를 좀 더 잘 학습할 가능성이 높아집니다. 그러나 은닉층이 많아지면 학습에 필요한 컴퓨팅 비용이 높아지고 사람이 학습을 추적하기가 매우 어려워집니다.

딥러닝은 이러한 심층 신경망을 이용한 머신러닝을 의미한다고 볼 수 있습니다. 은닉층의 수를 제외하고는 심층 신경망은 인공 신경망과 같아서 인공 신경망에서 사용하는 활성화 함수들을 그대로 적용할 수 있습니다.

1.3 딥러닝에 필요한 핵심 기술

인공 신경망 학습을 위해 다양한 기법들이 연구되어 왔습니다. 그 중에서 주요 기법인 오차 역전파 기법, 최적해 탐색 기법, 과적합 해결 기법을 소개합니다.

1.3.1 오차 역전파 기법

오차 역전파 기법은 인공 신경망의 가중치 학습[2]에 사용되는 기법입니다. 가장 마지막 계층부터 앞쪽 계층으로 순전파(forward)된 가중치를 편미분한 값을 역전파(backward) 형태로 곱해 나가면서 그 최종 역전파 값을 가중치에 더해서 조정합니다. 표 1.2는 활성화 함수별 순전파 계산식과 역전파 계산식을 보여줍니다.

표 1.2 활성화 함수의 역전파

활성화 함수	순전파	역전파
계단 함수	$h(z)=\begin{cases}1, z>0\text{이면}\\-1, \text{그 외}\end{cases}$	$\frac{\partial y}{\partial z}=\begin{cases}0, z>0\text{이면}\\0, \text{그 외}\end{cases}=0$
렐루 함수	$h(z)=\begin{cases}z, z>0\text{이면}\\0, \text{그 외}\end{cases}$	$\frac{\partial y}{\partial z}=\begin{cases}1, z>0\text{이면}\\0, \text{그 외}\end{cases}$
시그모이드 함수	$h(z)=\frac{1}{1+e^{-z}}$	$\frac{\partial y}{\partial z}=y(1-y)$
쌍곡탄젠트 함수	$h(z)=\frac{1-e^{-2z}}{1+e^{-2z}}$	$\frac{\partial y}{\partial z}=1-y^2$
소프트맥스 함수	$h(z)=\frac{e^{z_i}}{\sum_{i=1}^{n}e^{z_i}}$	$\frac{\partial y_i}{\partial z_i}=y_i-t_i$ (t_i는 i번째 레이블)

계단 함수의 순전파를 미분하면 모두 0이 되어 가중치에 영향을 주지 않습니다. 렐루 함수는 가중치 곱의 합이 양수일 때만 뒤쪽 계층에서 넘어온 역전파를 그대로 넘깁니다. 시그모이드

2 인공지능에서의 학습이란 곧 가중치를 조율하는 일에 다름 아니어서, 가중치 조율을 통한 인공지능의 학습 과정을 간단히 '가중치 학습'이라고 부를 수 있습니다.

함수와 쌍곡탄젠트 함수는 순전파 값에만 영향이 있습니다. 소프트맥스 함수에서 i번째 레이블에 대한 역전파 값은 i번째 레이블에 대한 순전파 값의 i번째 레이블 값(정답 값)이 됩니다.

사실 오차 역전파를 깊이 이해하려면 더 자세한 설명이 필요합니다. 그러나 이 책의 범위를 벗어나므로 깊이 있는 설명에 한계가 있습니다. 이 책에서 다루는 강화 학습 방식 주식 투자에서 오차 역전파를 자세히 이해할 필요는 없지만, 궁금하신 분들은 사이토 고키의 저서, ≪밑바닥부터 시작하는 딥러닝≫(한빛미디어)을 참고하시길 추천드립니다.

1.3.2 최적해 탐색 기법

신경망 학습은 신경망의 가중치를 반복해서 갱신하는 과정입니다. 학습 데이터는 입력 데이터와 레이블(정답 출력)로 구성되어 있습니다. 학습을 통해 입력 데이터에 대한 신경망의 출력과 레이블과의 차이를 줄여 나갑니다. 이러한 차이를 비용(cost) 또는 손실(loss)이라고 합니다. 앞으로는 손실로 용어를 통일합니다.

손실 함수(loss function)는 입력 데이터와 레이블로 손실을 구하는 함수입니다. 일반적인 손실 함수를 나타낸 수식은 다음과 같습니다.

$$L(w) = \frac{1}{m}\sum_{i=1}^{m}(h(z_i) - t_i)^2$$

w는 신경망의 가중치이며 z는 일련의 입력 데이터의 가중치 곱의 합이고 t는 일련의 레이블입니다. m은 z와 t의 크기입니다. 정책 신경망의 출력과 레이블을 뺀 값에 제곱을 취한 값의 평균이 손실입니다. 이러한 계산 방식을 오차제곱합(sum of squared error, SSE)이라고 합니다.

다시 신경망 학습을 정의해 보면 학습 데이터에 대한 신경망의 손실을 최소화하는 과정이라고 할 수 있습니다.

대표적인 신경망 학습 기법으로 경사 하강법(gradient descent, GD)을 들 수 있습니다. 본 장에서는 경사 하강법과, 정확성을 조금 양보하면서 효율을 높인 확률적 경사 하강법(stochastic gradient descent, SGD)을 소개합니다.

- **경사 하강법**

  경사 하강법은 손실 함수의 기울기에 학습 속도(learning rate)를 곱하여 가중치에서 빼 줌으로써 가중치를 갱신합니다. 수식으로 표현하면 다음과 같습니다.

$$w \leftarrow w - \alpha \frac{\delta}{\delta w} L(w)$$

  여기서 w는 가중치이며 w에 손실 함수의 기울기와 학습 속도 α를 곱한 값을 빼 줌으로써 갱신합니다. 이 과정을 계속 반복했을 때 손실이 줄어들어야 하며 손실에 변화가 거의 없어질 때에는 학습이 더 이상 이루어지지 않는다고 보고 학습을 종료합니다.

- **확률적 경사 하강법**

  확률적 경사 하강법은 경사 하강법과 학습 방법이 동일하지만 학습 수행 시점과 그 양에서 차이가 있습니다. 경사 하강법은 학습 데이터에 포함된 샘플 하나 하나를 가중치 갱신에 사용하는 반면, 확률적 경사 하강법은 미니 배치(mini batch)라고 하는 학습 데이터의 부분 집합을 한번의 가중치 갱신에 사용합니다.

  확률적 경사 하강법은 경사 하강법에 비해 학습량이 적으며 그만큼 수행 속도(performance)가 빠릅니다. 대신 샘플 하나 하나로 가중치를 갱신하지 않는 만큼 학습 정확도는 조금 떨어질 수 있습니다.

  그러한 이유로 학습 데이터가 작으면 경사 하강법을, 학습 데이터가 크면 확률적 경사 하강법을 사용하는 것이 좋습니다. 최근 대부분의 신경망 학습이 방대한 양의 학습 데이터를 사용하는 추세이고, 이에 따라 확률적 경사 하강법이 보편적으로 많이 사용되고 있습니다.

1.3.3 과적합 해결 기법

과적합(overfitting)은 학습 데이터를 과도하게 학습해서 신경망의 일반성(generality)을 떨어뜨리는 결과를 초래하는 현상을 의미합니다. 머신러닝에서 과적합을 잘 피해야 제대로 된 학습을 할 수 있습니다.

그림 1.13는 일반화된 모델과 과적합된 모델의 차이를 보여줍니다.

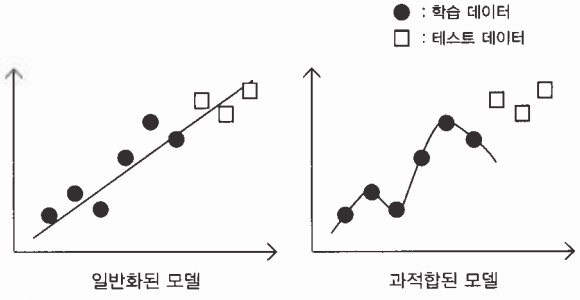


그림 1.13 일반화된 모델과 과적합된 모델

일반화된 모델은 학습 데이터의 패턴을 잘 찾아내서 테스트 데이터에도 잘 적용이 되지만, 과적합된 모델은 각각의 학습 데이터에 과도하게 맞춰서 테스트 데이터에는 작 적용되지 않습니다.

과적합을 피하는 방법은 여러 가지가 있으며 본 장에서 정규화(regularization)와 드롭아웃(dropout)이라는 대표적인 과적합 회피 기법 두 가지를 소개합니다.

- **정규화**

  정규화는 손실 함수에 람다(lambda) 상수와 가중치 제곱의 합을 곱하여 더하는 방법입니다.

$$L(w) = \frac{1}{m}\sum_{i=1}^{m}(h(z_i) - t_i)^2 + \lambda \sum w^2$$

  손실값을 줄이는 방향으로 학습이 진행됨과 동시에 가중치의 값도 줄여 나가는 것입니다.

- **드롭아웃**

  드롭아웃은 신경망의 일부 뉴런을 생략하고 학습을 진행하는 방법입니다. 간단하면서도 강력한 과적합 회피 방법으로 많이 사용되고 있습니다. 그림 1.14는 일부 뉴런을 드롭아웃한 상태를 보여 줍니다.

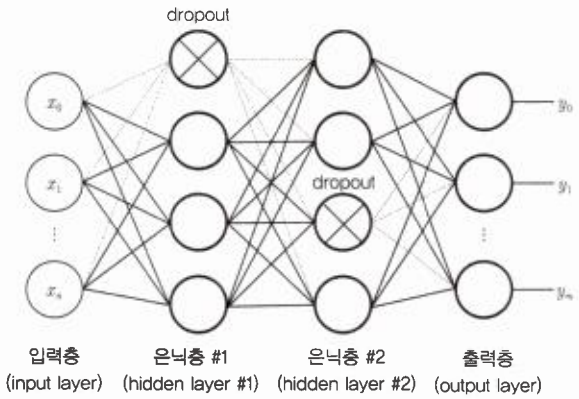


그림 1.14 신경망 드롭아웃

이렇게 드롭아웃이 된 뉴런에서 발생될 출력을 다음 레이어에 넘겨주지 않음으로써 과도하게 학습 데이터에 맞춰지지 않도록 할 수 있습니다.

1.4 고급 인공 신경망 구조

인공 신경망은 특정 데이터를 더 잘 처리하기 위해서 다양한 형태로 발전되어 왔습니다. 대표적으로 순환 신경망(recurrent neural network)과 합성곱 신경망(convolutional neural network)이 있습니다. 이것들을 간단히 소개하고 넘어가겠습니다.

1.4.1 순환 신경망

순환 신경망은 그림 1.15와 같이 은닉층에서의 출력 값을 다음 학습 때 다시 입력에 추가하는 형태의 인공 신경망입니다. 즉 이전의 상태를 기억하여 학습에 반영하는 것입니다.

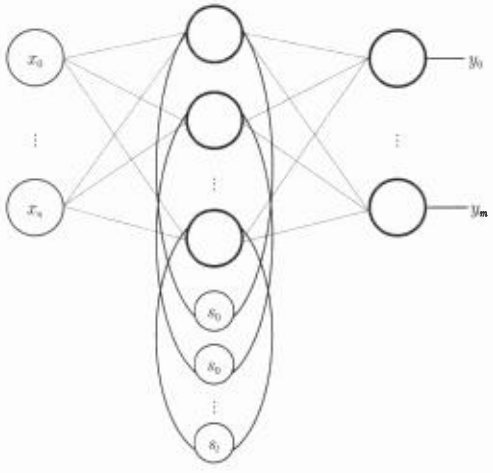


그림 1.15. 순환 신경망의 기본 구조

순환 신경망은 이전의 상태를 기억하고 있어서, 순서가 중시되는 데이터를 학습하기 위해 주로 사용됩니다. 이러한 특성에 의해 순환 신경망은 음성, 문장 등의 순차적(sequential) 데이터를 학습하는 데 주로 사용됩니다. 대표적인 분야로 음성 인식, 번역 등이 있습니다.

1.4.2 LSTM 신경망

LSTM 신경망은 RNN의 일종으로 이전 상태들을 더 잘 기억할 수 있도록 개선된 신경망입니다. LSTM 신경망은 그림 1.16과 같이 LSTM 유닛으로 구성됩니다.

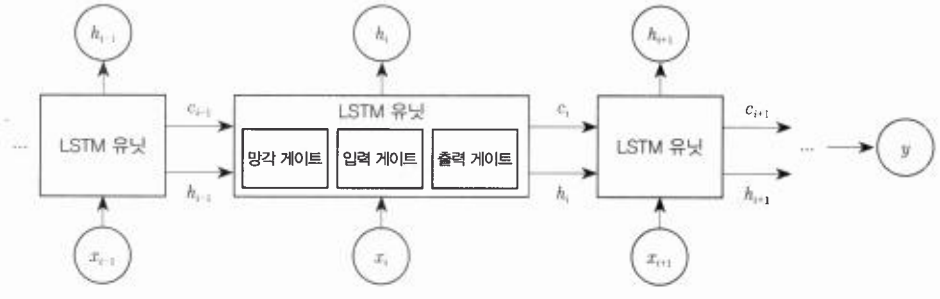


그림 1.16 LSTM 유닛

LSTM 유닛에는 망각 게이트(forget gate), 입력 게이트(input gate), 출력 게이트(output gate)라는 세 가지 게이트가 있습니다. 이들 게이트는 시그모이드 함수를 통해 잊을 값, 받아들일 값, 내보낼 값을 정합니다.

c_i는 i번째 셀 상태(cell state), h_i는 i번째 유닛의 은닉 상태(hidden state)입니다. 셀 상태는 크게 변형되지 않고 그 다음 유닛으로 전파가 쉽게 이루어지기 때문에 LSTM 신경망이 이전 상태들을 더 잘 기억할 수 있습니다.

1.4.3 합성곱 신경망

합성곱 신경망은 이미지 데이터 처리에 주로 사용되는 심층 신경망입니다. 컨볼루션(convolution) 계층과 풀링(pooling) 계층을 이용하여 입력 데이터의 분량을 줄이면서도 복잡한 특징을 추출합니다. 그림 1.17은 합성곱 신경망의 구조의 예를 보여줍니다.

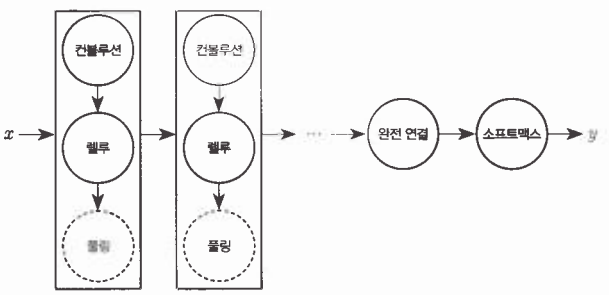


그림 1.17 합성곱 신경망 구조의 예

기존 인공 신경망과의 가장 큰 차이점은 입력 계층 다음에 컨볼루션, 렐루, 풀링 계층이 처리되고 이어서 완전 연결(fully connected) 계층이 이어지고 최종적으로 소프트맥스 계층을 거쳐서 결과가 출력된다는 것입니다.

컨볼루션 계층은 입력 데이터에 컨볼루션 필터를 적용합니다. 그림 1.16과 같이 가중치 행렬인 컨볼루션 필터를 이미지의 각 부분에 순차적으로 내적합니다.

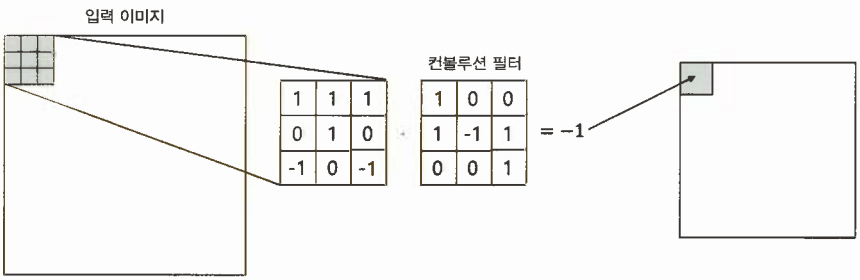


그림 1.18 컨볼루션 필터 적용의 예

컨볼루션 필터 적용의 의의는 이미지의 인접한 공간이 가중치를 공유하고 입력 데이터의 크기를 줄이는 데 있습니다.

풀링 계층은 입력 데이터의 크기를 줄이는 역할을 합니다. 예를 들어 그림 1.19와 같이 인접한 4개의 데이터를 하나의 데이터로 줄일 수 있습니다.

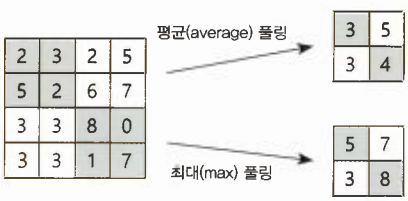


그림 1.19 평균 풀링과 최대 풀링

평균 풀링은 인접한 데이터들의 평균을 고려하고 최대 풀링은 인접한 데이터들 중에서 최대값만을 고려하여 데이터의 크기를 줄이는 기법입니다.

완전 연결 계층은 일반 인공 신경망에서 쓰이는 가중치 합의 곱을 구하는 계층으로 그 내용을 앞에서 이미 다루었기 때문에 설명을 생략합니다.

1.5 딥러닝 적용 사례

딥러닝은 다양한 분야에서 이미 사용되어오고 있습니다. 대표적인 분야로 기계 번역, 음성 인식, 이미지 인식을 들 수 있습니다.

1.5.1 기계 번역

기계 번역은 기존에는 주로 통계 기반 번역(statistical machine translation, SMT)에 의존해 왔습니다. 그러나 최근에는 인공 신경망 기반 번역(neural machine translation, NMT)을 적용하고 있습니다. SMT에서는 전체 문장에 대한 맥락을 이해하기가 어려웠는데, 이러한 맹점을 NMT로 해결할 수 있습니다. NMT는 문장의 단어, 단어들의 순서 등을 분석하여 번역의 품질을 크게 높입니다.

대표적인 NMT 서비스로 구글 번역과 네이버 파파고를 들 수 있습니다. 두 서비스 모두 훌륭한 번역 결과를 보여줍니다. NMT를 사용할 때 번역의 성능은 학습 데이터가 좌우하는데, 일부 어려운 한국어 문장 번역에서 파파고가 더 나은 번역 결과를 보여주기도 합니다.

'아버지가방에들어가신다'는 띄어쓰기를 하지 않았기 때문에 '아버지가 방에 들어가신다', '아버지 가방에 들어가신다'와 같이 다양한 해석이 가능합니다.

그림 1.20과 같이 구글은 '아버지 가방에 들어가신다'로 해석하여 'He goes into his father's bag'으로 해석하고 있습니다. NMT 모델이 계속 개선될 것이기 때문에 이 결과는 바뀔 수 있습니다.

그림 1.20 구글 번역(https://translate.google.co.kr/)

반면 그림 1.21과 같이 네이버 파파고는 '아버지가 방에 들어가신다'로 해석하여 'Father is entering the room'으로 번역하고 있습니다. 이 결과 역시 NMT 모델이 지속적으로 개선되고 있을 것이기 때문에 바뀔 수 있습니다.

그림 1.21 네이버 파파고(https://papago.naver.com/)

1.5.2 음성 인식

음성 인식은 음성 오디오 데이터를 텍스트로 변환하는 머신러닝 문제이고 스피치투텍스트(speech to text, STT)라고도 합니다. 음성 인식은 구글, 애플, 마이크로소프트, IBM 등 많은 기업들이 오랫동안 연구하고 서비스로도 제공하고 있습니다. 특히 애플은 모바일 비서 시스템인 시리(Siri)로 높은 음성 인식 기술력을 보여주었습니다.

음성 인식은 주로 순환 신경망을 사용합니다. 그 이유는 앞서 설명한 것과 같이 언어의 특성 때문입니다. 언어는 전후 맥락(context)이 중요한데 순환 신경망이 이러한 특징을 학습에 잘 반영할 수 있는 구조이기 때문입니다.

다양한 음성 인식 API들이 존재합니다.

- Google Speech API: 유료, 구글에서 제공하는 STT API, 매우 다양한 언어를 지원
- Bing Speech API: 제한적 무료, 마이크로소프트에서 제공하는 STT API로 스트리밍뿐 아니라 오디오 파일을 전송하여 텍스트로 변환 가능

- Watson Speech API: 제한적 무료, IBM에서 제공하는 STT API
- Clova Speech Recognition API: 제한적 무료, 네이버에서 제공하는 STT API

1.5.3 이미지 인식

흔히 Full HD라고 부르는 해상도는 1920×1080 픽셀입니다. Full HD 이미지를 데이터로 처리하고자 하면 그 배열의 크기는 얼마나 될까요? 흑백일 경우 2,073,600개의 크기를 가지는 배열이 될 것이고, RGB(red, green, blue) 이미지의 경우 흑백 크기의 3배인 6,220,800가 됩니다. 32×32 픽셀로 이뤄진 작은 이미지라고 해도 3,072개 크기의 배열이 필요합니다. 이미지 인식은 수많은 이미지를 학습하여 이미지를 분류해 내는데, 1,000개의 32×32 RGB 이미지가 있다고 할 때 그 데이터의 크기는 3,072,000개가 됩니다. 이렇게 학습 데이터의 크기가 커서 방대한 컴퓨팅 연산을 필요로 하여 이미지 인식은 머신러닝의 어려운 분야 중 하나입니다.

또한 이미지가 무엇을 나타내는 이미지인지를 기계가 학습 과정에서 알게 하기 위해 사람이 이미지에 레이블을 달아 주어야 하는데 이는 많은 시간과 노력이 필요하여 이미지 인식 분야의 걸림돌 중에 하나였습니다.

이미지 인식이 빠르게 발전할 수 있었던 이유는 이러한 어려움들이 점점 극복되어 가고 있기 때문입니다. CPU와 GPU의 발전, 합성곱 신경망 기술의 발전으로 방대한 크기의 학습 데이터를 효과적으로 학습할 수 있게 되었습니다. 그리고 학습에 사용할 수 있는 이미지들도 많이 생겼습니다. 그 대표적인 예로 ImageNet이라는 이미지 데이터베이스가 있습니다. 그림 1.22는 ImageNet의 탐색창을 보여줍니다. 2010년 4월 30일 기준 14,197,122개의 사람이 레이블을 입력해 놓은 이미지가 ImageNet에 저장되어 있었다고 알려져 있으니 현재는 그 이상의 데이터가 저장되어 있을 것입니다.

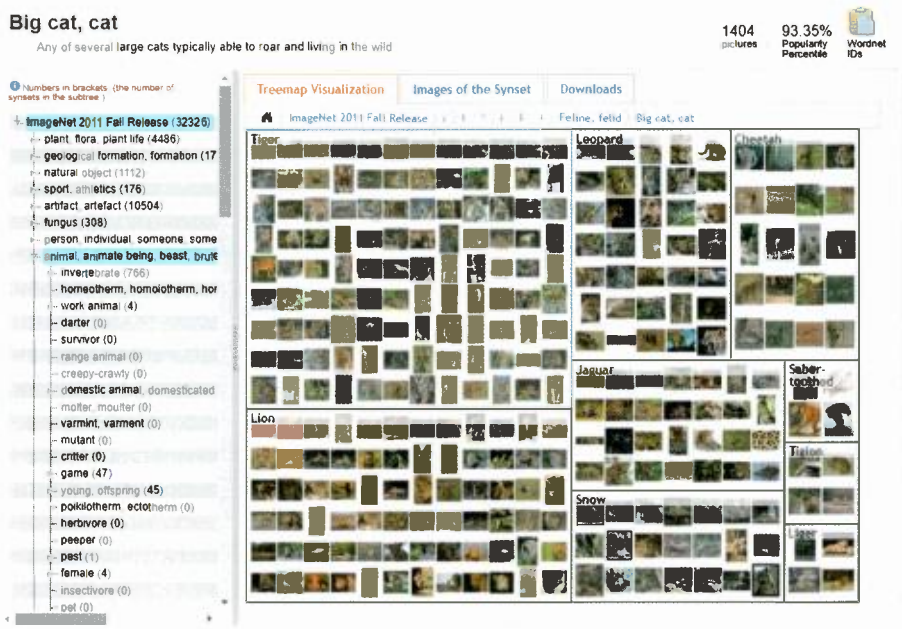


그림 1.22 이미지 데이터베이스인 ImageNet의 탐색 창

이미지 인식을 실제로 서비스하는 예로 구글의 이미지 검색, 네이버의 스마트렌즈 검색 등을 들 수 있습니다. 그리고 최근 활발히 연구 중인 자율주행 자동차 분야에도 카메라로 찍은 이미지에서 사물을 실시간으로 인지하는 기술이 포함됩니다.

1.6 이번 장의 요점

- 딥러닝은 오랜 역사를 가진 머신러닝 분야에 속한 기술입니다.
- 1950년대부터 인공 신경망 연구가 시작되었지만 2010년도에 아주 크게 주목받았습니다.
- 딥러닝으로 분류(classification), 군집화(clustering), 회귀(regression) 문제를 풀 수 있습니다.
- 딥러닝의 핵심 기술들로 오차 역전파, 최적해 탐색 기법, 과적합 해결 기법을 살펴봤습니다.
- 고급 인공 신경망의 구조들, 딥러닝 적용 사례들을 다뤄서 딥러닝에 대한 이해를 더했습니다.

02장

배경 이론 2: 강화학습이란?

강화학습(reinforcement learning)은 머신러닝 기법 중 한 가지로서, 어떠한 환경에서 어떠한 행동을 했을 때 그것이 잘된 행동인지 잘못된 행동인지를 나중에 판단하고 보상(또는 벌칙)을 줌으로써 스스로 시행착오를 해 가며 학습하게 하는 분야입니다.

강화학습에는 그림 2.1과 같이 두 가지 구성 요소로 환경과 에이전트가 있습니다.

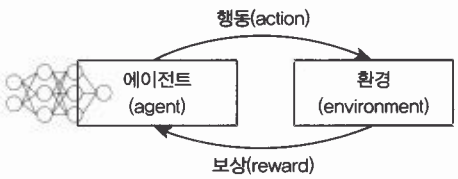


그림 2.1. 강화학습의 개념도

에이전트는 특정 환경에서 행동을 결정하고 환경은 그 결정에 대한 보상(reward)을 내립니다. 이 보상은 행동 즉시 결정되기보다는 여러 행동들을 취한 후에 한꺼번에 결정되는 경우가 많습니다. 특정 행동을 취했을 때 바로 그 행동에 대한 평가를 내릴 수 없는 경우가 많기 때문입니다.

강화학습은 앞서 다룬 딥러닝과 밀접한 관계가 있습니다. 에이전트가 행동을 결정하고 환경이 주는 보상으로 스스로 학습할 때 주로 딥러닝에서 다룬 인공 신경망을 사용합니다. 환경과 에이전트의 상태 등을 입력값으로 인공 신경망이 행동을 결정하고 보상이 있으면 이전의 입력값과 행동들을 긍정적으로 학습합니다.

2.1 강화학습의 기초가 된 마르코프 의사결정 과정

강화학습은 마르코프 의사결정 과정(Markov decision process, MDP)에 학습의 개념을 넣은 것이라 할 수 있습니다. 그러므로 MDP에 대해 잘 이해하는 것이 강화학습 시스템 개발에 있어 중요합니다. 그럼 MDP를 포함하여 강화학습의 주요 이론을 살펴보겠습니다.

2.1.1 마르코프 가정

마르코프 가정(Markov assumption)은 상태가 연속적인 시간에 따라 이어질 때 어떠한 시점의 상태는 그 시점 바로 이전의 상태에만 영향을 받는다는 가정입니다. 현재의 상태가 바로 이전 상태에만 영향을 받는다고 가정하는 것이 어떻게 보면 무리한 가정일 수 있습니다. 그러나 직관적으로 생각해 보면 현재 상태는 이전 상태에 가장 큰 영향을 받고, 이전 상태는 그 이전 상태에 큰 영향을 받고, 더 이전의 상태들도 이러한 식으로 그보다 이전 상태에 가장 큰 영향을 받으므로 그래도 합리적인 가정이라고 볼 수 있습니다. 결정적으로 마르코프 가정으로 어려운 문제들을 단순화하고 만족스러운 결과를 얻는 경우가 많습니다.

마르코프 가정의 수식은 다음과 같습니다.

$$P(x_t \mid x_1, x_2, \cdots, x_{t-1}) = (x_t \mid x_{t-1})$$

위 수식에서 좌변은 어떠한 시점 t에서의 상태 x_t는 최초의 상태 x_1에서 바로 이전의 상태 x_{t-1}까지에 영향을 받는다는 뜻입니다. 이는 연속적으로 존재하며 일어나는 일련의 상태들을 확실하게 표현하지만 이를 실제로 계산하기는 매우 어렵습니다. 그래서 상태 x_t는 바로 이전 상태인 x_{t-1}에 가장 큰 영향을 받고 x_{t-1}에서 x_{t-2}를 반영하는 등 연쇄적으로 모든 이전 상태들이 반영된다고 가정하는 마르코프 가정을 적용하면 위 수식의 우변과 같이 단순화할 수 있습니다.

2.1.2 마르코프 과정

마르코프 과정(Markov process)은 마르코프 가정을 만족하는 연속적인 일련의 상태입니다. 마르코프 과정은 일련의 상태 $<x_1, x_2, \cdots, x_n>$와 상태 전이 확률(state transition probability) p로 구성됩니다.

상태 전이 확률 $p_{ij}=P(x_{t+1}=j|x_t=i)$은 어떠한 상태가 일 때 그 다음 상태가 j가 될 확률을 의미합니다. 여기서 $P(A|B)$는 조건부 확률로 B일 때 A일 확률을 의미합니다. 마르코프 가정을 반영했기 때문에 이렇게 계산 가능한 상태 전이 확률을 정의할 수 있는 것입니다.

2.1.3 마르코프 의사결정 과정

마르코프 의사결정 과정(Markov decision process, MDP)은 마르코프 과정을 기반으로 한 의사결정 모델입니다. MDP는 아래 수식과 같이 상태(state) 집합 S, 행동(action) 집합 A, 상태 전이 확률(state transition probability) 행렬 P, 보상(reward) 함수 R, 할인 요인(discount factor) γ로 구성되어 있습니다.

$$MDP = (S,A,P,R,\gamma)$$

상태 집합은 MDP에서 가질 수 있는 모든 상태의 집합 $\{s_1,s_2,\cdots,s_{|S|}\}$입니다. 아래 식과 같이 어떠한 시점에서의 상태 x_t는 상태 집합 S에 포함된 특정 상태가 됩니다.

$$x_t = s, s \in S$$

행동 집합은 행동 주체인 에이전트가 할 수 있는 모든 행동들의 집합 $\{a_1,a_2,\cdots,a_{|A|}\}$입니다. 에이전트는 어떠한 시점에서 행동 a를 취하는 것입니다.

MDP에서의 상태 전이 확률은 마르코프 과정의 상태 전이 확률보다 조금 더 복잡합니다. MDP에서의 상태 전이 확률 수식은 아래와 같습니다.

$$P^{a}_{s_i,s_j} = P(x_{t+1} = s_j \mid x_t = s_i, A_t = a)$$

$P^{a}_{s_i,s_j}$는 에이전트가 어떠한 상태 s_i에서 행동 a를 취했을 때 상태 s_j로 변할 확률입니다.

보상 함수는 에이전트가 어떠한 상태에서 취한 행동에 대한 보상을 내리기 위한 함수입니다. 그 수식은 아래와 같습니다.

$$R^{a}_{s}: s, a \rightarrow \mathbb{R}$$

보상 함수 R^{a}_{s}는 상태 s에서 행동 a를 했을 때의 보상을 수치로 반환합니다.

할인 요인은 과거의 행동들을 얼마나 반영할지를 정하는 값으로 0에서 1 사이의 값입니다. 과거 5번의 행동에 대한 보상을 1씩 받았다고 했을 때 할인 요인 γ이 1이면 〈1, 1, 1, 1, 1〉

이 되고 할인 요인 γ이 0.9이면 〈1, 0.9, 0.81, 0.729, 0.6561〉이 됩니다. 즉 먼 과거에 대한 보상일수록 깎아서 반영하는 것입니다.

그림 2.2와 같은 MDP의 간단한 예로 위의 설명들을 정리해 보겠습니다.

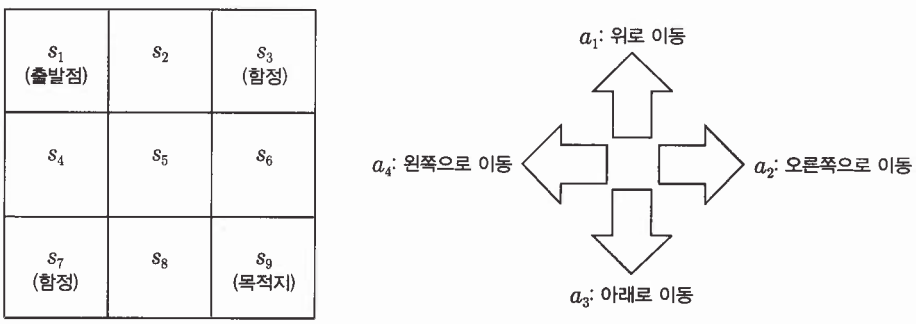


그림 2.2 MDP 예제의 상태 집합과 행동 집합

이 예제는 에이전트가 위, 오른쪽, 아래, 왼쪽으로 움직이면서 출발점에서 목적지까지 함정을 피하면서 도착하는 것을 목표로 합니다.

에이전트는 상태 전이 확률 $P^{a}_{s_i,s_j}$에 따라서 행동을 결정하고 함정에 다다르면 −0.1을 보상으로 주고 목적지에 다다르면 0.1을 보상으로 주어 이 상태 전이 확률을 조정해 나갑니다.

출발점 s_1에서 a_2, a_2를 취하여 s_3이 되면 −0.1을 보상으로 주는데, 이때 다음과 같이 $P^{a_2}_{s_1,s_2}$와 $P^{a_2}_{s_2,s_3}$의 값을 할인 요인 0.9를 곱하여 줄여서 다시 똑같이 함정에 빠지지 않는 방향으로 학습시킵니다.

$$P^{a_2}_{s_1,s_2} := P^{a_2}_{s_1,s_2} \times (1 - 0.1 \times 0.9)$$
$$P^{a_2}_{s_2,s_3} := P^{a_2}_{s_2,s_3} \times (1 - 0.1)$$

s_2에서 s_3으로 이동한 것이 가장 최근 행동이기 때문에 보상을 그대로 주고 그 전의 행동인 s_1에서 s_2로 이동한 것은 할인 요인을 곱하여 보상을 반영합니다. 함정이 아니라 목적지에 다다르게 되면 취했던 행동들의 확률 값을 높이게 됩니다.

이런 식으로 경험을 쌓다 보면 상태 전이 확률이 함정을 피하면서 목적지로 갈 수 있도록 조정되게 됩니다.

2.2 주요 강화학습 기법

강화학습은 MDP의 개념과 큰 차이는 없습니다만 복잡한 문제를 스스로 풀기 위해서 다음과 같은 특징을 가집니다.

- 상태 집합이 (거의) 무한하다.
- 상태 전이 확률을 인공 신경망으로 구한다.

앞서 설명한 MDP에서는 상태가 불연속적(discrete)이었습니다. 그러나 강화학습을 적용하는 대다수 문제는 상태가 연속적(continuous)이어서 가능한 상태의 수가 매우 많습니다. 예를 들어 두 개의 실수로 구성할 수 있는 공간은 $\mathbb{R}^2$으로 무한합니다. 상태를 구성하는 요소 개수가 많아질수록 그 공간은 방대해지는 것입니다. 그래서 상태 전이 확률을 일일이 정의하는 것이 불가능하므로 인공 신경망을 사용하는 것입니다.

주요 강화학습 기법으로는 Q 러닝과 정책 경사(policy gradient)가 있습니다.

2.2.1 Q 러닝 강화학습

Q 러닝은 회귀(regression) 문제를 푸는 강화학습으로 볼 수 있습니다. 어떠한 상태에서 어떠한 행동을 했을 때의 예상 가치(value)를 상태-행동 가치(state-action value)라고 부릅니다. Q 러닝은 이 상태-행동 가치가 가장 높은 행동을 취합니다. 복잡한 문제에서 상태 공간이 매우 크기 때문에 상태-행동 가치를 인공 신경망으로 모델링합니다.

2.2.2 정책 경사 강화학습

정책 경사는 분류(classification) 문제를 푸는 강화학습으로 볼 수 있습니다. 어떠한 상태에서 어떠한 행동을 결정하는 것이 가장 좋을지를 판단하는 것입니다. 정책 경사도 어떠한 상태에서의 행동에 대한 확률 값들을 인공 신경망으로 모델링합니다.

2.3 강화학습 적용 사례

현재까지의 강화학습은 주로 게임 인공지능에서 사용되어 왔습니다. 그 사례로 벽돌 깨기와 유명한 알파고가 있습니다. 각 사례에서 에이전트, 환경, 인공 신경망의 입력과 결정될 행동의 예를 들어 보겠습니다.

2.3.1 벽돌 깨기

구글의 계열사인 딥마인드(DeepMind)에서 2013년도에 발표한 〈Playing Atari with Deep Reinforcement Learning〉라는 논문에서 Q 러닝의 일종인 DQN(deep Q network)을 공개하였습니다. 이 논문에서 발표한 DQN으로 강화학습하여 벽돌 깨기 게임에서 사람보다 뛰어난 점수를 달성했습니다.

벽돌 깨기에서 환경은 그림 2.3과 같이 남은 벽돌, 공 위치, 바닥 위치 등이 될 수 있습니다. 그리고 에이전트는 바닥을 움직이는 역할을 합니다. 공이 움직이기 때문에 환경이 계속 변하는데, 연속된 환경을 입력 데이터로 인공 신경망에서 행동을 결정합니다. 행동은 가만히 있기, 왼쪽이나 오른쪽으로 움직이기가 될 수 있습니다. 여기서 보상은 게임이 끝났을 때의 점수가 될 수 있습니다. 처음에는 마구잡이로 게임을 진행하다가 운으로 점수가 높은 상황이 되면 그때 수행한 행동들은 긍정적으로 학습될 것입니다. 이런 식으로 수천 수만 번 반복하면 인공 신경망이 똑똑해져서 점수를 많이 획득하게 될 것입니다.

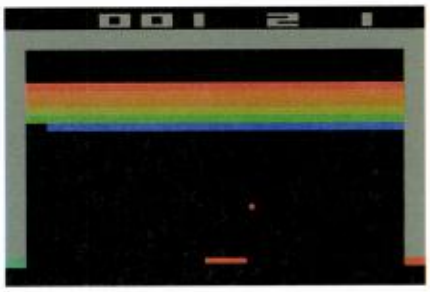


그림 2.3. 벽돌 깨기 게임

2.3.2 알파고

알파고는 구글 딥마인드에서 개발한 바둑 인공지능 프로그램입니다. 알파고는 2016년 3월 한국의 프로 9단 이세돌과의 대국으로 세간의 주목을 끌었습니다. 그림 2.4는 이 세기의 대국의 중계 화면입니다. 결과는 5전 4승 1패로 알파고의 승리였습니다.

그림 2.4 알파고와 이세돌 9단의 대국

1997년 IBM의 딥블루(Deep Blue)라는 인공지능 체스 프로그램이 체스 챔피언을 이길 때만 해도 바둑 인공지능은 불가능한 영역이라 여겨졌습니다. 그 이유는 바둑돌을 두는 경우의 수가 무한하기 때문입니다. 게다가 바둑은 돌을 계속 두기만 하는 것이 아니라 돌을 잡아서 빼내기도 합니다. 그러나 알파고의 등장으로 이러한 생각은 무너지게 되었습니다.

CPU, GPU의 비약적인 발전으로 컴퓨터의 연산 능력이 매우 좋아졌고 병렬 처리 기술 또한 발전하여 컴퓨터의 연산 효율을 크게 늘릴 수도 있게 되었습니다. 바둑 게임의 오랜 역사 동안 쌓인 수많은 기보를 학습 데이터로 사용할 수 있었습니다. 즉 딥러닝이 가능해진 것입니다.

이렇게 탄생한 알파고는 강화학습 기법으로 바둑 인공지능 프로그램끼리 서로 대국하여 지능을 높인 것으로 알려져 있습니다. 즉 알파고가 또 다른 알파고와 대국을 하면서 바둑의 수를 공부한 것입니다.

2.4 이번 장의 요점

- 강화학습의 기초 이론인 마르코프 가정, 마르코프 과정, 마르코프 의사결정 과정을 살펴보았습니다.
- 강화학습은 마르코프 의사결정 과정에 학습의 개념을 넣은 것이라 할 수 있습니다.
- 강화학습으로 복잡한 문제를 스스로 풀게 할 수 있습니다.
- 주요 강화학습 기법인 Q 러닝과 정책 경사 강화학습을 살펴보았습니다.
- 강화학습의 적용 사례로 구글 딥마인드에서 발표한 DQN과 알파고를 소개했습니다.

03장

배경 이론 3: 강화학습을 이용한 주식투자란?

이전 장에서 현재까지의 강화학습은 주로 게임용 인공지능에 사용되었다고 했습니다. 그러나 강화학습을 적용할 수 있는 범위는 무궁무진하다고 할 수 있습니다. 이번 장에서는 주식투자에 강화학습을 적용하는 방법에 대해 알아보겠습니다.

3.1 직관적으로 강화학습 전략 알아보기

강화학습을 주식에 적용하는 방법을 어려운 수식이나 용어를 빼고 살펴보고자 합니다. 강화학습으로 무작정 주식투자를 해 보고 경험을 쌓아 잘한 경우에 긍정적으로 보상하고 잘못한 경우엔 부정적으로 보상함으로써 일일이 학습 데이터를 만드는 수고를 없애면서도 효과적으로 주식투자 머신러닝을 수행할 수 있는 전략을 알아봅니다.

3.1.1 강화학습을 이용한 주식투자 구조

주식투자도 어떠한 환경에서 매수(buy), 매도(sell), 관망(hold)[3] 등을 판단하는 문제로서 강화학습을 적용할 수 있습니다. 주식투자에 강화학습을 적용했을 때 구성 요소는 그림 3.1과 같이 될 수 있습니다.

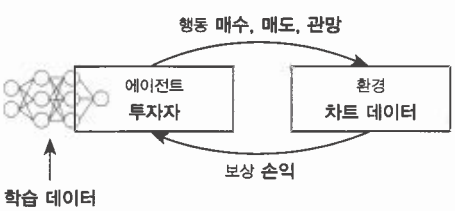


그림 3.1 주식투자에서의 강화학습 개념도

주식투자 강화학습에서 에이전트는 행동을 수행하는 주체인 투자자 역할을 합니다. 행동은 매수, 매도, 관망 등이 있을 수 있습니다. 행동은 정책 신경망으로 결정하고 정책 신경망은 투자를 진행하면서 발생하는 보상과 학습 데이터로 학습합니다.

---

3 hold를 보유로 번역해 쓰는 경우가 더 많지만 정확한 개념은 '관망'이나 '유보' 또는 '보류'입니다. 그래서 이 책에서는 그중에서 '관망'이라는 용어로 나타냅니다. 따라서 널리 알려진 '매수 후 보유' 전략은 정확한 개념에 맞춘 '매수 후 관망' 전략으로 표현될 수 있습니다.

주식투자 강화학습에서의 환경은 다양하게 정할 수 있지만 여기서는 한 종목의 차트 데이터를 환경으로 고려합니다. 에이전트가 수행하는 행동의 결과로 발생하는 수익 또는 손실을 보상으로 하여 정책 신경망을 학습합니다.

3.1.2 차트 데이터 이해하기

차트 데이터는 어떠한 종목의 시가, 고가, 저가, 종가, 거래량 등의 있는 그대로의 데이터입니다. 차트 데이터로 투자 손익을 계산합니다. 학습 데이터는 인공 신경망을 학습할 목적으로 가공한 데이터입니다. 차트 데이터와 학습 데이터에 대해서는 3.3절에서 상세히 다룹니다.

어떠한 차트 데이터상에서 투자자는 일명 묻지마 투자를 합니다. 마구잡이로 매수, 매도하는 것입니다. 그러면 일정 시간이 지났을 때 이익이나 손해가 발생하게 됩니다. 그 결과가 이익이면 보상, 손해면 처벌을 내림으로써 이전에 행했던 행동들을 학습 데이터와 보상으로 학습합니다. 시행착오를 반복하면서 에이전트의 인공 신경망이 점점 똑똑해지고 수익을 내는 방향으로 행동을 결정하게 됩니다.

주식의 차트는 주로 봉 차트(캔들 차트)로 그리게 됩니다. 봉 차트에 대해서 간단히 설명을 하고 넘어가겠습니다. 그림 3.2는 봉 차트에서 양봉과 음봉의 의미를 보여줍니다.

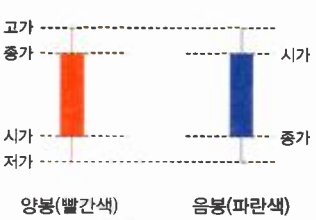


그림 3.2 봉 차트에서의 양봉과 음봉

한국의 봉 차트에서는 양봉을 빨간색으로 음봉을 파란색으로 그립니다. 미국에서는 주로 양봉을 초록색 음봉을 빨간색으로 표시하니 혼동되지 않게 유의해야 합니다.

양봉에서 굵은 부분의 아래쪽이 시가(始價)로 봉의 기간 중에서 가장 처음 거래된 가격을 의미합니다. 위 꼬리 부분은 고가(高價)로 봉의 기간 동안 거래된 가격 중에 가장 높은 가격을 의미합니다. 아래 꼬리는 저가(低價)로 봉의 기간 동안 가장 낮게 거래된 가격을 의미합니다. 굵은 부분의 위쪽은 종가(終價)로 봉의 기간 중에서 가장 마지막에 거래된 가격입니다.

음봉에서는 굵은 부분의 위쪽이 시가입니다. 고가와 저가는 양봉과 마찬가지로 각각 위쪽 꼬리와 아래쪽 꼬리입니다. 종가는 굵은 부분의 아래쪽입니다.

그림 3.3은 일봉 차트의 예입니다. 번호를 매긴 화살표가 가리키는 지점의 종가에서 어떤 투자 행동을 해야 좋을까요?

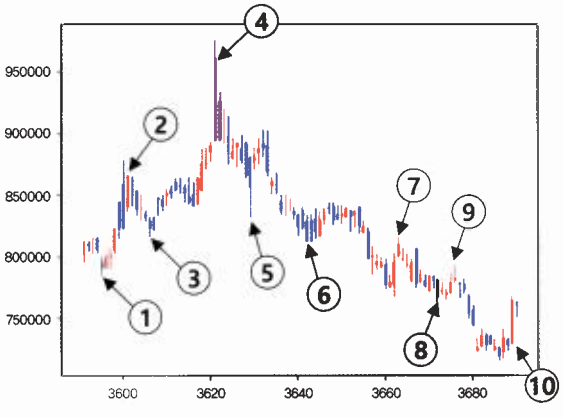


그림 3.3 일봉 차트에서 투자 행동

직관적으로 1번에서는 매수, 2번에서는 매도, 3번에서 매수, 4번부터 9번까지는 매도 및 관망, 10번에서는 매수를 하면 괜찮은 수익을 낼 수 있을 것 같습니다.

3.1.3 차트 데이터를 바탕으로 강화학습을 하는 방식

이 차트에서 강화학습으로 무작정 투자 행동을 결정해보고 어떤 식으로 학습을 할 수 있을지 살펴보겠습니다. 이 예제에서는 초기 자본으로 1,000만 원이 있고 거래에 대한 수수료와 세

금은 고려하지 않고 1주씩만 매수하거나 매도한다고 제한합니다. 그리고 2%의 손익이 발생했을 때 보상을 줍니다. 표 3.1은 매수만 해보는 경우의 결과를 보여줍니다.

표 3.1 강화학습 주식투자 예: 매수만 하는 경우

지점	종가(원)	행동	보유 주식 수		누적 손익률	보상 결정	누적 보상
1	791,000	매수	1	1	0%	–	매수[+1], 매도[0]
2	866,000	매수	2	1	0.75%	–	매수[+1], 매도[0]
3	824,000	매수	3	1	−0.09%	–	매수[+1], 매도[0]
4	960,000	매수	4	1	3.99%	보상 +1	매수[−1], 매도[0]
5	880,000	매수	5	1	0.79%	보상 −1	매수[−1], 매도[0]
6	813,000	매수	6	1	−2.56%	보상 −1	매수[−1], 매도[0]
7	810,000	매수	7	1	−2.74%	–	매수[−1], 매도[0]
8	767,000	매수	8	1	−5.75%	보상 −1	매수[0], 매도[0]
9	783,000	매수	9	1	−4.47%	–	매수[0], 매도[0]
10	765,000	매수	10	1	−6.09%	보상 0	매수[0], 매도[0]

이렇게 매수만 해 보니 4번 시점에서 2%가 넘는 3.99%의 수익이 발생하여 1~3번에서 했던 행동들을 +1로 보상하고, 5번 시점에서 −3.2% 손실이 발생하여 직전인 4번 행동을 −1로 보상합니다. 이런 식으로 10번까지 매수를 하면서 수익률 변동에 따라 보상을 주고 나면 각 괄호에 표시한 대로 누적 보상이 생깁니다.

표 3.2에서는 이전에 받은 보상을 기반으로 매수와 매도를 결정해 봅니다. 매수와 매도의 누적 보상이 같을 경우에 매수를 선택했습니다.

표 3.2 강화학습 주식투자 예: 두 번째 학습 과정

지점	종가(원)	행동	보유 주식 수		누적 손익률	보상 결정	누적 보상
1	791,000	매수	1	1	0%	–	매수[+2], 매도[0]
2	866,000	매수	2	1	0.75%	–	매수[+2], 매도[0]
3	824,000	매수	3	1	−0.09%	–	매수[+2], 매도[0]

지점	종가(원)	행동	보유 주식 수	누적 손익률	보상 결정	누적 보상
4	960,000	매도	2	3.99%	보상 +1	매수[−1], 매도[−1]
5	880,000	매도	1	2.39%	–	매수[−1], 매도[−1]
6	813,000	매도	0	1.72%	보상 −1	매수[−1], 매도[0]
7	810,000	매도	0	1.72%	–	매수[−1], 매도[0]
8	767,000	매도	0	1.72%	–	매수[0], 매도[0]
9	783,000	매도	0	1.72%	–	매수[0], 매도[0]
10	765,000	매수	1	1.72%	–	매수[0], 매도[0]

처음 매수만 결정했을 때보다는 투자를 잘 하는 것 같습니다. 여기서도 보상을 줘 보겠습니다. 4번 지점에서 3.99% 수익이 생겨서 직전 행동까지 +1을 보상합니다. 6번 지점에서는 4번 지점 대비 −2.27%의 손실이 생겨서 직전 행동까지 −1을 보상합니다.

그런데 잘 보면 4번 지점과 5번 지점에서는 행동 결정을 잘 했는데도 −1로 보상을 받게 됩니다. 이러한 문제는 학습을 반복적으로 진행하다 보면 대부분 해결됩니다. 이 상태에서 다시 누적 보상을 보고 투자를 진행해 보겠습니다. 그 결과는 표 3.3과 같습니다.

표 3.3 강화학습 주식투자 예: 세 번째 학습 과정

지점	종가(원)	행동	보유 주식 수	누적 손익률	보상 결정	누적 보상
1	791,000	매수	1	0%	–	매수[+3], 매도[0]
2	866,000	매수	2	0.75%		매수[+3], 매도[0]
3	824,000	매수	3	−0.09%		매수[+3], 매도[0]
4	960,000	매수	4	3.99%	보상 +1	매수[−2], 매도[−1]
5	880,000	매수	5	0.79%	보상 −1	매수[−2], 매도[−1]
6	813,000	매도	4	−2.56%	보상 −1	매수[−1], 매도[0]
7	810,000	매도	3	−2.68%	–	매수[−1], 매도[0]
8	767,000	매도	2	−3.97%	–	매수[0], 매도[0]
9	783,000	매도	1	−3.65%	–	매수[0], 매도[0]
10	765,000	매수	2	−3.83%	–	매수[0], 매도[0]

4번 지점과 5번 지점에서 누적 보상이 매수는 −2, 매도는 −1로 다시 매도의 보상이 더 높아졌습니다. 최종적으로 2번째 학습 과정에서의 행동 결정을 하도록 학습되었습니다. 이제는 더 학습하여도 똑같은 결과만을 주게 될 것입니다.

3.1.4 거래 수수료와 거래세

실제 투자에서는 거래 수수료와 거래세를 비용으로 고려해야 합니다. 적용하는 수수료와 세금은 다음과 같습니다.

- 매수 수수료: 0.015%
  - 매수 시 발생하는 비용으로 증권사가 취하는 거래 수수료
  - 매수 거래 금액이 100만 원일 경우 수수료는 150원
- 매도 수수료: 0.015%
  - 매도 시 발생하는 비용으로 증권사가 취하는 거래 수수료
  - 매도 거래 금액이 100만 원일 경우 수수료는 150원
- 거래세: 0.3%
  - 매도 시 발생하는 비용으로 국가가 취하는 거래 수수료
  - 매도 거래 금액이 100만 원일 경우 거래세는 3,000원

매수 및 매도 수수료는 증권사마다 다르지만 일반적으로 저렴한 경우 0.015%입니다. 거래세는 국가에 내는 돈이기 때문에 증권사와 상관없이 동일합니다.

현재 주가가 10만 원인 주식을 10주 사서 100만 원어치 매수하고 그대로 100만 원에 판다고 가정해 보겠습니다. 수수료와 거래세를 적용하지 않을 경우 포트폴리오 가치(portfolio value PV)[4] 값은 변하지 않을 것입니다.

---

4 이 책에서는 PV를 포트폴리오 가치를 나타내는 데 사용하므로, 현재 가치(PV)와 혼동하지 않도록 주의해야 합니다.

- 매수 전 PV = 주식 잔고 0 × 현재 주가 100,000원 + 현금 잔고 1,000,000원 = 1,000,000원
- 매수 후 PV = 주식 잔고 10 × 현재 주가 100,000원 + 현금 잔고 0원 = 1,000,000원
- 매도 후 PV = 주식 잔고 0 × 현재 주가 100,000원 + 현금 잔고 1,000,000원 = 1,000,000원

하지만 현실에서는 매수와 매도에서 비용이 발생합니다. 위 경우에 수수료와 세금을 적용해 보면 다음과 같습니다.

- 매수 전 PV = 주식 잔고 0 × 현재 주가 100,000원 + 현금 잔고 1,000,000원 = 1,000,000원
- 매수 후 PV = 주식 잔고 10 × 현재 주가 100,000원 + 현금 잔고 0원 − 매수 수수료 150원 = 999,850원
- 매도 후 PV = 주식 잔고 0 × 현재 주가 100,000원 + 현금 잔고 999,850원 − 매도 수수료 150원 − 거래세 3,000원 = 996,700원

즉 거래를 적게 하는 것이 비용을 줄이는 길인 것입니다. 투자에서 매수와 매도를 신중하게 결정해야 하는 이유이기도 합니다.

3.1.5 무작위 행동 결정(탐험)과 무작위 행동 결정 비율(엡실론)

이 예제에서는 주식투자 강화학습을 직관적으로 이해해 보기 위해 아주 제한적인 상황에서 투자를 하여 세 번째 학습 과정 이후부터는 정체되지만, 실제로는 사람이 추적할 수 없을 정도의 수많은 반복과 방대한 데이터를 사용하고 일부 지점에서 무작위 투자를 진행하여, 이 예제와 같이 학습이 정체되지 않도록 합니다. 무작위로 행동을 결정하는 것을 탐험(exploration)이라 하고 무작위로 행동을 결정하는 비율을 ε(엡실론, epsilon)으로 표시합니다.

3.2 강화학습 효과를 차별화하는 요인들

주식투자 강화학습에서 강화학습 모델을 다양하게 구성할 수 있습니다. 강화학습 모델에서 고려할 학습 데이터, 보상 규칙, 행동의 종류, 정책 신경망, 강화학습 기법 등을 달리할 수 있으며 이에 따라서 강화학습의 효과가 달라질 수 있습니다. 이렇게 다양한 차별화 요인들이 존재하기 때문에 어떠한 강화학습 모델이 가장 주식투자를 잘 학습할 수 있을지는 다양한 실험을 하고 비교 분석을 해 보아야 합니다.

3.2.1 차별화 요인 1: 학습 데이터 구성

본 책에서는 강화학습 신경망을 학습할 때 사용하는 데이터로 당일 주가 및 거래량의 이전 주가 및 거래량 대비 비율을 주로 사용합니다.

다음은 학습 데이터에 포함된 주가 및 거래량과 관련된 특징들입니다.

- **전일 종가 대비 당일 시가 비율** (open / last close)

  당일 시가를 전일 종가로 나눈 값입니다. 주로 전일 종가에 비해 당일 시가가 상당히 높으면 '갭상승'이라 하고, 반대로 전일 종가보다 당일 시가가 많이 떨어져 있으면 '갭하락'이라 부릅니다.

- **당일 종가 대비 당일 고가 비율** (high / close)

  당일 고가를 당일 종가로 나눈 값입니다. 이 값만으로는 추세를 예측할 수 없지만 학습 데이터의 특징으로 가치가 있습니다. 당일 고가가 종가에 비해 상당히 높다면 많은 경우에 저항선을 뚫지 못해서 추가 상승이 어렵다고 여기기도 합니다. 이를 일봉의 윗꼬리가 길다라고도 표현합니다.

- **당일 종가 대비 당일 저가 비율** (low / close)

  당일 저가를 당일 종가로 나눈 값입니다. 당일 저가보다 당일 종가가 상당히 높으면 지지선이 하향돌파될 우려가 되므로 강화학습 학습 데이터에도 특징으로 추가했습니다.

- **당일 종가 대비 전일 종가 비율** (close / last close)

  당일 종가를 전일 종가로 나눈 값입니다. 일반적으로 주가 등락을 얘기할 때 당일 종가에서 전일 종가를 뺀 다음에 다시 전일 종가로 나누는데, 수식으로는 (close - last close / last close)가 됩니다. 같은 의미이지만 학습 데이터에는 비율값으로 사용했습니다.

- **전일 거래량 대비 당일 거래량 비율** (volume / last volume)

  당일 거래량을 전일 거래량으로 나눈 값입니다. 주가가 상승하면서 거래량이 늘면 주가 상승에 긍적적인 신호로 여겨지는데, 이를 포착하고자 이 값을 학습 데이터의 특징으로 추가했습니다.

- **5일 평균 종가 대비 당일 종가 비율** (close / MA5 close)

  당일 종가를 5일 동안의 평균 종가로 나눈 값입니다. 단기 주가 추세를 학습 데이터 특징으로 고려하기 위해 사용합니다.

- **10일 평균 종가 대비 당일 종가 비율** (close / MA10 close)

  당일 종가를 10일 동안의 평균 종가로 나눈 값입니다. 마찬가지로 단기 주가 추세를 고려하기 위함입니다.

- **20일 평균 종가 대비 당일 종가 비율** (close / MA20 close)

  당일 종가를 20일 동안의 평균 종가로 나눈 값입니다. 20일선은 주가 추세를 잘 보여주는 이동평균선입니다. 이 값 역시 학습 데이터에서 매우 가치 있을 것입니다.

- **60일 평균 종가 대비 당일 종가 비율** (close / MA60 close)

  당일 종가를 60일 동안의 평균 종가로 나눈 값입니다. 60일선 역시 대표적인 이동평균선입니다. 20일선이 60일선을 상향 돌파하면 골든 크로스(golden cross)라고 부를 정도로 매우 의미 있는 이동평균선입니다.

- **120일 평균 종가 대비 당일 종가 비율** (close / MA20 close)

  당일 종가를 120일 동안의 평균 종가로 나눈 값입니다. 120일은 주식 거래일 기준 약 6개월에 해당하는 기간입니다. 이 값은 중장기간의 평균값 대비 당일 종가의 위치를 의미합니다.

- **5일 평균 거래량 대비 당일 거래량 비율** (volume / MA5 volume)

  당일 거래량을 5일 동안의 평균 거래량으로 나눈 값입니다. 이 값은 단기적인 거래량 추세를 수치화한 것입니다. 거래량이 증가하고 있다는 것은 주가의 추세를 유지할 가능성이 높다고 기대할 수 있습니다. 주가가 상승하고 있는데 거래량도 늘고 있다면 주가가 더 상승할 가능성이 높다는 것입니다. 반대로 주가가 하락하고 있는데 거래량이 늘고 있으면 주가가 더 떨어질 가능성이 높다는 것입니다.

- **10일 평균 거래량 대비 당일 거래량 비율** (volume / MA10 volume)

  당일 거래량을 10일 동안의 평균 거래량으로 나눈 값입니다. 이 값도 단기적인 거래량 추세를 내포합니다.

- **20일 평균 거래량 대비 당일 거래량 비율** (volume / MA20 volume)

  당일 거래량을 20일 동안의 평균 거래량으로 나눈 값입니다. 주가에서 20일 이동평균선을 중요하게 보았듯이 거래량 역시 20일 이동평균선은 중요합니다.

- **60일 평균 거래량 대비 당일 거래량 비율** (volume / MA60 volume)

  당일 거래량을 60일 동안의 평균 거래량으로 나눈 값입니다. 주가와 더불어 거래량의 추세를 보는 것은 매우 중요합니다. 60일 이동평균선은 중기적인 추세를 볼 수 있는 좋은 지표입니다.

- **120일 평균 거래량 대비 당일 거래량 비율** (volume / MA120 volume)

  당일 거래량을 120일 동안의 평균 거래량으로 나눈 값입니다. 이 지표로 장기적인 관점에서 당일 거래량의 정도를 판단할 수 있습니다.

여기에 추가적으로 에이전트의 상태를 학습 데이터에서 고려합니다. 투자를 수행하는 주체인 에이전트의 상태에 따라 같은 환경에서도 다른 투자 결정을 하는 편이 더 올바를 것입니다. 예를 들어 환경이 매수하기에 좋은 상황이라 하더라도 이미 너무 많은 주식을 보유하고 있다면 리스크 관리 차원에서 매수를 보류(즉, 관망)할 수 있습니다.

학습 데이터에 고려하는 에이전트의 상태는 다음과 같습니다.

- 주식 보유 비율 **(현재 보유 주식 수 / 최대 보유 가능 주식 수)**

  현재 보유 중인 주식 수에 보유한 자산으로 최대로 보유할 수 있는 주식 수를 나눈 값입니다. 이 값을 자산 대비 주식을 얼마나 보유하고 있는지를 의미합니다. 주식 보유 비율이 높을 때에는 매도 관점에서 주식 시장을 바라보고 주식 보유 비율이 낮으면 공격적인 매수를 하는 등의 전략을 학습할 수 있도록 이 값을 학습 데이터에 특징으로 추가했습니다.

- **포트폴리오 가치 비율 (현재 포트폴리오 가치 / 기준 포트폴리오 가치)**

  현재 포트폴리오 가치에 기준 포트폴리오 가치를 나눈 값입니다. 기준 포트폴리오 가치는 현재의 포트폴리오 가치를 비교할 만한 이전의 포트폴리오 가치로, 정하기 나름입니다. 현재의 수익률 또는 손익률을 학습에서 고려하기 위해 이 값을 특징으로 추가합니다.

에이전트의 상태 역시 앞서 소개한 두 가지 외에도 다양하게 구성할 수 있습니다. 예를 들어, 에이전트가 연속으로 매수한 횟수, 연속으로 매도한 횟수, 20 거래일 동안의 매수 비율, 20 거래일 동안의 매도 비율 등을 특징으로 삼을 수도 있습니다.

3.2.2 차별화 요인 2: 보상 규칙

보상 규칙은 강화학습을 진행하면서 학습 수행 여부를 결정할 기준이 됩니다. 보상 규칙은 손익률 1%, 2%, 5%, 10% 등으로 정할 수 있습니다.

보상 규칙에서 손익률 5%라는 것은 이익률 5%에 달성하면 이전에 행했던 투자 행동들을 '긍정적이었다'고 학습하고, 손실률 5%에 달성하면 이전 투자 행동들이 '부정적이었다'고 학습함을 의미합니다.

보상규칙에서 이익률과 손실률을 다르게 부여할 수도 있습니다. 이익률을 5% 초과하거나 손실률을 3% 초과할 때 학습을 진행하게 할 수 있습니다. 혹은 긍정 보상만 고려하거나 부정 보상만을 고려할 수도 있습니다. 이렇게 보상 규칙을 변경해 보면서 학습 결과를 분석해 보는 것도 의미 있는 실험일 것입니다.

3.2.3 차별화 요인 3: 행동 종류

에이전트는 행동을 수행함으로써 투자를 진행합니다. 행동에는 크게 매수, 매도, 관망이 있습니다. 매수는 주식을 사들이는 행동, 매도는 주식을 파는 행동, 관망은 매수도 매도도 하지 않고 관망하는 행동입니다. 이를 공격적 매수, 방어적 매수, 공격적 매도, 방어적 매도, 관망으로 더 세분화해 볼 수도 있습니다.

공격적 매수에서는 한번에 많은 양의 주식을 사들이고, 방어적 매수에서는 주식을 조금만 사들이고, 공격적 매도에서는 한번에 많은 주식을 팔고, 방어적 매도에서는 주식을 조금 파는 것입니다.

매매할 최대 및 최소 투자 단위를 결정해 놓고 확률에 따라 최소에서 최대 사이의 단위로 투자를 진행할 수도 있습니다.

3.2.4 차별화 요인 4: 정책 신경망

정책 신경망은 강화학습에서 에이전트가 행동을 결정하기 위한 두뇌 역할을 하는 인공 신경망입니다. 정책 신경망으로 쓸 만한 인공 신경망으로는 심층 신경망(deep neural network), 순환 신경망(recurrent neural network), 합성곱 신경망(convolutional neural network)과 이것들을 혼용하는 앙상블 신경망이 있습니다.

- **심층 신경망**

  학습 데이터에는 여러 특징이 있는데, 이것들의 값이 달라짐에 따라 수많은 조합을 이루게 됩니다. 하나의 특징은 연속 값을 가지기 때문에 이들의 조합은 무한합니다. 심층 신경망은 이렇게 무한한 특징 조합들을 효과적으로 학습하기 위해 여러 계층과 많은 수의 퍼셉트론을 가집니다.

- **순환 신경망**

  순환 신경망은 현재의 상태와 이전에 결정한 행동을 함께 고려할 수 있습니다. 이 경우 [매수→매도→매수→매도→매수]와 같은 정신 없는 투자를 피하고 어느 정도의 투자 일관성을 가질 수 있습니다.

- **합성곱 신경망**

  합성곱 신경망으로 5일, 10일 등의 특정 기간 동안의 일련의 상태를 한꺼번에 고려할 수 있습니다. 이전 상태들의 흐름을 포괄적으로 고려하여 높은 학습 품질을 기대할 수 있습니다.

- **여러 신경망의 앙상블**

  심층 신경망, 순환 신경망, 합성곱 신경망을 모두 사용하여 결과를 다수결로 결정하는 방법을 사용할 수 있습니다. 이 경우 투자 결정에 대해 신뢰성을 높일 수 있습니다. 이러한 방법을 앙상블(ensemble)이라 하고 비슷한 방법으로 배깅(bagging), 부스팅(boosting) 등의 방법이 있습니다.

3.2.5 차별화 요인 5: 강화학습 기법인 Q 러닝과 정책 경사

강화학습 기법으로는 Q 러닝, 정책 경사 등을 사용할 수 있습니다. 두 기법 모두 주식투자 강화학습에서 적절한 방법이지만 다음과 같은 차이가 있습니다.

- **Q 러닝**

  Q 러닝은 특정 행동을 취했을 때 예측되는 포트폴리오 가치를 회귀(regression)하고자 사용합니다. 주식투자에서 손실을 보고자 투자를 하는 사람을 없을 것이기 때문에 예측한 값이 높은 쪽으로 행동을 취하면 됩니다. 즉 Q 러닝을 사용할 경우 행동들마다 기대 손익을 수치적으로 예측합니다.

- **정책 경사(policy gradient)**

  정책 경사는 어떤 행동이 현재 상태에서 가장 좋을지를 확률적으로 판단합니다. Q 러닝과는 다르게 기대 손익을 예측하는 것이 아니라 단순히 현 상황에서 어떤 행동이 더 좋은지를 판단하는 것입니다.

주식투자 시 일반적으로 매수/매도했을 때 수익률(가치)을 수치적으로 예측하기보다는 현재 상태에서 매수해야 좋을지 아니면 매도해야 좋을지를 판단하는 편이 더 쉽고 신뢰성이 높다고 생각하여 본 책에서는 여러 강화학습 기법 중에 정책 경사를 주식투자에 적용합니다. 물론 여러 강화학습 기법을 한꺼번에 사용할 수도 있습니다.

3.3 차트 데이터와 학습 데이터 살펴보기

강화학습에 사용할 데이터로 차트 데이터와 학습 데이터가 있습니다. 차트 데이터는 전처리를 하지 않은, '있는 그대로'의 데이터입니다. 데이터 과학에서는 보통 원시 데이터(original data) 또는 기초 데이터(raw data)라고 부릅니다. 학습 데이터는 차트 데이터를 전처리(pre-processing)한 데이터로서, 학습에 바로 사용할 수 있게 한 데이터입니다.

3.3.1 차트 데이터

차트 데이터는 그림 3.4와 같이 2차원 데이터로 체결일(date), 시가(open), 고가(high), 저가(low), 종가(close), 거래량(volume)이 연속되는(idx 0 … idx N) 식으로 구성된 데이터입니다.

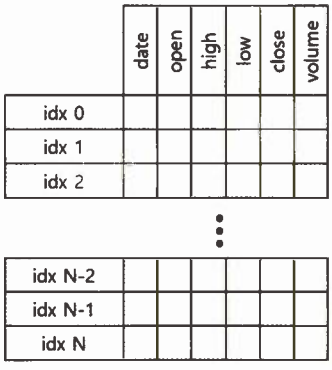


그림 3.4 차트 데이터의 구조

여기서 하나의 행(row)은 일 단위 데이터입니다. 여기서의 OHLC(open, high, low, close) 데이터는 일봉의 시가, 고가, 저가, 종가를 의미하고 volume은 하루 동안의 거래량입니다. 물론 차트 데이터를 60분봉, 20분봉, 5분봉, 1분봉 등으로 구성할 수도 있습니다.

차트 데이터를 구하는 방법은 다양한데 5장에서 대표적인 방법들을 상세히 다룹니다.

3.3.2 학습 데이터

정책 신경망이 학습하기 좋도록 차트 데이터를 전처리하여 학습 데이터를 만들어 내야 합니다. 먼저 5일, 10일, 20일, 60일, 120일 평균 종가와 평균 거래량을 구합니다. 5일 평균 데이터는 인덱스 4부터, 10일 평균 데이터는 인덱스 9부터, 20일 평균 데이터는 인덱스 19부터, 60일 평균 데이터는 인덱스 59부터, 120일 평균 데이터는 인덱스 119부터 계산이 가능합니다.

이 데이터는 아직 학습하기에 효율적인 형태는 아닙니다. 그 이유는 주가와 거래량의 단위가 각기 '원'과 '주'로서 서로 다르기 때문입니다. 그래서 효율적인 학습을 위해 그림 3.5와 같이 학습 데이터를 비율 값으로 구성합니다.

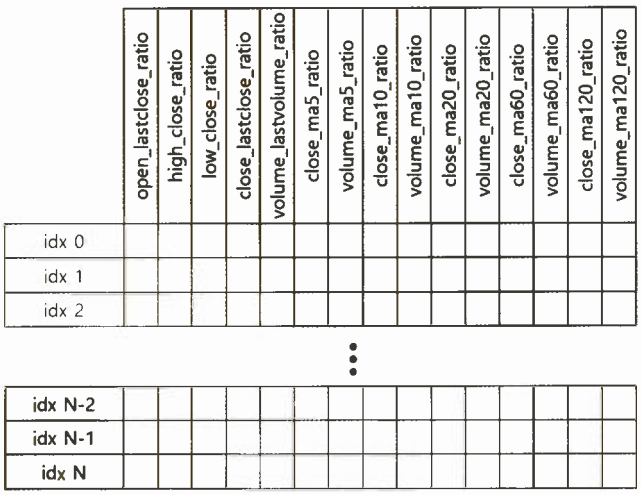


그림 3.5 학습 데이터의 구조

전처리 작업은 학습 데이터의 값의 범위를 비슷하게 만들기 때문에 학습했을 때 종목에 크게 의존되지 않게 할 수 있는 장점 또한 있습니다. 주식 데이터의 전처리 과정은 6.1절에서 확인할 수 있습니다.

3.4 주식투자 강화학습 절차

강화학습은 환경, 에이전트, 정책 신경망이 서로 상호작용하면서 학습을 수행하기 때문에 그 과정이 일반적인 머신러닝에 비해 복잡합니다. 여기서는 주식투자를 위한 강화학습 절차(process)를 순서도와 함께 살펴봅니다.

3.4.1 주식투자 강화학습 순서도

강화학습으로 주식투자하는 방법은 다른 여느 강화학습 문제와 다르지 않습니다. 강화학습 절차는 그림 3.6과 같습니다.

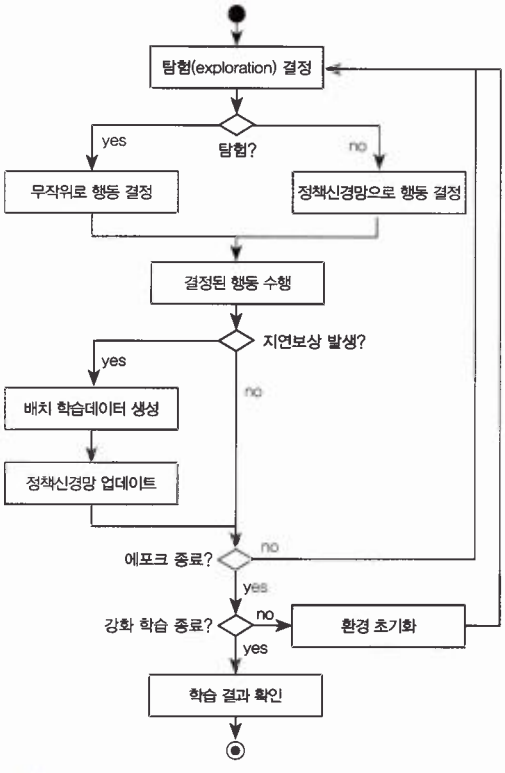


그림 3.6 강화학습 절차

강화학습은 경험을 학습하는 것입니다. 경험을 충분히 쌓기 위해서 많은 반복을 합니다. 학습 대상 데이터 전부를 대상으로 한 차례 반복한 과정을 에포크(epoch)[5]라고 합니다. 100번째 반복이라면 100 에포크라고 부르면 됩니다.

3.4.2 행동 결정

한 에포크에서 경험을 얻기 위해서 무작위로 행동을 해봐야 하는데 이를 탐험(exploration)이라 합니다. 일반적으로 강화학습 초반에는 탐험을 많이 하고 후반으로 갈수록 탐험을 적게 해 나갑니다.

---

5 에포크란 신기원(新紀元) 또는 새 시대라는 뜻으로 유사한 용어로는 유전 알고리즘에서 쓰는 세대(generation)와, 통계학의 연(連, run)이 있습니다. 각기 무언가를 한 차례 반복한다는 개념입니다.

탐험 비율을 엡실론(epsilon)이라 합니다. 1만 번을 반복한다고 했을 때 1 에포크에서는 엡실론을 30%로 정하고, 점점 엡실론을 줄여서 1만 번째 에포크에서는 0%가 되게 하면 에포크가 커질수록 무작위로 하는 행동이 줄어들게 됩니다. 경험이 쌓일수록 무작위로 하는 행동을 줄이는 것입니다.

무작위로 행동을 하지 않을 때는 정책 신경망으로 행동을 결정합니다. 정책 신경망의 출력값이 높은 행동을 선택합니다.

3.4.3 결정된 행동 수행

무작위로 결정한 행동이든 정책 신경망으로 결정한 행동이든 에이전트는 결정된 행동을 수행합니다.

매수의 경우 에이전트는 주식을 사들일 현금이 있는지 확인하고, 매수가 가능할 경우 매수를 수행하고 그렇지 않으면 관망합니다. 매수했을 경우 매수금만큼 현금을 줄이고 매수한 주식 수만큼 보유 주식 수를 늘려줍니다.

매도의 경우 에이전트는 보유한 주식이 있는지 확인하고 보유한 주식이 있을 경우 매도를 수행하고 그렇지 않으면 관망합니다. 매도했을 경우 매도한 주식 수만큼 보유 주식 수에서 빼주고 매도금만큼 현금보유액에 더해줍니다.

3.4.4 배치 학습 데이터 생성 및 정책 신경망 업데이트

이렇게 주식투자를 해 나가면서 지연 보상을 줄 수 있을지 판단합니다. 예를 들어 2% 이상의 이익 또는 손실을 지연 보상 기준으로 정해 보겠습니다. 즉, 투자를 해 나가다가 2% 이상의 이익이나 2% 이상의 손실이 발생하면 이때까지 상황과 행동들을 학습 데이터로서 생성합니다. 이 학습 데이터들을 한꺼번에 적용하여 정책 신경망을 업데이트합니다. 이런 학습 방법을 배치(batch) 학습이라고 부릅니다.

학습을 진행하고 나면 정책 신경망의 가중치들이 업데이트되어 이후에 진행되는 투자부터 바로 업데이트된 정책 신경망의 결과가 반영됩니다.

지연 보상 기준을 낮게 잡으면 작은 배치 학습 데이터로 자주 학습을 수행할 가능성이 높으며 지연 보상 기준이 높으면 큰 배치 학습 데이터로 덜 자주 학습을 진행할 가능성이 높습니다.

3.5 주식투자 강화학습 과정 및 결과 확인 방법

주식투자 강화학습을 진행하면 정해진 환경에서 매 순간마다 무작위로 행동을 결정하거나 정책 신경망으로 행동을 결정한 다음에 에이전트는 결정된 행동을 수행하고 그 결과로 에이전트의 상태가 변경됩니다. 이번 절에서는 주식투자 강화학습 과정을 확인하는 방법을 다룹니다.

3.5.1 강화학습 과정 확인의 필요성

강화학습이 잘 진행되고 있는지를 판단하려면 정책 신경망이 어떤 출력값을 반환하는지, 에이전트의 상태는 어떻게 변해가는지를 관찰해야 합니다.

강화학습으로 수백 수천 에포크를 거치며 주식투자를 (제대로) 학습하다 보면 처음에는 투자 손실을 보다가 에포크를 거쳐가면서 점점 수익을 내게 됩니다. 학습 시간은 인공 신경망의 복잡도, 학습 데이터의 크기, 에포크 수에 따라 수십 분에서 며칠까지도 걸릴 수 있습니다. 그렇기 때문에 제대로 된 학습을 진행하고 있는지 아닌지를 학습 과정이 진행되는 동안 관찰할 필요가 있습니다.

학습이 진행되어 가는데 정책 신경망의 학습이 제대로 되지 않아서 매수만 한다거나 매도만 하는 등의 이상 상황이 발생하면 중도에 학습을 멈추고 문제 파악 및 해결을 하고 다시 학습을 수행하게 하는 것이 시간을 절약하는 길일 것입니다.

3.5.2 강화학습 과정을 로그로 남기기

일반적으로 어떠한 과정을 텍스트로 기록한 것을 로그(log)라고 합니다. 로그는 데이터 크기가 (이미지에 비해) 상대적으로 작고 쉽게 기록할 수 있으며 상세한 정보를 담을 수 있는 장점이 있습니다. 단점으로는 한눈에 기록을 파악하기 힘듭니다.

강화학습 과정에서 로그로 남기는 값들은 다음과 같습니다.

- **파라미터**

  학습 속도(learning rate), 할인 요인(discount factor), 최소/최대 투자 단위(trading unit), 지연 보상 임계치(delayed reward threshold) 등의 파라미터를 기록해 두어야 파라미터에 따라 결과가 어떻게 달라지는지를 확인할 수 있습니다.

- **에포크 결과**

  전체 에포크 중에서 몇 번째 에포크인지 먼저 기록하고 탐험률, 탐험 횟수, 매수 횟수, 매도 횟수, 관망 횟수, 보유 주식 수, 포트폴리오 가치, 긍정적 학습 횟수, 부정적 학습 횟수, 학습 과정에서 발생한 손실 등을 기록하여 에포크가 진행되어 가면서 이 값들이 어떻게 변해가는지 확인합니다.

- **최종 학습 결과**

  학습 과정 중에서 달성한 최대 포트폴리오 가치, 수익 발생 횟수 등 전체 학습이 완료되고 나서야 알 수 있는 통계치 등을 확인합니다.

3.5.3 강화학습 과정을 이미지로 가시화하기

각 에포크마다 보유 주식 수, 행동, 인공 신경망의 출력 값, 탐험, 수익과 손실을 관찰하기 위해 그림 3.7과 같은 가시화 방법을 사용합니다.

그림 3.7 강화학습 과정 가시화 방법

가장 상위의 차트는 주식 종목의 일봉 차트를 보여줍니다. 즉, 강화학습의 환경(environment)이라고 할 수 있습니다.

두 번째 차트는 에이전트가 수행한 행동을 배경 색으로 보여주고 보유 주식 수를 실선으로 보여줍니다. 매수했으면 보유 주식 수가 증가할 것이고 매도했으면 보유 주식 수는 줄어들 것입니다.

세 번째 차트는 인공 신경망의 출력 값을 보여줍니다. 빨간색 점은 매수에 대한 확률값, 파란색 점은 매도에 대한 확률값을 보여줍니다. 즉 빨간색 점이 파란색 점보다 위에 있으면 매수를, 그 반대면 매도를 수행합니다. 수행할 행동을 배경색으로 보여줍니다. 여기서 무작위 투자, 즉 탐험을 했으면 노란색으로 표시합니다.

마지막 차트는 포트폴리오 가치를 보여줍니다. 즉, 투자 결과인 손익을 보는 것입니다. PV값이 기준선(초기 투자금) 위면 빨간색으로, 기준선 아래면 파란색으로 사이 공간을 채워서 손익을 더 쉽게 파악하게 합니다.

이 화면을 에포크마다 한 번씩 생성해 각 에포크에서 에이전트가 수행한 행동들, 인공 신경망의 출력, 투자 결과인 수익을 한눈에 볼 수 있습니다. 이 화면을 생성하는 방법은 4.6절에서 다룹니다.

3.6 이번 장의 요점

- 주식투자 강화학습 전략을 직관적으로 살펴보았습니다.
- 주식투자 강화학습의 효과를 달라지게 할 수 있는 가변점(variation point)을 확인했습니다.
- 주식투자 강화학습에서 사용하는 차트 데이터와 학습 데이터를 살펴봤습니다.
- 주식투자 강화학습 절차(process)를 전체적으로 살펴보고 주요 단계를 상세히 다루었습니다.
- 주식투자 강화학습의 결과를 로그와 가시화된 이미지로 확인하는 방법을 살펴보았습니다.

04장

모듈 개발: 강화학습 기반 주식투자 시스템 개발

이번 장에서는 파이썬으로 강화학습 기반 주식투자 시스템인 RLTrader를 개발하는 과정을 소스코드 수준에서 상세히 설명합니다.

같은 이름의 다른 프로젝트들이 깃허브(Github) 등에 올라와 있을 수 있는데, 여기서 다루는 RLTrader는 저자가 작성한 강화학습 주식투자 시뮬레이션 프로그램으로 https://github.com/quantylab/rltrader에서 소스코드를 확인할 수 있습니다.

이 프로젝트는 다음 라이선스를 따릅니다.

RLTrader는 실제 주식 데이터에서 수많은 반복적 투자를 통해 더 좋은 투자 결정을 내리도록 스스로 학습합니다. 더 좋은 투자 결정이란 수익을 낼 가능성이 높은 결정을 말합니다.

4.1 RLTrader 개발에 필요한 환경

RLTrader는 파이썬 기반의 시스템이고 몇몇 대표적인 파이썬 라이브러리를 사용합니다. RLTrader의 개발 환경은 다음과 같습니다.

- 운영체제(OS): 윈도우 10 x64
- 통합 개발 환경(IDE): PyCharm
  - 다운로드 URL: https://www.jetbrains.com/pycharm/
- 개발 언어: 파이썬 3(아나콘다 3에 포함되어 있음)
  - 다운로드 URL: https://www.anaconda.com/download/

- 라이브러리
  - NumPy(아나콘다 3에 포함되어 있음)
  - Pandas(아나콘다 3에 포함되어 있음)
  - Matplotlib(아나콘다 3에 포함되어 있음)
  - 텐서플로
  - 케라스

OS와 IDE는 다르게 선택할 수 있지만 위와 같이 권장하는 이유는 다음과 같습니다. 우선 OS의 경우 주식 데이터를 가져오기 위해 사용하는 증권사 API를 윈도우 환경에서만 사용할 수 있습니다. PyCharm은 JetBrain에서 만든 매우 훌륭한 IDE이고 결정적으로 무료입니다. PyCharm은 무료 버전인 공동체 판(community edition)과 유료 버전인 전문가 판(professional edition)이 있습니다. RLTrader 개발에서는 무료 버전으로 충분합니다.

4.1.1 아나콘다 설치

개발 언어와 필요한 라이브러리들을 설치하기 위해서 아나콘다 3를 설치하는 것을 권장합니다. 아나콘다 3는 파이썬과 주요 라이브러리들을 한꺼번에 설치해 주는 도구입니다. 아나콘다 3는 설치 과정에서 변경하지 않았다면 'C:\Users\〈사용자이름〉\Anaconda3'에 설치되었을 것입니다.

설치한 아나콘다 3로 파이썬 환경을 정하기 위해서 윈도우 기본 터미널 대신에 'Anaconda Prompt'를 사용할 것을 권장합니다.

시작 메뉴에서 Anaconda3 (64-bit) 폴더에 'Anaconda Prompt'가 있습니다. 또 윈도우 10에서는 시작 키를 누르고 'anaconda'를 검색하면 그림 4.1과 같이 'Anaconda Prompt'가 검색되는 것을 볼 수 있습니다.

그림 4.1 'Anaconda Prompt' 검색 결과

이 실행 파일의 위치는 다음과 같습니다.

```
C:\Users\<사용자이름>\AppData\Roaming\Microsoft\Windows\Start Menu\Programs
\Anaconda3 (64-bit)
```

파일 탐색기에서 AppData 폴더가 안 보일 수 있는데 주소창에 직접 이 위치를 입력하여 들어갈 수 있습니다.

NumPy는 수치 계산에 필요한 다양한 기능을 제공하는 라이브러리이고 Pandas는 수치 계산에 효과적인 다양한 자료 구조를 제공하고 Matplotlib은 데이터를 가시화할 수 있는 다양한 차트를 제공합니다. 텐서플로는 인공 신경망을 구현하는 것을 도와주는 대표적인 라이브러리이고 케라스는 인공 신경망을 아주 간단하게 구현할 수 있게 합니다.

4.1.2 텐서플로와 케라스 설치

사용하는 라이브러리들 중에서 텐서플로와 케라스는 따로 설치해야 합니다. 파이썬 패키지 관리 명령인 pip를 사용하여 설치합니다.

다음과 같이 pip로 파이썬 모듈을 설치할 수 있습니다.

- $ pip install <설치할 파이썬 모듈>

한꺼번에 여러 모듈을 설치하기 위해서 띄어쓰기로 모듈 이름을 연결할 수 있습니다. 예를 들어서 다음과 같은 명령으로 텐서플로와 케라스를 한꺼번에 설치할 수 있습니다.

- $ pip install tensorflow keras

설치한 파이썬 모듈은 다음과 같은 명령으로 삭제할 수 있습니다.

- $ pip uninstall <삭제할 파이썬 모듈>

설치가 에러 없이 정상적으로 완료되었는지 확인해야 합니다. 'Anaconda Prompt'에서 python을 실행하고 'import keras'를 입력합니다. 이때 에러 없이 'Using TensorFlow backend.'라는 출력이 잘 뜨는지를 확인합니다.

4.2 RLTrader의 구조

RLTrader는 여러 파이썬 모듈로 구성되어 있습니다. 주식투자 강화학습에서 필요한 참여요소와 역할을 분석하여 모듈로 묶었습니다. 각 모듈마다 나름대로 역할을 담당하며 다른 모듈과 서로 연관관계(association)가 있기도 합니다.

4.2.1 모듈 구조

RLTrader에는 그림 4.2와 같이 강화학습 요소가 모듈로 구성되어 있으며, 그밖에 추가 모듈이 몇 개 더 있습니다.

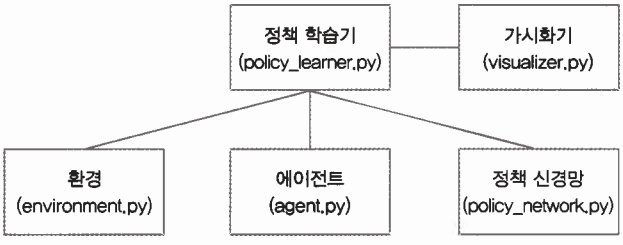


그림 4.2. RLTrader의 주요 모듈

하나의 상자는 하나의 파이썬 모듈로 구현됩니다. 실선으로 연결된 두 모듈 간에는 연관관계가 있습니다. 여기서 몸통이 되는 모듈은 정책 학습기 모듈이고 강화학습의 참여요소인 환경, 에이전트, 정책 신경망과 연관관계를 맺게 됩니다. 또한 학습과정 기록을 위한 가시화기 모듈 또한 사용합니다.

파이썬 프로그램은 패키지(package), 모듈(module), 클래스(class), 함수(function)로 구성됩니다. 이것들은 모두 데이터(상수 및 변수)와 기능을 묶는 것입니다. 함수에서 클래스로, 클래스에서 모듈로, 모듈에서 패키지로 그 묶음의 크기가 커집니다. 패키지는 여러 모듈을, 모듈은 여러 클래스를, 클래스는 여러 함수를 가질 수 있습니다.

4.2.2 디렉터리 구조

이 모듈들은 각기 파이썬 파일로 작성되는데, RLTrader에서는 하나의 모듈이 하나의 클래스를 가지도록 했습니다. 프로젝트의 디렉터리 구조는 다음과 같습니다.

- rltrader
  - chart_data/
  - logs/
  - epoch_summary/
  - models/
  - environment.py
  - agent.py
  - policy_network.py
  - visualizer.py
  - policy_learner.py

RLTrader의 프로젝트 최상위 폴더인 rltrader 내에 파이썬 모듈들인 environment.py, agent.py, policy_network.py, visualizer.py, policy_learner.py가 있고 chart_data, logs, epoch_summary, models 폴더들이 위치합니다.

chart_data 폴더에는 차트 데이터를 저장해 놓습니다. logs 폴더에는 강화학습 과정을 텍스트로 기록한 로그를 저장합니다. epoch_summary 폴더에는 에포크마다 발생하는 가시화 결과를 저장합니다. models 폴더에는 학습이 끝난 정책 신경망 모델을 저장합니다.

4.2.3 에이전트 모듈 개요

에이전트 모듈(agent.py)에는 에이전트 클래스(Agent)가 있습니다. 에이전트 클래스는 주식을 매수하거나 매도하는 투자자 역할을 하며 초기 자본금, 현금 잔고, 주식 잔고라는 상태가 있습니다. 현금 잔고와 주식 잔고의 평가액을 합하면 포트폴리오 가치(portfolio value)가 됩니다. 이를 PV라고 줄여 부르겠습니다.

$$PV = 주식\ 잔고 \times 현재\ 주가 + 현금\ 잔고$$

PV가 초기 자본금보다 높으면 수익이 발생한 것이고 낮으면 손실이 발생한 것입니다. 즉, 투자의 목표는 PV를 높여 나가는 것입니다.

에이전트가 매수를 하면 주식 잔고는 늘어나고 현금 잔고는 줄어들 것입니다. 반대로 에이전트가 매도를 하면 주식 잔고는 줄어들고 현금 잔고는 늘어납니다.

4.2.4 환경 모듈 개요

환경 모듈(environment.py)에는 환경 클래스(Environment)가 있습니다. 환경 클래스는 에이전트가 투자할 종목의 차트 데이터를 관리합니다. 에이전트가 과거로 돌아가서 투자하며 그 결과에 따라 학습하려는 것이 목적이므로 환경 클래스에 전체 차트 데이터가 있지만, 과거 시점부터 가장 최근 시점까지 순차적으로 데이터를 제공합니다. 즉 과거로 돌아간 에이전트가 미래의 차트 데이터는 알 수 없는 것입니다.

4.2.5 정책 신경망 모듈 개요

정책 신경망 모듈(policy_network.py)에는 정책 신경망 클래스(PolicyNetwork)가 있습니다. 정책 신경망 클래스는 특정 시점의 주식 데이터(sample)가 제공되었을 때 매수할지 또는 매도할지를 결정하는 에이전트의 뇌와 같은 역할을 합니다. 정책 신경망은 LSTM 신경망으로 구성되어 있고 매수와 매도 행위에 대해서 PV를 높일 수 있을지의 확률을 계산합니다.

주식을 매수하면 현금 잔고는 줄어들고 주식 잔고는 늘어납니다. 주식 잔고가 많은 상태에서 주가가 상승하면 PV는 크게 높아질 것입니다. 반면에 주가가 하락하면 PV는 크게 줄어들 수 있습니다. 그래서 주가가 상승할 것으로 생각되면 주식 잔고를 늘리고 하락할 것으로 예상되면 주식 잔고를 줄여서 리스크를 관리하는 것입니다.

RLTrader는 정책 신경망을 통해 주식 잔고를 늘릴지 줄일지를 결정합니다. 주식 데이터를 정책 신경망에 넣으면 매수와 매도에 대한 확률이 나오고 매수가 매도보다 확률이 높으면 매수를 그 반대의 경우엔 매도를 하는 것입니다.

4.2.6 가시화기 모듈 개요

가시화기 모듈(visualizer.py)은 주식 데이터 학습 과정을 직관적으로 파악하기 위해 환경, 에이전트 상태, 정책 신경망 출력 등을 그림 파일로 시각화합니다. 그림 파일을 순서대로 보면 에포크가 진행됨에 따라 정책 신경망이 결정하는 행동 확률이 포트폴리오 가치를 높이는 방향으로 변하고 그에 따라 포트폴리오 가치가 높아져 가는 것을 확인할 수 있습니다.

충분한 에포크가 지났음에도 포트폴리오 가치가 높아지지 않는다면, 학습이 제대로 이루어지지 않고 있다고 파악하고, 학습 데이터 모델링을 다시 하거나 알고리즘을 개선해야 하는지 고려해야 합니다.

4.2.7 정책 학습기 모듈 개요

정책 학습기 모듈(policy_learner.py)에는 정책 학습기 클래스(PolicyLearner)가 있습니다. 정책 학습기 클래스는 앞서 소개한 에이전트, 환경, 정책 신경망을 가지고 강화학습을 수행하는 RLTrader의 몸체라고 할 수 있습니다. 정책 학습기는 학습 데이터를 가지고 있고 보상이 결정되었을 때 학습 데이터로 정책 신경망을 학습시킵니다.

4.3 환경 모듈 개발

환경 모듈에 포함된 Environment 클래스의 속성과 함수를 살펴보고 파이썬으로 구현한 소스코드를 상세히 확인합니다.

4.3.1 환경 모듈의 주요 속성과 함수

환경 모듈(environment.py)은 투자할 종목의 차트 데이터를 관리하는 작은 모듈입니다. 환경 모듈에는 환경 클래스(Environment)가 있습니다. 환경 클래스의 주요 속성 및 함수는 다음과 같습니다.

- **속성**

  chart_data: 주식 종목의 차트 데이터

  observation: 현재 관측치

  idx: 차트 데이터에서의 현재 위치

- **함수**

  reset(): idx와 observation을 초기화

  observe(): idx를 다음 위치로 이동하고 observation을 업데이트

  get_price(): 현재 observation에서 종가를 획득

4.3.2 코드 조각: 환경 클래스의 전체 소스코드

예제 4.1은 환경 클래스의 전체 소스코드를 보여줍니다.

예제 4.1 Environment 클래스

https://github.com/quantylab/rltrader/blob/master/environment.py

```
1    class Environment:
2
```

```
3       PRICE_IDX = 4  # 종가의 위치
4
5       def __init__(self, chart_data=None):
6           self.chart_data = chart_data
7           self.observation = None
8           self.idx = -1
9
10      def reset(self):
11          self.observation = None
12          self.idx = -1
13
14      def observe(self):
15          if len(self.chart_data) > self.idx + 1:
16              self.idx += 1
17              self.observation = self.chart_data.iloc[self.idx]
18              return self.observation
19          return None
20
21      def get_price(self):
22          if self.observation is not None:
23              return self.observation[self.PRICE_IDX]
24          return None
```

5번 줄의 환경 클래스 생성자에서 관리할 차트 데이터를 할당합니다. 여기서는 차트 데이터인 chart_data를 입력으로 받아서 객체 내의 변수인 self.chart_data로 저장해 줍니다. 차트 데이터에서 현재 위치의 관측 값이 self.observation에 저장될 것입니다. 현재 위치는 self.idx에 저장됩니다.

파이썬 클래스의 생성자 __init__()은 클래스의 객체가 생성될 때 자동으로 호출되는 함수입니다. 보통 이 함수에서 입력으로 받은 값들을 객체 내의 변수로 할당해 줍니다.

차트 데이터(chart_data)는 2차원 배열이라고 생각하면 됩니다. RLTrader에서는 이러한 2차원 자료 구조를 효과적으로 처리하기 위해 Pandas의 DataFrame 클래스를 사용합니다. DataFrame의 사용법 등의 내용은 필요할 때마다 소개하겠습니다.

14번 줄의 observe() 함수에서 하루 앞으로 이동하며 차트 데이터에서 관측 데이터(observation)를 제공합니다. 더 이상 제공할 데이터가 없을 때는 None을 반환합니다.

len() 함수는 파이썬 내장 함수로 문자열, 리스트 등의 길이를 반환합니다.

차트 데이터의 전체 길이보다 다음 위치가 작을 경우 가져올 데이터가 있다는 뜻이고 이 경우 현재 위치 self.idx에 1을 더하고 차트 데이터에서 이 위치의 요소를 가져옵니다.

iloc() 함수는 DataFrame 함수로 특정 행의 데이터를 가져옵니다.

10번 줄의 reset() 함수에서는 다시 차트 데이터의 처음으로 돌아가게 합니다. 즉 현재 위치 self.idx를 -1로, self.observation을 None으로 초기화합니다.

21번 줄의 get_price() 함수에서는 관측 데이터로부터 종가를 가져와서 반환합니다. 종가 close의 위치가 5번째 열이기 때문에 0부터 시작하는 인덱스인 PRICE_IDX 값은 4입니다. self.observation은 하나의 행이고 여기서 인덱스 4의 값인 self.observation[4]가 종가에 해당됩니다.

4.4 에이전트 모듈 개발

에이전트 모듈에 포함된 에이전트 클래스(Agent)의 속성과 함수를 살펴보고 파이썬으로 구현한 소스코드를 상세히 확인합니다.

4.4.1 에이전트 모듈의 주요 속성과 함수

에이전트 모듈(agent.py)은 투자 행동을 수행하고 투자금과 보유 주식을 관리하기 위한 에이전트 클래스(Agent)를 가집니다. Agent 클래스의 속성, 함수는 다음과 같습니다.

- **속성**

  initial_balance: 초기 투자금

  balance: 현금 잔고

  num_stocks: 보유 주식 수

  portfolio_value: 포트폴리오 가치(투자금 잔고 + 주식 현재가 * 보유 주식 수)

- **함수**

  reset(): 에이전트의 상태를 초기화

  set_balance(): 초기 자본금을 설정

  get_states(): 에이전트 상태를 획득

  decide_action(): 탐험 또는 정책 신경망에 의한 행동 결정

  validate_action(): 행동의 유효성 판단

  decide_trading_unit(): 매수 또는 매도할 주식 수 결정

  act(): 행동 수행

에이전트 모듈은 환경 모듈보다는 규모 면에서 더 큽니다. 그래서 소스코드를 한 번에 보여주는 것이 효과적이지 않아서 상수 선언 부분 및 함수 단위로 설명합니다.

4.4.2 코드 조각 1: 에이전트 클래스의 상수 선언 부분

예제 4.2는 에이전트 클래스의 상수 선언부의 소스코드입니다.

예제 4.2 Agent 클래스의 상수 선언 부분

https://github.com/quantylab/rltrader/blob/master/agent.py

```
import numpy as np


class Agent:
    # 에이전트 상태가 구성하는 값 개수
```

```
6       STATE_DIM = 2  # 주식 보유 비율, 포트폴리오 가치 비율
7
8       # 매매 수수료 및 세금
9       TRADING_CHARGE = 0  # 거래 수수료 미고려 (일반적으로 0.015%)
10      TRADING_TAX = 0  # 거래세 미고려 (실제 0.3%)
11
12      # 행동
13      ACTION_BUY = 0  # 매수
14      ACTION_SELL = 1  # 매도
15      ACTION_HOLD = 2  # 관망
16      ACTIONS = [ACTION_BUY, ACTION_SELL]  # 인공 신경망에서 확률을 구할 행동들
17      NUM_ACTIONS = len(ACTIONS)  # 인공 신경망에서 고려할 출력값의 개수
```

에이전트 모듈은 NumPy를 사용하므로 우선 임포트합니다. 1번 줄에서 numpy 모듈을 np로 줄여서 정의했습니다.

import는 다른 패키지, 모듈, 클래스, 함수를 접근할 수 있도록 연결하는 명령어입니다. 임포트한 패키지, 모듈, 클래스, 함수 등을 as 키워드로 다른 이름으로 지정할 수도 있습니다. 예를 들어서 import numpy as np는 numpy 모듈을 np로 사용하겠다는 의미입니다.

Agent 클래스는 여러 상수들을 사용합니다. 우선 STATE_DIM은 에이전트의 상태의 차원입니다. RLTrader에서의 에이전트 상태는 주식 보유 비율과 포트폴리오 가치 비율로 2개의 값을 가지므로 2차원입니다. 에이전트의 상태에 대해서는 뒤에서 더 설명합니다.

에이전트는 매수와 매도를 수행하는 주체이기 때문에 매매 수수료 및 세금을 상수로서 가집니다. TRADING_CHARGE는 매수 및 매도 수수료를, TRADING_TAX는 거래세를 의미합니다.

사실 실제 주식 투자에서는 거래 수수료와 거래세는 수익성을 좌우하는 큰 요소입니다. 이 책에서 주식 투자 시뮬레이션을 할 때는 일봉 차트로 매일 주식을 매매하므로 거래 횟수가 아주 많습니다. 이렇게 거래를 많이 하는 것은 현실적으로 좋은 방법이 아닙니다.

이 책은 주식 투자 자체에 관점을 맞춘 것이 아니라 개인이 딥러닝을 주식투자에 (시험적으로) 적용해 볼 수 있도록 필요한 프로그래밍 지식을 전달하는 데 집중하고 있습니다. 따라서

이 책에서는 거래 수수료와 거래세를 고려하지 않습니다. 실전 투자에 RLTrader를 적용하고자 한다면 거래 횟수를 주 1회, 월 1회, 연 1회 등으로 줄이고 일봉 차트뿐만 아니라 다양한 지표를 활용해야 할 것입니다. 거래 수수료와 거래세를 고려하고 싶다면 9번 줄과 10번 줄을 수정하시길 바랍니다. 일반적으로 거래 수수료는 0.015%, 거래세는 0.3%로 고려하면 됩니다.

에이전트가 할 수 있는 행동은 매수, 매도, 관망입니다. 에이전트는 이 행동들에 특정 값을 부여하여 상수로 다룹니다. ACTION_BUY는 매수 행동을 의미하고 0을 값으로 가지고, ACTION_SELL은 매도 행동이고 1을 할당했고, ACTION_HOLD는 매수도 매도도 하지 않는 관망 행동으로 2의 값을 할당합니다. 이들 행동들 중에서 정책 신경망이 확률을 계산할 행동들을 ACTIONS 리스트에 저장합니다. RLTrader는 매수와 매도에 대한 확률만 계산하고 매수와 매도 중에서 결정한 행동을 할 수 없을 때만 관망 행동을 합니다.

4.4.3 코드 조각 2: 에이전트 클래스의 생성자 부분

이제 예제 4.3에서 에이전트 클래스의 생성자 함수를 살펴보겠습니다.

예제 4.3 Agent 클래스의 생성자 __init__()

https://github.com/quantylab/rltrader/blob/master/agent.py

```
19      def __init__(
20          self, environment, min_trading_unit = 1, max_trading_unit = 2,
21          delayed_reward_threshold = .05):
22          # Environment 객체
23          self.environment = environment  # 현재 주식 가격을 가져오기 위해 환경 참조
24
25          # 최소 매매 단위, 최대 매매 단위, 지연 보상 임계치
26          self.min_trading_unit = min_trading_unit  # 최소 단일 거래 단위
27          self.max_trading_unit = max_trading_unit  # 최대 단일 거래 단위
28          self.delayed_reward_threshold = delayed_reward_threshold  # 지연 보상 임계치
29
30          # Agent 클래스의 속성
31          self.initial_balance = 0  # 초기 자본금
```

```
        self.balance = 0  # 현재 현금 잔고
        self.num_stocks = 0  # 보유 주식 수
        self.portfolio_value = 0  # balance + num_stocks * {현재 주식 가격}
        self.base_portfolio_value = 0  # 직전 학습 시점의 PV
        self.num_buy = 0  # 매수 횟수
        self.num_sell = 0  # 매도 횟수
        self.num_hold = 0  # 관망 횟수
        self.immediate_reward = 0  # 즉시 보상

        # Agent 클래스의 상태
        self.ratio_hold = 0  # 주식 보유 비율
        self.ratio_portfolio_value = 0  # 포트폴리오 가치 비율
```

생성자의 파라미터로 environment, min_trading_unit, max_trading_unit을 받습니다. environment는 Environment 클래스의 객체입니다. min_trading_unit은 최소한의 매매 단위이고 max_trading_unit은 최대의 매매 단위로 각각 1과 2로 주었습니다. max_trading_unit을 크게 잡으면 결정한 행동에 대한 확신이 높을 때 더 많이 매수 또는 매도할 수 있게 설계했습니다. delayed_reward_threshold는 지연 보상 임계치로, 손익률이 이 값을 넘으면 지연 보상이 발생합니다.

initial_balance는 초기 자본금으로, 투자 시작 시점의 보유 현금을 의미합니다. balance는 현재의 현금 잔고입니다. num_stocks는 현재의 보유 주식 수입니다. portfolio_value는 포트폴리오 가치로, 보유 현금과 보유 주식 수에 현재 주가를 곱하여 더한 값입니다. 기준 포트폴리오 가치는 목표 수익률 또는 기준 손실률을 달성하기 전의 과거 포트폴리오 가치로, 현재 포트폴리오 가치가 증가했는지 감소했는지를 비교할 기준이 됩니다.

num_buy, num_sell, num_hold는 각기 에어전트가 행한 매수, 매도, 관망 횟수입니다. immediate_reward는 에이전트가 가장 최근에 행한 행동에 대한 즉시 보상 값입니다. 즉시 보상은 행동을 수행한 시점에서 수익이 발생한 상태면 1을, 아니었으면 −1을 주었습니다.

ratio_hold는 주식 보유 비율로 최대로 보유할 수 있는 주식 수 대비 현재 보유하고 있는 주식 수의 비율입니다. ratio_portfolio_value는 직전 지연 보상이 발생했을 때의 포트폴리오 가치 대비 현재의 포트폴리오 가치의 비율입니다.

4.4.4 코드 조각 3: 에이전트 클래스의 함수 부분

예제 4.4에서 에이전트 클래스의 속성 값 획득(get) 함수나 설정(set) 함수를 살펴보겠습니다.

예제 4.4 Agent 클래스의 함수: Get/Set 함수

https://github.com/quantylab/rltrader/blob/master/agent.py

```
45      def reset(self):
46          self.balance = self.initial_balance
47          self.num_stocks = 0
48          self.portfolio_value = self.initial_balance
49          self.base_portfolio_value = self.initial_balance
50          self.num_buy = 0
51          self.num_sell = 0
52          self.num_hold = 0
53          self.immediate_reward = 0
54          self.ratio_hold = 0
55          self.ratio_portfolio_value = 0
56
57      def set_balance(self, balance):
58          self.initial_balance = balance
59
60      def get_states(self):
61          self.ratio_hold = self.num_hold / int(
62              self.portfolio_value / self.environment.get_price())
63          self.ratio_portfolio_value = self.portfolio_value / self.initial_balance
64          return (
65              self.ratio_hold,
66              self.ratio_portfolio_value
67          )
```

45번 줄의 reset() 함수는 Agent 클래스의 속성들을 초기화해 줍니다. 학습 단계에서 한 에포크마다 에이전트의 상태를 초기화해야 합니다. 57번 줄의 set_balance() 함수는 에이전트의 초기 자본금을 설정합니다. 60번 줄의 get_states() 함수는 에이전트의 상태를 반환합니다. 상태는 다음 두 값으로 구성됩니다.

- **주식 보유 비율 = 보유 주식 수 / (포트폴리오 가치 / 현재 주가)**

  주식 보유 비율은 현재 상태에서 가장 많이 가질 수 있는 주식 수 대비 현재 보유한 주식의 비율입니다. 이 값이 0이면 주식을 하나도 보유하지 않은 것이고 0.5이면 최대 가질 수 있는 주식 대비 절반의 주식을 보유하고 있는 것이고 1이면 최대로 주식을 보유하고 있는 것입니다. 아무래도 주식 수가 너무 적으면 매수의 관점에서 투자에 임하고 주식 수가 너무 많으면 매도의 관점에서 투자에 임하게 됩니다. 즉 보유 주식 수를 투자 행동 결정에 영향을 주기 위해서 정책 신경망의 입력에 포함합니다.

- **포트폴리오 가치 비율 = 포트폴리오 가치 / 기준 포트폴리오 가치**

  포트폴리오 가치 비율은 기준 포트폴리오 가치 대비 현재 포트폴리오 가치의 비율입니다. 기준 포트폴리오 가치는 직전에 목표 수익 또는 손익률을 달성했을 때의 포트폴리오 가치입니다. 이 값은 현재 수익이 발생했는지 아니면 손실이 발생했는지를 판단할 수 있는 값이 됩니다. 포트폴리오 가치 비율이 0에 가까우면 손실이 큰 것이고 1보다 크면 수익이 발생했다는 뜻입니다. 수익률이 목표 수익률에 가까우면 매도의 관점에서 투자를 하곤 합니다. 수익률이 투자 행동 결정에 영향을 줄 수 있기 때문에 이 값을 에이전트의 상태로 정하고 정책 신경망의 입력값으로 포함합니다.

int()는 숫자를 정수로 캐스팅(즉, 형식 변환)하는 파이썬 내장 함수입니다. 예를 들어, int(0.5)는 0을, int(1.3)은 1을 반환합니다.

파이썬에서 (a, b, …)는 튜플(tuple)을 의미합니다. 튜플은 리스트 [a, b, …]와 비슷합니다. 차이점은 튜플은 요소를 추가, 변경, 삭제하는 처리가 불가능하고 리스트는 가능하다는 것입니다. 튜플이 메모리를 적게 사용하기 때문에 요소를 수정할 필요가 없으면 튜플을 사용하는 것이 효율적입니다.

예제 4.5는 에이전트가 행동을 결정하고 결정한 행동의 유효성을 검사하는 함수를 보여줍니다.

예제 4.5 Agent 클래스의 함수: 행동 결정 및 유효성 검사 함수

https://github.com/quantylab/rltrader/blob/master/agent.py

```
69      def decide_action(self, policy_network, sample, epsilon):
70          confidence = 0.
71          # 탐험 결정
72          if np.random.rand() < epsilon:
73              exploration = True
74              action = np.random.randint(self.NUM_ACTIONS)  # 무작위로 행동 결정
75          else:
76              exploration = False
77              probs = policy_network.predict(sample)  # 각 행동에 대한 확률
78              action = np.argmax(probs)
79              confidence = probs[action]
80          return action, confidence, exploration
81
82      def validate_action(self, action):
83          validity = True
84          if action == Agent.ACTION_BUY:
85              # 적어도 1주를 살 수 있는지 확인
86              if self.balance < self.environment.get_price() * (
87                  1 + self.TRADING_CHARGE) * self.min_trading_unit:
88                  validity = False
89          elif action == Agent.ACTION_SELL:
90              # 주식 잔고가 있는지 확인
91              if self.num_stocks <= 0:
92                  validity = False
93          return validity
```

69번 줄의 decide_action()은 입력으로 들어온 엡실론(epsilon)의 확률로 무작위로 행동을 결정하고, 그렇지 않은 경우에 정책 신경망을 통해 행동을 결정합니다.

72번 줄에서 0에서 1 사이의 랜덤 값을 생성하고 이 값이 엡실론보다 작으면 무작위로 행동을 결정합니다. 여기서 self.NUM_ACTIONS는 2의 값을 가집니다. 그러므로 랜덤으로 행동을 결정하면 0(매수) 또는 1(매도)의 값을 결정하는 것입니다.

탐험을 하지 않는 경우 77번 줄에서 정책 신경망을 통해 행동을 결정합니다. 정책 신경망 클래스의 predict() 함수를 사용하여 현재의 상태에서 매수와 매도의 확률을 받아옵니다. 이렇게 받아온 확률 중에서 큰 값을 선택하여 행동으로 결정합니다. 정책 신경망 클래스의 함수는 4.5장에서 상세하게 다룹니다.

NumPy의 random 모듈은 랜덤 값 생성을 위한 rand() 함수를 제공합니다. 이 함수는 0에서 1 사이의 값을 생성하여 반환합니다. randint(low, high=None) 함수는 high를 넣지 않은 경우 0에서 low 사이의 정수를 랜덤으로 생성하고 high를 넣은 경우 low에서 high 사이의 정수를 생성합니다.

NumPy의 argmax(array) 함수는 입력으로 들어온 array에서 가장 큰 값의 위치를 반환합니다. 예를 들어, array가 [3, 5, 7, 0, -3]이면 가장 큰 값은 7이므로 그 위치인 2를 반환합니다. 파이썬에서 위치(Index)는 0부터 시작합니다.

RLTrader에서는 신용 매수나 공매도는 고려하지 않습니다. 신용 매수는 잔금이 부족하더라도 돈을 빌려서 매수를 하는 것이고, 공매도는 주식을 보유하지 않고 있더라도 미리 매도하고 나중에 주식을 사서 갚는 방식의 거래입니다.

그러므로 이렇게 결정한 행동은 특정 상황에서는 수행할 수 없을 수도 있습니다. 예를 들어, 매수를 결정했는데 잔금이 1주를 매수하기에도 부족한 경우나 매도를 결정했는데 보유하고 있는 주식이 하나도 없는 경우에 결정한 행동을 수행할 수 없습니다. 그래서 결정한 행동을 수행할 수 있는지를 확인하기 위해서 78번 줄의 validate_action() 함수를 사용합니다.

80번 줄에서는 매수 결정에 대해서 적어도 1주를 살 수 있는 잔금이 있는지 확인합니다. 이때 설정된 거래 수수료까지 고려합니다.

84번 줄에서는 매도 결정에 대해서 주식 잔고가 있는지 확인합니다. self.num_stocks 변수에서 현재 보유하고 있는 주식 수를 저장하고 있습니다.

예제 4.6에서는 정책 신경망에서 결정한 행동의 확률에 따라서 매수 또는 매도의 단위를 조정하는 함수를 살펴봅니다.

예제 4.6 Agent 클래스의 함수: 매수/매도 단위 결정 함수

https://github.com/quantylab/rltrader/blob/master/agent.py

```
95          def decide_trading_unit(self, confidence):
96              if np.isnan(confidence):
97                  return self.min_trading_unit
98              added_traiding = max(min(
99                  int(confidence * (self.max_trading_unit - self.min_trading_unit)),
100                 self.max_trading_unit - self.min_trading_unit
101             ), 0)
102             return self.min_trading_unit + added_traiding
```

95번 줄의 decide_trading_unit() 함수는 정책 신경망이 결정한 행동의 확률이 높을수록 매수 또는 매도하는 단위를 크게 정해줍니다. 높은 확률로 매수를 결정했으면 그에 맞게 더 많은 주식을 매수하고 높은 확률로 매도를 결정했으면 더 많은 보유 주식을 매도하는 것입니다.

예제 4.7에서는 투자 행동을 수행하는 함수의 초반부를 살펴봅니다.

예제 4.7 Agent 클래스의 함수: 투자 행동 수행 함수 (행동 준비)

https://github.com/quantylab/rltrader/blob/master/agent.py

```
104         def act(self, action, confidence):
105             if not self.validate_action(action):
106                 action = Agent.ACTION_HOLD
107
108             # 환경에서 현재 가격 얻기
109             curr_price = self.environment.get_price()
110
111             # 즉시 보상 초기화
112             self.immediate_reward = 0
```

104번 줄의 act() 함수는 에이전트가 결정한 행동을 수행합니다. 인자로는 action과 confidence를 받습니다. action은 탐험 또는 정책 신경망을 통해 결정한 행동으로 매수와 매도를 의미하는 0 또는 1의 값입니다. confidence는 정책 신경망을 통해 결정한 경우 결정한 행동에 대한 소프트맥스 확률 값입니다.

먼저 105번 줄에서 이 행동을 할 수 있는지 확인하고, 할 수 없는 경우 아무 행동도 하지 않도록 관망(hold)합니다.

109번 줄에서는 환경 객체에서 현재의 주가를 받아옵니다. 이 가격은 매수 금액, 매도 금액, 포트폴리오 가치를 계산할 때 사용됩니다.

즉시 보상은 에이전트가 행동을 할 때마다 결정되기 때문에 112번 줄에서 초기화합니다.

예제 4.8에서는 act() 함수의 매수 행동 수행 부분을 이어서 살펴봅니다.

예제 4.8 Agent 클래스의 함수: 투자 행동 수행 함수 (매수)

https://github.com/quantylab/rltrader/blob/master/agent.py

```
114             # 매수
115             if action == Agent.ACTION_BUY:
116                 # 매수할 단위를 판단
117                 trading_unit = self.decide_trading_unit(confidence)
118                 balance = self.balance - curr_price * (1 + self.TRADING_CHARGE) * trading_unit
119                 # 보유 현금이 모자랄 경우 보유 현금으로 가능한 만큼 최대한 매수
120                 if balance < 0:
121                     trading_unit = max(min(
122                         int(self.balance / (
123                             curr_price * (1 + self.TRADING_CHARGE))), self.max_trading_unit),
124                         self.min_trading_unit
125                     )
126                 # 수수료를 적용하여 총 매수 금액 산정
127                 invest_amount = curr_price * (1 + self.TRADING_CHARGE) * trading_unit
128                 self.balance -= invest_amount  # 보유 현금을 갱신
129                 self.num_stocks += trading_unit  # 보유 주식 수를 갱신
130                 self.num_buy += 1  # 매수 횟수 증가
```

수행할 행동이 매수인지를 115번 줄에서 확인합니다. 117번 줄에서 매수 단위(살 주식 수)를 정합니다. 그리고 118번 줄에서 매수 후의 잔금을 확인합니다. 매수 후에 잔금이 0보다 적을 수 없기 때문에 120번 줄에서 확인합니다.

121번 줄에서 결정한 매수 단위가 최대 단일 거래 단위를 넘어가면 최대 단일 거래 단위로 제한하고 최소 거래 단위보다 최소한 1주를 매수하도록 합니다.

파이썬의 내장 함수인 min(a, b)과 max(a, b) 함수에 대해서 알아보겠습니다. min(a, b)는 a와 b 중에서 작은 값을 반환합니다. 반대로 max(a, b)는 a와 b 중에서 큰 값을 반환합니다. 예를 들어, min(5, 10)은 5를, max(–5, –10)은 –5를 반환합니다.

127번 줄에서 매수할 단위에 수수료를 적용하여 총 투자 금액을 계산합니다. 그리고 128번 줄에서 이 금액을 현재 잔금에 빼고 129번 줄에서 주식 보유 수를 투자 단위만큼 늘려 줍니다. 그리고 130번 줄에서 통계 정보인 self.num_buy를 1만큼 증가시킵니다.

예제 4.9에서는 이어서 매도와 관망에 대한 부분을 살펴보겠습니다.

예제 4.9 Agent 클래스의 함수: 투자 행동 수행 함수 (매도 및 관망)

https://github.com/quantylab/rltrader/blob/master/agent.py

```
132             # 매도
133             elif action == Agent.ACTION_SELL:
134                 # 매도할 단위를 판단
135                 trading_unit = self.decide_trading_unit(confidence)
136                 # 보유 주식이 모자랄 경우 가능한 만큼 최대한 매도
137                 trading_unit = min(trading_unit, self.num_stocks)
138                 # 매도
139                 invest_amount = curr_price * (
140                     1 - (self.TRADING_TAX + self.TRADING_CHARGE)) * trading_unit
141                 self.num_stocks -= trading_unit  # 보유 주식 수를 갱신
142                 self.balance += invest_amount  # 보유 현금을 갱신
143                 self.num_sell += 1  # 매도 횟수 증가
144
145             # 관망
146             elif action == Agent.ACTION_HOLD:
147                 self.num_hold += 1  # 관망 횟수 증가
```

133번 줄에서 매도인지 확인합니다. 매수 때와 마찬가지로 투자 단위를 판단합니다. 여기서는 투자 단위가 매도할 주식 수가 됩니다. 현재 가지고 있는 주식 수보다 결정한 매도 단위가 많으면 안 되기 때문에 137번 줄에서 현재 보유 주식 수를 최대 매도 단위로 제한합니다.

139번 줄에서 매도 금액을 계산합니다. 이 때 정해준 매도 수수료와 거래세를 모두 고려하여 계산합니다. 그리고 141번 줄에서 보유 주식 수를 매도한만큼 빼고 142번 줄에서 매도하여 현금화한 금액을 잔고에서 더해 줍니다. 143번 줄에서 통계를 위한 변수인 self.num_sell을 1만큼 증가시킵니다.

146번 줄에서는 관망으로 결정된 경우를 판별합니다. 관망(hold)은 결정한 매수(buy)나 매도(sell) 행동을 수행할 수 없는 경우에 적용됩니다. 관망은 아무것도 하지 않는 것이기 때문에 보유 주식 수나 잔고에 영향을 미치지 않습니다. 대신 가격 변동은 있을 수 있으므로 포트폴리오 가치는 변동됩니다. 147번 줄에서 관망 횟수를 1만큼 증가시킵니다.

예제 4.10에서는 이어서 act() 함수의 후반부를 살펴봅니다.

예제 4.10 Agent 클래스의 함수: 투자 행동 수행 함수 (포트폴리오 가치 갱신 및 지연 보상 판단)

https://github.com/quantylab/rltrader/blob/master/agent.py

```
149             # 포트폴리오 가치 갱신
150             self.portfolio_value = self.balance + curr_price * self.num_stocks
151             profitloss = (
152                 (self.portfolio_value - self.base_portfolio_value) / self.base_portfolio_value)
153
154             # 즉시 보상 판단
155             self.immediate_reward = 1 if profitloss >= 0 else -1
156
157             # 지연 보상 판단
158             if profitloss > self.delayed_reward_threshold:
159                 delayed_reward = 1
160                 # 목표 수익률을 달성하여 기준 포트폴리오 가치 갱신
161                 self.base_portfolio_value = self.portfolio_value
162             elif profitloss < -self.delayed_reward_threshold:
163                 delayed_reward = -1
164                 # 손실 기준치를 초과하여 기준 포트폴리오 가치 갱신
165                 self.base_portfolio_value = self.portfolio_value
166             else:
167                 delayed_reward = 0
168             return self.immediate_reward, delayed_reward
```

이렇게 매수, 매도, 또는 관망을 하면 포트폴리오 가치에 변동이 생깁니다. 포트폴리오 가치는 150번 줄에서 잔고, 주식 보유 수, 현재 주식 가격에 의해 결정됩니다. 151번 줄에서는 기준 포트폴리오 가치에서 현재 포트폴리오 가치의 등락률을 계산합니다. 기준 포트폴리오 가치는 과거에 학습을 수행한 시점의 포트폴리오 가치를 의미한다고 했습니다.

155번 줄에서 즉시 보상을 판단합니다. 즉시 보상은 수익이 발생한 상태면 1을, 그렇지 않으면 −1을 부여했습니다.

158번 줄에서는 지연 보상을 판단합니다. 지연 보상은 포트폴리오 가치의 등락률인 profitloss가 지연 보상 임계치인 delayed_reward_threshold를 수익으로 초과하는 경우 1을, 손실로 초과하는 경우 −1을, 그 외의 경우엔 0을 반환합니다. RLTrader는 지연 보상이 0이 아닌 경우 학습을 수행합니다. 즉 지연 보상 임계치를 초과하는 수익이 났으면 이전에 했던 행동들이 잘했다고 보고 긍정적으로(positive) 학습하고, 지연 보상 임계치를 초과하는 손실이 났으면 이전 행동들에 문제가 있다고 보고 부정적으로(negative) 학습합니다.

4.5 정책 신경망 모듈 개발

이번 절에서는 정책 신경망 모듈에 포함된 정책 신경망 클래스의 속성과 함수를 살펴보고 파이썬으로 구현한 소스코드를 상세히 확인합니다.

4.5.1 정책 신경망 모듈의 주요 속성과 함수

정책 신경망 모듈(policy_network.py)은 투자 행동을 결정하기 위해 신경망을 관리하는 정책 신경망 클래스(PolicyNetwork)를 가집니다. 정책 신경망 클래스의 속성과 함수는 다음과 같습니다.

- **속성**

  model: 케라스 라이브러리로 구성한 LSTM 신경망 모델

  prob: 가장 최근에 계산한 투자 행동별 확률

- **함수**

  reset(): prob 변수를 초기화

  predict(): 신경망을 통해 투자 행동별 확률 계산

  train_on_batch(): 배치 학습을 위한 데이터 생성

  save_model(): 학습한 신경망을 파일로 저장

  load_model(): 파일로 저장한 신경망을 로드

4.5.2 정책 신경망에서 사용하는 LSTM 신경망의 구조

그림 4.3은 정책 신경망에서 사용하는 LSTM 신경망의 구조를 보여줍니다.

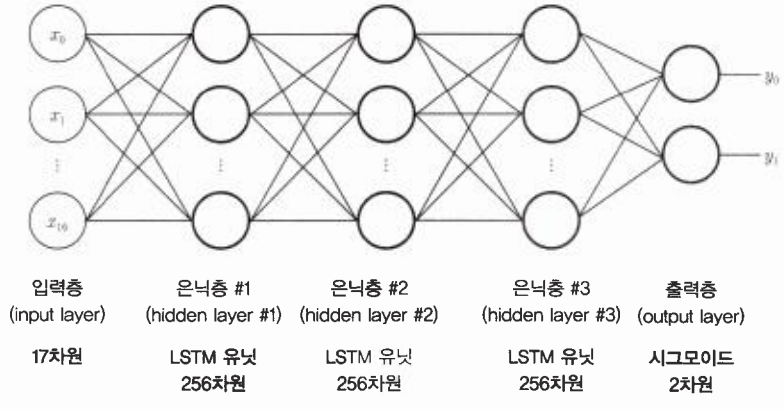


그림 4.3 정책 신경망의 LSTM 신경망 구조

총 5개 층으로 구성되고 입력층은 17차원이고 출력층은 2차원입니다. 입력층은 학습 데이터 차원인 15차원과 에이전트 상태 차원인 2차원을 합한 17차원인 것이고 출력층은 투자 행동인 매수와 매도로 2차원을 가집니다. 여기서 은닉층의 수와 차원은 조절할 수 있습니다.

4.5.3 코드 조각 1: 정책 신경망 클래스의 생성자 부분

예제 4.11은 정책 신경망 클래스의 생성자 부분 소스코드입니다.

예제 4.11 PolicyNetwork 클래스 생성자

https://github.com/quantylab/rltrader/blob/master/policy_network.py

```
import numpy as np
from keras.models import Sequential
from keras.layers import Activation, LSTM, Dense, BatchNormalization
from keras.optimizers import sgd


class PolicyNetwork:
    def __init__(self, input_dim=0, output_dim=0, lr=0.01):
        self.input_dim = input_dim
        self.lr = lr

        # LSTM 신경망
        self.model = Sequential()

        self.model.add(LSTM(256, input_shape=(1, input_dim),
                            return_sequences=True, stateful=False, dropout=0.5))
        self.model.add(BatchNormalization())
        self.model.add(LSTM(256, return_sequences=True, stateful=False, dropout=0.5))
        self.model.add(BatchNormalization())
        self.model.add(LSTM(256, return_sequences=False, stateful=False, dropout=0.5))
        self.model.add(BatchNormalization())
        self.model.add(Dense(output_dim))
        self.model.add(Activation('sigmoid'))

        self.model.compile(optimizer=sgd(lr=lr), loss='mse')
        self.prob = None
```

PolicyNetwork 클래스는 파이썬 딥러닝 라이브러리인 케라스를 사용하여 LSTM 신경망을 구성합니다. 세 개의 LSTM 층을 256 차원으로 구성했으며 드롭아웃을 50%로 정하여 과적합을 피합니다.

파이썬에서는 딥러닝을 위해 주로 텐서플로, 띠아노(Theano), 케라스, 라자냐(Lasagne), 카페(Caffe) 등의 라이브러리를 사용합니다. 이들 중 케라스는 텐서플로 또는 띠아노를 백엔드(backend)로 사용하는 상위 라이브러리입니다. 즉, 케라스는 텐서플로나 띠아노를 좀 더 쉽게 사용할 수 있게 해 주는 래퍼(wrapper) 라이브러리입니다. Lasagne도 비슷하게 띠아노를 좀 더 쉽게 사용할 수 있게 해줍니다. 각 라이브러리들의 URL은 다음과 같습니다.

- 텐서플로: https://www.tensorflow.org/
- 띠아노: http://deeplearning.net/software/theano/
- 케라스: https://keras.io/
- 라자냐: https://github.com/Lasagne/Lasagne
- 카페: http://caffe.berkeleyvision.org/

다양한 딥러닝 라이브러리들 중에서 깃허브 저장소(Github repository)의 참여(fork) 횟수나 인기(star) 점수를 근거로 사용자가 가장 많다고 판단되는 라이브러리는 텐서플로입니다. 2017년 11월 기준 텐서플로의 참여 횟수와 인기 점수는 각각 약 38,000회, 약 78,000점입니다. 띠아노의 경우 참여 횟수는 약 2,000회, 인기 점수는 약 7,000점입니다. 카페의 경우 참여 횟수는 약 13,000회 인기 점수는 약 21,000점입니다. 텐서플로가 띠아노에 비해 10배가 넘게 선호되고 카페에 비해 거의 3배 이상의 선호가 있는 셈입니다. 그래서 텐서플로를 쓰기로 결정을 했고 이를 더 쉽고 간단하게 사용하고 싶어서 케라스를 최종적으로 선택했습니다. 물론 다른 라이브러리를 사용하고 싶다면 이 모듈을 수정하면 됩니다.

13번 줄에서 23번 줄까지는 신경망의 층들을 구성하는 코드이고 25번 줄에서 최적화 알고리즘(optimization algorithm)과 학습 속도를 정하여 신경망 모델을 준비합니다. 여기서 기본 학습 알고리즘은 확률적 경사 하강법(SGD)을, 기본 학습 속도(learning rate, LR)는 0.01로 정했습니다.

케라스에서 Sequential 클래스는 전체 신경망을 구성하는 클래스입니다. 전체 신경망에서 하나의 노드를 LSTM 클래스로 구성합니다. LSTM 클래스는 그림 4.3에서 진한 원에 해당됩니다.

텐서플로, 케라스와 관련해 더 깊게 공부하고 싶다면 ≪파이썬을 이용한 머신러닝, 딥러닝 실전 개발 입문≫(위키북스, 2017)을 추천합니다. 텐서플로와 케라스의 공식 문서 사이트 또한 참고할 것을 추천합니다.

- 텐서플로 공식 문서 튜토리얼 사이트: https://www.tensorflow.org/tutorials/
- 케라스 공식 문서 사이트: https://keras.io/

4.5.4 코드 조각 2: 정책 신경망 클래스의 함수 선언 부분

예제 4.12에서는 정책 신경망 클래스의 함수들을 살펴봅니다.

예제 4.12 PolicyNetwork 클래스의 함수

https://github.com/quantylab/rltrader/blob/master/policy_network.py

```
28      def reset(self):
29          self.prob = None
30
31      def predict(self, sample):
32          self.prob = self.model.predict(np.array(sample).reshape((1, -1, self.input_dim)))[0]
33          return self.prob
34
35      def train_on_batch(self, x, y):
36          return self.model.train_on_batch(x, y)
37
38      def save_model(self, model_path):
39          if model_path is not None and self.model is not None:
40              self.model.save_weights(model_path, overwrite=True)
41
42      def load_model(self, model_path):
43          if model_path is not None:
44              self.model.load_weights(model_path)
```

28번 줄의 reset() 함수는 self.prob 변수를 초기화하는 간단한 역할만 합니다. 31번 줄의 predict() 함수는 신경망을 통해서 학습 데이터와 에이전트 상태를 합한 17 차원의 입력을 받아서 매수와 매도가 수익을 높일 것으로 판단되는 확률을 구합니다.

32번 줄에서 Sequential 클래스의 predict() 함수는 여러 샘플을 한꺼번에 받아서 신경망의 출력을 반환합니다. 하나의 샘플에 대한 결과만을 받고 싶어도 샘플의 배열로 입력값을 구성해야 하기 때문에 2차원 배열로 재구성했습니다.

NumPy의 array() 함수는 파이썬 리스트를 n차원 배열(약칭 ndarray) 형식으로 만들어 줍니다. 즉 파이썬 기본 리스트 자료구조를 NumPy 라이브러리에서 사용하는 자료구조로 변환하는 것입니다.

그림 4.4와 같이 sample의 크기는 self.input_dim의 값인 17입니다. 이 배열을 1행 17열인 2차원 배열로 모양을 변환합니다.

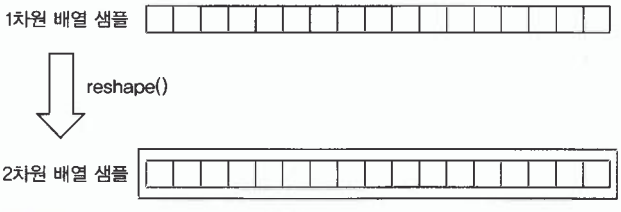


그림 4.4 sample을 케라스 입력 형식에 맞게 변환

NumPy의 ndarray를 reshape() 함수를 써서 다른 차원으로 변환할 수 있습니다. 예를 들어 1차원 배열인 [1, 2, 3, 4, 5, 6]이 있을 때 이 배열의 shape(즉, 모양)은 (6,)입니다. 이를 (3, 2)로 만들면 [[1, 2], [3, 4], [5, 6]]이 됩니다. 이 때 유의할 점은 배열의 총 크기는 변하지 않아야 합니다. 위 예에서는 변환한 후의 크기가 3 × 2 = 6 이므로 1차원 배열일 때와 같습니다.

35번 줄의 train_on_batch() 함수는 입력으로 들어온 학습 데이터 집합 x와 레이블 y로 정책 신경망을 학습시킵니다. 이 함수는 4.7장에서 다룰 정책 신경망 모듈에서 호출합니다. 입력으로 들어올 x와 y는 정책 학습기에서 준비합니다.

케라스의 Sequential 클래스 함수인 train_on_batch()는 입력으로 들어온 학습 데이터 집합(즉, 1개 배치)으로 신경망을 한 번 학습합니다.

상세한 설명은 https://keras.io/models/sequential/#train_on_batch에서 확인할 수 있습니다.

38번 줄의 save_model() 함수는 학습한 정책 신경망을 파일로 저장합니다. 인자로 넘겨지는 model_path는 저장할 파일명을 의미합니다. 42번 줄의 load_model()은 저장한 정책 신경망을 불러오기 위한 함수입니다.

케라스의 Sequential 클래스 함수인 save_weights()는 인공 신경망을 구성하기 위한 값들을 HDF5 파일로 저장합니다. 이렇게 저장한 파일을 load_weights() 함수로 불러올 수 있습니다. 즉, 훈련한 인공지능 신경망 모델을 HDF5 파일 형식으로 저장해 둘 수 있는 것입니다. 그랬다가 해당 모델을 활용해야 할 때 HDF5 파일을 load_weights() 함수로 불러 오면 즉각 활용할 수 있습니다.

상세한 설명은 https://keras.io/models/about-keras-models/#about-keras-models에서 확인할 수 있습니다.

정책 신경망을 학습하는 데는 많은 시간과 컴퓨팅 자원이 소모되므로, 한번 학습한 정책 신경망을 저장해 놓고 필요할 때 불러와서 사용하는 것은 필수적인 기능입니다.

현재까지의 데이터로 정책 신경망을 학습하고 파일로 저장한 다음 앞으로의 투자에서 저장한 정책 신경망 모델을 불러와서 사용할 수 있습니다. 불러온 정책 신경망에서 추가적으로 학습을 진행하여 개선된 정책 신경망을 만들 수도 있습니다.

4.6 가시화기 모듈 개발

가시화기 모듈에 포함된 가시화기 클래스의 속성과 함수를 살펴보고 파이썬으로 구현한 소스코드를 상세히 확인합니다.

4.6.1 가시화기 모듈의 주요 속성과 함수

가시화기 모듈(visualizer.py)은 정책 신경망을 학습하는 과정에서 에이전트의 투자 상황, 정책 신경망의 투자 결정 상황, 포트폴리오 가치의 상황을 시간에 따라 연속적으로 보여주기 위해 시각화 기능을 담당하는 가시화기 클래스(Visualizer)를 가집니다. 가시화기 클래스의 속성, 함수는 다음과 같습니다.

- **속성**

  fig: 캔버스 같은 역할을 하는 Matplotlib의 Figure 클래스 객체

  axes: 차트를 그리기 위한 Matplotlib의 Axes 클래스 객체

- **함수**

  prepare(): Figure를 초기화하고 일봉 차트를 출력

  plot(): 일봉 차트를 제외한 나머지 차트들을 출력

  save(): Figure를 그림 파일로 저장

  clear(): 일봉 차트를 제외한 나머지 차트들을 초기화

4.6.2 가시화기 모듈이 만들어 내는 정보

가시화기 모듈이 만들어내는 결과는 다음 정보를 포함합니다.

- Figure 제목: 에포크 및 탐험률
- Axes 1: 종목의 일봉 차트
- Axes 2: 보유 주식 수 및 에이전트 행동 차트
- Axes 3: 정책 신경망 출력 및 탐험 차트
- Axes 4: 포트폴리오 가치 차트

에포크 및 탐험률은 Figure의 제목으로 표시합니다. 그 외 각 차트들은 각기 Axes로 구성되어 있습니다. Axes 하나가 하나의 차트라고 보면 되며 최종 Figure의 구조는 그림 4.5와 같습니다.

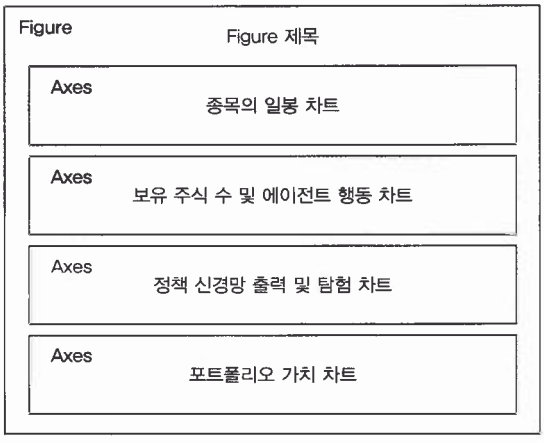


그림 4.5 Visualizer가 그리는 최종 Figure 구조

Figure는 Axes 네 개를 포함합니다. 4행 1열 구조로 Axes들이 세로로 나열되어 있습니다. 위에서부터 종목의 일봉 차트, 보유 주식 수 및 에이전트 행동 차트, 정책 신경망 출력 및 탐험 차트, 포트폴리오 가치 차트 순으로 가시화합니다.

4.6.3 코드 조각 1: 가시화기 클래스의 생성자 부분

일봉 차트를 그리기 위해서 mpl_finance 모듈을 사용합니다. 이 모듈은 원래 Matplotlib 라이브러리에 포함되어 있었지만 Matplotlib 2.0 버전부터 향후 삭제 예정(deprecated) 상태가 되었습니다. 향후 삭제 예정 상태는 해당 기능의 중요도가 떨어져 언젠가는 해당 기능을 없앨 것이므로 더 이상 사용하지 말기를 권장한다는 의미입니다. 저자는 그 이유를 도메인에 특화된 코드를 별도로 관리하여 Matplotlib의 코드를 깔끔하게 유지하기 위해서라 생각합니다.

mpl_finance 모듈은 다음 깃허브 저장소에서 내려받을 수 있습니다.

- https://github.com/matplotlib/mpl_finance

내려받은 후 'Anaconda Prompt'에서 mpl_finance 폴더로 이동하고 다음과 같이 설치할 수 있습니다.

- $ python setup.py install

Visualizer 클래스는 NumPy, Matplotlib, mpl_finance 모듈을 사용합니다. 예제 4.13에서 이 모듈들을 임포트하는 부분과 Visualizer 클래스의 선언부분을 보여줍니다.

예제 4.13 Visualizer 클래스의 생성자

https://github.com/quantylab/rltrader/blob/master/visualizer.py

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_finance import candlestick_ohlc


class Visualizer:

    def __init__(self):
        self.fig = None  # 캔버스 같은 역할을 하는 Matplotlib의 Figure 클래스 객체
        self.axes = None  # 차트를 그리기 위한 Matplotlib의 Axes 클래스 객체
```

Visualizer 클래스는 fig와 axes를 속성으로 가집니다. fig는 Figure 클래스의 객체로 전체 가시화 결과를 관리하고 axes는 Axes 클래스의 객체로 fig에 포함되는 차트의 배열입니다.

4.6.4 코드 조각 2: 일봉 차트 가시화 함수 부분

예제 4.14는 전체 차트를 그릴 준비를 하고 에포크에 상관없이 공통된 첫 번째 차트인 종목 일봉 차트를 그립니다.

예제 4.14 Visualizer 클래스의 prepare() 함수

https://github.com/quantylab/rltrader/blob/master/visualizer.py

```
    def prepare(self, chart_data):
        # 캔버스를 초기화하고 4개의 차트를 그릴 준비
        self.fig, self.axes = plt.subplots(nrows=4, ncols=1, facecolor='w', sharex=True)
        for ax in self.axes:
            # 보기 어려운 과학적 표기 비활성화
            ax.get_xaxis().get_major_formatter().set_scientific(False)
```

```
18              ax.get_yaxis().get_major_formatter().set_scientific(False)
19          # 차트 1. 일봉 차트
20          self.axes[0].set_ylabel('Env.')  # y 축 레이블 표시
21          # 거래량 가시화
22          x = np.arange(len(chart_data))
23          volume = np.array(chart_data)[:, -1].tolist()
24          self.axes[0].bar(x, volume, color='b', alpha=0.3)
25          # ohlc란 open, high, low, close의 약자로 이 순서로 구성된 2차원 배열
26          ax = self.axes[0].twinx()
27          ohlc = np.hstack((x.reshape(-1, 1), np.array(chart_data)[:, 1:-1]))
28          # self.axes[0]에 봉 차트 출력
29          # 양봉은 빨간색으로, 음봉은 파란색으로 표시
30          candlestick_ohlc(ax, ohlc, colorup='r', colordown='b')
```

14번 줄에서 4행 1열의 구조를 가지는 Figure를 생성합니다. 각 행에 해당하는 차트는 Axes 객체의 배열로 반환됩니다.

Matplotlib의 subplots() 함수는 여러 Axes로 구성된 Figure를 생성할 때 효과적입니다. 인자로 들어가는 nrows는 행 개수, ncols는 열 개수를 의미합니다. nrows가 2이고 ncols가 3이라면 2x3 Axes, 즉 6개의 Axes로 구성된 Figure가 생성됩니다.

subplots() 함수는 2개의 변수를 Tuple로 반환합니다. 첫 번째는 Figure 객체이고 두 번째는 이 Figure 클래스 객체에 포함된 Axes 객체의 배열입니다.

Matplotlib에서 주요 색깔 이름을 다음과 같이 줄여서 사용할 수 있습니다.

- 검정색: k, 흰색: w, 빨간색: r, 파란색: b, 초록색: g, 노란색: y

15번 줄부터 18번 줄까지는 모든 Axes가 숫자 표기 단위로 과학적 표기를 쓰지 않도록 합니다. 이 차트들이 주로 원, 확률 등을 표기하기 때문에 과학적 표기로 변환되면 보기가 어렵기 때문에 과학적 표기법을 비활성화시킵니다.

20번 줄부터는 첫 번째 차트인 일봉 차트를 가시화하는 코드입니다. 20번 줄에서는 y축 레이블을 지정합니다. 여기서는 Environment의 약자로 Env.라고 표기했습니다. 거래량을 표시하기 위해 24번 줄에서 막대 차트를 그립니다. 그리고 일봉 차트를 구성하기 위해 27번

줄에서 ohlc 데이터를 구성합니다. ohlc란 open, high, low, close의 약자로 N×5 차원의 배열입니다. 열의 수가 5인 이유는 첫 번째 열은 인덱스이기 때문입니다. N은 일봉의 수입니다.

ohlc 배열을 만들기 위해서 그림 4.6과 같이 1차원 인덱스 배열을 만들고 chart_data의 4번째 열까지를 수평적으로 붙입니다.

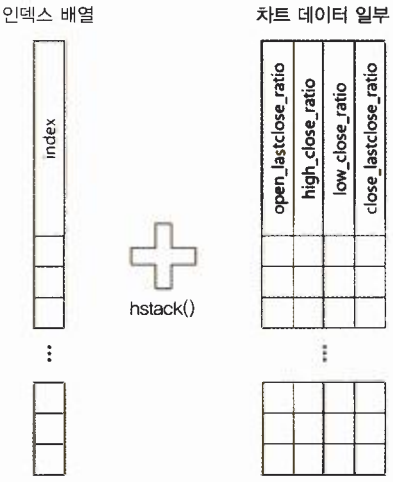


그림 4.6 수평적으로 인덱스 배열과 차트 데이터 일부를 붙여서 ohlc 데이터 구성

NumPy의 arrange() 함수는 0부터 입력으로 들어온 값만큼 순차적으로 값을 생성해서 배열로 반환합니다. 만약 np.arrange(3)이면 NumPy 배열 [0, 1, 2]를 반환합니다. 시작 값(start), 종료 값(stop), 값 사이 간격(step)을 옵션으로 넣을 수 있습니다.

mpl_finance 모듈의 candlestick_ohlc() 함수의 입력은 Axes 객체, ohlc 데이터입니다. 여기서 옵션으로 양봉(colorup) 및 음봉(colordown)의 색을 지정할 수 있습니다.

4.6.5 코드 조각 3: 전체 차트 가시화 함수 선언 부분

예제 4.15는 plot() 함수의 초반 부분을 보여줍니다.

예제 4.15 Visualizer 클래스의 plot() 함수 (1)

https://github.com/quantylab/rltrader/blob/master/visualizer.py

```
32      def plot(self, epoch_str=None, num_epoches=None, epsilon=None,
33              action_list=None, actions=None, num_stocks=None,
34              outvals=None, exps=None,
35              initial_balance=None, pvs=None):
36          x = np.arange(len(actions))  # 모든 차트가 공유할 x축 데이터
37          actions = np.array(actions)  # 에이전트의 행동 배열
38          outvals = np.array(outvals)  # 정책 신경망의 출력 배열
39          pvs_base = np.zeros(len(actions)) + initial_balance  # 초기 자본금 배열
```

plot() 함수의 인자는 다음과 같습니다.

- epoch_str: Figure 제목으로 표시할 에포크
- num_epoches: 총 수행할 에포크 수
- epsilon: 탐험률
- action_list: 에이전트가 수행할 수 있는 전체 행동 리스트
- actions: 에이전트가 수행한 행동 배열
- num_stocks: 주식 보유 수 배열
- outvals: 정책 신경망의 출력 배열
- exps: 탐험 여부 배열
- initial_balance: 초기 자본금
- pvs: 포트폴리오 가치 배열

먼저 36번 줄에서 모든 차트가 공유할 x축 데이터를 생성합니다. actions, num_stocks, outvals, exps, pvs의 모든 크기가 같기 때문에 이들 중 하나인 actions의 크기만큼 배열을 생성하여 x축으로 사용합니다.

그리고 Matplotlib이 NumPy 배열을 입력으로 받기 때문에 37번, 38번 줄에서 리스트들을 모두 NumPy 배열로 감싸줍니다.

포트폴리오 가치 차트에서 초기 자본금에 직선을 그어서 포트폴리오 가치와 초기 자본금을 쉽게 비교할 수 있도록 배열(pvs_base)로 준비합니다.

NumPy의 zeros() 함수는 인자로 배열의 형태인 shape를 받아서 0으로 구성된 NumPy 배열을 반환합니다. 예를 들어, zeros(3)은 [0, 0, 0]을, zeros((2, 2))는 [[0, 0], [0, 0]]을 반환합니다. 주의할 점은 다차원 배열의 경우 그 형태를 튜플로 넘겨줘야 한다는 것입니다.

4.6.6 코드 조각 4: 에이전트 상태 가시화 부분

예제 4.16은 에이전트 상태 차트를 그리는 코드를 보여줍니다. 에이전트가 수행한 행동을 배경의 색으로 표시하고 보유 주식 수를 라인 차트로 그립니다.

예제 4.16 Visualizer 클래스의 plot() 함수 (2)

https://github.com/quantylab/rltrader/blob/master/visualizer.py

```
41          # 차트 2. 에이전트 상태 (행동, 보유 주식 수)
42          colors = ['r', 'b']
43          for actiontype, color in zip(action_list, colors):
44              for i in x[actions == actiontype]:
45                  self.axes[1].axvline(i, color=color, alpha=0.1)  # 배경 색으로 행동 표시
46          self.axes[1].plot(x, num_stocks, '-k')  # 보유 주식 수 그리기
```

45번 줄에서 매수 행동의 배경 색을 빨간색으로, 매도 행동의 배경 색을 파란색으로 그립니다.

zip()은 파이썬 내장 함수로 두 개의 배열에서 같은 인덱스의 요소를 순서대로 묶어 줍니다. 예를 들어서 zip([1,2,3], [4,5,6])은 [(1, 4), (2, 5), (3, 6)]이 됩니다.

Matplotlib의 axvline()은 x축 위치에서 세로로 선을 긋는 함수입니다. 이 선의 색깔은 color 인자로, 선의 투명도를 alpha로 정해줄 수 있습니다.

그리고 46번 줄에서 보유 주식 수를 실선으로 그립니다.

Matplotlib의 plot() 함수는 x축의 데이터, y축의 데이터, 차트의 스타일을 인자로 받습니다. x축 데이터와 y축 데이터의 크기가 같아야 합니다. 스타일은 실선, 점선 등의 선 형태와 선의 색깔을 축약적으로 정의합니다. 예를 들어서 '-k'는 검정색 실선을 의미합니다.

4.6.7 코드 조각 5: 정책 신경망 출력 결과 및 탐험 수행 가시화 부분

예제 4.17은 정책 신경망의 출력 결과와 탐험을 수행한 부분을 가시화하는 소스코드를 보여줍니다.

예제 4.17 Visualizer 클래스의 plot() 함수 (3)

https://github.com/quantylab/rltrader/blob/master/visualizer.py

```
48          # 차트 3. 정책 신경망의 출력 및 탐험
49          for exp_idx in exps:
50              # 탐험을 노란색 배경으로 그리기
51              self.axes[2].axvline(exp_idx, color='y')
52          for idx, outval in zip(x, outvals):
53              color = 'white'
54              if outval.argmax() == 0:
55                  color = 'r'  # 매수면 빨간색
56              elif outval.argmax() == 1:
57                  color = 'b'  # 매도면 파란색
58              # 행동을 빨간색 또는 파란색 배경으로 그리기
59              self.axes[2].axvline(idx, color=color, alpha=0.1)
60          styles = ['.r', '.b']
61          for action, style in zip(action_list, styles):
62              # 정책 신경망의 출력을 빨간색, 파란색 점으로 그리기
63              self.axes[2].plot(x, outvals[:, action], style)
```

exps 배열이 탐험을 수행한 x축 인덱스를 가지고 있습니다. 51번 줄에서 탐험을 한 x축 인덱스에 대해서 배경을 노란색으로 표시합니다.

탐험을 하지 않은 지점에서는 에이전트의 행동을 52번 줄부터 59번 줄까지의 코드로 매수는 빨간색, 매도는 파란색으로 표시합니다.

63번 줄에서는 정책 신경망의 출력을 표시합니다. 매수에 대한 정책 신경망의 출력 값을 빨간색 점으로, 매도에 대한 정책 신경망의 출력 값을 파란색 점으로 그립니다. 빨간색 점이 파란색 점보다 위에 위치하면 에이전트가 매수를 했을 것이고 그 반대의 경우 에이전트는 매도를 수행했을 것입니다.

Matplotlib의 plot() 함수에서 스타일은 다양하게 구성할 수 있습니다. 표시할 다양한 모양과 색깔을 조합할 수 있습니다. 예를 들어, '-'은 선, '.'은 점, 'r'은 빨간색, 'b'는 파란색을 의미하는데, 이들의 조합인 '-r', '-b', '.r', '.b'는 각각 빨간 선, 파란 선, 빨간 점, 파란 점을 의미합니다.

4.6.8 코드 조각 6: 포트폴리오 가치 및 기타 정보 가시화 부분

예제 4.18은 포트폴리오 가치 차트를 그리고 제목을 포함하여 Figure를 최종적으로 구성합니다.

예제 4.18 Visualizer 클래스의 plot() 함수 (4)

https://github.com/quantylab/rltrader/blob/master/visualizer.py

```
65          # 차트 4. 포트폴리오 가치
66          self.axes[3].axhline(initial_balance, linestyle='-', color='gray')
67          self.axes[3].fill_between(x, pvs, pvs_base,
68                                    where=pvs > pvs_base, facecolor='r', alpha=0.1)
69          self.axes[3].fill_between(x, pvs, pvs_base,
70                                    where=pvs < pvs_base, facecolor='b', alpha=0.1)
71          self.axes[3].plot(x, pvs, '-k')
72          for learning_idx, delayed_reward in learning:
73              # 학습 위치 표시
74              if delayed_reward > 0:
75                  self.axes[3].axvline(learning_idx, color='r', alpha=0.1)
76              else:
77                  self.axes[3].axvline(learning_idx, color='b', alpha=0.1)
78
79          # 에포크 및 탐험 비율
80          self.fig.suptitle('Epoch %s/%s (e=%.2f)' % (epoch_str, num_epoches, epsilon))
81          # 캔버스 레이아웃 조정
82          plt.tight_layout()
83          plt.subplots_adjust(top=.9)
```

66번 줄에서 초기 자본금을 가로로 곧게 그어서 손익을 쉽게 파악할 수 있게 합니다. 67번 줄에서는 포트폴리오 가치가 초기 자본금보다 높은 부분은 빨간색, 69번 줄에서 포트폴리오 가치가 초기 자본금보다 낮은 부분을 파란색으로 배경을 칠합니다. 71번 줄에서 포트폴리오 가치를 검정색 실선으로 그립니다. 72번에서 77번 줄에서 학습을 수행한 위치를 표시합니다. 80번 줄에서 에포크와 탐험 비율을 적어 줍니다.

Matplotlib의 fill_between() 함수는 x 축 배열과 두 개의 y 축 배열을 입력으로 받습니다. 두 개의 y 축 배열의 같은 인덱스 위치의 값 사이에 색을 칠합니다. where 옵션으로 색을 칠할 조건을 추가할 수 있고 facecolor 옵션으로 칠할 색을 지정하고 alpha 옵션으로 투명도를 조정할 수 있습니다.

파이썬에서 문자열에 값을 넣어주는 방법으로 %를 사용할 수 있습니다. 문자열 내에서 %s는 문자열을, %d는 정수를, %f는 실수를 입력하는 자리입니다. 예를 들어서, ['Hello %s' % 'World!']는 ['Hello World!']가 됩니다. 정수와 실수의 경우 ['%d / %d = %.2f' % (10, 3, 10/3)]이 ['10 / 3 = 3.33']으로 됩니다.

다른 방법으로 format() 함수를 사용할 수 있습니다. 이 함수에서는 값을 넣을 자리를 {}로 표현합니다. 예를 들어서 ['Hello {}'.format('World!')]는 ['Hello World!']가 됩니다. 이 자리에 이름을 정할 수도 있습니다. 예를 들어서 ['Hello {tail}'.format(tail='World!')]는 ['Hello World!']가 됩니다.

Matplotlib tight_layout() 함수는 Figure의 크기에 알맞게 내부 차트들의 크기를 조정해 줍니다.

4.6.9 코드 조각 7: 차트 초기화 및 저장 함수 부분

예제 4.19는 Figure를 초기화하고 저장하는 함수를 보여줍니다.

예제 4.19 Visualizer 클래스의 clear() 함수 및 save() 함수

https://github.com/quantylab/rltrader/blob/master/visualizer.py

```
85          def clear(self, xlim):
86              for ax in self.axes[1:]:
87                  ax.cla()  # 그린 차트 지우기
88                  ax.relim()  # limit를 초기화
89                  ax.autoscale()  # 스케일 재설정
90              # y축 레이블 재설정
91              self.axes[1].set_ylabel('Agent')
92              self.axes[2].set_ylabel('PG')
```

```
93            self.axes[3].set_ylabel('PV')
94            for ax in self.axes:
95                ax.set_xlim(xlim)  # x축 limit 재설정
96                ax.get_xaxis().get_major_formatter().set_scientific(False) # 과학적 표기 비활성화
97                ax.get_yaxis().get_major_formatter().set_scientific(False)
98                ax.ticklabel_format(useOffset=False)  # x축 간격을 일정하게 설정
99
100       def save(self, path):
101           plt.savefig(path)
```

85번 줄에서는 학습 과정에서 변하지 않는 환경에 관한 차트를 제외하고 그 외 차트들을 초기화합니다. 입력으로 받는 xlim은 모든 차트의 x축 값 범위를 설정해 줄 튜플입니다.

87번 줄에서 이전에 그린 차트를 지우고 88번 줄에서 설정한 차트의 x축과 y축의 값 범위를 초기화합니다. 89번 줄에서 자동 크기 조정 기능을 활성화합니다.

91번에서 93번 줄까지에서는 차트들의 y축 레이블을 설정해 줍니다. 95번 줄에서는 x축 값의 범위를 설정해 줍니다. 96번과 97번 줄에서 x축과 y축의 값을 있는 그대로 보여주기 위해 과학적 표기 기능을 비활성화합니다. 98번 줄에서는 x축 간격을 일정하게 설정합니다. 그림 4.7은 x축 간격을 값에 맞게 조정한 결과와 x축 간격을 일정하게 설정한 결과의 차이를 보여줍니다.

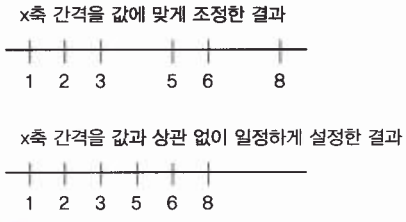


그림 4.7 x축 간격을 값에 맞게 조정한 결과와 일정하게 설정한 결과의 차이

이렇게 x축 간격을 일정하게 하는 이유는 일봉 차트의 경우 x축을 날짜로 지정하면 토요일과 일요일과 같이 휴장하는 날인 부분엔 차트가 비게 되어서 보기가 좋지 않기 때문입니다. 대부분의 서비스에서 일봉 차트는 휴장일 부분은 표시하지 않습니다.

101번 줄에서는 Figure를 그림파일로 저장합니다. 그림 4.8은 이렇게 생성된 가시화 결과를 보여줍니다.

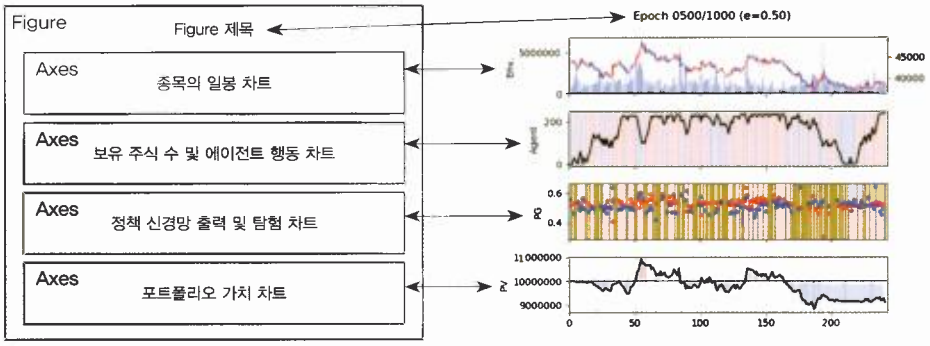


그림 4.8 가시화기 모듈에서 생성한 그림 파일

첫 번째 차트는 prepare() 함수에서 그려준 종목의 일봉 차트입니다. 이 차트는 전체 학습 과정에서 동일하기 때문에 한 번만 호출되는 prepare() 함수에서 그려줍니다. 그 외의 차트들은 에포크마다 다르기 때문에 plot() 함수에서 그려줍니다.

4.7 정책 학습기 모듈 개발

정책 학습기 모듈(policy_learner.py)은 정책 학습기 클래스(PolicyLearner)를 가지고 일련의 학습 데이터를 준비하고 정책 신경망을 학습합니다. 정책 신경망 클래스의 속성과 함수를 살펴보고 파이썬으로 구현한 소스코드를 상세히 확인합니다.

4.7.1 코드 조각 1: 정책 학습기 모듈의 의존성 임포트 부분

예제 4.20은 정책 학습기 모듈의 의존성 임포트 부분을 보여줍니다.

예제 4.20 policy_learner.py 모듈의 의존 패키지, 모듈, 클래스 임포트 부분

https://github.com/quantylab/rltrader/blob/master/policy_learner.py

```
1    import os
2    import locale
3    import logging
4    import time
5    import datetime
6    import numpy as np
7    import settings
8    from environment import Environment
9    from agent import Agent
10   from policy_network import PolicyNetwork
11   from visualizer import Visualizer
12
13
14   logger = logging.getLogger(__name__)
```

1번부터 5번 줄에서 파이썬 기본 모듈인 os, locale, logging, time, datetime을 임포트합니다. os는 폴더 생성, 파일 경로 준비 등을 위해 사용합니다. locale은 통화(currency) 문자열 포맷을 위해서 사용합니다. logging은 학습 과정 중에 정보를 기록하기 위해서 사용합니다. time과 datetime은 시간 값을 얻어오고 시간 문자열 포맷을 위해 사용합니다.

7번과 8번 줄에서 파이썬 라이브러리인 NumPy와 Pandas를 임포트합니다. 이들은 배열 자료구조를 조작하고, 저장 및 불러오기를 위해서 사용합니다.

10번 줄에서 14번 줄까지는 RLTrader의 모듈들을 임포트한 코드입니다. settings는 투자 설정, 로깅 설정 등을 하기 위한 모듈로서 여러 상수 값들을 포함합니다. 부록을 참조하면 자세한 사항을 확인할 수 있습니다. 이전 장 들에서 다룬 environment, agent, policy_network, visualizer 모듈을 모두 임포트합니다.

임포트에서 사용하는 키워드는 import, from, as입니다. import는 패키지, 모듈, 클래스, 함수를 임포트하기 위한 키워드입니다. from은 임포트할 모듈의 상위 패키지, 임포트할 클래스의 상위 모듈, 또는 임포트할 함수의 상위 모듈을 지정하기 위한 키워드입니다. as 키워드는 임포트한 패키지, 모듈, 클래스, 함수를 다른 이름으로 사용하기 위한 키워드입니다.

4.7.2 코드 조각 2: 정책 학습기 클래스의 생성자 부분

예제 4.21은 PolicyLearner 클래스의 생성자를 보여줍니다. 필수 인자는 stock_code, chart_data입니다. 학습 과정에서는 training_data도 넣어 줘야 합니다.

예제 4.21 PolicyLearner 클래스의 생성자

https://github.com/quantylab/rltrader/blob/master/policy_learner.py

```
17  class PolicyLearner:
18
19      def __init__(self, stock_code, chart_data, training_data=None,
20                   min_trading_unit=1, max_trading_unit=2,
21                   delayed_reward_threshold=.05, lr=0.01):
22          self.stock_code = stock_code  # 종목코드
23          self.chart_data = chart_data
24          self.environment = Environment(chart_data)  # 환경 객체
25          # 에이전트 객체
26          self.agent = Agent(self.environment,
27                             min_trading_unit=min_trading_unit,
28                             max_trading_unit=max_trading_unit,
29                             delayed_reward_threshold=delayed_reward_threshold)
30          self.training_data = training_data  # 학습 데이터
31          self.sample = None
32          self.training_data_idx = -1
33          # 정책 신경망; 입력 크기 = 학습 데이터의 크기 + 에이전트 상태 크기
34          self.num_features = self.training_data.shape[1] + self.agent.STATE_DIM
35          self.policy_network = PolicyNetwork(
36              input_dim=self.num_features, output_dim=self.agent.NUM_ACTIONS, lr=lr)
37          self.visualizer = Visualizer()  # 가시화 모듈
```

인자로 받은 chart_data는 24번 줄에서 Environment 클래스 객체를 생성할 때 넣어줍니다. Environment 클래스는 차트 데이터를 순서대로 읽으면서 주가, 거래량 등의 환경을 제공합니다.

학습 데이터는 학습에 사용할 특징(feature)들을 포함합니다. 이 특징들은 2장에서 상세하게 다루었습니다.

학습에 사용할 최종 특징 개수는 학습 데이터에 포함된 15개의 특징과 에이전트의 상태인 2개 특징을 더해서 17개입니다. 35번과 36번 줄에서 이 특징들을 학습하기 위한 PolicyNetwork 클래스 객체를 생성합니다.

NumPy 배열은 배열의 모양을 의미하는 shape 변수를 가집니다. N차원 배열이면 shape 변수는 N차원 튜플입니다. 예를 들어 1차원 배열의 shape는 1차원 튜플, 2차원 배열의 shape는 2차원 튜플입니다. [[1, 2, 3], [4, 5, 6]]의 shape는 (2, 3)입니다.

37번 줄에서는 학습 과정을 가시화하기 위해 Visualizer 클래스 객체를 생성합니다. 이 객체를 통해 에포크마다 투자 결과를 가시화합니다.

4.7.3 코드 조각 3: 에포크 초기화 함수 부분

예제 4.22는 에포크마다 호출하는 reset() 함수를 보여줍니다.

예제 4.22 PolicyLearner 클래스의 reset() 함수

https://github.com/quantylab/rltrader/blob/master/policy_learner.py

```
39        def reset(self):
40            self.sample = None
41            self.training_data_idx = -1
```

이 함수는 학습 데이터를 다시 처음부터 읽기 위해서 self.training_data_idx를 −1로 재설정합니다. 학습 데이터를 읽어가면서 이 값은 1씩 증가합니다. 읽어 온 데이터는 self.sample에 저장되는데 초기화 단계에서는 읽어온 학습 데이터가 없기 때문에 None으로 할당합니다.

4.7.4 코드 조각 4: 학습 함수 선언 부분

예제 4.23은 fit() 함수의 선언과 함수 인자 설명을 보여줍니다. fit() 함수는 PolicyLearner 클래스의 핵심 함수이며 그 길이 역시 상대적으로 깁니다. 그래서 여러 코드 조각으로 나눠서 설명을 하고자 합니다.

예제 4.23 PolicyLearner 클래스의 fit() 함수 (1)

https://github.com/quantylab/rltrader/blob/master/policy_learner.py

```
43      def fit(
44          self, num_epoches=1000, max_memory=60, balance=10000000,
45          discount_factor=0, start_epsilon=.5, learning=True):
46          logger.info("LR: {lr}, DF: {discount_factor}, "
47                      "TU: [{min_trading_unit}, {max_trading_unit}], "
48                      "DRT: {delayed_reward_threshold}".format(
49              lr=self.policy_network.lr,
50              discount_factor=discount_factor,
51              min_trading_unit=self.agent.min_trading_unit,
52              max_trading_unit=self.agent.max_trading_unit,
53              delayed_reward_threshold=self.agent.delayed_reward_threshold
54          ))
```

num_epoches는 수행할 반복 학습의 전체 횟수입니다. 반복 학습을 거치면서 정책 신경망이 점점 포트폴리오 가치를 높이는 방향으로 갱신되기 때문에 충분한 반복 횟수를 정해줘야 합니다. 그러나 num_epoches를 너무 크게 잡으면 학습에 소요되는 시간이 너무 길어지게 되므로 적절하게 정해줘야 합니다. 얼마나 많은 양의 데이터를 학습하는가에 따라 다르지만 여기서는 기본값을 1,000번으로 정합니다.

max_memory는 배치 학습 데이터를 만들기 위해 과거 데이터를 저장할 배열입니다. 배치 학습 데이터를 만드는 함수인 _get_batch()는 뒤에서 설명합니다.

balance는 에이전트의 초기 투자 자본금을 정해주기 위한 인자입니다. RLTrader에서는 신용 거래와 같이 보유 현금을 넘어서는 투자는 고려하지 않습니다. 보유 현금이 부족하면 정책 신경망 결과 매수가 좋아도 관망합니다.

지연 보상이 발생했을 때 그 이전 지연 보상이 발생한 시점과 현재 지연 보상이 발생한 시점 사이에서 수행한 행동들 전체에 현재의 지연 보상을 적용합니다. 이때 과거로 갈수록 현재 지연 보상을 적용할 판단 근거가 흐려지기 때문에 먼 과거의 행동일수록 할인 요인을 적용하여 지연 보상을 약하게 적용합니다. discount_factor로 할인 요인을 정합니다.

start_epsilon은 초기 탐험 비율을 의미합니다. 학습이 전혀 되어 있지 않은 초기에는 탐험 비율을 크게 해서 더 많은 탐험, 즉 무작위 투자를 수행하도록 해야 합니다. 탐험을 통해 특정 상황에서 좋은 행동과 그렇지 않은 행동을 결정하기 위한 경험을 쌓습니다.

learning은 학습 유무를 정하는 불(boolean) 값입니다. 불 값이란 참(true) 또는 거짓(false)을 가지는 이진 값입니다. 학습을 마치면 학습된 정책 신경망 모델이 만들어집니다. 이렇게 학습을 해서 정책 신경망 모델을 만들고자 한다면 learning을 True로, 학습된 모델을 가지고 투자 시뮬레이션만 하려 한다면 learning을 False로 줍니다.

4.7.5 코드 조각 5: 학습 함수 초반 부분

예제 4.24는 fit() 함수의 초반 부분을 보여줍니다.

예제 4.24 PolicyLearner 클래스의 fit() 함수 (2)

https://github.com/quantylab/rltrader/blob/master/policy_learner.py

```
56          # 가시화 준비
57          # 차트 데이터는 변하지 않으므로 미리 가시화
58          self.visualizer.prepare(self.environment.chart_data)
59
60          # 가시화 결과 저장할 폴더 준비
61          epoch_summary_dir = os.path.join(
62              settings.BASE_DIR, 'epoch_summary/%s/epoch_summary_%s' % (
63                  self.stock_code, settings.timestr))
64          if not os.path.isdir(epoch_summary_dir):
65              os.makedirs(epoch_summary_dir)
66
67          # 에이전트 초기 자본금 설정
68          self.agent.set_balance(balance)
69
70          # 학습에 대한 정보 초기화
71          max_portfolio_value = 0
72          epoch_win_cnt = 0
```

58번 줄에서 종목의 분봉 차트를 먼저 가시화합니다. 이 차트는 학습 과정에서 변하지 않기 때문에 한 번만 가시화합니다.

61번에서 65번 줄까지는 가시화 결과를 저장할 폴더를 준비하기 위한 코드입니다. 폴더의 이름에 날짜 및 시간을 넣어서 중복되는 일이 없도록 합니다.

settings.timestr은 폴더 이름에 포함할 날짜와 시간입니다. 여기서 사용하는 문자열 형식은 "%Y%m%d%H%M%S"입니다. 4자리 연도 2자리 월, 2자리 일, 2자리 시, 2자리 분, 2자리 초를 연결하여 201712011530과 같은 문자열을 구성합니다.

프로그래밍을 하다 보면 시간 값을 다룰 일이 종종 생깁니다. 주로 시간 값을 처리하는 일은 Unix Timestamp로 불리는 숫자 형식의 시간 값, 사람이 보기 쉬운 문자열(string) 형식의 시간 값, 시간 값을 다루는 클래스 객체인 datetime 사이의 변환 작업입니다.

피이썬에서 시간 값과 관련된 모듈로 time과 datetime이 있습니다. 유닉스의 timestamp는 주로 time 모듈로 다루고, String 형식과 datetime 객체는 datetime 모듈로 다룹니다. 여기서 주의할 점은 datetime 모듈 안에 같은 이름으로 datetime 클래스가 속해 있어서 혼동하지 말아야 합니다.

- datetime에서 timestamp로 변환

```
dt = datetime.datetime(2010, 2, 25, 23, 23)
ts = time.mktime(dt.timetuple())
```

- timestamp에서 datetime으로 변환

```
ts = 1284101485
dt = datetime.datetime.fromtimestamp(ts)
```

- datetime에서 string으로 변환

```
dt = datetime.datetime(2010, 2, 25, 23, 23)
str_time = dt.strftime('%Y-%m-%d %H:%M:%S')
```

- strftime을 string formatting with time의 의미로 생각하면 이해하기 편합니다.

- string에서 datetime으로 변환

```
str_time = '2010-02-25 23:23:00'
dt = datetime.datetime.strptime(str_time, "%Y-%m-%d %H:%M:%S")
```

- strptime을 string parsing to time(즉, 문자열을 시간으로 구문분석)이라는 의미로 생각하면 이해하기 편합니다.

61번 줄에서는 폴더의 경로를 변수로 저장합니다. 경로는 종목코드, 날짜, 시간으로 구성하여 중복이 없도록 합니다.

파일 경로는 윈도우 운영체제에서는 \로 구분되고 리눅스 계열의 운영체제에서는 /로 구분됩니다. 이렇게 운영체제마다 경로 구성법이 다르기 때문에 파이썬에서는 os.path.join() 함수로 경로를 구성하는 것이 좋습니다. os.path.join('parent', 'child')는 윈도우에서는 'parent\child'가 되고 리눅스에서는 'parent/child'가 됩니다.

64번과 65번 줄에서는 해당 경로가 없으면 이 경로를 구성하는 폴더들을 생성합니다.

os.path 모듈에는 경로와 관련된 다양한 함수들이 있습니다. os.path.isdir(path) 함수는 path가 존재하고 폴더인지 확인합니다. os.path.isfile(path)는 path가 존재하는 파일인지 확인합니다.

os.mkdirs(path) 함수는 path에 포함된 폴더들이 없을 경우에 생성해 줍니다. path가 '/a/b/c'이고 현재 '/a'라는 경로만 존재한다면 '/a' 폴더 하위에 'b' 폴더를 생성하고 'b' 폴더 하위에 'c' 폴더를 생성하여 최종적으로 '/a/b/c' 경로가 존재하도록 만듭니다.

68번 줄에서는 초기 자본금을 설정합니다. 71번 줄에서는 학습 과정에서 달성한 최대 포트폴리오 가치를 저장하는 변수를 초기화하고 72번 줄에서는 수익이 발생한 에포크 수를 저장하는 변수를 초기화합니다.

4.7.6 코드 조각 6: 학습 함수의 로컬 변수 초기화 부분

예제 4.25는 num_epoches만큼 반복 학습을 시작하는 부분을 보여줍니다. 79번 줄부터 193번 줄까지의 코드가 모두 이 반복문 내에 포함되어 있음을 유의해야 합니다.

파이썬에서 코드 블록은 들여쓰기(indent)로 구분합니다. 특히 class, def, if, elif, else, for, while, try, except, final, with 등을 사용할 때 들여쓰기를 유의해야 합니다.

예제 4.25 PolicyLearner 클래스의 fit() 함수 (3)

https://github.com/quantylab/rltrader/blob/master/policy_learner.py

```
74          # 학습 반복
75          for epoch in range(num_epoches):
76              # 에포크 관련 정보 초기화
77              loss = 0.
78              itr_cnt = 0
79              win_cnt = 0
80              exploration_cnt = 0
81              batch_size = 0
82              pos_learning_cnt = 0
83              neg_learning_cnt = 0
84
85              # 메모리 초기화
86              memory_sample = []
87              memory_action = []
88              memory_reward = []
89              memory_prob = []
90              memory_pv = []
91              memory_num_stocks = []
92              memory_exp_idx = []
93              memory_learning_idx = []
```

75번 줄에서 반복 학습을 시작합니다. 77번 줄에서 83번 줄까지는 에포크 관련 정보를 초기화합니다. loss는 정책 신경망의 결과가 학습 데이터와 얼마나 차이가 있는지를 저장하는 변수입니다. loss 값이 학습이 진행되면서 줄어드는 것이 좋습니다.

itr_cnt 변수는 수행한 에포크 수를 저장합니다. win_cnt 변수에는 수행한 에포크 중에서 수익이 발생한 에포크 수를 저장합니다. 즉, 포트폴리오 가치가 초기 자본금보다 높아진 에포크 수입니다. exploration_cnt 변수는 무작위 투자를 수행한 횟수를 저장합니다. epsilon이 0.1이고 100번의 투자 결정이 있다고 한다면 약 10번의 무작위 투자를 할 것입니다.

pos_learning_cnt는 수익이 발생하여 긍적적 지연 보상을 준 수이고 neg_learning_cnt는 손실이 발생하여 부정적 지연 보상을 준 수입니다.

85번에서 93번 줄까지는 메모리를 초기화하는 부분입니다. 메모리 리스트에 저장하는 데이터는 샘플, 행동, 즉시보상, 정책 신경망의 출력, 포트폴리오 가치, 보유 주식 수, 탐험 위치, 학습 위치입니다.

4.7.7 코드 조각 7: 학습 함수의 연관 객체 초기화 및 탐험 비율 설정 부분

예제 4.26에서 fit() 함수를 이어서 살펴보겠습니다. 여기서는 무작위 투자 비율인 epsilon 값을 정하고 투자 과정 데이터를 저장할 메모리를 초기화하고 가시화기에서 분봉 차트 외의 차트들을 초기화합니다.

예제 4.26 PolicyLearner 클래스의 fit() 함수 (4)

https://github.com/quantylab/rltrader/blob/master/policy_learner.py

```
95              # 환경, 에이전트, 정책 신경망 초기화
96              self.environment.reset()
97              self.agent.reset()
98              self.policy_network.reset()
99              self.reset()
100
101             # 가시화기 초기화
102             self.visualizer.clear([0, len(self.chart_data)])
103
104             # 학습을 진행할수록 탐험 비율 감소
105             if learning:
106                 epsilon = start_epsilon * (1. - float(epoch) / (num_epoches - 1))
107             else:
108                 epsilon = 0
```

95번 줄에서 99번 줄까지 환경 객체, 에이전트 객체, 정책 신경망 객체, 정책 학습기 객체를 초기화합니다. 102번 줄에서는 가시화기의 clear() 함수를 호출하여 2, 3, 4번째 차트를 초기화합니다. x축 데이터 범위를 파라미터로 넣어 줍니다.

106번 줄에서 epsilon 값을 정하는데 이때 fit() 함수의 인자로 넘어오는 최초 무작위 투자 비율인 start_epsilon 값에 현재 epoch 수에 학습 진행률을 곱해서 정합니다. 예를 들어, start_epsilon이 0.3이면 첫 번째 에포크에서는 30%의 확률로 무작위 투자를 진행합니다. 수행할 에포크 수가 100이라고 했을 때 50번째 에포크에서는 $0.3 \times \left(1 - \frac{49}{99}\right) \approx 0.51$이 됩니다.

4.7.8 코드 조각 8: 학습 함수의 에포크 수행 while 문 초반부

예제 4.27에서 하나의 에포크를 수행하는 while 문의 초반부를 보여줍니다.

예제 4.27 PolicyLearner 클래스의 fit() 함수 (5)

https://github.com/quantylab/rltrader/blob/master/policy_learner.py

```
110             while True:
111                 # 샘플 생성
112                 next_sample = self._build_sample()
113                 if next_sample is None:
114                     break
115
116                 # 정책 신경망 또는 탐험에 의한 행동 결정
117                 action, confidence, exploration = self.agent.decide_action(
118                     self.policy_network, self.sample, epsilon)
119
120                 # 결정한 행동을 수행하고 즉시 보상과 지연 보상 획득
121                 immediate_reward, delayed_reward = self.agent.act(action, confidence)
```

행동을 결정하기 위한 데이터인 샘플을 112번 줄에서 준비합니다. next_sample이 None이라면 마지막까지 데이터를 다 읽은 것이므로 110번 줄의 반복문을 종료합니다.

117번 줄에서 투자 행동을 결정합니다. 여기서는 매수와 매도 중 하나를 결정합니다. 이 행동 결정은 무작위 투자 비율인 epsilon 값의 확률로 무작위로 하거나 그렇지 않은 경우 정책 신경망의 출력을 통해 결정됩니다. 정책 신경망의 출력은 매수를 했을 때와 매도를 했을 때의 포트폴리오 가치를 높일 확률을 의미합니다. 즉 매수에 대한 정책 신경망 출력이 매도에 대한 출력보다 높으면 매수를, 그 반대의 경우 매도를 선택합니다.

decide_action() 함수가 반환하는 값은 세 가지입니다. 결정한 행동인 action, 결정에 대한 확신도인 confidence, 무작위 투자 유무인 exploration입니다.

121번 줄에서 결정한 행동을 수행하도록 에이전트의 act() 함수를 호출합니다. act() 함수는 행동을 수행하고 즉시보상과 지연 보상을 반환합니다.

for와 while 반복문에서 break로 반복문을 끝낼 수 있고 continue로 continue 이후의 코드를 건너뛸 수 있습니다.

4.7.9 코드 조각 9: 학습 함수의 행동과 그 결과를 저장하는 부분

예제 4.28은 행동과 행동에 대한 결과를 메모리에 저장하는 소스코드를 보여줍니다.

예제 4.28 PolicyLearner 클래스의 fit() 함수 (5)

https://github.com/quantylab/rltrader/blob/master/policy_learner.py

```
123                 # 행동 및 행동에 대한 결과를 기억
124                 memory_sample.append(next_sample)
125                 memory_action.append(action)
126                 memory_reward.append(immediate_reward)
127                 memory_pv.append(self.agent.portfolio_value)
128                 memory_num_stocks.append(self.agent.num_stocks)
129                 memory = [(
130                     memory_sample[i],
131                     memory_action[i],
132                     memory_reward[i])
133                     for i in list(range(len(memory_action)))[-max_memory:]
134                 ]
135                 if exploration:
136                     memory_exp_idx.append(itr_cnt)
137                     memory_prob.append([np.nan] * Agent.NUM_ACTIONS)
138                 else:
139                     memory_prob.append(self.policy_network.prob)
```

memory는 학습 데이터의 샘플, 에이전트 행동, 즉시보상, 포트폴리오 가치, 보유 주식 수를 저장하는 배열입니다.

124번에서 128번 줄까지는 각 데이터를 메모리에 추가하는 부분입니다. 129번 줄에서 이들을 모아서 하나의 2차원 배열로 만들어 줍니다.

무작위 투자로 행동을 결정한 경우에 136번 줄에서 현재의 인덱스를 memory_exp_idx에 저장합니다. memory_prob는 정책 신경망의 출력을 그대로 저장하는 배열입니다. 무작위 투자에서는 정책 신경망의 출력이 없기 때문에 NumPy의 Not A Number(nan) 값으로 넣어 줍니다.

파이썬에서는 리스트에 곱하기를 하면 똑같은 리스트를 뒤에 붙여줍니다. 예를 들어서 [1, 2, 3] * 3은 [1, 2, 3, 1, 2, 3, 1, 2, 3]이 됩니다.

무작위 투자가 아닌 경우 139번 줄에서 정책 신경망의 출력을 그대로 memory_prob에 저장합니다.

메모리 변수들은 두 가지 목적으로 (1) 학습에서 배치 학습 데이터로 사용하고 (2) 가시화기에서 차트를 그릴 때 사용합니다.

4.7.10 코드 조각 10: 학습 함수의 반복 정보 갱신 부분

예제 4.29에서는 배치 크기 batch_size, 반복 카운팅 횟수 itr_cnt, 무작위 투자 횟수 exploration_cnt, 수익이 발생한 횟수 win_cnt를 증가시킵니다.

예제 4.29 PolicyLearner 클래스의 fit() 함수 (6)

https://github.com/quantylab/rltrader/blob/master/policy_learner.py

```
141                 # 반복에 대한 정보 갱신
142                 batch_size += 1
143                 itr_cnt += 1
144                 exploration_cnt += 1 if exploration else 0
145                 win_cnt += 1 if delayed_reward > 0 else 0
```

exploration_cnt의 경우 탐험을 한 경우에만 1을 증가시키고 그렇지 않으면 0을 더해서 변화가 없게 합니다. win_cnt도 지연 보상이 0보다 큰 경우에만 1을 증가시키고 그렇지 않으면 0을 더해서 변화가 없게 합니다.

파이썬에서는 if else 문을 한 줄에서 쓸 수 있습니다. 예를 들어 x가 0보다 크거나 같을 경우에 1을 증가시키고 0보다 작을 경우에 1을 감소시키는 코드를 작성해 보겠습니다. 일반적인 방법으론 다음과 같이 작성할 수 있습니다.

```
if x >= 0:
    x += 1
else:
    x -= 1
```

이 코드를 한줄로 다음과 같이 작성할 수 있습니다.

```
x += 1 if x >= 0 else -1
```

4.7.11 코드 조각 11: 학습 함수의 정책 신경망 학습 부분

예제 4.30은 지연 보상이 발생한 경우에 학습을 수행하는 부분을 보여줍니다.

예제 4.30 PolicyLearner 클래스의 fit() 함수 (7)

https://github.com/quantylab/rltrader/blob/master/policy_learner.py

```
147                 # 학습 모드이고 지연 보상이 존재할 경우 정책 신경망 갱신
148                 if delayed_reward == 0 and batch_size >= max_memory:
149                     delayed_reward = immediate_reward
150                 if learning and delayed_reward != 0:
151                     # 배치 학습 데이터 크기
152                     batch_size = min(batch_size, max_memory)
153                     # 배치 학습 데이터 생성
154                     x, y = self._get_batch(
155                         memory, batch_size, discount_factor, delayed_reward)
156                     if len(x) > 0:
157                         if delayed_reward > 0:
158                             pos_learning_cnt += 1
159                         else:
```

```
160                     neg_learning_cnt += 1
161                 # 정책 신경망 갱신
162                 loss += self.policy_network.train_on_batch(x, y)
163                 memory_learning_idx.append([itr_cnt, delayed_reward])
164             batch_size = 0
```

148번 줄에서는 학습 없이 메모리가 최대 크기만큼 다 찼을 경우 즉시 보상으로 지연 보상을 대체하여 학습을 진행하도록 합니다.

학습 모드에서 지연 보상이 발생했으면 151번 줄에서 164번 줄까지를 수행합니다. 152번 줄에서는 배치 학습에 사용할 배치 데이터 크기를 결정합니다. 배치 데이터 크기는 memory 변수의 크기인 max_memory보다는 작아야 합니다.

154번 줄에서 _get_batch() 함수를 통해 배치 데이터를 준비합니다. 먼저 학습 데이터 샘플, 에이전트 행동, 즉시 보상을 담고 있는 memory, 생성할 배치 데이터 크기, 할인 요인(discount factor), 지연 보상을 인자로 넣어줍니다. 할인 요인 이론에 대해서는 2.1.3장에서 다루었습니다.

로그 기록을 위해 158번과 160번 줄에서 긍정(positive) 학습과 부정(negative) 학습 횟수를 셉니다.

162번 줄에서 준비한 배치 데이터로 학습을 진행합니다. _get_batch() 함수는 뒤에서 설명합니다. 학습은 정책 신경망 객체의 train_on_batch() 함수로 수행합니다. 163번 줄에서 학습이 진행된 인덱스를 저장합니다.

학습을 수행했으니 164번 줄에서 배치 데이터 크기를 0으로 초기화합니다.

4.7.12 코드 조각 12: 에포크 결과 가시화 부분

예제 4.31은 하나의 에포크가 완료되어 에포크 관련 정보를 가시화하는 부분입니다.

예제 4.31 PolicyLearner 클래스의 fit() 함수 (8)

https://github.com/quantylab/rltrader/blob/master/policy_learner.py

```
166                 # 에포크 관련 정보 가시화
167                 num_epoches_digit = len(str(num_epoches))
168                 epoch_str = str(epoch + 1).rjust(num_epoches_digit, '0')
169
170                 self.visualizer.plot(
171                     epoch_str=epoch_str, num_epoches=num_epoches, epsilon=epsilon,
172                     action_list=Agent.ACTIONS, actions=memory_action,
173                     num_stocks=memory_num_stocks, outvals=memory_prob,
174                     exps=memory_exp_idx, learning=memory_learning_idx,
175                     initial_balance=self.agent.initial_balance, pvs=memory_pv
176                 )
177                 self.visualizer.save(os.path.join(
178                     epoch_summary_dir, 'epoch_summary_%s_%s.png' % (
179                         settings.timestr, epoch_str)))
```

167번 줄에서는 총 에포크 수의 문자열 길이를 확인합니다. 총 에포크 수가 1,000이면 길이는 4가 됩니다. 168번 줄에서는 현재 에포크 수를 4자리 문자열로 만들어 줍니다. 첫 번째 에포크의 경우 epoch는 0이기 때문에 1을 더해 주고 앞에 '0'을 채워서 '0001'로 만들어 줍니다.

rjust() 함수는 문자열을 자리수에 맞게 오른쪽으로 정렬해 주는 함수입니다. 예를 들어 "1".rjust(5)를 하면 '   1'이 됩니다. 앞에 빈칸 4자리를 채워주고 1을 붙여서 5자리 문자열이 되는 것입니다. 두 번째 인자로 빈칸 대신 채워줄 문자를 정해줄 수도 있습니다. "1".rjust(5, '0')는 '00001'이 됩니다.

비슷한 함수로 ljust() 함수가 있습니다. 이 함수는 기존 문자열 앞에 빈칸 또는 특정 문자를 채워줍니다.

이렇게 기존 문자 앞 또는 뒤에 어떠한 문자를 채워주는 것을 패딩(padding)이라고 합니다.

170번 줄에서 가시화기의 plot() 함수를 호출하여 에포크 수행 결과를 가시화합니다. 4.6절에서 plot() 함수에 대해서 상세히 다루었습니다. 177번 줄에서는 가시화한 에포크 수행 결과를 파일로 저장합니다.

4.7.13 코드 조각 13: 에포크 결과 로그 기록 부분

예제 4.32는 에포크 수행 결과를 로그로 기록하는 부분입니다.

예제 4.32 PolicyLearner 클래스의 fit() 함수 (9)

https://github.com/quantylab/rltrader/blob/master/policy_learner.py

```
181             # 에포크 관련 정보 로그 기록
182             if pos_learning_cnt + neg_learning_cnt > 0:
183                 loss /= pos_learning_cnt + neg_learning_cnt
184             logger.info("[Epoch %s/%s]\tEpsilon:%.4f\t#Expl.:%d/%d\t"
185                         "#Buy:%d\t#Sell:%d\t#Hold:%d\t"
186                         "#Stocks:%d\tPV:%s\t"
187                         "POS:%s\tNEG:%s\tLoss:%10.6f" % (
188                             epoch_str, num_epoches, epsilon, exploration_cnt, itr_cnt,
189                             self.agent.num_buy, self.agent.num_sell, self.agent.num_hold,
190                             self.agent.num_stocks,
191                             locale.currency(self.agent.portfolio_value, grouping=True),
192                             pos_learning_cnt, neg_learning_cnt, loss))
```

여기서는 총 에포크 중에서 몇 번째 에포크를 수행했는지, 탐험률, 탐험 횟수, 매수 횟수, 매도 횟수, 관망 횟수, 보유 주식 수, 포트폴리오 가치, 긍정적 학습 횟수, 부정적 학습 횟수, 학습 손실을 로그로 남깁니다.

4.7.14 코드 조각 14: 학습 통계 정보 갱신 부분

예제 4.33은 하나의 에포크 수행이 완료되었기 때문에 전체 학습에 대한 통계 정보를 갱신합니다.

예제 4.33 PolicyLearner 클래스의 fit() 함수 (10)

https://github.com/quantylab/rltrader/blob/master/policy_learner.py

```
194             # 학습 관련 정보 갱신
195             max_portfolio_value = max(
196                 max_portfolio_value, self.agent.portfolio_value)
197             if self.agent.portfolio_value > self.agent.initial_balance:
198                 epoch_win_cnt += 1
```

관리하고 있는 학습 통계 정보는 달성한 최대 포트폴리오 가치 max_portfolio_value와 수익이 발생한 에포크의 수 epoch_win_cnt입니다. 198번 줄까지가 75번 줄의 학습 반복 for 문의 코드 블록입니다. 이는 학습이 끝나지 않았다면 198번 줄 다음에 75번 줄로 돌아간다는 뜻입니다.

4.7.15 코드 조각 15: 최종 학습 결과 통계 정보 로그 기록 부분

예제 4.34는 fit() 함수의 마지막 부분을 보여줍니다. 75번 줄의 학습 반복 for 문을 벗어나고 최종 로그를 기록합니다.

예제 4.34 PolicyLearner 클래스의 fit() 함수 (11)

https://github.com/quantylab/rltrader/blob/master/policy_learner.py

```
200             # 학습 관련 정보 로그 기록
201             logger.info("Max PV: %s, \t # Win: %d" % (
202                 locale.currency(max_portfolio_value, grouping=True), epoch_win_cnt))
```

201번 줄에서 달성한 최대 포트폴리오 가치 max_portfolio_value와 수익이 발생한 에포크의 수 epoch_win_cnt를 로깅합니다.

이 외의 다양한 통계 정보를 학습 과정에서 관리하고 이 부분에서 출력해 줄 수 있습니다.

4.7.16 코드 조각 16: 미니 배치 데이터 생성 함수 부분

예제 4.35는 미니 배치 데이터를 생성하는 _get_batch()를 보여줍니다.

예제 4.35 PolicyLearner 클래스의 _get_batch() 함수

https://github.com/quantylab/rltrader/blob/master/policy_learner.py

```
204         def _get_batch(self, memory, batch_size, discount_factor, delayed_reward):
205             x = np.zeros((batch_size, 1, self.num_features))
206             y = np.full((batch_size, self.agent.NUM_ACTIONS), 0.5)
207
```

```
208             for i, (sample, action, reward) in enumerate(
209                     reversed(memory[-batch_size:])):
210                 x[i] = np.array(sample).reshape((-1, 1, self.num_features))
211                 y[i, action] = (delayed_reward + 1) / 2
212                 if discount_factor > 0:
213                     y[i, action] *= discount_factor ** i
214             return x, y
```

205번 줄의 x는 일련의 학습 데이터 및 에이전트 상태이고, 206번 줄의 y는 일련의 지연 보상입니다. x 배열의 형태는 배치 데이터 크기, 학습 데이터 특징 크기로 2차원으로 구성됩니다. y 배열의 형태는 배치 데이터 크기, 정책 신경망이 결정하는 에이전트 행동의 수로 2차원으로 구성됩니다. y는 0.5로 일괄적으로 채워놓습니다.

NumPy의 full() 함수는 첫 번째 인자로 배열의 형태인 shape를 받아서 두 번째 인자로 입력된 값으로 채워진 NumPy 배열을 반환합니다. 예를 들어, full(3, 1)은 [1, 1, 1]을, zeros((2, 2), 0.5)는 [[0.5, 0.5], [0.5, 0.5]]을 반환합니다. 주의할 점은 다차원 배열의 경우 그 형태를 튜플로 넘겨줘야 합니다.

배치 데이터 크기는 지연 보상이 발생될 때 결정되기 때문에 매번 다르고 학습 데이터 특징의 크기와 에이전트 행동 수는 17과 2로 고정되어 있습니다.

210번 줄에서 특징벡터를 지정하고 211번 줄에서 지연 보상으로 정답(레이블)을 설정하여 학습 데이터를 구성합니다. 지연 보상이 1인 경우 1로, 지연 보상이 −1인 경우 0으로 레이블을 지정합니다. 할인 요인이 있을 경우 213번 줄에서 적용합니다.

4.7.17 코드 조각 17: 학습 데이터 샘플 생성 부분

예제 4.36은 학습 데이터를 구성하는 샘플 하나를 생성하는 함수인 _build_sample() 함수를 보여줍니다.

예제 4.36 PolicyLearner 클래스의 _build_sample() 함수

https://github.com/quantylab/rltrader/blob/master/policy_learner.py

```
216         def _build_sample(self):
217             self.environment.observe()
218             if len(self.training_data) > self.training_data_idx + 1:
219                 self.training_data_idx += 1
220                 self.sample = self.training_data.iloc[self.training_data_idx].tolist()
221                 self.sample.extend(self.agent.get_states())
222                 return self.sample
223             return None
```

217번 줄에서 환경 객체의 observe() 함수를 호출하여 차트 데이터의 현재 인덱스에서 다음 인덱스 데이터를 읽도록 합니다. 218번 줄에서는 학습 데이터의 다음 인덱스가 존재하는지 확인합니다.

학습 데이터에 다음 인덱스 데이터가 존재하면 training_data_idx 변수를 1 증가시키고 training_data 배열에서 training_data_idx 인덱스의 데이터를 받아와서 sample로 저장합니다. 현재까지는 sample 데이터는 15개의 값으로 구성되어 있습니다. 221번 줄에서 sample에 에이전트 상태를 추가하여 17개의 값으로 구성하도록 합니다. 그리고 이 sample을 반환합니다.

4.7.18 코드 조각 18: 투자 시뮬레이션을 하는 trade() 함수 부분

예제 4.37은 학습된 정책 신경망 모델로 주식투자 시뮬레이션을 진행하기 위한 trade() 함수를 보여줍니다.

예제 4.37 PolicyLearner 클래스의 trade() 함수

https://github.com/quantylab/rltrader/blob/master/policy_learner.py

```
225         def trade(self, model_path=None, balance=2000000):
226             if model_path is None:
227                 return
228             self.policy_network.load_model(model_path=model_path)
229             self.fit(balance=balance, num_epoches=1, learning=False)
```

먼저 학습된 정책 신경망 모델을 정책 신경망 객체의 load_model로 적용시켜 줍니다.

이 함수는 학습된 정책 신경망으로 투자 시뮬레이션을 하는 것이므로 반복 투자를 할 필요가 없기 때문에 총 에포크 수 num_epoches를 1로 주고 learning 인자에 False를 넘겨줍니다. 이렇게 하면 학습을 진행하지 않고 정책 신경망에만 의존하여 투자 시뮬레이션을 진행합니다. 물론 무작위 투자는 수행하지 않습니다.

4.8 이번 장의 요점

- RLTrader의 주요 개발 환경은 아나콘다, 텐서플로, 케라스입니다.
- RLTrader의 구성 모듈은 강화학습의 참여요소인 환경, 에이전트의 역할을 고려하여 설계하였습니다.
- RLTrader는 정책 학습기, 환경, 에이전트, 정책 신경망, 가시화기로 구성됩니다.
- RLTrader의 정책 신경망에서는 LSTM 인공 신경망을 사용합니다.
- 심층 신경망, 합성곱 신경망 등 다른 신경망을 RLTrader의 정책 신경망으로 적용할 수 있습니다.

05장

데이터 준비: 주식 데이터 획득

강화학습 주식투자 시뮬레이션을 해보려면 먼저 분석할 주식 데이터를 확보해야 합니다. 주식 데이터를 획득하는 방법은 다음과 같이 다양합니다.

- 방법 1. 증권사 HTS 사용
- 방법 2. 증권사 API 사용
- 방법 3. 포털 사이트 사용

방법 1은 증권사 HTS를 설치하고 증권 계좌만 있으면 쉽게 주식 데이터를 얻을 수 있는 방법입니다. 그러나 증권 계좌를 개설해야 하는 번거로움이 있고, 프로그램을 사용해 데이터를 획득하기가 쉽지 않습니다.

방법 2는 증권사 API를 사용하여 프로그램적으로 데이터를 획득할 수 있습니다. 그러나 증권 계좌를 개설해야 하고 증권사 API를 호출할 프로그램을 작성해야 합니다.

방법 3은 구글, 야후, 네이버, 다음 등의 포털 사이트에서 주식 데이터를 획득하는 방법입니다. 이 방법은 증권사 계좌를 개설할 필요가 없으며 프로그램적으로 데이터를 획득할 수 있고, 증권사 API 사용에 비해 개발 환경에 큰 제약이 없습니다. 그러나 포털 사이트에서 데이터를 가져올 프로그램을 작성해야 합니다. 대표적인 주식 포털 사이트인 구글과 야후의 경우 이미 오픈소스로 구현되어 있는 데이터 획득 코드들이 많습니다. 그러나 포털 사이트가 데이터 제공 구조를 종종 바꾸고 API를 지속적으로 제공하지 않기 때문에 이 방법은 안정적이지는 않습니다.

직접 프로그램을 개발한다면 주식 데이터를 담고 있는 웹페이지를 크롤링(crawling, 긁어오기)하는 모듈과 가져온 웹페이지를 파싱(parsing, 구문분석)하는 모듈을 개발해야 하기 때문에 비교적 많은 노력이 들어갑니다. 그리고 분석하는 웹페이지의 형식이 바뀌면 파싱 로직도 다시 맞춰줘야 하기 때문에 비효율적입니다.

RLTrader가 필요로 하는 주식 데이터는 종목의 일봉 차트 및 일별 거래량입니다. 여기 제시한 각 방법으로 주식 데이터를 획득해 보겠습니다.

5.1 방법 1. 증권사 HTS 사용

증권사 HTS를 사용하면 주식 데이터를 쉽게 확인하고 받아올 수 있습니다. 이 방법은 프로그래밍이 익숙하지 않거나 번거롭다고 생각되면 고려할 만합니다.

5.1.1 증권사 HTS 다운로드

대부분의 증권사 HTS가 제공하는 기능이 비슷하므로 여기서는 키움증권의 영웅문4를 대상으로 설명합니다. 저자와 키움증권과는 아무런 이해관계가 없음을 밝힙니다.

먼저 영웅문4를 설치하기 위해 웹 브라우저에서 키움증권 홈페이지(https://www.kiwoom.com/)에 접속합니다. 키움증권 홈페이지 우측에서 그림 5.1과 같은 화면을 볼 수 있습니다.

그림 5.1 키움증권 HTS 다운로드 화면

여기서 '영웅문4 다운로드' 버튼을 누르면 영웅문4 설치 파일(KiwoomHero4Setup.exe)이 받아집니다. 내려받은 설치 파일을 실행해서 영웅문4를 설치합니다. 설치 경로는 기본적으로 'C:\KiwoomHero4'로 정해집니다.

5.1.2 증권 계좌 개설

키움증권 HTS를 사용하기 위해선 키움증권 계좌를 개설해야 합니다. 키움증권 홈페이지 메인화면의 우측에 그림 5.2와 같이 '키움 계좌 개설' 버튼이 있습니다.

그림 5.2 키움 계좌 개설 화면

키움 계좌 개설 화면에 들어가면 그림 5.3과 같이 계좌 개설 절차에 대한 설명을 볼 수 있습니다.

그림 5.3 키움 계좌 개설 절차

계좌 개설은 스마트폰(안드로이드 또는 아이폰) 앱이나 키움증권 홈페이지에서 진행할 수 있으나 'STEP 4 본인인증'을 거치려면 스마트폰에 설치된 키움 계좌 개설 앱이 꼭 필요합니다.

5.1.3 종목 차트 데이터 확인

계좌를 개설했다면 컴퓨터에 설치한 키움증권 HTS 영웅문4를 실행하여 로그인합니다. 영웅문4 창의 좌측 상단에 그림 5.4와 같은 툴바가 있습니다.

그림 5.4 영웅문4의 상단 툴바 검색 창

표시한 부분에 0600을 입력하면 그림 5.6과 같은 키움종합차트창이 뜹니다. 키움종합차트창은 자주 쓰이는 기능이라 그림 5.5와 같이 상단 툴바의 메뉴로도 있습니다.

그림 5.5 영웅문4의 상단 툴바 메뉴

이렇게 키움종합차트 창을 띄우고 시가, 고가, 저가, 종가, 거래량, 5일 평균 거래량, 20일 평균 거래량, 60일 평균 거래량, 120일 평균 거래량을 표시해 줍니다. 여기서 차트 데이터를 구성하는 시가, 고가, 저가, 종가, 거래량이 필요합니다.

거래량이 표시되지 않는다면 추가로 설정해 줘야 합니다. 좌측 사이드바에서서 '거래량지표'를 선택하고 '거래량' 항목을 클릭하면 그림 5.6과 같이 거래량 차트가 표시됩니다.

그림 5.6 키움종합차트창의 일봉 캔들차트 및 거래량차트

이렇게 설정을 마치고 나면 이제 이 데이터들을 복사할 수 있습니다. 그런데 그 전에 주의할 점은 이 차트는 최근 600 거래일의 데이터만 읽어온 상태라는 점입니다. 더 이전의 데이터를 읽어 오기 위해서 그림 5.6의 우측 상단에 빨간색 상자로 표시한 연속조회 버튼을 누릅니다. 이 버튼은 더 읽어올 데이터가 있으면 활성화되어 있고 모든 데이터를 다 읽어왔으면 비활성화됩니다.

이 상태에서 차트에 마우스 오른쪽 버튼을 클릭하면 그림 5.7과 같이 팝업 메뉴가 뜨고 '텍스트창' 항목이 있습니다.

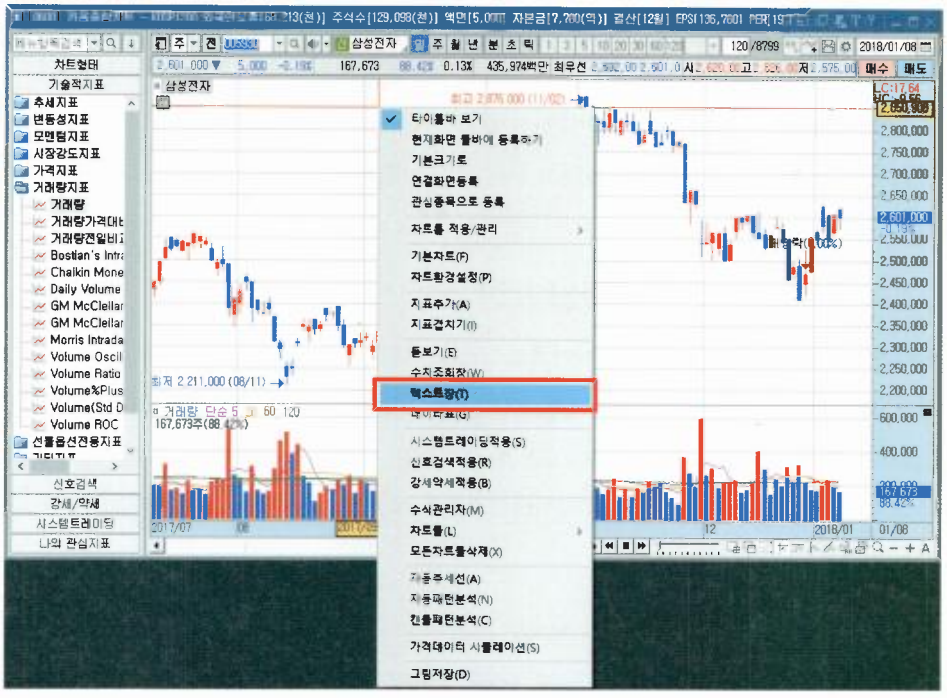


그림 5.7 종합차트의 연속조회 및 팝업 메뉴

'텍스트창'을 클릭하면 차트 데이터를 복사할 수 있는 메뉴가 그림 5.8과 같이 표시됩니다.

일자	시가	고가	저가	종가	거래량	단순 5
2018/01/08	2,620,000	2,626,000	2,575,000	2,601,000	167,673	192,192,00
2018/01/05	2,565,000	2,606,000	2,560,000	2,606,000	189,623	194,599,20
2018/01/04	2,606,000	2,609,000	2,532,000	2,554,000	233,909	199,649,00
2018/01/03	2,627,000	2,628,000	2,571,000	2,581,000	200,270	217,026,60
2018/01/02	2,569,000	2,570,000	2,539,000	2,551,000	169,485	221,771,20
2017/12/28	2,478,000	2,548,000	2,475,000	2,548,000	179,709	250,371,40
2017/12/27	2,448,000	2,478,000	2,423,000	2,468,000	214,872	254,751,80
2017/12/26	2,488,000	2,505,000	2,410,000	2,410,000	320,797	259,691,80
2017/12/22	2,470,000	2,498,000	2,462,000	2,485,000	223,993	224,933,40
2017/12/21	2,550,000	2,553,000	2,455,000	2,457,000	312,486	239,849,00
2017/12/20	2,575,000	2,588,000	2,541,000	2,544,000	201,611	258,593,40
2017/12/19	2,577,000	2,604,000	2,576,000	2,578,000	239,572	263,606,00
2017/12/18	2,531,000	2,562,000	2,531,000	2,560,000	147,005	250,751,80
2017/12/15	2,562,000	2,574,000	2,526,000	2,531,000	298,571	253,854,20

그림 5.8 종합차트의 엑셀 저장 메뉴

'이전 일자부터 보임'을 클릭하여 이전 일자부터 최근 일자 순으로 나열되게 데이터 정렬 순서를 변경합니다. 그리고 '데이터를 클립보드로 복사'를 클릭하면 차트 데이터가 복사됩니다. 여기서 '데이터를 엑셀로 저장'을 사용하지 않는 이유는 차트 데이터가 xls 형식의 파일로만 저장되고 xls 형식의 파일을 파이썬에서 읽어 오기가 번거롭기 때문입니다.

5.1.4 일별 데이터 엑셀 파일 저장

이렇게 차트 데이터를 복사하고 마이크로소프트 오피스 엑셀(Microsoft Office Excel), 아파치 오픈오피스 칼크(Apache OpenOffice Calc), 리브레 오피스 칼크(LibreOffice Calc) 등에 붙여 넣습니다. 마이크로소프트 오피스는 유료이고, 아파치 오픈오피스와 리브레 오피스는 무료입니다. 여기서는 마이크로소프트 오피스 엑셀을 기준으로 설명을 이어 나가겠습니다.

엑셀에 복사한 차트 데이터를 붙여 넣으면 그림 5.9와 같은 모습일 것입니다. 여기서 차트 데이터 구성에 포함되지 않는 5일, 20일, 60일, 120일 평균 거래량 열을 삭제합니다.

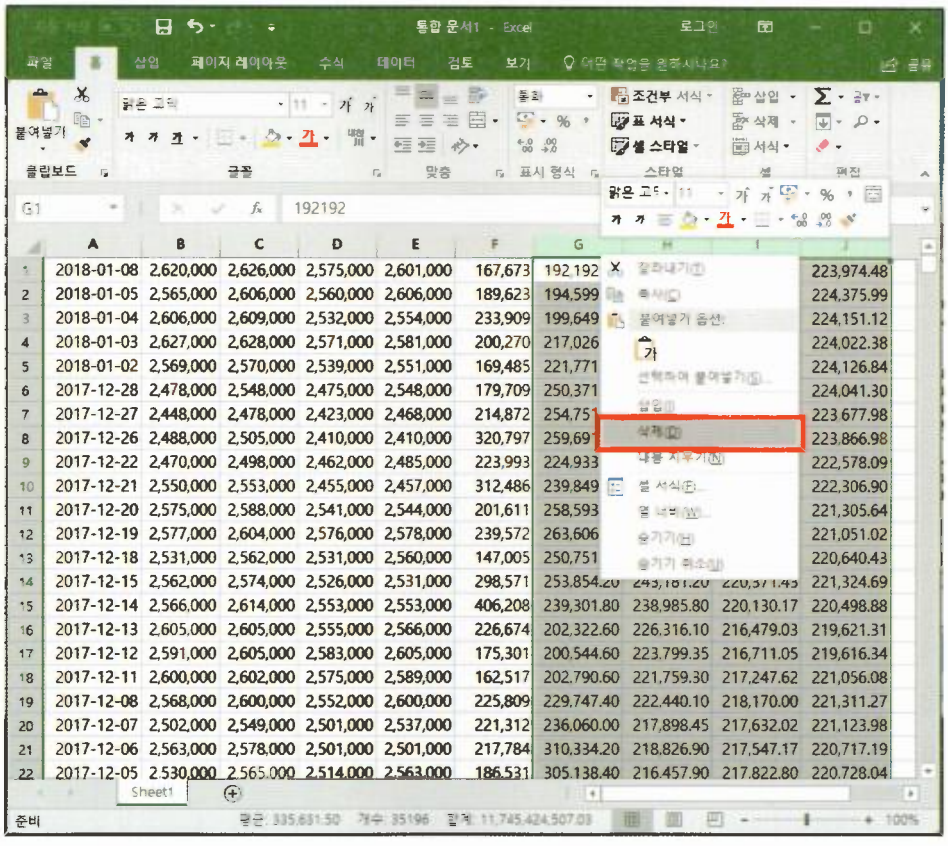


그림 5.9 엑셀에 차트 데이터 붙여넣기

키보드 F12를 누르면 그림 5.10과 같이 저장 윈도우가 나타납니다. 여기서 파일 이름을 적당히 정하고 파일 형식을 'CSV UTF-8(쉼표로 분리)'로 선택하여 저장합니다.

그림 5.10 차트 데이터를 CSV로 저장

여기서는 파일명을 종목코드로 정했습니다. 파일명은 자유롭게 정할 수 있으나 한글을 쓰지 않는 것을 권장합니다. 그 이유는 Pandas 라이브러리에서 한글 파일명을 읽을 때 에러가 발생할 수 있기 때문입니다.

파일 형식에 인코딩을 UTF-8로 할지 윈도우 기본(CP949 또는 EUC-KR)으로 할지를 정해줍니다. 파이썬에서 저장한 CSV 파일을 읽을 때 인코딩 관련 오류가 발생하면 인코딩을 지정해 줘야 할 수 있습니다.

5.2 방법 2. 증권사 API 사용

증권사에서 제공하는 API를 사용하면 프로그래밍으로 주식 데이터를 동적으로 획득할 수 있습니다. 이 방법은 실시간으로 주식 데이터를 확인해야 하는 요구가 있을 경우 매우 유용한 방법입니다.

5.2.1 증권사 API 설치

증권사 API는 대표적으로 대신증권 API, 키움증권 API, 이베스트투자증권 API를 들 수 있습니다. 저자는 이 세 가지 API를 모두 사용해 보았는데 대신증권 API를 추천합니다. 대신증권 API가 가장 직관적이었고, 구현하기 쉬웠습니다. 참고로 저자는 위 증권사들과 어떤 이해관계도 없음을 알립니다.

먼저 대신증권 크레온 API 설치 파일을 다운로드합니다. 대신증권 크레온 홈페이지(https://www.creontrade.com)에 접속하고 그림 5.11과 같은 하단부에서 '다운로드센터' 링크를 클릭합니다.

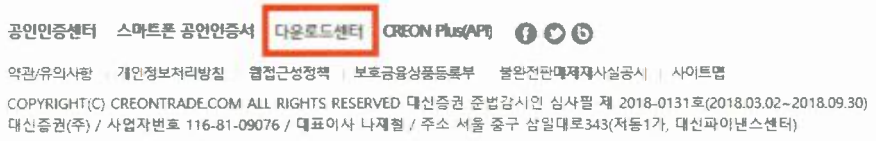


그림 5.11 대신증권 크레온 홈페이지 하단부

다운로드센터 페이지로 이동하게 되면 그림 5.12와 같이 제품별 다운로드 버튼들이 표시됩니다.

그림 5.12 크레온 HTS 다운로드

여기서 'CREON HTS' 칸에 있는 '다운로드' 버튼을 클릭하여 대신증권 크레온 HTS 설치파일을 받습니다. 내려받은 설치파일로 크레온 HTS 설치를 진행합니다. 설치과정 설명은 생략하겠습니다.

5.2.2 대신증권 크레온 API 사용 환경 준비

대신증권 크레온을 설치했으면 API를 사용할 준비가 된 것입니다. API를 부를 프로그램은 파이썬으로 작성할 것입니다. 그런데 파이썬에서 대신증권 크레온 API를 호출하기 위해서 준비해야 하는 사항들이 있습니다. 바로 파이썬 32비트 버전과 win32com 패키지를 설치해야 한다는 것입니다.

아나콘다(Anaconda) 32비트 버전을 설치하면 파이썬 32비트 버전 및 win32com 패키지를 한꺼번에 설치하고 그 외의 주요 모듈들을 같이 설치할 수 있습니다. 그림 5.13은 아나콘다 다운로드 페이지(https://www.anaconda.com/download/)입니다.

그림 5.13 아나콘다 32비트 다운로드

아나콘다 32비트 설치파일을 다운로드해서 설치를 진행합니다. 설치 과정에 대한 설명은 생략하겠습니다.

5.2.3 대신증권 크레온 HTS 실행

대신증권 크레온을 실행시키면 그림 5.14와 같은 화면이 나타납니다. 여기서 상단의 'creon plus' 탭을 선택해 줍니다.

그림 5.14 크레온 플러스로 HTS 로그인

아이디와 비밀번호를 입력해 주고[6] '조회전용'을 체크하여 로그인합니다. 로그인에 성공하게 되면 윈도우 화면 우측하단에 있는 시스템 트레이에 그림 5.15와 같이 대신증권 크레온 플러스 아이콘이 뜨게 됩니다.

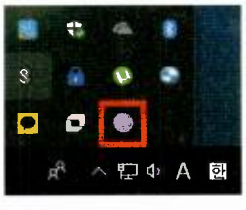


그림 5.15 대신증권 크레온 플러스 아이콘

이 아이콘이 제대로 생겼는지 확인하시기 바랍니다.

6 이미 대신증권에 회원으로 가입되어 있다고 가정합니다.

5.2.4 대신증권 크레온 API를 이용한 차트 데이터 획득 프로그램 작성

이제 대신증권 크레온 API를 사용할 준비를 마쳤습니다. 파이썬 프로그램을 작성하여 크레온 API로 주식 데이터를 획득해 보겠습니다.

이를 위해 모듈 creon.py에 Creon 클래스를 작성합니다. 예제 5.1은 Creon 클래스의 생성자를 보여줍니다.

예제 5.1 Creon 클래스의 생성자

https://github.com/quantylab/rltrader/blob/master/creon.py

```
import time
import win32com.client
import pandas as pd


class Creon:
    def __init__(self):
        self.obj_CpCodeMgr = win32com.client.Dispatch('CpUtil.CpCodeMgr')
        self.obj_CpCybos = win32com.client.Dispatch('CpUtil.CpCybos')
        self.obj_StockChart = win32com.client.Dispatch('CpSysDib.StockChart')
```

Creon 클래스는 win32com 패키지의 client 모듈을 이용합니다. Dispatch 클래스의 인스턴스로 대신증권 크레온 모듈을 사용할 수 있습니다. 여기서는 'CpUtil.CpCodeMgr', 'CpUtil.CpCybos', 'CpSysDib.StockChart' 모듈을 사용합니다.

각 모듈의 상세정보는 그림 5.16과 같이 '대신증권 CREON Plus' 웹페이지에서 확인할 수 있습니다. 그림 5.11에서 볼 수 있는 'CREON Plus(API)' 링크로 이 웹페이지에 접근할 수 있습니다.

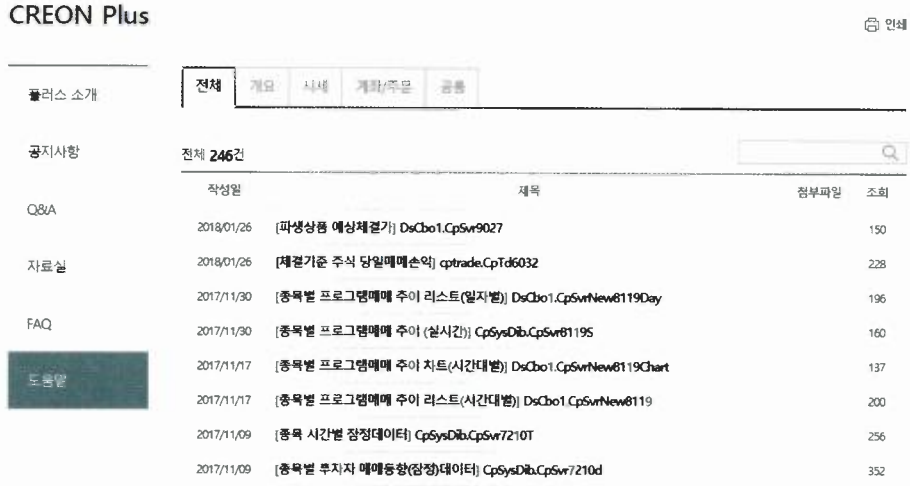


그림 5.16 '대신증권 CREON Plus' 웹페이지

이어서 차트 데이터를 획득하기 위한 함수를 설명합니다. 예제 5.2는 이 함수의 초반부를 보여줍니다.

예제 5.2 Creon 클래스의 creon_7400_주식차트조회() 함수 (1)

https://github.com/quantylab/rltrader/blob/master/creon.py

```
12      def creon_7400_주식차트조회(self, code, date_from, date_to):
13          b_connected = self.obj_CpCybos.IsConnect
14          if b_connected == 0:
15              print("연결 실패")
16              return None
17
18          list_field_key = [0, 1, 2, 3, 4, 5, 8]
19          list_field_name = ['date', 'time', 'open', 'high', 'low', 'close', 'volume']
20          dict_chart = {name: [] for name in list_field_name}
```

이 함수의 파라미터로 가져올 주식 종목의 코드(code), 가져올 데이터의 시작일(date_from), 가져올 데이터의 종료일(date_to)을 지정해 줍니다.

먼저 13번 줄에서 대신증권 크레온 프로그램에 연결되어 있는지 확인합니다. 연결이 되었으면 가져올 필드 키(field key)를 정해줍니다. 여기서는 date, time, open, high, low, close, volume 필드를 가져오도록 했습니다.

예제 5.3은 차트 데이터를 받아오기 위해 API를 호출하는 부분을 보여줍니다.

예제 5.3 Creon 클래스의 creon_7400_주식차트조회() 함수 (2)

https://github.com/quantylab/rltrader/blob/master/creon.py

```
22          self.obj_StockChart.SetInputValue(0, 'A'+code)
23          self.obj_StockChart.SetInputValue(1, ord('1'))  # 0: 개수, 1: 기간
24          self.obj_StockChart.SetInputValue(2, date_to)  # 종료일
25          self.obj_StockChart.SetInputValue(3, date_from)  # 시작일
26          self.obj_StockChart.SetInputValue(5, list_field_key)  # 필드
27          self.obj_StockChart.SetInputValue(6, ord('D'))  # 'D', 'W', 'M', 'm', 'T'
28          self.obj_StockChart.BlockRequest()
```

22번에서 28번 줄까지는 API를 호출하기 위해 입력값들을 설정해 주는 부분입니다. 23번 줄에서는 데이터를 가져오는 방식을 기간으로 설정합니다. 24번 줄에서는 종료일을, 25번 줄에서는 시작일을 입력합니다. 25번 줄에서는 획득할 필드를 리스트로 입력합니다. 26번 줄에서는 차트 데이터의 단위를 일(Day)로 설정합니다. 그리고 28번 줄에서 입력한 설정에 따라 데이터를 요청합니다.

예제 5.4는 요청 결과 및 출력물을 가져오는 부분입니다.

예제 5.4 Creon 클래스의 creon_7400_주식차트조회() 함수 (3)

https://github.com/quantylab/rltrader/blob/master/creon.py

```
30          status = self.obj_StockChart.GetDibStatus()
31          msg = self.obj_StockChart.GetDibMsg1()
32          print("통신상태: {} {}".format(status, msg))
33          if status != 0:
34              return None
35
36          cnt = self.obj_StockChart.GetHeaderValue(3)  # 수신개수
37          for i in range(cnt):
```

```
38              dict_item = {
39                  {name: self.obj_StockChart.GetDataValue(pos, i)
40                  for pos, name in zip(range(len(list_field_name)), list_field_name)}
41              }
42              for k, v in dict_item.items():
43                  dict_chart[k].append(v)
44
45          print("차트: {} {}".format(cnt, dict_chart))
46          return pd.DataFrame(dict_chart, columns=list_field_name)
```

30번 줄에서 요청 결과 상태(status)를 받아옵니다. 이 값이 0이 아니면 이상이 있음을 의미합니다. 36번 줄에서 결과 출력물의 개수를 확인합니다. 39번 줄에서 'CpSysDib.StockChart'의 GetDataValue() 함수로 값을 받아옵니다.

43번 줄에서 받아온 값을 dict_chart 딕셔너리(dictionary)에 추가해 줍니다. 46번 줄에서 이렇게 구성된 dict_chart 딕셔너리를 Pandas DataFrame 객체로 만들어서 반환합니다.

예제 5.5는 작성한 Creon 클래스를 사용하는 방법을 보여줍니다.

예제 5.5 Creon 클래스 사용법

https://github.com/quantylab/rltrader/blob/master/creon.py

```
49  if __name__ == '__main__':
50      creon = Creon()
51      print(creon.creon_7400_주식차트조회('035420', 20150101, 20171201))
```

먼저 Creon 클래스의 객체를 생성하고 이 객체의 creon_7400_주식차트조회() 함수를 호출합니다. 이 때, 차트 데이터를 받아올 대상 종목의 코드값과 가져올 값의 기간을 파라미터로 넣어 줍니다.

주의할 점은 Creon 클래스를 호출할 때는 win32com 패키지로 대신증권 크레온 API 모듈을 사용하기 때문에 관리자 권한이 필요하다는 것입니다. 그러므로 Anaconda Prompt를 관리자 권한으로 띄워서 이 코드를 실행하는 것을 권장합니다.

5.3 방법 3. 포털 사이트 사용

포털 사이트에서 주식 데이터를 받아오는 방법을 살펴봅니다. 여기서는 데이터 획득을 위한 대표적인 파이썬 라이브러리인 'pandas-datareader'를 주로 다룹니다.

5.3.1 pandas-datareader, fix_yahoo_finance 설치하기

Anaconda Prompt에서 pandas-datareader, fix_yahoo_finance 패키지를 설치합니다. 다음 명령으로 설치합니다.

- pandas-datareader 설치 명령: 'pip install pandas-datareader'
- fix_yahoo_finance 설치 명령: 'pip install fix_yahoo_finance --upgrade --no-cache-dir'

pandas-datareader는 대표적인 오픈 데이터를 쉽게 가져올 수 있게 작성된 라이브러리입니다. 야후 금융(Yahoo Finance) 데이터 역시 제공했으나 최근 야후의 데이터 제공 방식이 변경됨에 따라 현재는 pandas-datareader에서 제공하지 않습니다. fix_yahoo_finance는 pandas-datareader에서 야후 금융에서 데이터를 크롤링 방식으로 획득할 수 있도록 합니다.

설치가 성공적으로 이루어졌는지 확인하시기 바랍니다. 각 명령의 마지막 출력 메시지가 다음과 같아야 합니다.

- pandas-datareader 설치 확인 메시지: Successfully installed pandas-datareader-0.6.0
- fix_yahoo_finance 설치 확인 메시지: Successfully installed fix-yahoo-finance-0.0.21 multitasking-0.0.7

5.3.2 Google Finance에서 주식 데이터 획득하기

pandas_datareader에서 안정적이지는 않지만 구글에서 주식 데이터를 획득할 수 있습니다. 안정적인 방법이 아니기 때문에 이 방법을 권장하지는 않겠습니다. 예제 5.6은 pandas_datareader로 구글에서 주식 데이터를 가져오는 소스코드를 보여줍니다.

예제 5.6 구글에서 주식 데이터 획득하기

https://github.com/quantylab/rltrader/blob/master/portal.py

```
1    from pandas_datareader import data
2    chart_data = data.DataReader("KRX:005930", "google")
```

데이터 획득에 성공하면 pandas DataFrame으로 결과를 확인할 수 있습니다. 그러나 최근 많은 경우에 다음과 같은 에러 메시지를 출력하며 데이터 획득에 실패하곤 합니다.

```
b'<html><head><meta http-equiv="content-type" content="text/html; charset=utf-8"/><title>Sorry...</title><style> body { font-family: verdan
a, arial, sans-serif; background-color: #fff; color: #000; }</style></head><body><div><table><tr><td><b><font face=sans-serif size=10><font
 color=#4285f4>G</font><font color=#ea4335>o</font><font color=#fbbc05>o</font><font color=#4285f4>g</font><font color=#34a853>l</font><font
 color=#ea4335>e</font></font></b></td><td style="text-align: left; vertical-align: bottom; padding-bottom: 15px; width: 50%"><div style="bo
rder-bottom: 1px solid #dfdfdf;">Sorry...</div></td></tr></table></div><div style="margin-left: 4em;"><h1>We'll're sorry...</h1><p>... but you
r computer or network may be sending automated queries. To protect our users, we can't process your request right now.</p></div><div style
="margin-left: 4em;">See <a href="https://support.google.com/websearch/answer/86640">Google Help</a> for more information.<br/><br/></div><d
iv style="text-align: center; border-top: 1px solid #dfdfdf;"><a href="https://www.google.com">Google Home</a></div></body></html>'
```

이 메시지는 구글에서 데이터 요청을 거부했음을 의미합니다. (다시 말하지만, 이 방법을 사용하지 않는 게 좋습니다.)

5.3.3 Yahoo Finance에서 주식 데이터 획득하기

야후의 경우 최근 데이터 제공 API 서비스를 중단했습니다. 그림 5.17에서 볼 수 있듯이 야후측에서 공식적으로 주식 데이터 제공 서비스를 중단했음을 밝히고 있습니다.

그림 5.17 야후 금융 API 서비스 중단

따라서 pandas_datareader에서 야후 API를 사용하는 기능이 더 이상 동작하지 않습니다. fix_yahoo_finance는 기존의 야후 API를 사용하는 pandas_datareader의 기능을 그대로 사용할 수 있도록 데이터 획득을 야후 API가 아닌 크롤링 방식으로 변경해 줍니다. 예제 5.7은 야후에서 주식 데이터를 획득하는 소스코드를 보여줍니다.

예제 5.7 야후에서 주식 데이터 획득하기

https://github.com/quantylab/rltrader/blob/master/portal.py

```
10  from pandas_datareader import data
11  import fix_yahoo_finance
12  fix_yahoo_finance.pdr_override()
13
14  chart_data = data.get_data_yahoo(
15      '005930.KS', # 코스피: KS, 코스닥: KQ
16      '2016-01-01',
17      '2017-12-31'
18  )
```

먼저 12번 줄에서 fix_yahoo_finance 모듈의 pdr_override() 함수를 호출하여 야후에서 데이터를 획득하는 방식을 크롤링으로 변경합니다.

14번 줄에서 데이터를 획득하는데, get_data_yahoo() 함수에 종목 코드, 시작 일자, 종료 일자를 입력해 줍니다. 코스피의 경우 '.KS'를 코스닥의 경우 '.KQ'를 종목 코드 뒤에 붙여줍니다.

이렇게 하면 그림 5.18과 같이 pandas의 DataFrame으로 데이터를 받아 올 수 있습니다.

Date	Open	High	Low	Close	Adj Close	Volume
2016-01-04	1260000.0	1260000.0	1205000.0	1205000.0	1164615.000	306939
2016-01-05	1202000.0	1218000.0	1186000.0	1208000.0	1167514.375	216002
2016-01-06	1208000.0	1208000.0	1168000.0	1175000.0	1135620.375	366752
2016-01-07	1166000.0	1183000.0	1151000.0	1163000.0	1124022.625	282388
2016-01-08	1163000.0	1186000.0	1163000.0	1171000.0	1131754.625	257763

그림 5.18 야후에서 획득한 주식 데이터

여기서 Adj Close는 수정주가로 기업이 증자, 액면분할 등으로 인해 주가에 변화가 생겼을 때 주가를 현재주가를 기준으로 조정해 준 가격입니다. 일반적으로 주가라 함은 수정주가를 의미합니다.

fix-yahoo-finance 문서(https://pypi.python.org/pypi/fix-yahoo-finance)에서 더 상세한 사용법을 확인할 수 있습니다.

5.4 이번 장의 요점

- 주식 데이터를 획득하는 방법은 다양합니다.
- 주로 증권사 HTS, 증권사 API, 포털 사이트를 이용하여 주식 데이터를 획득할 수 있습니다.
- 증권사 HTS에서 쉽게 주식 데이터를 획득할 수 있지만 증권사 계좌를 개설해야 합니다.
- 증권사 API로 프로그램적으로 주식 데이터를 획득할 수 있지만 증권사 계좌를 개설해야 합니다.
- 포털 사이트에서 주식 데이터를 획득하는 방법은 간편하지만 안정성이 떨어질 수 있습니다.

06장

모델 구축:
투자 시뮬레이션

이전 장에서 주식투자 시뮬레이션을 하기 위한 RLTrader의 모듈들을 모두 구현했습니다. 이번 장에서는 실제 주식 데이터를 받아오는 방법과 이 데이터를 전처리하여 학습에 사용하는 방법과 학습 과정과 그 결과를 확인하는 방법을 다룹니다.

6.1 주식 데이터 전처리

이번 장에서는 준비한 주식 데이터를 RLTrader에서 사용하기 위해 하나의 추가 모듈을 설명합니다. data_manager 모듈은 준비한 CSV 파일을 읽고 전처리하여 학습 데이터를 만듭니다.

6.1.1 코드 조각 1: CSV 파일을 읽는 부분

먼저 준비한 CSV 파일을 파이썬 Pandas 라이브러리로 읽어옵니다. Pandas의 read_csv() 함수로 CSV 파일을 쉽게 읽어올 수 있습니다. 여기서 주의할 점은 CSV 파일명에 한글을 포함하지 않아야 한다는 것입니다.

예제 6.1은 저장한 CSV 파일을 읽어 오기 위한 간단한 파이썬 코드를 보여줍니다.

예제 6.1 data_manager 모듈의 load_chart_data() 함수

https://github.com/quantylab/rltrader/blob/master/data_manager.py

```
1   import pandas as pd
2   import numpy as np
3
4   def load_chart_data(fpath):
5       chart_data = pd.read_csv(fpath, thousands=',', header=None)
6       chart_data.columns = ['date', 'open', 'high', 'low', 'close', 'volume']
7       return chart_data
```

load_chart_data() 함수는 CSV 파일 경로를 입력으로 받습니다. Pandas의 read_csv() 함수의 첫 번째 인자에 이 파일 경로를 입력합니다. thousands 파라미터로 ','를 넣어주면 1,234,567과 같이 천 단위로 콤마(,)가 붙은 값을 숫자로 인식합니다. header 파라미터에

None을 넣어 준 것은 저장한 CSV에 헤더가 없다는 것을 명시해 주기 위해서입니다. CSV 파일에 헤더 값을 넣어 줬다면 이 파라미터는 넣지 않아도 됩니다.

Pandas의 read_csv() 함수의 명세는 다음 웹사이트에서 확인할 수 있습니다.

https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html

read 파라미터로 sep, header, engine, thousands, encoding 등이 있습니다.

CSV를 읽을 때 다음과 같은 인코딩 관련 에러가 발생하면 CSV 파일을 저장할 때 어떤 인코딩을 선택했는지 확인해야 합니다.

UnicodeDecodeError: 'utf-8' codec can't decode byte 0xc5 in position 0: invalid continuation byte

위 에러는 윈도우 기본인 CP949 또는 EUC-KR로 CSV를 저장했는데 UTF-8로 CSV를 읽을 때 발생합니다. 이 경우 Pandas read_csv() 함수 파라미터에 encoding='CP949'를 추가해서 read_csv(〈파일명〉, encoding='CP949')와 같이 호출하면 해결됩니다.

CSV 파일에 한글이 없으면 인코딩 에러는 발생하지 않을 것입니다.

파일명에 한글이 포함된 경우 Pandas의 read_csv()를 호출할 때 다음과 같은 에러가 발생할 수 있습니다.

File "pandas/_libs/parsers.pyx", line 394, in pandas._libs.parsers.TextReader.__cinit__ (pandas_libs\parsers.c:4209)
File "pandas/_libs/parsers.pyx", line 712, in pandas._libs.parsers.TextReader._setup_parser_source (pandas_libs\parsers.c:8895)
OSError: Initializing from file failed

이 경우 파일명을 영문과 숫자로만 구성하거나 read_csv() 함수에 engine='python'를 넣어서 read_csv(〈파일명〉, engine='python')과 같이 호출하면 됩니다. 꼭 파일명에 한글을 넣을 필요가 없다면 파일명을 영문과 숫자로만 구성할 것을 권장합니다.

이렇게 CSV 파일을 읽으면 Pandas의 DataFrame 객체를 얻을 수 있습니다. 이 DataFrame 객체를 출력해보면 그림 6.1과 같습니다.

	date	open	high	low	close	volume
0	1985-01-04	6511	6511	6473	6473	2232
1	1985-01-05	6434	6465	6427	6427	2167
2	1985-01-07	6434	6511	6427	6442	15417
3	1985-01-08	6434	6434	6358	6358	16879
4	1985-01-09	6312	6312	6090	6136	6488

...

	date	open	high	low	close	volume
8794	2018-01-02	2569000	2570000	2539000	2551000	169485
8795	2018-01-03	2627000	2628000	2571000	2581000	200270
8796	2018-01-04	2606000	2609000	2532000	2554000	233909
8797	2018-01-05	2565000	2606000	2560000	2606000	189623
8798	2018-01-08	2620000	2626000	2575000	2601000	167673

그림 6.1 차트 데이터 예제의 첫 부분과 끝 부분

과거의 주가와 현재의 주가가 크게 차이가 나기 때문에 이 값을 그대로 학습에 사용하기는 어렵습니다. 그러므로 현재 종가와 전일 종가의 비율, 이동평균 종가의 비율, 현재 거래량과 전일 거래량의 비율, 이동평균 거래량의 비율을 학습에서 사용합니다.

6.1.2 코드 조각 2: 종가와 거래량의 이동 평균 구하기

예제 6.2는 종가와 거래량의 이동 평균값을 구하는 소스코드를 보여줍니다.

예제 6.2 data_manager 모듈의 preporcess() 함수

https://github.com/quantylab/rltrader/blob/master/data_manager.py

```
11    def preprocess(chart_data):
12        prep_data = chart_data
13        windows = [5, 10, 20, 60, 120]
14        for window in windows:
15            prep_data['close_ma{}'.format(window)] = prep_data['close'].rolling(window).mean()
16            prep_data['volume_ma{}'.format(window)] = (
17                prep_data['volume'].rolling(window).mean())
18        return prep_data
```

구하려고 하는 이동평균 윈도우는 5, 10, 20, 60, 120으로 11번 줄에서 리스트로 가지고 있습니다. 각 이동평균 크기에 대해서 'close_ma{이동평균크기}'와 'volume_ma{이동평균크기}' 데이터를 15번과 16번 줄에서 만들어 냅니다.

Pandas의 rolling(window) 함수는 window 크기만큼 데이터를 묶어서 합, 평균, 표준편차 등을 계산할 수 있게 준비합니다. 이를 이동합, 이동평균, 이동표준편차라고 합니다

- 이동합(moving sum): 〈Pandas 객체〉.rolling().sum()
- 이동평균(moving average): 〈Pandas 객체〉.rolling().mean()
- 이동표준편차(moving standard deviation): 〈Pandas 객체〉.rolling().std()

이동합은 롤링합(rolling sum), 이동평균은 롤링평균(rolling mean, rolling average), 이동표준편차는 롤링표준편차(rolling standard deviation)라고도 합니다.

그림 6.2는 차트 데이터를 전처리한 후에 추가되는 열들을 보여줍니다. 종가와 거래량과는 다르게 이동평균값은 float 값으로 소수점을 가질 수 있습니다.

close_ma5	volume_ma5	close_ma10	volume_ma10	close_ma20	volume_ma20	close_ma60	volume_ma60	close_ma120	volume_ma120
2492400.0	221771.2	2513200.0	230810.1	2537750.0	234245.50	2.666183e+06	220081.966667	2.552550e+06	224126.841667
2511600.0	217026.6	2518200.0	220980.0	2539700.0	231390.40	2.666483e+06	219411.316667	2.554233e+06	224022.383333
2540400.0	199649.0	2517600.0	229670.4	2539050.0	228220.80	2.666317e+06	219006.666667	2.555492e+06	224151.116667
2568000.0	194599.2	2520400.0	224675.5	2541200.0	228375.40	2.665750e+06	215327.116667	2.557267e+06	224375.991667
2578600.0	192192.0	2526100.0	221281.7	2546200.0	225869.85	2.663567e+06	213570.950000	2.558667e+06	223974.475000

그림 6.2 차트 데이터 전처리 후 추가되는 열

6.1.3 코드 조각 3: 주가와 거래량의 비율 구하기

예제 6.3은 전처리된 데이터로 학습 데이터를 만드는 함수의 초반부를 보여줍니다. 이 함수에서 3.2장에서 정한 특징(feature)을 계산합니다.

예제 6.3 data_manager 모듈의 build_training_data() 함수 (1)

https://github.com/quantylab/rltrader/blob/master/data_manager.py

```
21  def build_training_data(prep_data):
22      training_data = prep_data
23
24      training_data['open_lastclose_ratio'] = np.zeros(len(training_data))
25      training_data['open_lastclose_ratio'].iloc[1:] = \
26          (training_data['open'][1:].values - training_data['close'][:-1].values) / \
27          training_data['close'][:-1].values
28      training_data['high_close_ratio'] = \
29          (training_data['high'].values - training_data['close'].values) / \
30          training_data['close'].values
31      training_data['low_close_ratio'] = \
32          (training_data['low'].values - training_data['close'].values) / \
33          training_data['close'].values
34      training_data['close_lastclose_ratio'] = np.zeros(len(training_data))
35      training_data['close_lastclose_ratio'].iloc[1:] = \
36          (training_data['close'][1:].values - training_data['close'][:-1].values) / \
37          training_data['close'][:-1].values
38      training_data['volume_lastvolume_ratio'] = np.zeros(len(training_data))
39      training_data['volume_lastvolume_ratio'].iloc[1:] = \
40          (training_data['volume'][1:].values - training_data['volume'][:-1].values) / \
41          training_data['volume'][:-1]\
42              .replace(to_replace=0, method='ffill')\
43              .replace(to_replace=0, method='bfill').values
```

build_training_data() 함수의 입력으로 들어오는 prep_data는 전처리된 후의 데이터입니다. 24번 줄에서 27번 줄은 시가/전일종가 비율(open_lastclose_ratio)을 구합니다. 첫 번째 행은 전일 값이 없거나 그 값이 있더라도 알 수 없기 때문에 전일 대비 종가 비율을 구하지 못합니다. 그래서 두 번째 행부터 마지막 행까지 'open_lastclose_ratio' 열에 시가/전일종가 비율을 저장합니다. 시가/전일종가 비율을 구하는 방식은 현재 종가에 전일 종가를 빼고 전일 종가로 나누어 주는 것입니다.

이어서 비슷한 방식으로 고가/종가 비율(high_close_ratio), 저가/종가 비율(low_close_ratio), 종가/전일종가 비율(close_lastclose_ratio), 거래량/전일거래량 비율(volume_lastvolume_ratio)을 계산합니다. 다만 거래량/전일거래량 비율을 구할 때는 거래량 값이 0이면 이전의 0이 아닌 값으로 바꾸어 줍니다.

그림 6.3은 이렇게 계산된 후의 학습 데이터 일부를 보여줍니다.

	open_lastclose_ratio	high_close_ratio	low_close_ratio	close_lastclose_ratio	volume_lastvolume_ratio
8794	0.008242	0.007448	-0.004704	0.001177	-0.056892
8795	0.029792	0.018210	-0.003874	0.011760	0.181638
8796	0.009686	0.021535	-0.008614	-0.010461	0.167968
8797	0.004307	0.000000	-0.017652	0.020360	-0.189330
8798	0.005372	0.009612	-0.009996	-0.001919	-0.115756

그림 6.3 학습 데이터의 예 (1)

Pandas의 DataFrame 데이터에서 일부 데이터를 잘라서 가져올 수 있습니다. 이를 슬라이스(slice, 조각 자르기)라고 합니다. DataFrame은 행과 열로 구성되어 있는데, loc() 함수로 행과 열을 각각 슬라이스할 수 있습니다.

예를 들어 다음과 같은 DataFrame 객체를 data_frame이라고 할 때,

행 \ 열	'a'	'b'	'c'
0	1	2	3
1	4	5	6
2	7	8	9

b번 열에서 1, 2 행을 가져오고 싶다면 다음과 같이 loc() 함수를 쓸 수 있습니다.

- data_frame.loc[1:, 'b']
- data_frame.loc[[1, 2], 'b']

두 경우 다 결과는 다음과 같이 같으나 그 의미는 다릅니다.

행 \ 열	'b'
1	5
2	8

loc() 함수의 첫 번째 인자에 '1:'을 넣으면 두 번째 행부터 마지막 행까지 슬라이스하고, [1, 2]를 넣으면 두 번째와 세 번째 행을 슬라이스합니다.

Pandas의 replace() 함수로 특정 값을 바꿀 수 있습니다. 특정 값을 이전의 값으로 변경하고자 할 때 ffill 메서드를 사용하고, 이후의 값으로 변경하고자 할 때 bfill 메서드를 사용합니다. series가 [1, 3, 0, 5, 7]일 때 결과는 다음과 같습니다.

- series.replace(to_replace=0, method='ffill')

  결과: [1, 3, 3, 5, 7]

- series.replace(to_replace=0, method='bfill')

  결과: [1, 3, 5, 5, 7]

6.1.4 코드 조각 4: 주가와 거래량의 이동 평균 비율 구하기

예제 6.4는 이어서 이동평균 종가 비율과 이동평균 거래량 비율을 구하는 소스코드를 보여줍니다.

예제 6.4 data_manager 모듈의 build_training_data() 함수 (2)

https://github.com/quantylab/rltrader/blob/master/data_manager.py

```
45        windows = [5, 10, 20, 60, 120]
46        for window in windows:
47            training_data['close_ma%d_ratio' % window] = \
48                (training_data['close'] - training_data['close_ma%d' % window]) / \
49                training_data['close_ma%d' % window]
50            training_data['volume_ma%d_ratio' % window] = \
51                (training_data['volume'] - training_data['volume_ma%d' % window]) / \
52                training_data['volume_ma%d' % window]
53
54        return training_data
```

이동평균 윈도우는 5, 10, 20, 60, 120로 각 윈도우에 대해서 이동평균 종가 비율, 이동평균 거래량 비율을 구합니다. 즉, close_ma5_ratio, close_ma10_ratio, close_ma20_ratio, close_ma60_ratio, close_ma120_ratio, volume_ma5_ratio, volume_ma10_ratio, volume_ma20_ratio, volume_ma60_ratio, volume_ma120_ratio를 구합니다.

이동평균 종가 비율을 구하는 방법은 현재 종가에 이동평균 값을 빼고 그 값에 이동평균 값을 나누는 것입니다. 이동평균 거래량 비율 역시 같은 방식으로 계산합니다.

그림 6.4는 위에서 구한 학습 데이터의 특징들의 예를 보여줍니다.

close_ma5_ratio	volume_ma5_ratio	close_ma10_ratio	volume_ma10_ratio	close_ma20_ratio
0.023511	-0.235766	0.015041	-0.265695	0.005221
0.027632	-0.077210	0.024938	-0.093719	0.016262
0.005353	0.171601	0.014458	0.018455	0.005888
0.014798	-0.025572	0.033963	-0.156014	0.025500
0.008687	-0.127576	0.029650	-0.242264	0.021522

volume_ma20_ratio	close_ma60_ratio	volume_ma60_ratio	close_ma120_ratio	volume_ma120_ratio
-0.276464	-0.043202	-0.229901	-0.000607	-0.243799
-0.134493	-0.032058	-0.087239	0.010479	-0.106027
0.024924	-0.042124	0.068045	-0.000584	0.043533
-0.169687	-0.022414	-0.119372	0.019057	-0.154887
-0.257657	-0.023490	-0.214907	0.016545	-0.251375

그림 6.4 학습 데이터의 예 (2)

6.2 주식 데이터 학습

준비한 차트 데이터와 학습 데이터로 강화학습을 진행합니다. 본 장에서는 주식 데이터를 읽고, 차트 데이터와 학습 데이터를 준비하고, RLTrader로 주식투자 강화학습을 실행하는 main 모듈을 설명합니다.

6.2.1 코드 조각 1: 강화학습을 실행하는 메인(main) 모듈

예제 6.5는 차트 데이터, 학습 데이터를 입력으로 받아서 학습을 시작하는 main 모듈의 main 문을 보여줍니다.

예제 6.5 주식투자 강화학습 main 영역 (1)

https://github.com/quantylab/rltrader/blob/master/main.py

```
1    import logging
2    import os
3    import settings
4    import data_manager
5    from policy_learner import PolicyLearner
6
7
8    if __name__ == '__main__':
9        stock_code = '005930'  # 삼성전자
10
11       # 로그 기록
12       log_dir = os.path.join(settings.BASE_DIR, 'logs/%s' % stock_code)
13       timestr = settings.get_time_str()
14       file_handler = logging.FileHandler(filename=os.path.join(
15           log_dir, "%s_%s.log" % (stock_code, timestr)), encoding='utf-8')
16       stream_handler = logging.StreamHandler()
17       file_handler.setLevel(logging.DEBUG)
18       stream_handler.setLevel(logging.INFO)
19       logging.basicConfig(format="%(message)s",
20           handlers=[file_handler, stream_handler], level=logging.DEBUG)
```

9번 줄에서 투자 시뮬레이션 대상 종목의 종목코드를 정해줍니다. 여기서는 삼성전자의 종목코드인 '005930'을 입력해 주었습니다. 12번 줄부터 20번 줄까지는 로그를 기록하기 위한 코드입니다.

대부분의 프로그래밍 언어에서 프로그램을 시작하기 위한 부분을 main이라고 합니다. 파이썬에서는 if __name__ == "__main__": 구문 안에 작성된 코드가 main의 역할을 합니다.

강화학습의 최종 산출물은 학습된 정책 신경망 모델이라 할 수 있습니다. 나중에 이 학습된 정책 신경망을 사용하기 위해서는 이 정책 신경망 모델을 파일로 저장해 둬야 합니다. 먼저 저장할 파일명을 정합니다. 여기서는 'models/{종목코드}' 폴더에 파일명을 'model_{모델 저장일시}'로 저장합니다.

6.2.2 코드 조각 2: 강화학습에 필요한 주식 데이터 준비 부분

예제 6.6은 강화학습에 사용할 주식 데이터를 준비하는 소스코드입니다.

예제 6.6 주식투자 강화학습 main 영역 (2)

https://github.com/quantylab/rltrader/blob/master/main.py

```
22      # 주식 데이터 준비
23      chart_data = data_manager.load_chart_data(
24          os.path.join(settings.BASE_DIR,
25                       'data/chart_data/{}.csv'.format(stock_code)))
26      prep_data = data_manager.preprocess(chart_data)
27      training_data = data_manager.build_training_data(prep_data)
28
29      # 기간 필터링
30      training_data = training_data[(training_data['date'] >= '2017-01-01') &
31                                    (training_data['date'] <= '2017-12-31')]
32      training_data = training_data.dropna()
```

23번 줄에서는 5장에서 준비한 차트 데이터를 Pandas DataFrame 객체로 불러옵니다. 26번 줄에서 불러온 차트 데이터를 전처리해서 학습 데이터를 만들 준비를 합니다. 27번 줄에서는 학습 데이터에 포함될 열들을 추가합니다. 이렇게 준비된 training_data는 차트 데이터의 열들, 전처리에서 추가된 열들, 학습 데이터의 열들이 모두 포함된 데이터입니다.

30번 줄에서는 특정 기간의 데이터만 학습에 사용하기 위해 필터링을 합니다. 여기서는 2017년 전체 데이터를 사용합니다. 그리고 32번 줄에서는 거래 정지 등에 의해 값이 없는 데이터를 제거합니다.

32번 줄에서는 차트 데이터의 열들을 지정하고 33번 줄에서 이 열들만 포함하도록 차트 데이터 객체를 제한해 줍니다. 36번 줄에서는 학습 데이터의 열들을 지정하고 37번 줄에서 이 열들만 포함하도록 학습 데이터 객체를 제한합니다.

6.2.3 코드 조각 3: 데이터를 차트 데이터와 학습 데이터로 분리하는 부분

예제 6.7은 준비한 주식 데이터를 차트 데이터와 학습 데이터로 분리하는 소스코드입니다.

예제 6.7 주식투자 강화학습 main 영역 (3)

https://github.com/quantylab/rltrader/blob/master/main.py

```
34        # 차트 데이터 분리
35        features_chart_data = ['date', 'open', 'high', 'low', 'close', 'volume']
36        chart_data = training_data[features_chart_data]
37
38        # 학습 데이터 분리
39        features_training_data = [
40            'open_lastclose_ratio', 'high_close_ratio', 'low_close_ratio',
41            'close_lastclose_ratio', 'volume_lastvolume_ratio',
42            'close_ma5_ratio', 'volume_ma5_ratio',
43            'close_ma10_ratio', 'volume_ma10_ratio',
44            'close_ma20_ratio', 'volume_ma20_ratio',
45            'close_ma60_ratio', 'volume_ma60_ratio',
46            'close_ma120_ratio', 'volume_ma120_ratio'
47        ]
48        training_data = training_data[features_training_data]
```

35번 줄에서 차트 데이터에 포함되어야 할 열들을 지정하고 36번 줄에서 차트 데이터를 분리합니다. 39번 줄에서 학습 데이터의 열들을 지정하고 48번 줄에서 학습 데이터를 분리합니다. 이렇게 최종적으로 강화학습에 필요한 차트 데이터와 학습 데이터가 준비되었습니다.

6.2.4 코드 조각 4: 강화학습을 시작하는 부분

예제 6.8은 주식투자 강화학습을 시작하는 부분을 보여줍니다.

예제 6.8 주식투자 강화학습 main 영역 (4)

https://github.com/quantylab/rltrader/blob/master/main.py

```
50        # 강화학습 시작
51        policy_learner = PolicyLearner(
```

```
52          stock_code=stock_code, chart_data=chart_data, training_data=training_data,
53          min_trading_unit=1, max_trading_unit=2, delayed_reward_threshold=.2, lr=.001)
54      policy_learner.fit(balance=10000000, num_epoches=1000,
55                         discount_factor=0, start_epsilon=.5)
56
57      # 정책 신경망을 파일로 저장
58      model_dir = os.path.join(settings.BASE_DIR, 'models/%s' % stock_code)
59      model_path = os.path.join(model_dir, 'model_%s.h5' % timestr)
60      policy_learner.policy_network.save_model(model_path)
```

51번 줄에서 정책 학습기 객체를 생성합니다. 이때 종목 코드, 차트 데이터, 학습 데이터, 최소 투자 단위, 최대 투자 단위, 지연 보상 임계치, 학습 속도를 정해 줍니다.

54번 줄에서 생성한 정책 학습기 객체의 fit() 함수를 호출합니다. 이때 초기 자본금, 수행할 에포크 수, 할인 요인, 초기 탐험률을 정해 줍니다.

학습이 완료되면 학습 결과인 정책 신경망 모델을 저장합니다. 58번 줄에서 저장할 폴더 경로를 정해 주고, 59번 줄에서 정책 신경망 모델을 저장할 파일명을 정해 주고, 60번 줄에서 이 경로에 정책 신경망 모델을 저장하기 위해 save_model() 함수를 호출합니다.

6.3 학습 과정 및 결과 확인

학습을 수행했으면 그 결과를 확인할 차례입니다. 결과는 두 가지 방법으로 확인할 수 있습니다. 텍스트 형식인 로그를 확인하는 방법과 가시화된 이미지를 확인하는 방법입니다.

6.3.1 콘솔에 출력되는 로그의 의미

강화학습이 시작되면 콘솔에 그림 6.5와 같은 로그가 출력되는 것을 볼 수 있습니다.

```
[Epoch 0050/1000]  Epsilon:0.4755  #Expl.:122/246  #Buy:100  #Sell:98  #Hold:48  #Stocks:10  PV:₩10,100,000  POS:19  NEG:16  Loss:  0.171609
[Epoch 0051/1000]  Epsilon:0.4750  #Expl.:110/246  #Buy:116  #Sell:103  #Hold:27  #Stocks:65  PV:₩10,122,500  POS:23  NEG:22  Loss:  0.181179
[Epoch 0052/1000]  Epsilon:0.4745  #Expl.:122/246  #Buy:109  #Sell:98  #Hold:39  #Stocks:55  PV:₩10,175,000  POS:23  NEG:22  Loss:  0.170261
[Epoch 0053/1000]  Epsilon:0.4740  #Expl.:106/246  #Buy:111  #Sell:97  #Hold:38  #Stocks:70  PV:₩10,507,500  POS:21  NEG:21  Loss:  0.166023
[Epoch 0054/1000]  Epsilon:0.4735  #Expl.:121/246  #Buy:115  #Sell:105  #Hold:26  #Stocks:50  PV:₩10,232,500  POS:18  NEG:17  Loss:  0.170147
[Epoch 0055/1000]  Epsilon:0.4730  #Expl.:110/246  #Buy:111  #Sell:100  #Hold:35  #Stocks:55  PV:₩10,735,000  POS:19  NEG:18  Loss:  0.173739
[Epoch 0056/1000]  Epsilon:0.4725  #Expl.:116/246  #Buy:109  #Sell:97  #Hold:40  #Stocks:60  PV:₩10,227,500  POS:21  NEG:20  Loss:  0.165752
[Epoch 0057/1000]  Epsilon:0.4720  #Expl.:112/246  #Buy:115  #Sell:103  #Hold:28  #Stocks:60  PV:₩10,587,500  POS:20  NEG:21  Loss:  0.166072
[Epoch 0058/1000]  Epsilon:0.4715  #Expl.:119/246  #Buy:114  #Sell:109  #Hold:23  #Stocks:25  PV:₩9,612,500  POS:19  NEG:19  Loss:  0.166146
[Epoch 0059/1000]  Epsilon:0.4710  #Expl.:129/246  #Buy:119  #Sell:107  #Hold:20  #Stocks:60  PV:₩10,662,500  POS:19  NEG:15  Loss:  0.155420
[Epoch 0060/1000]  Epsilon:0.4705  #Expl.:120/246  #Buy:118  #Sell:112  #Hold:16  #Stocks:30  PV:₩9,615,000  POS:12  NEG:15  Loss:  0.170788
```

그림 6.5 학습 과정 로그

먼저 총 수행할 에포크 중에서 몇 번째 에포크인지 표시합니다. [Epoch 0050/1000] 줄은 1,000번의 에포크 중에서 50번째 에포크의 수행 결과를 표시합니다.

각 에포크에서 적용된 무작위 투자 비율, 즉 엡실론(epsilon) 값을 표시합니다. Epsilon은 0에서 1 사이의 실수로 0.5000은 50% 확률로, 0.1000은 10% 확률로 무작위 투자를 수행했다는 것을 의미합니다.

#Expl.은 탐험을 수행한 횟수를 보여줍니다. 110/246일 경우 총 투자 결정 횟수 246번 중에서 110번을 무작위로 투자를 했다는 의미입니다.

#Buy는 매수를 결정한 횟수, #Sell은 매도를 결정한 횟수, #Hold는 매수도 매도도 하지 않은 횟수입니다. 관망(Hold)은 매수를 결정했지만 매수가 불가능하고, 매도를 결정했지만 매도가 불가능한 경우에 발생합니다. 즉, 매수를 결정했는데 잔고가 부족하거나 매도를 결정했는데 보유 주식 잔고가 없는 경우에 관망을 수행합니다. #Stocks는 에포크를 마친 후 최종적으로 보유하고 있는 주식 수입니다.

PV는 에포크를 마친 후의 포트폴리오 가치입니다. 포트폴리오 가치가 높아지도록 학습합니다. 주식투자에서는 정답이 없기 때문에 더욱 학습이 어려우나 강화학습으로 확률적으로 좋은 결정을 내리게 함으로써 수익이 발생하도록 합니다.

POS는 에포크에서 수행한 긍정적 학습의 횟수이고 NEG는 부정적 학습의 횟수입니다. Loss는 정책 신경망 배치 학습에서 발생한 손실의 평균입니다. 즉 학습 데이터와 정책 신경망의 괴리가 얼마나 되었는지를 알려줍니다.

6.3.2 가시화 결과가 저장되는 그림 파일

각 에포크 수행 종료 시점에서 차트 데이터, 에이전트 상태, 에이전트가 수행한 행동, 정책 신경망 출력, 탐험 여부, 포트폴리오 가치를 가시화합니다. 가시화한 결과는 그림 6.6에 나타낸 것처럼 그림 파일로 저장됩니다.

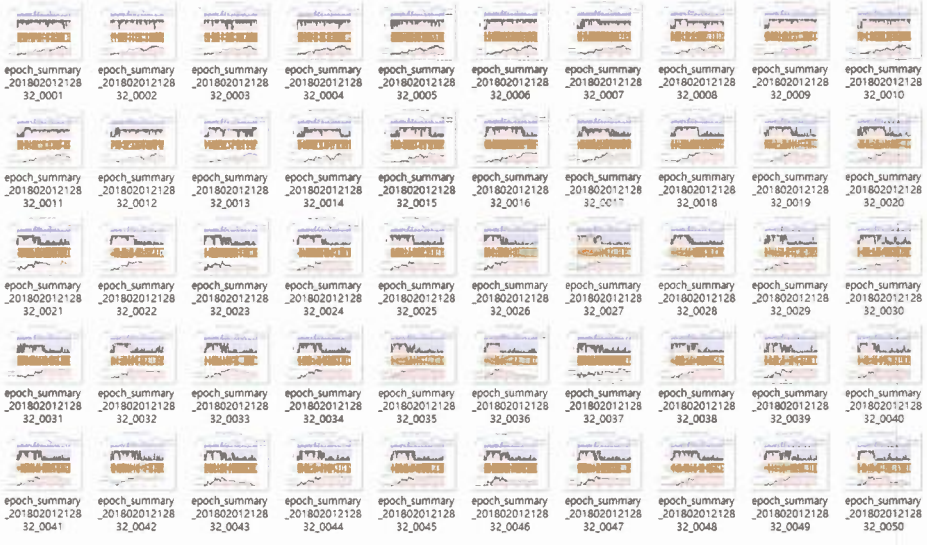


그림 6.6 에포크 가시화 저장 결과

하나의 에포크당 하나의 가시화 결과가 그림 파일로 저장되므로 수행한 에포크 수가 1,000이면 1,000개의 그림 파일이 생성됩니다.

저장된 가시화 결과를 보고 학습이 진행되면서 에이전트 행동의 변화, 정책 신경망 출력의 변화, 포트폴리오 가치의 변화 등을 확인할 수 있으며 학습이 제대로 진행되었는지도 판단해 볼 수 있습니다.

6.4 이번 장의 요점

- 주식 투자 시뮬레이션을 위해 주식 데이터 전처리, 주식 데이터 학습, 학습 과정 및 결과 확인 단계를 수행했습니다.
- 주식 데이터 전처리를 위해 차트 데이터를 읽고 이동 평균을 구합니다.
- 학습 데이터에서 주가와 이전 주가의 비율, 거래량과 이전 거래량의 비율을 특징으로 사용합니다.
- 학습 데이터에서 주가와 주가 이동평균의 비율, 거래량과 거래량 이동평균의 비율을 특징으로 사용합니다.
- 강화학습을 시작하는 메인 모듈을 살펴봤습니다.
- 학습 과정과 결과를 콘솔과 가시화된 그림 파일로 확인하는 방법을 살펴봤습니다.

07장

모델 검증: 투자 시뮬레이션

이번 장에서는 RLTrader에서 코스피 종목 중 시가총액 상위 종목들의 실제 주식 데이터로 주식투자 강화학습을 수행한 결과를 보여줍니다. 여기서는 2016년 한 해 동안의 주식 데이터를 사용하여 강화학습을 진행합니다. 이번 장에서 학습한 정책 신경망 모델을 활용하여 8장에서 2017년 주식 데이터로 투자 시뮬레이션을 진행합니다.

이와 같이 구축한 투자 모델(이 책의 경우에는 6장에서 구축한 모델)이 적합한 모델인지를 과거 데이터(old data), 즉 이미 거래가 성립한 이전 데이터(back data)를 이용해 시험(test)해 보아야 합니다. 이와 같은 과정을 백테스트(back test)라고 합니다.

백테스트를 하려면 몇 가지 기본 환경 설정 변수를 설정해 두어야 합니다. 이때 설정하는 변수를 통계학이나 데이터 과학 분야에서는 보통 '하이퍼 파라미터'라고 부릅니다. RLTrader의 하이퍼 파라미터로는 '초기 자본금', '최소 투자 단위', '최대 투자 단위', '총 에포크 수', '초기 탐험률', '지연 보상 임계치', '학습 속도', '할인 요인'이 있습니다. 이 책에서는 이러한 하이퍼 파라미터를 특별히 학습 파라미터라고 부르겠습니다. 이 파라미터들은 모델 사용자가 일일이 설정해 주어야 하는 것으로, 이 설정 값에 따라 모델의 정확도가 달라지게 되므로, 여러 차례 검증을 해 보아 최적값을 찾아내는 게 좋습니다.

여기서는 학습 속도를 0.0001로 설정한 확률적 경사 하강법(SGD)으로 학습합니다. 초기 투자금은 1,000만 원으로, 에포크는 1,000번, 초기 탐험률은 50%로 동일하게 정했습니다.

주식투자의 경우 지연 보상 발생 시점의 바로 직전 행동이 그 이전의 행동보다 더 중요하다고 볼 수는 없습니다. 그러므로 할인 요인을 0으로 정하여 지연 보상 발생까지의 행동들을 동등하게 보상합니다.

지연 보상 임계치, 최소 및 최대 투자 단위 등은 종목의 특성에 맞게 설정했습니다.

투자 시뮬레이션 대상 주식 종목은 다음과 같이 임의로 정했습니다. 여기서 정한 종목들과 저자와는 아무런 이해관계가 없음을 밝힙니다.

- 삼성전자(005930): 2016년, 2017년 상승 추세 종목
- SK하이닉스(000660): 2016년, 2017년 상승 추세 종목

- 현대차(005380): 2016년, 2017년 박스권 종목
- LG화학(051910): 2016년 하락 추세, 2017년 상승 추세 종목
- NAVER(035420): 2016년 상승 추세, 2017년 박스권 종목
- KT(030200): 2016년, 2017년 박스권 종목

7.1 투자 시뮬레이션 결과 1: 삼성전자(005930)

삼성전자(005930)는 2016년, 2017년 상승 추세 종목입니다. 2016년 동안의 데이터를 학습하여 정책 신경망을 구축합니다. 여기서 학습시킨 정책 신경망을 활용하여, 8.2절에서는 2017년도 데이터를 활용해 투자 시뮬레이션을 해 봅니다.

7.1.1 종목의 개요

그림 7.1은 삼성전자의 2016년 차트입니다. 차트는 주가를 보여주는 일봉차트와 거래량을 보여주는 막대 그래프로 구성됩니다. 차트의 Y축으로 왼쪽은 거래량을, 오른쪽은 주가를 보여줍니다.

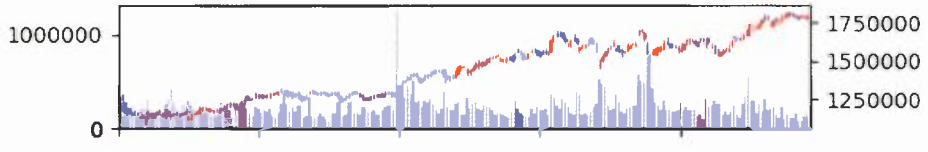


그림 7.1 삼성전자(005930) 종목의 2016년 차트

삼성전자 종목은 2016년 한 해 동안 꾸준한 주가 상승 추세를 보이고 있습니다. 그리고 2분기에서 대량 거래가 발생한 것을 볼 수 있습니다.

종가만을 두고 봤을 때 그림 7.2와 같이 주가의 최고가는 1,812,000원, 최저가는 1,126,000원, 최대낙폭(maximum draw-down, MDD)은 약 13.2%입니다.

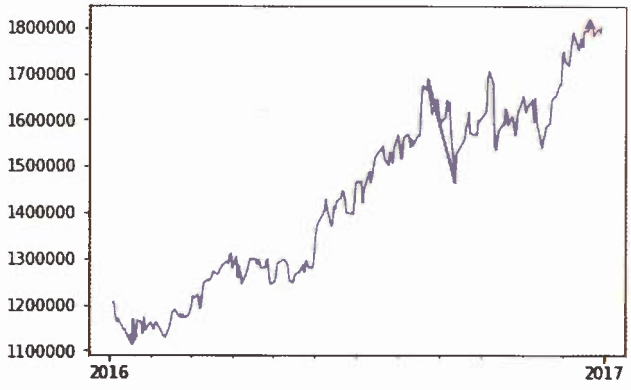


그림 7.2 삼성전자(005930) 종목의 2016년 최고가, 최저가, MDD

여기서 빨간색 삼각형은 최고가 지점을, 파란색 역삼각형은 최저가 지점을, 파란색 실선은 MDD의 시작점과 끝지점을 연결한 선입니다.

7.1.2 주식 데이터 전처리

삼성전자(005930)의 차트 데이터 뒷부분(tail)은 그림 7.3과 같습니다.

	date	open	high	low	close	volume
8546	2016-12-23	1801000	1804000	1780000	1782000	166697
8547	2016-12-26	1780000	1800000	1778000	1798000	96472
8548	2016-12-27	1799000	1810000	1793000	1799000	93069
8549	2016-12-28	1792000	1799000	1780000	1788000	133258
8550	2016-12-29	1771000	1802000	1770000	1802000	150329

그림 7.3 삼성전자(005930) 종목의 차트 데이터

차트 데이터는 일자(date), 시가(open), 고가(high), 저가(low), 종가(close), 거래량(volume)으로 구성됩니다. 이 데이터는 환경(environment) 모듈이 관리하며 매수금, 손익액 등을 계산할 때 사용합니다.

차트 데이터를 전처리하면 그림 7.4와 같이 이동평균 값들이 추가됩니다.

	close_ma5	volume_ma5	close_ma10	volume_ma10	close_ma20	volume_ma20	close_ma60	volume_ma60	close_ma120	volume_ma120
8546	1800600.0	137170.2	1785000.0	164735.6	1761700.0	232371.70	1.660300e+06	261656.566667	1.609967e+06	249185.941667
8547	1801200.0	134721.4	1789600.0	151553.9	1767750.0	223901.25	1.663633e+06	259343.533333	1.612733e+06	248657.375000
8548	1798600.0	122874.4	1792900.0	137491.9	1773850.0	210473.40	1.666717e+06	256543.650000	1.615483e+06	248121.225000
8549	1795200.0	123317.6	1794000.0	135907.5	1775950.0	188601.10	1.669533e+06	254607.800000	1.618542e+06	246440.183333
8550	1793800.0	127965.0	1798300.0	139393.3	1778600.0	180582.20	1.671383e+06	247260.566667	1.621475e+06	245777.058333

그림 7.4 삼성전자(005930) 종목의 이동평균값 전처리

전처리 결과, 종가의 5일 평균(close_ma5), 거래량의 5일 평균(volume_ma5), 종가의 10일 평균(close_ma10), 거래량의 10일 평균(volume_ma10), 종가의 20일 평균(close_ma20), 거래량의 20일 평균(volume_ma20), 종가의 60일 평균(close_ma60), 거래량의 60일 평균(volume_ma60), 종가의 120일 평균(close_ma120), 거래량의 120일 평균(volume_ma120)이 계산됩니다.

전처리 후의 데이터에 그림 7.5와 같이 학습 데이터 준비를 위해 필요한 값들을 추가합니다.

	open_lastclose_ratio	high_close_ratio	low_close_ratio	close_lastclose_ratio	volume_lastvolume_ratio
8546	-0.004422	0.012346	-0.001122	-0.014925	0.311625
8547	-0.001122	0.001112	-0.011123	0.008979	-0.421273
8548	0.000556	0.006115	-0.003335	0.000556	-0.035274
8549	-0.003891	0.006152	-0.004474	-0.006115	0.431819
8550	-0.009508	0.000000	-0.017758	0.007830	0.128105

	close_ma5_ratio	volume_ma5_ratio	close_ma10_ratio	volume_ma10_ratio	close_ma20_ratio	volume_ma20_ratio
8546	-0.010330	0.215257	-0.001681	0.011906	0.011523	-0.282628
8547	-0.001777	-0.283915	0.004694	-0.363448	0.017112	-0.569131
8548	0.000222	-0.242568	0.003402	-0.323095	0.014178	-0.557811
8549	-0.004011	0.080608	-0.003344	-0.019495	0.006785	-0.293440
8550	0.004571	0.174767	0.002057	0.078452	0.013156	-0.167531

	close_ma60_ratio	volume_ma60_ratio	close_ma120_ratio	volume_ma120_ratio
8546	0.073300	-0.362917	0.106855	-0.331034
8547	0.080767	-0.628015	0.114877	-0.612028
8548	0.079368	-0.637220	0.113599	-0.624905
8549	0.070958	-0.476615	0.104698	-0.459268
8550	0.078149	-0.392022	0.111334	-0.388352

그림 7.5 삼성전자(005930) 종목의 비율값 전처리

학습 데이터에는 전일 종가 대비 시가(open_lastclose_ratio), 종가 대비 고가(high_close_ratio), 종가 대비 저가(low_close_ratio), 전일 종가 대비 당일 종가(close_lastclose_ratio), 전일 거래량 대비 당일 거래량(volume_lastvolume_ratio), 5일 평균 종가 대비 당일 종가(close_ma5_ratio), 5일 평균 거래량 대비 당일 거래량(volume_ma5_ratio), 10일 평균 종가 대비 당일 종가(close_ma10_ratio), 10일 평균 거래량 대비 당일 거래량(volume_ma10_ratio), 20일 평균 종가 대비 당일 종가(close_ma20_ratio), 20일 평균 거래량 대비 당일 거래량(volume_ma20_ratio), 60일 평균 종가 대비 당일 종가(close_ma60_ratio), 60일 평균 거래량 대비 당일 거래량(volume_ma60_ratio), 120일 평균 종가 대비 당일 종가(close_ma120_ratio), 120일 평균 거래량 대비 당일 거래량(volume_ma120_ratio)이 계산되어 추가됩니다.

7.1.3 학습 파라미터 설정

삼성전자(005930)를 대상으로 투자 시뮬레이션을 할 때에 설정한 학습 파라미터는 다음과 같습니다.

초기 자본금	10,000,000원
최소 투자 단위	1주
최대 투자 단위	1주
총 에포크 수	1,000회
초기 탐험률	0.5 (50%)
지연 보상 임계치	0.05 (5%)
학습 속도	0.0001
할인 요인	0% (미적용)

우리는 초기 자본금을 10,000,000원으로 설정해 주식투자 시뮬레이션을 합니다. 삼성전자의 경우 2018년 5월 4일에 액면분할을 했지만 2016년, 2017년의 삼성전자 주가는 100만 원이 넘는 고가였습니다. 그래서 투자 단위를 1주로 정했습니다. 1,000회 시뮬레이션을 반복하며 학습을 진행하고 초기 탐험률을 50%로 정하고 에포크가 지나가면서 동일 비율로 탐험률을 줄여나갑니다. 최종 에포크에서는 탐험률이 0이 되게 합니다.

이익 또는 손실이 기준점 대비 5% 초과로 발생했을 때 학습을 수행하도록 지연 보상 임계치를 0.05로 정하고 0.0001의 학습 속도로 학습을 진행합니다.

탐험률에 따라 무작위 투자를 진행하기 때문에 동일한 실험 환경을 갖추더라도 실험 결과가 다를 수 있습니다.

7.1.4 에포크 10일 때의 결과

총 1,000번의 에포크 중에서 에포크 10에서의 학습 결과를 그림 7.6에서 보여줍니다. 아직 많은 학습이 진행되지는 않았지만 2016년의 삼성전자가 워낙 상승 추세가 뚜렷한 종목이다 보니 정책 신경망이 어느 정도 적합화(fitting)되었습니다.

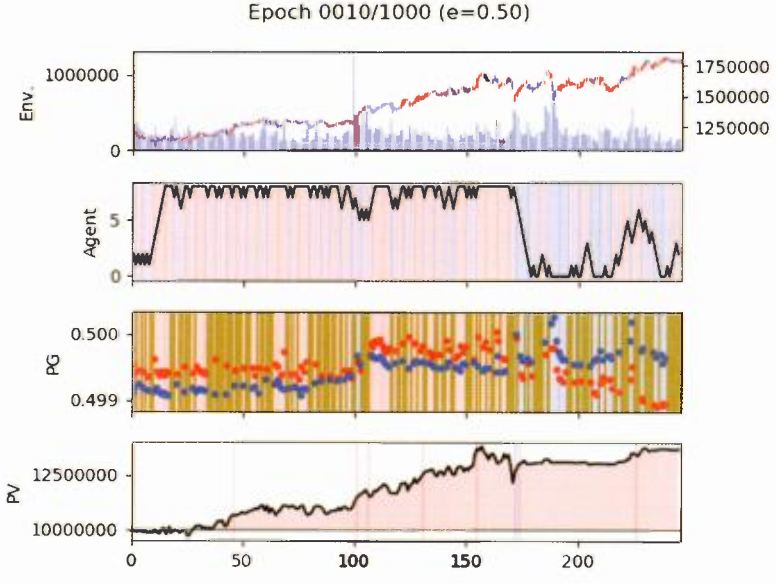


그림 7.6 삼성전자(005930) 종목의 강화학습 투자 시뮬레이션: 에포크 10/1000

50%의 확률로 무작위 투자를 진행했습니다. 세 번째 차트(PG)에서 노란색으로 표시되어 있는 부분이 무작위로 투자를 한 부분입니다. 그 외의 부분에서 매수와 매도에 대한 정책 신경망의 출력값이 빨간색점과 파란색 점으로 각각 표시됩니다.

정책 신경망이 3분기까지 대부분의 구간에서 매수 신호를 보내고, 4분기쯤부터는 매도 신호를 보내고 있습니다. 그 결과 네 번째 차트인 포트폴리오 가치(PV)를 보면 수익이 발생하고 있음을 알 수 있습니다.

삼성전자의 경우 상승 추세 종목이었기 때문에 아주 비정상적인 투자를 하지 않았으면 수익이 발생했을 것입니다. 에포크 10에서의 결과처럼 50% 가까이 무작위로 투자했는데도 큰 수익이 발생한 것을 볼 수 있습니다.

에포크 10에서의 로그는 다음과 같습니다.

```
[Epoch 0010/1000] Epsilon:0.4955 #Expl.:129/246 #Buy:79 #Sell:77 #Hold:90 #Stocks:2
PV:₩13,688,000 POS:7 NEG:1 Loss: 0.426497
```

탐험률은 49.55%임을 알 수 있고. 총 246번의 행동 중에서 129번을 무작위로 결정하여 수행했습니다. 79번의 매수, 77번의 매도를 했습니다. 결정된 매수, 매도를 수행할 수 없는 상황에서 관망을 90번 했습니다. 최종적으로 보유한 주식 수는 2주고 최종 포트폴리오 가치는 13,688,000원입니다.

긍정적 학습을 일곱 번, 부정적 학습을 한 번 했습니다. 학습과정에서 발생한 손실(loss)[7]은 0.426497입니다. 손실이 학습이 진행됨에 따라 더 줄어들 것입니다.

7.1.5 에포크 200일 때의 결과

총 1,000번의 에포크 중에서 에포크 200에서의 학습 결과를 그림 7.7에서 보여줍니다.

---

7 학습에서의 손실(loss)과 주식 투자에서의 손실을 혼동하지 않도록 유의해야 합니다. 학습에서의 손실은 학습과정에서 발생한 정답과의 오차 정도를 의미하고 주식 투자에서의 손실은 투자 과정에서 발생한 금전적인 손실을 의미합니다.

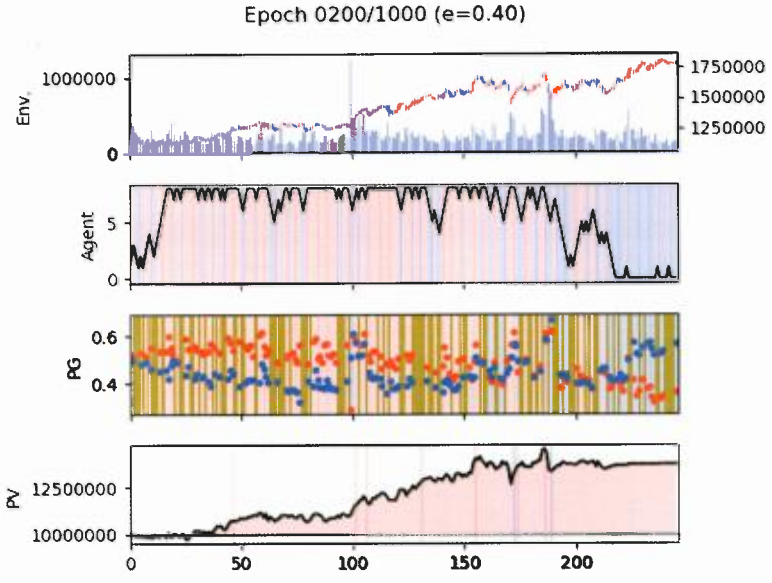


그림 7.7 삼성전자(005930) 종목의 강화학습 투자 시뮬레이션: 에포크 200/1000

에포크 10의 결과와 큰 차이는 없지만 세 번째 차트에서 정책 신경망의 출력을 보면 4분기 초반에서도 매수 신호를 보내고 있게 정책 신경망이 적합화된 것을 볼 수 있습니다.

에포크 200의 로그는 다음과 같습니다.

```
[Epoch 0200/1000] Epsilon:0.4004 #Expl.:110/246 #Buy:74 #Sell:74 #Hold:98 #Stocks:0
PV:₩13,735,000 POS:7 NEG:2 Loss: 0.527588
```

탐험률은 약 40%입니다. 총 246번의 행동 수행에서 110번의 행동을 무작위 결정했습니다.

74번 매수하고 74번 매도했습니다. 행동한 결정을 수행할 수 없어서 98번을 관망했습니다. 에포크 200에서의 최종 보유 주식은 없고 최종 포트폴리오 가치는 13,735,000원으로 에포크 10일때보다 조금 상승했습니다.

정책 신경망은 일곱 차례의 긍정적 학습과 두 차례에 걸친 부정적 학습으로 업데이트되었습니다. 손실은 0.527588로 에포크 10일 때의 손실보다 오히려 조금 높지만 추세적으로 줄어듦을 이후의 에포크 로그에서 확인하겠습니다.

7.1.6 에포크 600일 때의 결과

총 1,000번의 에포크 중에서 에포크 600에서의 학습 결과를 그림 7.8에서 보여줍니다.

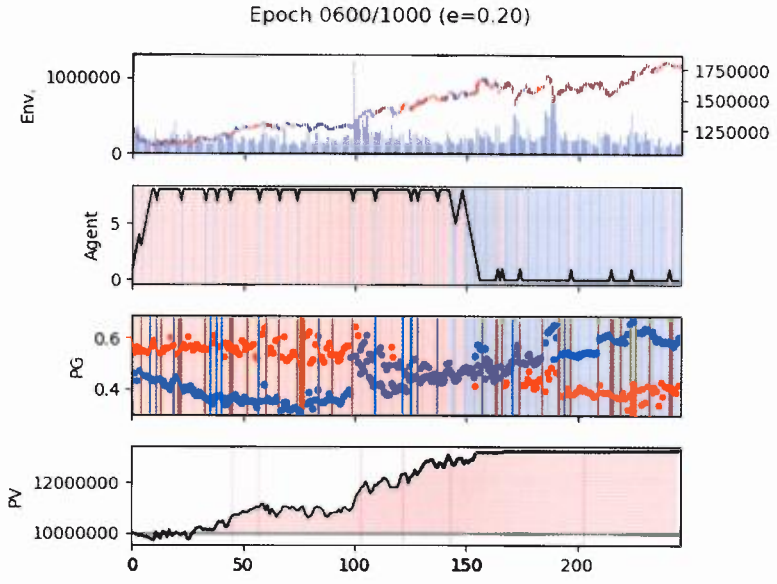


그림 7.8 삼성전자(005930) 종목의 강화학습 투자 시뮬레이션: 에포크 600/1000

이제 정책 신경망이 많은 적합화 과정을 거쳐서 견고해졌습니다. 그런데 정책 신경망이 에포크 200일 때보다 오히려 주가는 상승하고 있지만 매도 신호를 일찍 보내고 있어서 결과적으로 최종 포트폴리오 가치의 수익률이 적어졌습니다. 3분기 후반과 4분기 초반에서 나타나는 급락 부분에서 부정적 학습이 이루어진 결과입니다.

에포크 600의 로그는 다음과 같습니다.

```
[Epoch 0600/1000] Epsilon:0.2002 #Expl.:48/246 #Buy:32 #Sell:32 #Hold:182 #Stocks:0
PV:₩13,293,000 POS:6 NEG:0 Loss: 0.303930
```

탐험률은 약 20%이고 총 246회의 행동 수행에서 48번은 무작위로 수행했습니다. 32번의 매수, 32번의 매도를 수행했습니다. 182번은 결정된 행동을 수행할 수 없어서 관망했습니

다. 최종 보유 주식은 없으며 최종 포트폴리오 가치는 13,293,000원으로 약 33%의 수익률이 발생했습니다.

6번의 긍정적 학습으로 정책 신경망이 추가로 적합화되었습니다. 학습 손실은 0.303930으로 줄어들었습니다.

7.1.7 에포크 1000일 때의 결과

그림 7.9는 마지막 에포크에서의 학습 결과를 보여줍니다.

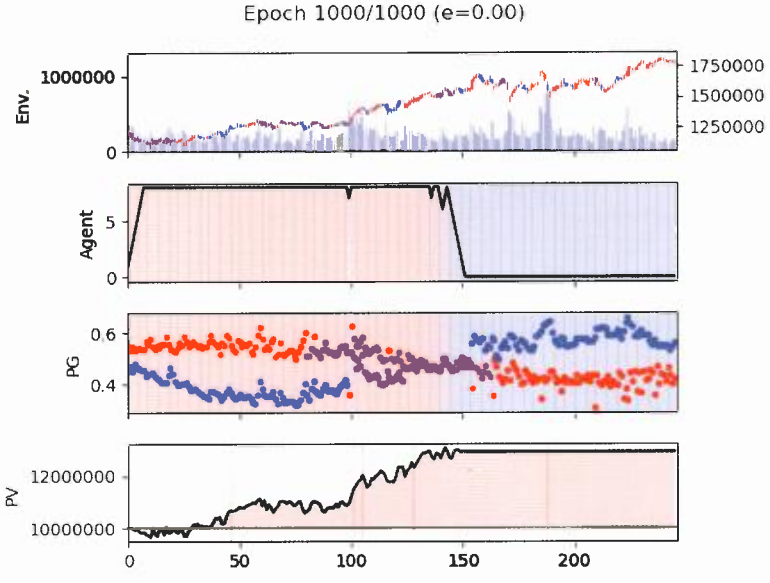


그림 7.9 삼성전자(005930) 종목의 강화학습 투자 시뮬레이션: 에포크 1000/1000

에포크 600일 때와 큰 차이가 없는 결과를 보여줍니다. 여기서는 무작위 투자를 하지 않고 오로지 정책 신경망의 출력만을 근거로 투자 행동을 결정했습니다.

최종 에포크에서의 로그는 다음과 같습니다.

```
[Epoch 1000/1000] Epsilon:0.0000 #Expl.:0/246 #Buy:12 #Sell:12 #Hold:222 #Stocks:0
PV:₩12,911,000 POS:5 NEG:0 Loss: 0.285672
```

탐험률은 0%이고 246번의 행동을 정책 신경망을 통해서 결정하였습니다. 12번의 매수와 12번의 매도를 하였고 결정한 행동을 수행할 수 없었던 222번을 관망했습니다. 최종 보유 주식 수는 없습니다. 최종 포트폴리오 가치는 12,911,000원입니다.

5번의 긍정적 학습으로 정책 신경망이 추가로 적합화되었습니다. 학습 손실은 0.285672로 줄어들었습니다.

7.1.8 총평

적합화 과정을 통해 손실을 줄이면서 정책 신경망이 견고해졌습니다. 삼성전자의 2016년 주가는 뚜렷한 상승 추세를 보이고 있으므로 투자 시뮬레이션 결과에서도 꽤 큰 수익률을 보여주고 있습니다.

그러나 4분기 후반에서의 주가 상승을 학습에 제대로 반영하지 못한 아쉬운 부분이 있습니다. 3분기 후반과 4분기 초반에서 나타나는 급락 부분에서 부정적 학습이 많이 수행되면서 4분기에서 주식을 보유하지 않으려는 성향이 나타났습니다. 이 때문에 4분기 후반부에서 포트폴리오 가치에 변화가 없고 그 결과 학습도 수행하지 않게 되었기 때문입니다.

이럴 때 몇 가지 아이디어를 내 볼 수 있습니다. 현재 학습을 포트폴리오 가치의 변동률을 기준으로 하고 있는데, 주가가 상승할 때 주식을 보유하고 있지 않았으면 부정적 학습을 추가로 수행해 볼 수도 있습니다.

7.2 투자 시뮬레이션 결과 2: SK하이닉스(000660)

SK하이닉스(000660)는 2016년, 2017년 상승 추세 종목입니다. 2016년 동안의 데이터를 학습하여 정책 신경망을 구축합니다. 여기서 학습시킨 정책 신경망을 활용하여, 8.2장에서는 2017년도 데이터를 활용해 투자 시뮬레이션을 해 봅니다.

7.2.1 종목의 개요

그림 7.10은 SK하이닉스의 2016년 차트입니다. 차트는 주가를 보여주는 일봉차트와 거래량을 보여주는 막대 그래프로 구성됩니다. 차트의 Y축 중에 왼쪽은 거래량을, 오른쪽은 주가를 보여줍니다.

2016년도 SK하이닉스는 삼성전자와 유사한 주가 추세를 보입니다. SK하이닉스의 주가가 삼성전자에 비해 주가가 가벼워서 상승률이 더 큽니다.

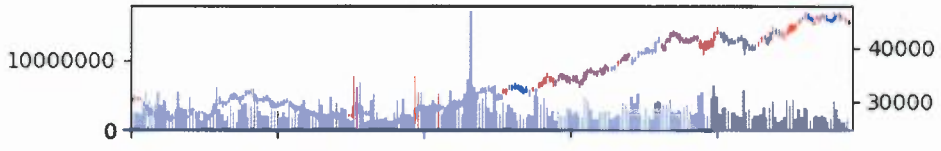


그림 7.10 SK하이닉스(000660) 종목의 2016년 차트

SK하이닉스 종목은 2016년 한 해 동안 꾸준한 주가 상승 추세를 보이고 있습니다. 그림 7.11에서 표시한 주가의 최고가는 46,400원, 최저가는 25,750원, 최대낙폭(maximum draw-down)은 약 19.4%입니다.

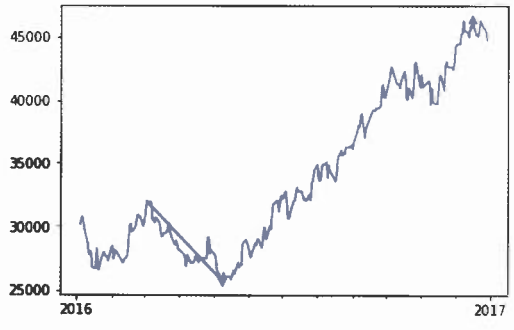


그림 7.11 SK하이닉스(000660) 종목의 2016년 최고가, 최저가, MDD

여기서 빨간색 삼각형은 최고가 지점을, 파란색 역삼각형은 최저가 지점을, 파란색 실선은 MDD의 시작점과 끝지점을 연결한 선입니다.

7.2.2 주식 데이터 전처리

SK하이닉스(000660)의 차트 데이터 뒷부분(tail)은 그림 7.12와 같습니다.

	date	open	high	low	close	volume
5033	2016-12-23	46200	46400	45950	46300	2982024
5034	2016-12-26	46200	46600	45250	45650	1603230
5035	2016-12-27	45550	45950	45400	45650	1175619
5036	2016-12-28	45350	45750	45100	45350	1296254
5037	2016-12-29	45150	45200	44600	44700	1808856

그림 7.12 SK하이닉스(000660) 종목의 차트 데이터

차트 데이터는 일자(date), 시가(open), 고가(high), 저가(low), 종가(close), 거래량(volume)으로 구성됩니다. 이 데이터는 환경(environment) 모듈이 관리하며 매수금, 손익액 등을 계산할 때 사용합니다.

차트 데이터를 전처리하면 그림 7.13과 같이 이동평균 값들이 추가됩니다.

	close_ma5	volume_ma5	close_ma10	volume_ma10	close_ma20	volume_ma20	close_ma60	volume_ma60	close_ma120	volume_ma120
5033	45420.0	2359578.6	45520.0	2100179.4	44915.0	2472822.35	42571.666667	2.893672e+06	38888.750000	2.924908e+06
5034	45520.0	2280785.8	45555.0	2116443.5	45067.5	2477323.10	42662.500000	2.857647e+06	38996.250000	2.911668e+06
5035	45620.0	2293759.0	45625.0	2051710.2	45230.0	2447230.05	42726.666667	2.816577e+06	39111.250000	2.899988e+06
5036	45690.0	2220607.8	45575.0	1980264.8	45352.5	2295766.55	42772.500000	2.765059e+06	39234.166667	2.858170e+06
5037	45530.0	1773196.6	45485.0	1978266.1	45377.5	2211650.10	42808.333333	2.748282e+06	39351.250000	2.850618e+06

그림 7.13 SK하이닉스(000660) 종목의 이동평균값 전처리

전처리 결과, 종가의 5일 평균(close_ma5), 거래량의 5일 평균(volume_ma5), 종가의 10일 평균(close_ma10), 거래량의 10일 평균(volume_ma10), 종가의 20일 평균(close_ma20), 거래량의 20일 평균(volume_ma20), 종가의 60일 평균(close_ma60), 거래량의 60일 평균(volume_ma60), 종가의 120일 평균(close_ma120), 거래량의 120일 평균(volume_ma120)이 계산됩니다.

전처리 후의 데이터에 그림 7.14와 같이 학습 데이터 준비를 위해 필요한 값들을 추가합니다.

	open_lastclose_ratio	high_close_ratio	low_close_ratio	close_lastclose_ratio	volume_lastvolume_ratio
5033	0.015385	0.002160	-0.007559	0.017582	-0.262954
5034	-0.002160	0.020811	-0.008762	-0.014039	-0.462369
5035	-0.002191	0.006572	-0.005476	0.000000	-0.266718
5036	-0.006572	0.008820	-0.005513	-0.006572	0.102614
5037	-0.004410	0.011186	-0.002237	-0.014333	0.395449

	close_ma5_ratio	volume_ma5_ratio	close_ma10_ratio	volume_ma10_ratio	close_ma20_ratio	volume_ma20_ratio
5033	0.019375	0.263795	0.017135	0.419890	0.030836	0.205919
5034	0.002856	-0.297071	0.002085	-0.242489	0.012925	-0.352838
5035	0.000658	-0.487471	0.000548	-0.427005	0.009286	-0.519612
5036	-0.007441	-0.416262	-0.004937	-0.345414	-0.000055	-0.435372
5037	-0.018230	0.020110	-0.017258	-0.085636	-0.014930	-0.182124

	close_ma60_ratio	volume_ma60_ratio	close_ma120_ratio	volume_ma120_ratio
5033	0.087578	0.030533	0.190576	0.019527
5034	0.070026	-0.438968	0.170625	-0.449377
5035	0.068419	-0.582607	0.167183	-0.594612
5036	0.060261	-0.531202	0.155880	-0.546474
5037	0.044189	-0.341823	0.135923	-0.365451

그림 7.14 SK하이닉스(000660) 종목의 비율값 전처리

학습 데이터에는 전일 종가 대비 시가(open_lastclose_ratio), 종가 대비 고가(high_close_ratio), 종가 대비 저가(low_close_ratio), 전일 종가 대비 당일 종가(close_lastclose_ratio), 전일 거래량 대비 당일 거래량(volume_lastvolume_ratio), 5일 평균 종가 대비 당일 종가(close_ma5_ratio), 5일 평균 거래량 대비 당일 거래량(volume_ma5_ratio), 10일 평균 종가 대비 당일 종가(close_ma10_ratio), 10일 평균 거래량 대비 당일 거래량(volume_ma10_ratio), 20일 평균 종가 대비 당일 종가(close_ma20_ratio), 20일 평균 거래량 대비 당일 거래량(volume_ma20_ratio), 60일 평균 종가 대비 당일 종가(close_ma60_ratio), 60일 평균 거래량 대비 당일 거래량(volume_ma60_ratio), 120일 평균 종가 대비 당일 종가(close_ma120_ratio), 120일 평균 거래량 대비 당일 거래량(volume_ma120_ratio)이 계산되어 추가됩니다.

7.2.3 학습 파라미터 설정

SK하이닉스(000660)를 대상으로 투자 시뮬레이션을 할 때에 설정한 학습 파라미터는 다음과 같습니다.

초기 자본금	10,000,000원
최소 투자 단위	10주
최대 투자 단위	10주
총 에포크 수	1,000회
초기 탐험률	0.5 (50%)
지연 보상 임계치	0.05 (5%)
학습 속도	0.0001
할인 요인	0% (미적용)

10,000,000원으로 주식투자 시뮬레이션을 합니다. SK하이닉스는 2016년 주가가 10만 원이 되지 않기 때문에 투자 단위를 10주로 정했습니다. 1,000회 시뮬레이션을 반복하며 학습을 진행하고 초기 탐험률을 50%로 정하고 에포크가 지나가면서 동일 비율로 탐험률을 줄여나갑니다. 최종 에포크에서는 탐험률이 0이 되게 합니다.

이익 또는 손실이 기준점 대비 5% 초과로 발생했을 때 학습을 수행하도록 지연 보상 임계치를 0.05로 정하고 0.0001의 학습 속도로 학습을 진행합니다.

탐험률에 따라 무작위 투자를 진행하기 때문에 동일한 실험 환경을 갖추더라도 실험 결과가 다를 수 있습니다.

7.2.4 에포크 10일 때의 결과

총 1,000번의 에포크 중에서 에포크 10에서의 학습 결과를 그림 7.15에서 보여줍니다. 에포크 10에서는 아직 적합화가 충분히 되지 않아서 정책 신경망이 매도 신호만을 보내고 있습니다. 정책 신경망이 적합화가 되어 있지 않을수록 무작위 투자를 많이 수행해야 하는 이유입니다.

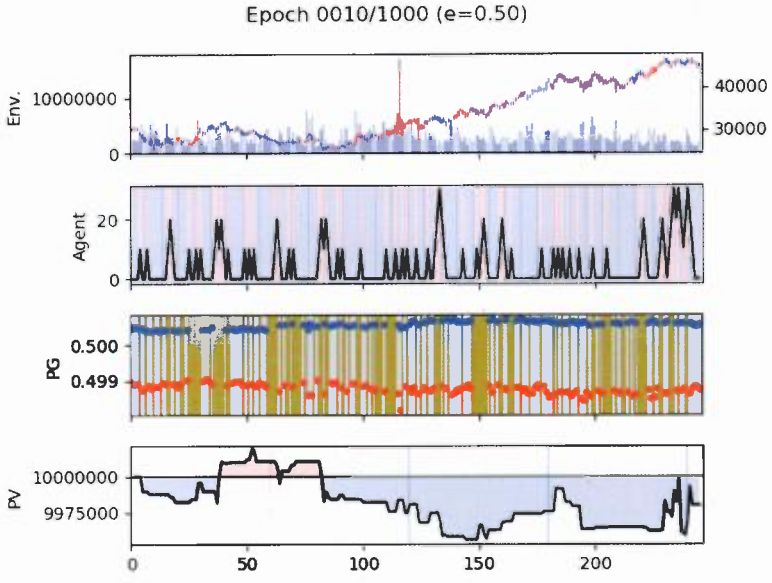


그림 7.15 SK하이닉스(000660) 종목의 강화학습 투자 시뮬레이션: 에포크 10/1000

정책 신경망이 매도 신호만 보내고 있고 약 50%로 무작위 투자를 진행하여 부분적으로 매수를 하였습니다. 그 결과 한 번의 긍정적 학습과 세 번의 부정적 학습을 수행할 수 있었습니다.

에포크 10에서의 로그는 다음과 같습니다.

```
[Epoch 0010/1000] Epsilon:0.4955 #Expl.:121/246 #Buy:58 #Sell:58 #Hold:130 #Stocks:0
PV:₩9,980,000 POS:1 NEG:3 Loss: 1.083758
```

49.55%로 무작위 투자를 수행했습니다. 246번의 행동 중에서 121번을 무작위로 수행했습니다. 이 과정에서 58번 매수하고 58번 매도했습니다. 결정한 매수 또는 매도 행동을 수행할 수 없어서 130번 관망했습니다. 최종 보유 주식 수는 없고 결과적으로 포트폴리오 가치는 9,980,000원으로 약간의 손실이 발생했습니다.

한 번의 긍정적 학습과 세 번의 부정적 학습을 수행했습니다. 이 때의 학습 손실은 1.083758입니다. 이 손실은 앞으로 학습이 진행되면서 줄어들 것입니다.

7.2.5 에포크 200일 때의 결과

총 1,000번의 에포크 중에서 에포크 200에서의 학습 결과를 그림 7.16에서 보여줍니다.

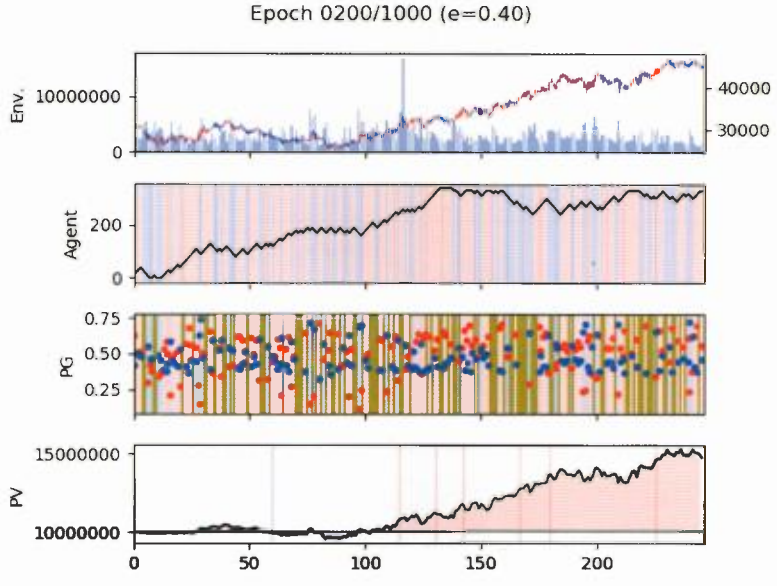


그림 7.16 SK하이닉스(000660) 종목의 강화학습 투자 시뮬레이션: 에포크 200/1000

이제 정책 신경망이 어느 정도 적합화되어 의미 있는 신호를 보내고 있습니다. 결과적으로 포트폴리오 가치는 크게 상승했으나 과하게 매수 신호를 내보내고 있습니다.

에포크 200에서의 로그는 다음과 같습니다.

```
[Epoch 0200/1000] Epsilon:0.4004 #Expl.:99/246 #Buy:129 #Sell:96 #Hold:21 #Stocks:330
PV:₩14,757,500 POS:6 NEG:1 Loss: 0.415333
```

탐험률은 40.04%입니다. 246번의 행동 중에서 99번을 무작위로 수행했습니다. 129번 매수했고 96번 매도했고 21번 관망했습니다. 최종 보유 주식 수는 330주이며 최종 포트폴리오 가치는 14,757,500원으로 약 47.6%의 수익이 발생했습니다.

에포크 200에서는 6번의 긍정적 학습과 한 번의 부정적 학습을 수행했습니다. 이때의 손실은 0.415333으로 에포크 10일 때에 비해 크게 줄었습니다

7.2.6 에포크 600일 때의 결과

총 1,000번의 에포크 중에서 에포크 600에서의 학습 결과를 그림 7.17에서 보여줍니다.

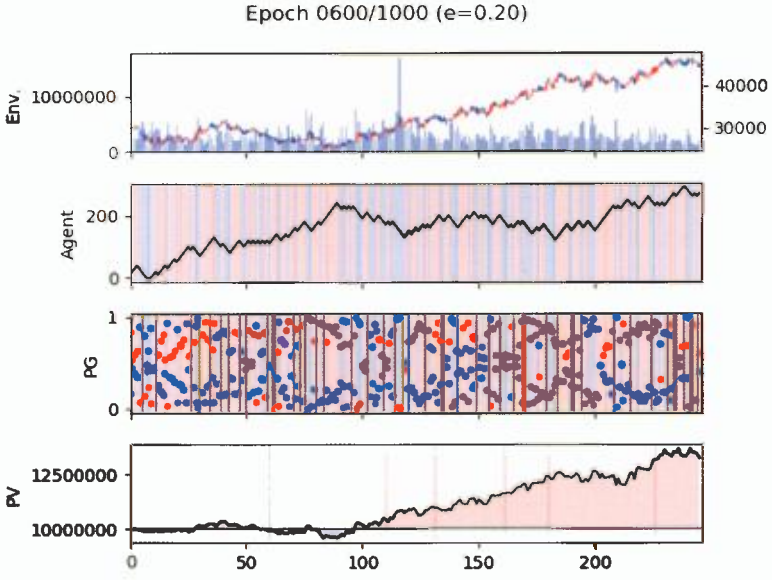


그림 7.17 SK하이닉스(000660) 종목의 강화학습 투자 시뮬레이션: 에포크 600/1000

정책 신경망의 출력을 보면 1과 0에 가까운 출력이 많아졌습니다. 그 의미는 정책 신경망이 결정한 행동의 확률이 높다고 예측한 것입니다.

에포크 600에서의 로그는 다음과 같습니다.

```
[Epoch 0600/1000] Epsilon:0.2002 Loss: 0.424583 #Expl.:48/246 #Stocks:270 #Buy:135
#Sell:108 #Hold:3 PV:₩13,264,000 POS:5 NEG:1
```

약 20%로 무작위 투자를 했습니다. 총 246번의 행동 중에서 48번을 무작위로 수행했습니다. 135번의 매수와 108번의 매도를 수행했고 3번 관망했습니다. 최종 보유 주식은 270주고 최종 포트폴리오 가치는 13,264,000원으로 약 32.6%의 수익이 발생했습니다.

이 에포크에서는 5번의 긍정적 학습과 한 번의 부정적 학습이 진행되었습니다. 이때의 학습 손실은 0.424583으로 에포크 200에 비해 조금 상승했습니다. 주식투자 시뮬레이션 과정에서 학습 데이터가 매우 다양하게 발생하기 때문에 손실이 줄어들었다가 늘어났다가 하기도 합니다. 그러나 학습을 진행하면서 추세적으로 줄어드는 것이 정상입니다. 이후의 에포크에 대해 언급하자면 에포크 900대에서는 손실이 0.1에서 0.15 사이로 줄어들어 있음을 확인했습니다.

7.2.7 에포크 1000일 때의 결과

마지막 에포크에서의 학습 결과를 그림 7.18에서 보여줍니다.

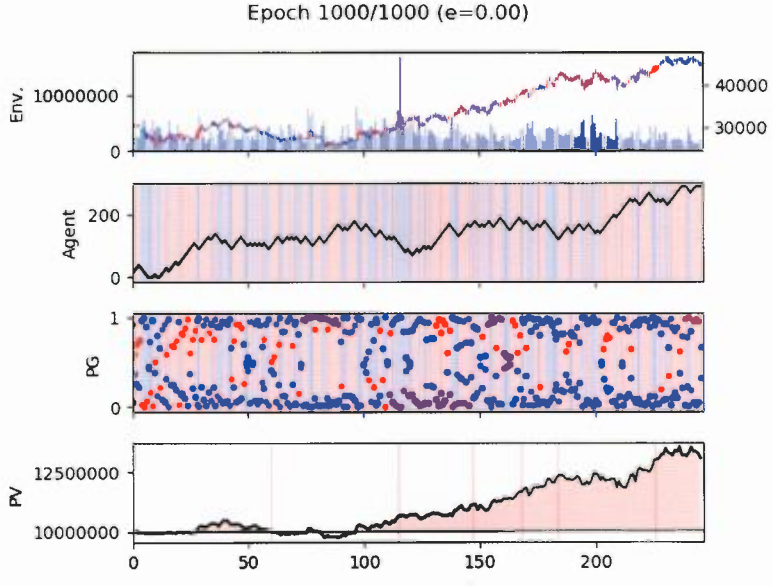


그림 7.18 SK하이닉스(000660) 종목의 강화학습 투자 시뮬레이션: 에포크 1000/1000

마지막 에포크에서는 무작위 투자를 하지 않고 정책 신경망의 출력에 의해 결정한 투자 행동만을 수행합니다.

에포크 1000의 로그는 다음과 같습니다.

```
[Epoch 1000/1000]  Epsilon:0.0000  #Expl.:0/246  #Buy:134  #Sell:105  #Hold:7  #Stocks:290
PV:₩13,071,000  POS:6  NEG:0  Loss:  0.125779
```

탐험률은 0%로 무작위 투자를 수행하지 않습니다. 134번의 매수, 105번의 매도, 7번의 관망을 했습니다. 최종 보유 주식 수는 290주이고 최종 포트폴리오 가치는 13,071,000원으로 약 30.7%의 수익이 발생했습니다.

6번의 긍정적 학습이 진행되었습니다. 이때의 손실은 0.125779로 이전에 살펴본 에포크 10, 에포크 200, 에포크 600에 비해 크게 줄었습니다

7.2.8 총평

에포크 10일 때는 정책 신경망이 제대로 적합화될 만큼 학습이 이루어지지 않아서 뚜렷하게 상승 추세를 보이는 SK하이닉스 종목에서도 수익이 발생하지 않았었습니다. 학습이 어느 정도 이루어진 에포크 200에서 약 47.6%에 달하는 큰 수익이 발생했지만 정책 신경망이 너무 매수에 편중되어 있었습니다. 물론 상승 추세이기 때문에 매수만 하면 수익이 발생하겠지만 정책 신경망이 한쪽으로 편중되면 다른 데이터를 적용했을 때 결과가 좋지 않을 수 있습니다.

에포크 600에서는 에포크 200과 큰 차이가 없었고 에포크 1000에서 학습 손실이 크게 줄어들어서 학습이 잘 진행되었음을 확인했습니다.

7.3 투자 시뮬레이션 결과 3: 현대차(005380)

현대차(005380)는 2016년, 2017년 박스권 종목입니다. 2016년 동안의 데이터를 학습하여 정책 신경망을 구축합니다. 여기서 학습시킨 정책 신경망을 활용하여, 8.2절에서는 2017년도 데이터를 활용해 투자 시뮬레이션을 해 봅니다.

7.3.1 종목의 개요

그림 7.19는 현대차 종목의 2016년 차트를 보여줍니다. 차트는 주가를 보여주는 일봉차트와 거래량을 보여주는 막대 그래프로 구성됩니다. 차트의 Y축으로 왼쪽은 거래량을, 오른쪽은 주가를 보여줍니다.

2016년도 현대차는 1분기 말에 큰폭의 상승을 겪고 2분기 초에 큰폭의 하락을 겪는 등 큰 변동성을 보이다 3분기, 4분기에는 횡보하는 모양세입니다.

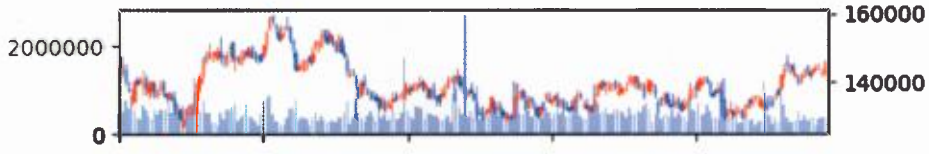


그림 7.19 현대차(005380) 종목의 2016년 차트

그림 7.20에서 표시한 바와 같이 현대차 종목의 2016년 주가의 최고가는 159,000원, 최저가는 129,000원, 최대낙폭은 약 18.9%입니다.

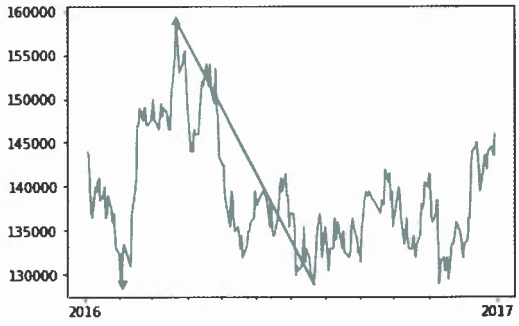


그림 7.20 현대차(005380) 종목의 2016년 최고가, 최저가, MDD

여기서 빨간색 삼각형은 최고가 지점을, 파란색 역삼각형은 최저가 지점을, 파란색 실선은 MDD의 시작점과 끝지점을 연결한 선입니다.

7.3.2 주식 데이터 전처리

현대차(005380)의 차트 데이터 뒷부분(tail)은 그림 7.21과 같습니다.

	date	open	high	low	close	volume
8546	2016-12-23	143000	144000	142000	144000	357914
8547	2016-12-26	143500	145000	143500	144500	206099
8548	2016-12-27	144500	145000	144000	144500	337224
8549	2016-12-28	142500	144000	141500	143500	377265
8550	2016-12-29	142500	146000	142000	146000	381283

그림 7.21 현대차(005380) 종목의 차트 데이터

차트 데이터는 일자(date), 시가(open), 고가(high), 저가(low), 종가(close), 거래량(volume)으로 구성됩니다. 이 데이터는 환경(environment) 모듈이 관리하며 매수금, 손익액 등을 계산할 때 사용합니다.

차트 데이터를 전처리하면 그림 7.22와 같이 이동평균 값들이 추가됩니다.

	close_ma5	volume_ma5	close_ma10	volume_ma10	close_ma20	volume_ma20	close_ma60	volume_ma60	close_ma120	volume_ma120
8546	143000.0	339163.8	142950.0	342871.7	139525.0	425157.25	136616.666667	460799.783333	135725.000000	497759.000000
8547	143500.0	321610.0	142950.0	325673.9	140000.0	422645.15	136766.666667	453939.233333	135787.500000	495151.650000
8548	143700.0	309152.6	142900.0	331003.0	140500.0	423895.55	136858.333333	447766.933333	135866.666667	494717.958333
8549	143700.0	322568.0	142900.0	342591.1	141025.0	383320.45	136916.666667	444894.583333	135979.166667	488708.891667
8550	144500.0	331957.0	143300.0	352145.7	141725.0	378466.50	137033.333333	446719.166667	136104.166667	488958.650000

그림 7.22 현대차(005380) 종목의 이동평균값 전처리

전처리 결과, 종가의 5일 평균(close_ma5), 거래량의 5일 평균(volume_ma5), 종가의 10일 평균(close_ma10), 거래량의 10일 평균(volume_ma10), 종가의 20일 평균(close_ma20), 거래량의 20일 평균(volume_ma20), 종가의 60일 평균(close_ma60), 거래량의 60일 평균(volume_ma60), 종가의 120일 평균(close_ma120), 거래량의 120일 평균(volume_ma120)이 계산됩니다.

전처리 후의 데이터에 그림 7.23과 같이 학습 데이터 준비를 위해 필요한 값들을 추가합니다.

	open_lastclose_ratio	high_close_ratio	low_close_ratio	close_lastclose_ratio	volume_lastvolume_ratio
8546	0.007042	0.000000	-0.013889	0.014085	0.070515
8547	-0.003472	0.003460	-0.006920	0.003472	-0.424166
8548	0.000000	0.003460	-0.003460	0.000000	0.636223
8549	-0.013841	0.003484	-0.013937	-0.006920	0.118737
8550	-0.006969	0.000000	-0.027397	0.017422	0.010650

	close_ma5_ratio	volume_ma5_ratio	close_ma10_ratio	volume_ma10_ratio	close_ma20_ratio	volume_ma20_ratio
8546	0.006993	0.055284	0.007345	0.043872	0.032073	-0.158161
8547	0.006969	-0.359165	0.010843	-0.367161	0.032143	-0.512359
8548	0.005567	0.090801	0.011197	0.018794	0.028470	-0.204464
8549	-0.001392	0.169567	0.004199	0.101211	0.017550	-0.015797
8550	0.010381	0.148592	0.018842	0.082742	0.030164	0.007442

	close_ma60_ratio	volume_ma60_ratio	close_ma120_ratio	volume_ma120_ratio
8546	0.054044	-0.223277	0.060969	-0.280949
8547	0.056544	-0.545977	0.064163	-0.583766
8548	0.055836	-0.246876	0.063543	-0.318351
8549	0.048083	-0.152013	0.055309	-0.228037
8550	0.065434	-0.146482	0.072708	-0.220214

그림 7.23 현대차(005380) 종목의 비율값 전처리

학습 데이터에는 전일 종가 대비 시가(open_lastclose_ratio), 종가 대비 고가(high_close_ratio), 종가 대비 저가(low_close_ratio), 전일 종가 대비 당일 종가(close_lastclose_ratio), 전일 거래량 대비 당일 거래량(volume_lastvolume_ratio), 5일 평균 종가 대비 당일 종가(close_ma5_ratio), 5일 평균 거래량 대비 당일 거래량(volume_ma5_ratio), 10일 평균 종가 대비 당일 종가(close_ma10_ratio), 10일 평균 거래량 대비 당일 거래량(volume_ma10_ratio), 20일 평균 종가 대비 당일 종가(close_ma20_ratio), 20일 평균 거래량 대비 당일 거래량(volume_ma20_ratio), 60일 평균 종가 대비 당일 종가(close_ma60_ratio), 60일 평균 거래량 대비 당일 거래량(volume_ma60_ratio), 120일 평균 종가 대비 당일 종가(close_ma120_ratio), 120일 평균 거래량 대비 당일 거래량(volume_ma120_ratio)이 계산되어 추가됩니다.

7.3.3 학습 파라미터 설정

현대차(005380) 를 대상으로 투자 시뮬레이션을 할 때에 설정한 학습 파라미터는 다음과 같습니다.

초기 자본금	10,000,000원
최소 투자 단위	1주
최대 투자 단위	1주
총 에포크 수	1,000회
초기 탐험률	0.5 (50%)
지연 보상 임계치	0.05 (5%)
학습 속도	0.0001
할인 요인	0% (미적용)

10,000,000원으로 주식투자 시뮬레이션을 합니다. 단순하게 현대차의 2016년 주가가 10만원이 넘어서 투자 단위를 1주로 정했습니다. 1,000회 시뮬레이션을 반복하며 학습을 진행하고 초기 탐험률을 50%로 정하고 에포크가 지나가면서 동일 비율로 탐험률을 줄여나갑니다. 최종 에포크에서는 탐험률이 0이 되게 합니다.

이익 또는 손실이 기준점 대비 5% 초과로 발생했을 때 학습을 수행하도록 지연 보상 임계치를 0.05로 정하고 0.0001의 학습 속도로 학습을 진행합니다.

탐험률에 따라 무작위 투자를 진행하기 때문에 동일한 실험 환경을 갖추더라도 실험 결과가 다를 수 있습니다.

7.3.4 에포크 10일 때의 결과

총 1,000번의 에포크 중에서 에포크 10에서의 학습 결과를 그림 7.24에서 보여줍니다. 2분기에 급락하는 구간이 있어서 부정적 학습을 많이 해서인지 정책 신경망이 매도 신호만을 보내고 있습니다. 학습 횟수가 아직 적기 때문에 에포크를 더 진행해야 정책 신경망이 의미 있는 신호를 줄 것으로 보입니다.

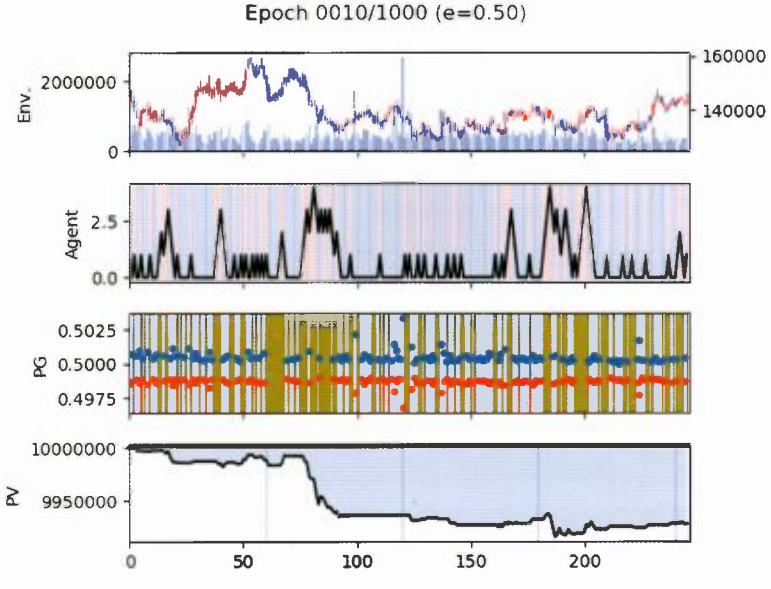


그림 7.24 현대차(005380) 종목의 강화학습 투자 시뮬레이션: 에포크 10/1000

특히 거래량이 솟아오른 직후에서 매도 신호를 강하게 주고 있습니다. 특징이 뚜렷한 지점에서 부정적 학습을 많이 해서 나타나는 현상으로 보입니다.

에포크 10에서의 로그는 다음과 같습니다.

```
[Epoch 0010/1000] Epsilon:0.4955 #Expl.:119/246 #Buy:66 #Sell:65 #Hold:115 #Stocks:1
PV:₩9,927,500 POS:0 NEG:4 Loss: 1.343468
```

49.55%로 무작위 투자를 수행했고. 246번의 행동 중에서 119번을 무작위로 수행했습니다. 이 과정에서 66번 매수하고 65번 매도했습니다. 결정한 매수 또는 매도 행동을 수행할 수 없어서 115번 관망했습니다. 최종 보유 주식 수는 1주이고 결과적으로 포트폴리오 가치는 9,927,500원으로 약간의 손실이 발생했습니다.

네 번의 부정적 학습을 수행했습니다. 이때의 학습 손실은 1.343468입니다. 이 손실은 앞으로 학습이 진행되면서 줄어들 것입니다.

7.3.5 에포크 200일 때의 결과

에포크 200에서의 학습 결과를 그림 7.25에서 보여줍니다. 이제 정책 신경망이 어느 정도 학습됐습니다.

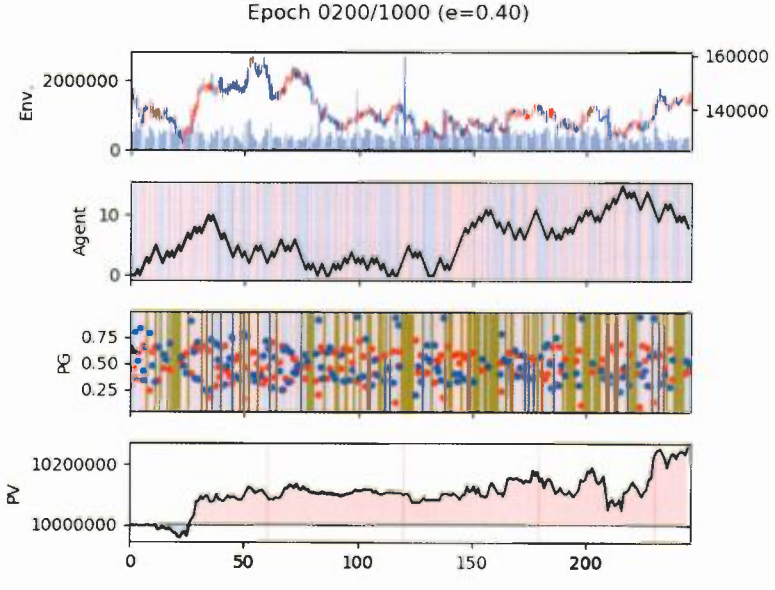


그림 7.25 현대차(005380) 종목의 강화학습 투자 시뮬레이션: 에포크 200/1000

1분기에서 단기간 가파르게 상승하는 구간에서 매수하여 수익을 내고 있고 2분기에서 단기간 가파르게 하락하는 구간에서 매도하여 방어하는 모습입니다.

이때의 로그는 다음과 같습니다.

```
[Epoch 0200/1000] Epsilon:0.4004 #Expl.:98/246 #Buy:123 #Sell:115 #Hold:8 #Stocks:8
PV:₩10,257,000 POS:4 NEG:0 Loss: 0.256396
```

탐험률은 약 40%입니다. 246회의 행동 중에서 98번을 무작위로 결정했습니다. 123번의 매수와 115번의 매도, 그리고 8번의 관망이 있었습니다. 최종적으로 8주를 보유하고 있습니다. 최종 포트폴리오 가치는 10,257,000원입니다.

에포크 동안 4번의 긍정적 학습이 수행됐습니다. 학습 손실은 에포크 10에 비해 크게 떨어진 0.256396입니다.

7.3.6 에포크 600일 때의 결과

그림 7.26은 에포크 600을 가시화한 결과입니다. 전반적으로 주식을 사들이면서 보유 주식이 점점 늘어가고 있습니다.

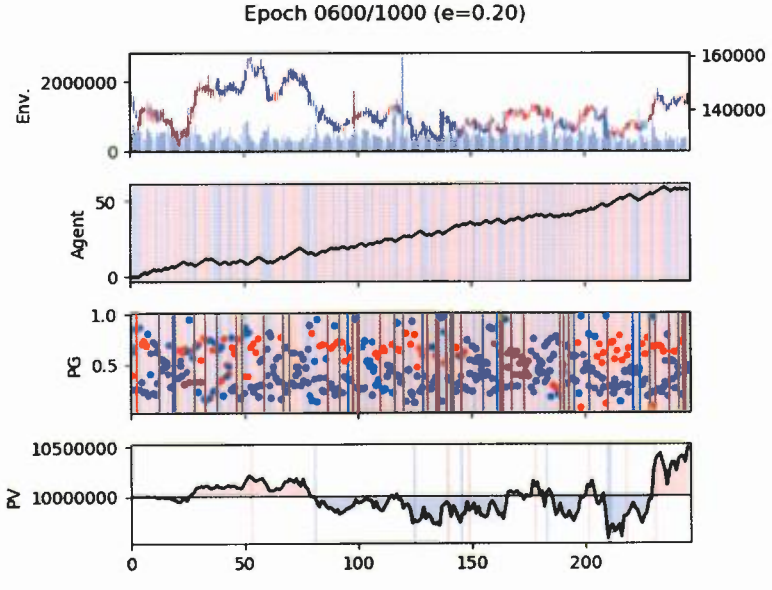


그림 7.26 현대차(005380) 종목의 강화학습 투자 시뮬레이션: 에포크 600/1000

정책 신경망이 2분기의 급락 구간에서 매도 신호를 짧게 보내고는 있으나 이 급락구간을 충분히 방어하지 못하는 모습입니다.

에포크 600의 로그는 다음과 같습니다.

```
[Epoch 0600/1000]  Epsilon:0.2002  #Expl.:49/246  #Buy:149  #Sell:92  #Hold:5  #Stocks:57
PV:₩10,483,500  POS:8  NEG:6  Loss:  0.697765
```

약 20%로 무작위투자를 진행하였습니다. 총 246회의 행동 중에서 49회를 무작위로 결정했습니다. 149번의 매수, 92번의 매도, 5번의 관망을 했습니다. 최종적으로 57주를 보유합니다. 최종 포트폴리오 가치는 10,483,500원으로 약 5%의 수익이 발생했습니다.

8번의 긍정적 학습과 6번의 부정적 학습이 수행되었습니다. 학습 손실은 0.697765입니다.

7.3.7 에포크 1000일 때의 결과

현대차 종목의 마지막 에포크의 결과를 그림 7.27에서 보여줍니다. 에포크 600의 결과와 유사한 결과를 보여줍니다.

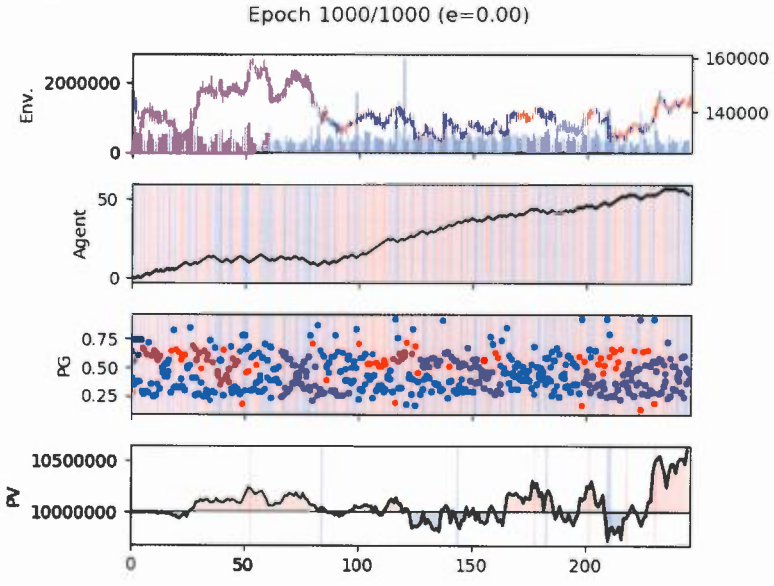


그림 7.27 현대차(005380) 종목의 강화학습 투자 시뮬레이션: 에포크 1000/1000

2분기 급락구간에서 매도 신호를 좀 더 길게 유지하면서 에포크 600인 경우보다는 좀 더 방어하고 있습니다.

마지막 에포크의 로그는 다음과 같습니다.

```
[Epoch 1000/1000]  Epsilon:0.0000  #Expl.:0/246  #Buy:148  #Sell:95  #Hold:3  #Stocks:53
PV:₩10,609,500  POS:5  NEG:5  Loss:  0.780274
```

탐험률은 0%로 정책 신경망의 신호로만 행동을 결정합니다. 148회 매수, 95회 매도, 3회 관망했습니다. 최종적으로 53주를 보유합니다. 최종 포트폴리오 가치는 10,609,500원입니다.

5번의 긍정적 학습과 5번의 부정적 학습이 있었습니다. 학습 손실은 0.780274입니다.

7.3.8 총평

주가와 거래량의 특징만으로 학습한 현대차 종목 투자의 결과는 그다지 만족스럽지 못합니다. 2분기의 급락 구간에서 조금 방어하긴 하지만 손실을 피하지 못했고 4분기의 급락 구간에서도 손실이 발생했습니다.

학습에 사용할 다양한 특징을 발굴할 필요가 있음을 잘 알려주는 시뮬레이션 결과입니다. 기관과 외국인의 순매수량을 고려할 수도 있고 PER, PBR 등의 다양한 지표들을 학습에 반영할 수도 있습니다.

7.4 투자 시뮬레이션 결과 4: LG화학(051910)

LG화학(005380)은 2016년 하락 추세, 2017년 상승 추세 종목입니다. 2016년 동안의 데이터를 학습하여 정책 신경망을 구축합니다. 여기서 학습시킨 정책 신경망을 활용하여, 8.2절에서는 2017년도 데이터를 활용해 투자 시뮬레이션을 해 봅니다.

7.4.1 종목의 개요

LG화학 종목의 2016년 차트는 그림 7.28과 같습니다. 차트는 주가를 보여주는 일봉차트와 거래량을 보여주는 막대 그래프로 구성됩니다. 차트의 Y축으로 왼쪽은 거래량을, 오른쪽은 주가를 보여줍니다.

2016년도 LG화학은 지속적으로 하락하고 있습니다. 이때 수익을 본 LG화학 투자자는 많지 않았을 것으로 보입니다. 이 상황에서는 강화학습에서도 원금을 잃지 않는다면 성공적인 학습일 것입니다.

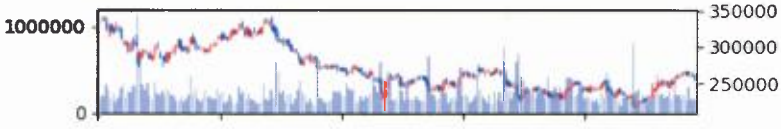


그림 7.28 LG화학(005380) 종목의 2016년 차트

그림 7.29와 같이 LG화학 종목의 2016년 주가의 최고가는 341,500원, 최저가는 219,500원, 최대낙폭(maximum draw-down)은 약 35.7%입니다. 엄청난 낙폭입니다.

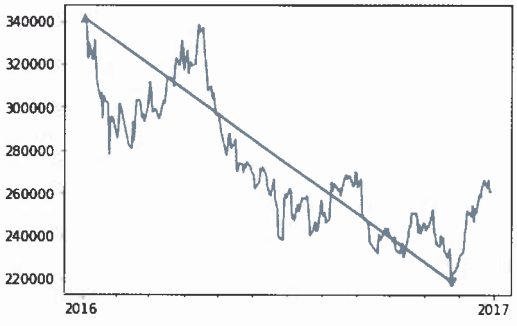


그림 7.29 LG화학(005380) 종목의 2016년 최고가, 최저가, MDD

여기서 빨간색 삼각형은 최고가 지점을, 파란색 역삼각형은 최저가 지점을, 파란색 실선은 MDD의 시작점과 끝지점을 연결한 선입니다.

7.4.2 주식 데이터 전처리

LG화학(005380)의 차트 데이터 뒷부분(tail)은 그림 7.30과 같습니다.

	date	open	high	low	close	volume
3881	2016-12-23	266500	266500	263500	266500	216506
3882	2016-12-26	266500	267000	261000	263000	145603
3883	2016-12-27	264500	266500	261500	266500	299387
3884	2016-12-28	263000	265000	260000	261000	166227
3885	2016-12-29	263000	263000	255500	261000	165816

그림 7.30 LG화학(005380) 종목의 차트 데이터

차트 데이터는 일자(date), 시가(open), 고가(high), 저가(low), 종가(close), 거래량(volume)으로 구성됩니다. 이 데이터는 환경(environment) 모듈이 관리하며 매수금, 손익액 등을 계산할 때 사용합니다.

차트 데이터를 전처리하면 그림 7.31과 같이 이동평균 값들이 추가됩니다.

	close_ma5	volume_ma5	close_ma10	volume_ma10	close_ma20	volume_ma20	close_ma60	volume_ma60	close_ma120	volume_ma120
3881	262400.0	233022.8	256700.0	219103.8	246375.0	229186.25	241883.333333	250485.383333	247620.833333	253471.050000
3882	263100.0	191625.8	258050.0	211128.3	248300.0	222440.60	242241.666667	248874.216667	247625.000000	253250.450000
3883	264700.0	209982.8	259250.0	222611.4	250325.0	229995.80	242725.000000	249661.483333	247666.666667	254427.916667
3884	264200.0	199802.6	260650.0	219427.4	252050.0	227814.45	243075.000000	249560.400000	247770.833333	252237.066667
3885	263600.0	198707.8	261450.0	210516.6	253625.0	226976.10	243525.000000	245150.650000	247866.666667	251508.483333

그림 7.31 LG화학(005380)종목의 이동평균값 전처리

전처리 결과, 종가의 5일 평균(close_ma5), 거래량의 5일 평균(volume_ma5), 종가의 10일 평균(close_ma10), 거래량의 10일 평균(volume_ma10), 종가의 20일 평균(close_ma20), 거래량의 20일 평균(volume_ma20), 종가의 60일 평균(close_ma60), 거래량의 60일 평균(volume_ma60), 종가의 120일 평균(close_ma120), 거래량의 120일 평균(volume_ma120)이 계산됩니다.

전처리 후의 데이터에 그림 7.32와 같이 학습 데이터 준비를 위해 필요한 값들을 추가합니다.

	open_lastclose_ratio	high_close_ratio	low_close_ratio	close_lastclose_ratio	volume_lastvolume_ratio
3881	0.009470	0.000000	-0.011257	0.009470	0.263973
3882	0.000000	0.015209	-0.007605	-0.013133	-0.327487
3883	0.005703	0.000000	-0.018762	0.013308	1.056187
3884	-0.013133	0.015326	-0.003831	-0.020638	-0.444775
3885	0.007663	0.007663	-0.021073	0.000000	-0.002473

	close_ma5_ratio	volume_ma5_ratio	close_ma10_ratio	volume_ma10_ratio	close_ma20_ratio	volume_ma20_ratio
3881	0.015625	-0.070881	0.038177	-0.011856	0.081684	-0.055327
3882	-0.000380	-0.240170	0.019182	-0.310358	0.059203	-0.345430
3883	0.006800	0.425769	0.027965	0.344886	0.064616	0.301706
3884	-0.012112	-0.168044	0.001343	-0.242451	0.035509	-0.270340
3885	-0.009863	-0.165528	-0.001721	-0.212338	0.029078	-0.269456

	close_ma60_ratio	volume_ma60_ratio	close_ma120_ratio	volume_ma120_ratio
3881	0.101771	-0.135654	0.076242	-0.145835
3882	0.085693	-0.414953	0.062090	-0.425063
3883	0.097950	0.199172	0.076043	0.176707
3884	0.073743	-0.333921	0.053393	-0.340989
3885	0.071759	-0.323616	0.052985	-0.340714

그림 7.32 LG화학(005380) 종목의 비율값 전처리

학습 데이터에는 전일 종가 대비 시가(open_lastclose_ratio), 종가 대비 고가(high_close_ratio), 종가 대비 저가(low_close_ratio), 전일 종가 대비 당일 종가(close_lastclose_ratio), 전일 거래량 대비 당일 거래량(volume_lastvolume_ratio), 5일 평균 종가 대비 당일 종가(close_ma5_ratio), 5일 평균 거래량 대비 당일 거래량(volume_ma5_ratio), 10일 평균 종가 대비 당일 종가(close_ma10_ratio), 10일 평균 거래량 대비 당일 거래량(volume_ma10_ratio), 20일 평균 종가 대비 당일 종가(close_ma20_ratio), 20일 평균 거래량 대비 당일 거래량(volume_ma20_ratio), 60일 평균 종가 대비 당일 종가(close_ma60_ratio), 60일 평균 거래량 대비 당일 거래량(volume_ma60_ratio), 120일 평균 종가 대비 당일 종가(close_ma120_ratio), 120일 평균 거래량 대비 당일 거래량(volume_ma120_ratio)이 계산되어 추가됩니다.

7.4.3 학습 파라미터 설정

LG화학(005380)을 대상으로 투자 시뮬레이션을 할 때에 설정한 학습 파라미터는 다음과 같습니다.

초기 자본금	10,000,000원
최소 투자 단위	1주
최대 투자 단위	1주
총 에포크 수	1,000회
초기 탐험률	0.5 (50%)
지연 보상 임계치	0.05 (5%)
학습 속도	0.0001
할인 요인	0% (미적용)

10,000,000원으로 주식투자 시뮬레이션을 합니다. 2016년 LG화학은 20~30만 원대의 주가를 가지므로 투자 단위를 1주로 정했습니다. 1,000회 시뮬레이션을 반복하며 학습을 진행하고 초기 탐험률을 50%로 정하고 에포크가 지나가면서 동일 비율로 탐험률을 줄여나갑니다. 최종 에포크에서는 탐험률이 0이 되게 합니다.

이익 또는 손실이 기준점 대비 5% 초과로 발생했을 때 학습을 수행하도록 지연 보상 임계치를 0.05로 정하고 0.0001의 학습 속도로 학습을 진행합니다.

탐험률에 따라 무작위 투자를 진행하기 때문에 동일한 실험 환경을 갖추더라도 실험 결과가 다를 수 있습니다.

7.4.4 에포크 10일 때의 결과

에포크 10의 결과를 그림 7.33에서 보여줍니다. 아직 많은 학습이 진행된 상태가 아니지만 정책 신경망이 3분기 후반과 4분기 초반에서 매수 신호를 주는 방향으로 학습이 되었습니다.

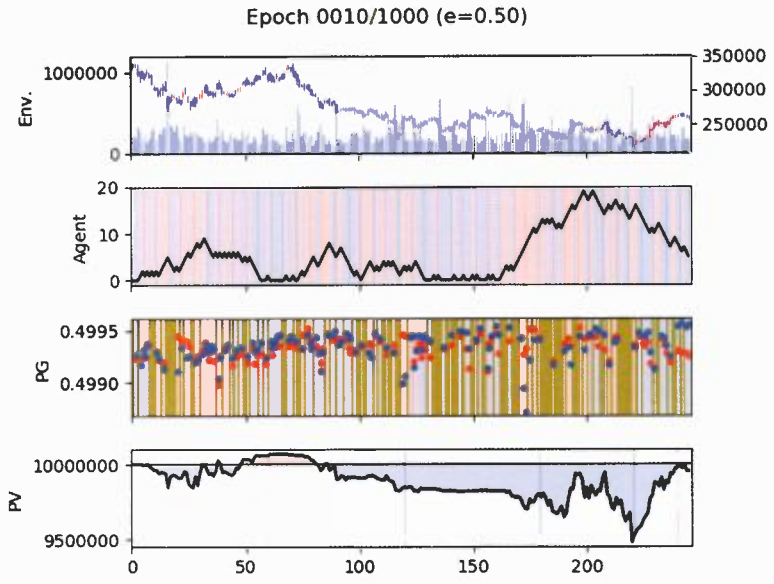


그림 7.33 LG화학(005380) 종목의 강화학습 투자 시뮬레이션: 에포크 10/1000

2016년의 LG화학이 하락 추세의 종목이라 수익은 없으나 원금 손실이 크지는 않아 보입니다.

다음은 에포크 10에서의 로그입니다.

```
[Epoch 0010/1000] Epsilon:0.4955 #Expl.:122/246 #Buy:110 #Sell:105 #Hold:31 #Stocks:5
PV:₩9,952,000 POS:2 NEG:3 Loss: 0.916424
```

49.55%로 무작위 투자를 했습니다. 총 246번의 투자 행동 중에서 122회를 무작위로 수행했습니다. 110번 매수, 105번 매도, 21번 관망했습니다. 최종 보유 주식 수는 5주이고 최종 포트폴리오 가치는 약 4.8% 하락한 9,952,000원입니다.

2번의 긍정적 학습과 3번의 부정적 학습을 수행했습니다. 학습 손실은 0.916424입니다.

7.4.5 에포크 200일 때의 결과

에포크 200의 결과를 그림 7.34에서 보여줍니다. 하락 추세가 뚜렷하여 최종적으로 여전히 손실을 보고 있습니다.

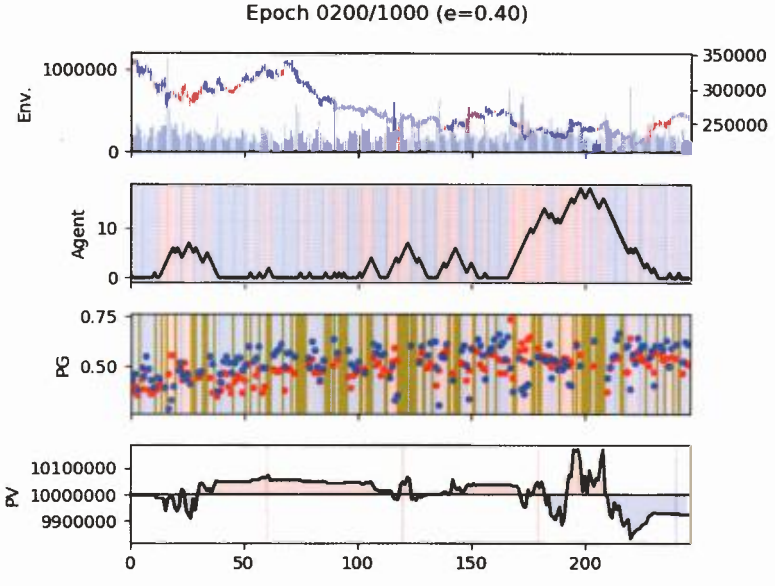


그림 7.34 LG화학(005380) 종목의 강화학습 투자 시뮬레이션: 에포크 200/1000

3분기 후반에 횡보하는 구간에서 매집하고 4분기부터는 매도하고 있습니다.

에포크 200에서의 로그는 다음과 같습니다.

```
[Epoch 0200/1000] Epsilon:0.4004 #Expl.:89/246 #Buy:84 #Sell:84 #Hold:78 #Stocks:0
PV:₩9,924,500 POS:3 NEG:1 Loss: 0.561698
```

약 40%로 무작위 투자를 수행합니다. 총 246번의 행동 중에서 89번을 무작위로 수행했습니다. 84번의 매수, 84번의 매도, 78번의 관망을 수행했습니다. 최종 보유 주식 수는 없으며 포트폴리오 가치는 9,924,500원으로 약간의 손실이 발생했습니다.

3번의 긍정적 학습과 한번의 부정적 학습을 수행했습니다. 학습 손실은 0.561698로 에포크 10에 비해 줄어들었습니다.

7.4.6 에포크 600일 때의 결과

에포크 600에서의 결과를 그림 7.35에서 보여줍니다. 3분기 후반에서 좀 더 뚜렷하게 매수세를 유지합니다.

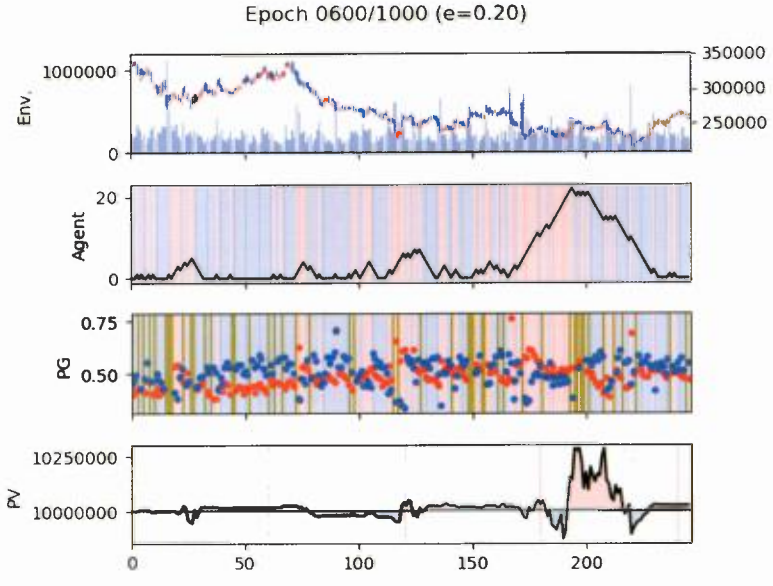


그림 7.35 LG화학(005380) 종목의 강화학습 투자 시뮬레이션: 에포크 600/1000

정책 신경망이 1분기~2분기의 하락하는 구간에서 대부분 매도 신호를 보내고 있습니다. 3분기 후반에서 매수세를 가지지만 4분기에서 다시 매도세로 돌아섭니다.

이때의 로그는 다음과 같습니다.

```
[Epoch 0600/1000] Epsilon:0.2002 #Expl.:48/246 #Buy:89 #Sell:89 #Hold:68 #Stocks:0
PV:₩10,023,500 POS:4 NEG:0 Loss: 0.286769
```

약 20%로 무작위 투자를 진행합니다. 246번의 행동 중에서 48번을 무작위 투자를 수행했습니다. 최종 보유 주식 수는 없고 89번 매수, 89번 매도, 68번 관망했습니다. 포트폴리오 가치는 10,023,500원으로 거의 보합입니다.

4번의 긍정적 학습을 진행했습니다. 학습 손실은 0.286769입니다.

7.4.7 에포크 1000일 때의 결과

그림 7.36은 마지막 에포크인 에포크 1000의 결과를 보여줍니다. 3분기 후반에서 더 일정하게 매수세를 유지합니다. 정책 신경망의 출력을 봐도 매수 신호를 확실히 주고 있습니다.

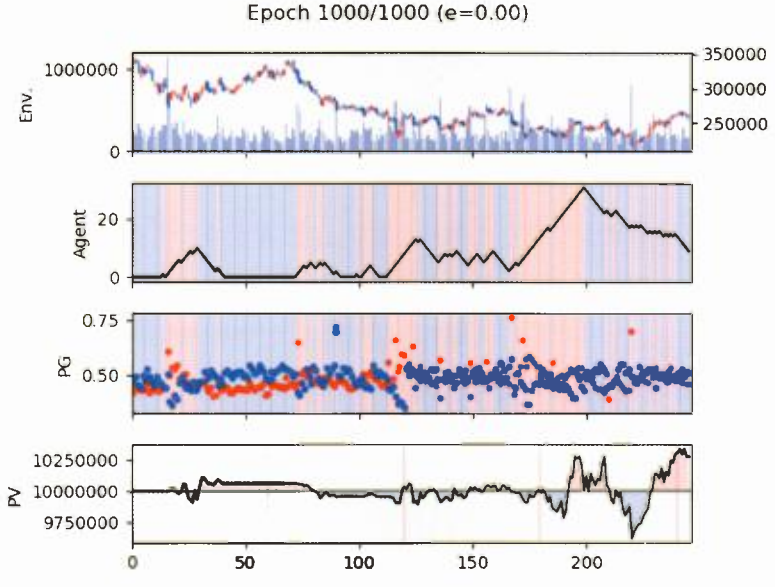


그림 7.36 LG화학(005380) 종목의 강화학습 투자 시뮬레이션: 에포크 1000/1000

3분기 후반에서 보이는 특징이 매수해야 하는 방향으로 학습되었습니다. 우선 주가가 횡보하고 있고 거래량이 지속적으로 많습니다.

에포크 1000의 로그는 다음과 같습니다.

```
[Epoch 1000/1000] Epsilon:0.0000 #Expl.:0/246 #Buy:100 #Sell:91 #Hold:55 #Stocks:9
PV:₩10,277,500 POS:4 NEG:0 Loss: 0.287880
```

정책 신경망의 출력에만 의존하여 투자 행동을 수행하였습니다. 100번의 매수를, 91번의 매도를, 55번의 관망을 수행했습니다. 최종 보유 주식 수는 9주입니다. 포트폴리오 가치는 10,277,500원으로 약 2.8%의 수익이 발생했습니다.

추가적으로 4번의 긍정적 학습이 있었습니다. 학습 손실은 0.287880입니다.

7.4.8 총평

2016년의 LG화학 종목은 워낙 하락 추세가 뚜렷했기 때문에 투자하기 좋은 종목은 아니었습니다. 상승 구간에서는 기울기가 완만하였고 하락 구간에서는 가파르게 하락했기 때문에 투자자들의 근심이 많았을 것으로 생각됩니다.

RLTrader는 1분기 상승 초반에 잠깐 매수 신호를 주었고 2분기 후반에서 또 잠깐 매수 신호를 주었고 3분기 후반에서 강한 매수 신호를 주고 있습니다. 2017년도의 LG화학의 주가가 크게 상승했는데, RLTrader가 저점에서 잘 매수한 것입니다. 이렇게 학습한 LG화학 투자 정책 신경망 모델을 2017년도에 적용한 결과는 8장에서 다룹니다.

7.5 투자 시뮬레이션 결과 5: NAVER(035420)

NAVER(005380)는 2016년 상승 추세, 2017년 박스권 종목입니다. 2016년 동안의 데이터를 학습하여 정책 신경망을 구축합니다. 여기서 학습시킨 정책 신경망을 활용하여, 8.2절에서는 2017년도 데이터를 활용해 투자 시뮬레이션을 해 봅니다.

7.5.1 종목의 개요

NAVER 종목의 2016년 차트는 그림 7.37과 같습니다. 차트는 주가를 보여주는 일봉차트와 거래량을 보여주는 막대 그래프로 구성됩니다. 차트의 Y축으로 왼쪽은 거래량을, 오른쪽은 주가를 보여줍니다.

2016년도에 NAVER의 주가는 중간중간에 조정 구간이 있지만 전체적으로 상승 추세를 보입니다.

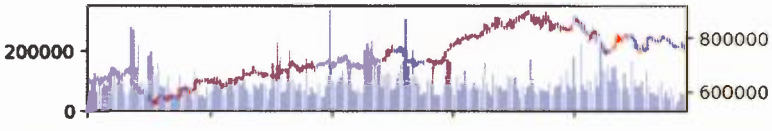


그림 7.37 NAVER(035420) 종목의 2016년 차트

그림 7.38에서 2016년 NAVER 종목의 최저가, 최고가, MDD를 표시합니다.

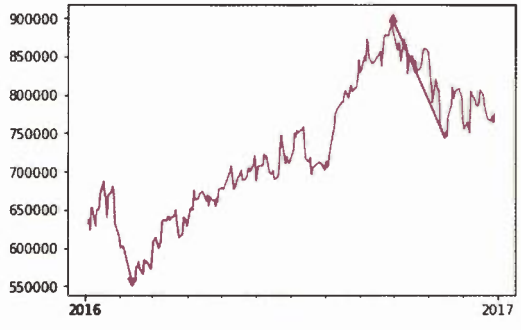


그림 7.38 NAVER(035420) 종목의 2016년 최고가, 최저가, MDD

최고가는 900,000원, 최저가는 556,000원, MDD는 약 17.3%입니다. 최고가 90만 원을 찍고 저항을 받아 17% 이상이 하락하면서 MDD를 형성했습니다.

7.5.2 주식 데이터 전처리

현대차(005380)의 차트 데이터 뒷부분(tail)은 그림 7.39와 같습니다.

	date	open	high	low	close	volume
3510	2016-12-23	771000	777000	762000	768000	53375
3511	2016-12-26	766000	773000	765000	765000	20665
3512	2016-12-27	766000	773000	763000	772000	31962
3513	2016-12-28	785000	785000	763000	763000	57737
3514	2016-12-29	758000	777000	754000	775000	56324

그림 7.39 NAVER(035420) 종목의 차트 데이터

차트 데이터는 일자(date), 시가(open), 고가(high), 저가(low), 종가(close), 거래량(volume)으로 구성됩니다. 이 데이터는 환경(environment) 모듈이 관리하며 매수금, 손익액 등을 계산할 때 사용합니다.

차트 데이터를 전처리하면 그림 7.40과 같이 이동평균 값들이 추가됩니다.

	close_ma5	volume_ma5	close_ma10	volume_ma10	close_ma20	volume_ma20	close_ma60	volume_ma60	close_ma120	volume_ma120
3510	781400.0	57929.8	786200.0	64791.9	783100.0	79198.35	809550.000000	95291.833333	797000.000000	97171.350000
3511	774400.0	52505.0	783500.0	60806.3	780950.0	76424.00	807583.333333	92761.033333	797316.666667	96361.583333
3512	771400.0	45774.6	782100.0	57922.8	779500.0	75191.05	806150.000000	91473.733333	797500.000000	95569.050000
3513	768200.0	48483.2	779900.0	58419.8	777750.0	69648.55	804416.666667	91327.300000	797650.000000	95029.808333
3514	768600.0	44012.6	778800.0	55655.8	778300.0	67271.65	802983.333333	91351.983333	797833.333333	94928.208333

그림 7.40 NAVER(035420) 종목의 이동평균값 전처리

전처리 결과, 종가의 5일 평균(close_ma5), 거래량의 5일 평균(volume_ma5), 종가의 10일 평균(close_ma10), 거래량의 10일 평균(volume_ma10), 종가의 20일 평균(close_ma20), 거래량의 20일 평균(volume_ma20), 종가의 60일 평균(close_ma60), 거래량의 60일 평균(volume_ma60), 종가의 120일 평균(close_ma120), 거래량의 120일 평균(volume_ma120)이 계산됩니다.

전처리 후의 데이터에 그림 7.41과 같이 학습 데이터 준비를 위해 필요한 값들을 추가합니다.

	close_ma5	volume_ma5	close_ma10	volume_ma10	close_ma20	volume_ma20	close_ma60	volume_ma60	close_ma120	volume_ma120
3510	781400.0	57929.8	786200.0	64791.9	783100.0	79198.35	809550.000000	95291.833333	797000.000000	97171.350000
3511	774400.0	52505.0	783500.0	60806.3	780950.0	76424.00	807583.333333	92761.033333	797316.666667	96361.583333
3512	771400.0	45774.6	782100.0	57922.8	779500.0	75191.05	806150.000000	91473.733333	797500.000000	95569.050000
3513	768200.0	48483.2	779900.0	58419.8	777750.0	69648.55	804416.666667	91327.300000	797650.000000	95029.808333
3514	768600.0	44012.6	778800.0	55655.8	778300.0	67271.65	802983.333333	91351.983333	797833.333333	94928.208333

	close_ma5_ratio	volume_ma5_ratio	close_ma10_ratio	volume_ma10_ratio	close_ma20_ratio	volume_ma20_ratio
3510	-0.017149	-0.078626	-0.023149	-0.176209	-0.019282	-0.326059
3511	-0.012138	-0.606418	-0.023612	-0.660150	-0.020424	-0.729601
3512	0.000778	-0.301753	-0.012914	-0.448197	-0.009622	-0.574923
3513	-0.006769	0.190866	-0.021669	-0.011688	-0.018965	-0.171024
3514	0.008327	0.279724	-0.004879	0.012006	-0.004240	-0.162738

	close_ma60_ratio	volume_ma60_ratio	close_ma120_ratio	volume_ma120_ratio
3510	-0.051325	-0.439879	-0.036386	-0.450713
3511	-0.052729	-0.777223	-0.040532	-0.785547
3512	-0.042362	-0.650588	-0.031975	-0.665561
3513	-0.051487	-0.367801	-0.043440	-0.392433
3514	-0.034849	-0.383440	-0.028619	-0.406667

그림 7.41 NAVER(035420) 종목의 비율값 전처리

학습 데이터에는 전일 종가 대비 시가(open_lastclose_ratio), 종가 대비 고가(high_close_ratio), 종가 대비 저가(low_close_ratio), 전일 종가 대비 당일 종가(close_lastclose_ratio), 전일 거래량 대비 당일 거래량(volume_lastvolume_ratio), 5일 평균 종가 대비 당일 종가(close_ma5_ratio), 5일 평균 거래량 대비 당일 거래량(volume_ma5_ratio), 10일 평균 종가 대비 당일 종가(close_ma10_ratio), 10일 평균 거래량 대비 당일 거래량(volume_ma10_ratio), 20일 평균 종가 대비 당일 종가(close_ma20_ratio), 20일 평균 거래량 대비 당일 거래량(volume_ma20_ratio), 60일 평균 종가 대비 당일 종가(close_ma60_ratio), 60일 평균 거래량 대비 당일 거래량(volume_ma60_ratio), 120일 평균 종가 대비 당일 종가(close_ma120_ratio), 120일 평균 거래량 대비 당일 거래량(volume_ma120_ratio)이 계산되어 추가됩니다.

7.5.3 학습 파라미터 설정

NAVER(035420)를 대상으로 투자 시뮬레이션을 할 때에 설정한 학습 파라미터는 다음과 같습니다.

초기 자본금	10,000,000원
최소 투자 단위	1주
최대 투자 단위	1주
총 에포크 수	1,000회
초기 탐험률	0.5 (50%)
지연 보상 임계치	0.05 (5%)
학습 속도	0.0001
할인 요인	0% (미적용)

10,000,000원으로 주식투자 시뮬레이션을 합니다. 2016년 NAVER는 50~80만 원 대의 주가를 가지므로 투자 단위를 1주로 정했습니다. 1,000회 시뮬레이션을 반복하며 학습을 진행하고 초기 탐험률을 50%로 정하고 에포크가 지나가면서 동일 비율로 탐험률을 줄여나갑니다. 최종 에포크에서는 탐험률이 0이 되게 합니다.

이익 또는 손실이 기준점 대비 5% 초과로 발생했을 때 학습을 수행하도록 지연 보상 임계치를 0.05로 정하고 0.0001의 학습 속도로 학습을 진행합니다.

탐험률에 따라 무작위 투자를 진행하기 때문에 동일한 실험 환경을 갖추더라도 실험 결과가 다를 수 있습니다.

7.5.4 에포크 10일 때의 결과

그림 7.42는 에포크 10의 투자 시뮬레이션 결과를 보여줍니다. 정책 신경망이 아직 의미 있는 출력을 내보내고 있지 않습니다.

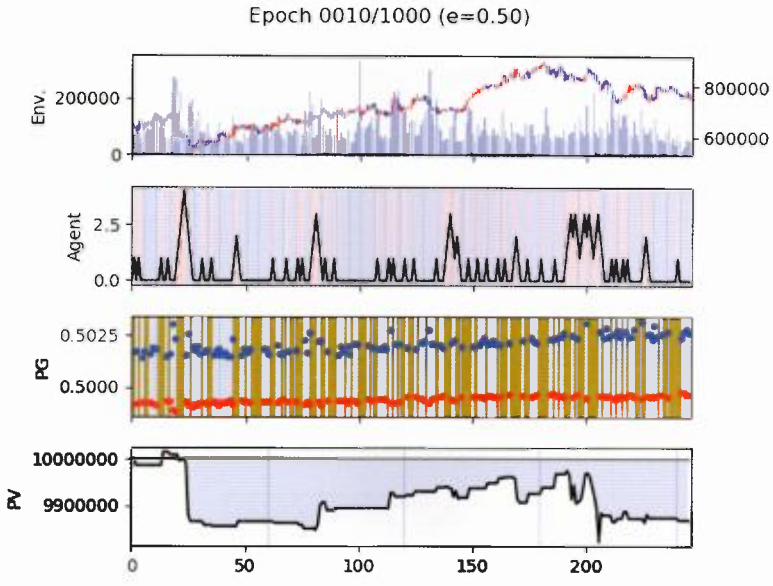


그림 7.42 NAVER(035420) 종목의 강화학습 투자 시뮬레이션: 에포크 10/1000

정책 신경망이 매도 신호만을 보내는 상황에서 무작위 투자에 의존하여 학습 데이터를 만들어 냅니다.

이때의 로그는 다음과 같습니다.

```
[Epoch 0010/1000]  Epsilon:0.4955  #Expl.:125/246  #Buy:57  #Sell:57  #Hold:132  #Stocks:0
PV:₩9,870,000  POS:0  NEG:4  Loss:  1.315884
```

약 50%의 확률로 무작위 투자를 수행합니다. 246회의 투자 행동 중에서 125회를 무작위로 정했습니다. 57회의 매수, 57회의 매도, 132회의 관망을 했습니다. 최종 보유 주식은 없고 최종적으로 포트폴리오 가치가 9,870,000원으로 13% 가량의 손실이 발생했습니다.

이 에포크에서는 4회의 부정적 학습을 진행했습니다. 학습 손실은 1.315884입니다.

7.5.5 에포크 200일 때의 결과

에포크 200의 결과를 그림 7.43에서 보여줍니다. 정책 신경망이 이제 의미 있는 출력을 보내고 있습니다. 1분기 하락구간에서 매도 신호를 보내며 잘 방어하였고 2~3분기 상승 구간에서는 주식을 매수하면서 수익을 내고 있습니다. 4분기 고점에서는 보유 주식을 일부 청산하고 있습니다.

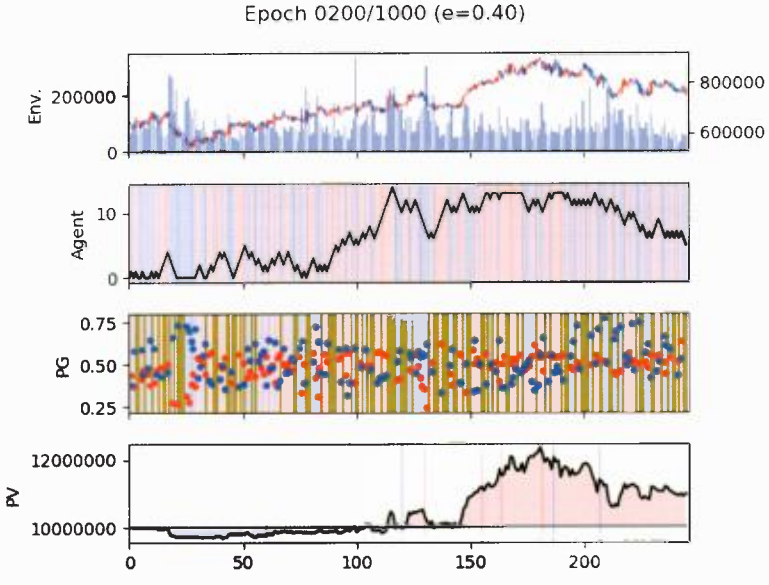


그림 7.43 NAVER(035420) 종목의 강화학습 투자 시뮬레이션: 에포크 200/1000

약 40%의 무작위 투자를 하고 있는데, 이미 정책 신경망이 꽤 적합화된 것으로 보입니다.

이 에포크의 로그는 다음과 같습니다.

```
[Epoch 0200/1000] Epsilon:0.4004 #Expl.:104/246 #Buy:111 #Sell:106 #Hold:29 #Stocks:5
PV:₩10,962,000 POS:4 NEG:4 Loss: 0.792779
```

약 40%로 무작위 투자를 하였습니다. 총 246회 투자 행동 중에서 104회를 무작위로 결정했습니다. 111회 매수, 106회 매도, 29회 관망하였습니다. 최종 보유 주식 수는 5주입니다. 최종 포트폴리오 가치가 10,962,000원으로 약 9.6%의 수익이 발생했습니다.

이 에포크에서는 4번의 긍정적 학습과 4번의 부정적 학습이 있었습니다. 손실은 0.792779로 줄었습니다.

7.5.6 에포크 600일 때의 결과

에포크 600의 결과를 그림 7.44에서 보여줍니다. 전반적으로 투자를 잘 수행했으나 4분기에서 망가지는 모습입니다.

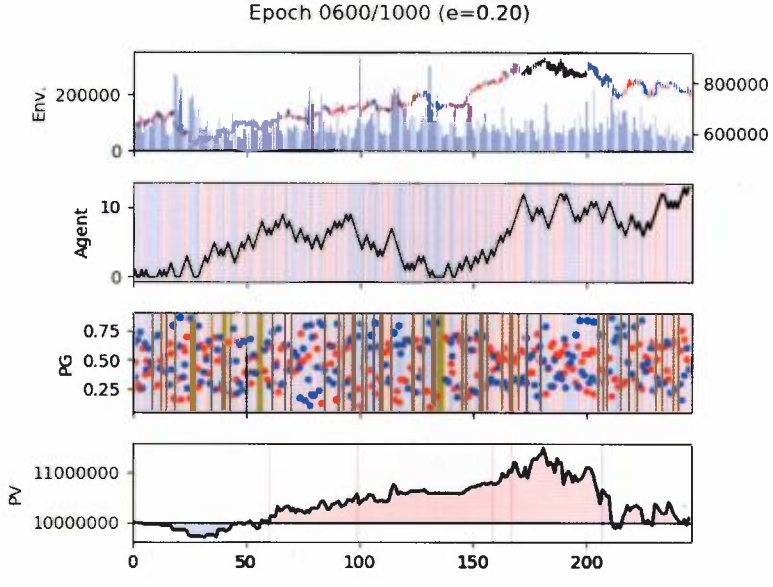


그림 7.44 NAVER(035420) 종목의 강화학습 투자 시뮬레이션: 에포크 600/1000

4분기의 하락 구간에서 매수, 매도하면서 갈팡질팡함에 따라 포트폴리오 가치가 크게 줄었습니다.

이 에포크의 로그는 다음과 같습니다.

```
[Epoch 0600/1000] Epsilon:0.2002 #Expl.:57/246 #Buy:121 #Sell:108 #Hold:17 #Stocks:13
PV:₩10,096,000 POS:4 NEG:1 Loss: 0.475351
```

약 20%의 무작위 투자를 진행했습니다. 총 246회의 행동 중에서 57번을 무작위로 결정했습니다. 121회 매수, 108회 매도, 17회 관망하였습니다. 최종적으로 13주를 보유합니다. 포트폴리오 가치가 10,096,000원으로 초기 자본금과 거의 보합입니다.

이 에포크에서는 4번의 긍정적 학습과 한 번의 부정적 학습이 수행되었습니다. 학습 손실은 0475351로 에포크 200에 비해 줄었습니다.

7.5.7 에포크 1000일 때의 결과

그림 7.45는 NAVER 종목의 2016년 투자 시뮬레이션에서 마지막 에포크 결과를 보여줍니다. 정책 신경망이 4분기 초의 하락 구간에서 매도 신호를 보내면서 에포크 600일 때보다는 덜 망가지고 있습니다.

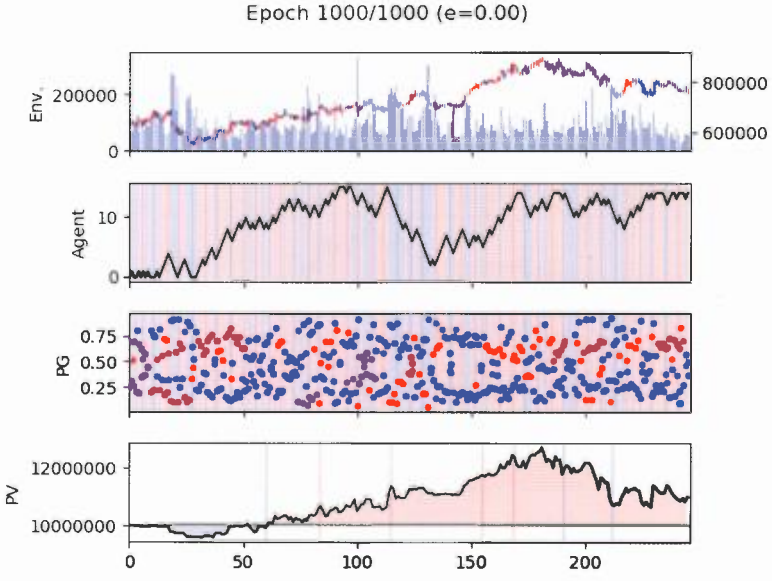


그림 7.45 NAVER(035420) 종목의 강화학습 투자 시뮬레이션: 에포크 1000/1000

에포크 1000의 로그는 다음과 같습니다.

```
[Epoch 1000/1000]  Epsilon:0.0000  #Expl.:0/246  #Buy:120  #Sell:106  #Hold:20  #Stocks:14
PV:₩11,007,000  POS:4  NEG:3  Loss:  0.680098
```

탐험률은 0%입니다. 120번 매수를, 106번 매도를, 20번 관망을 수행했습니다. 최종 보유 주식 수는 14주이고 최종 포트폴리오 가치는 11,007,000원으로 약 10%의 수익률을 보입니다.

추가적으로 4번의 긍정적 학습과 3번의 부정적 학습을 수행했습니다. 학습 손실은 0.680098입니다

7.5.8 총평

2016년도 NAVER 종목은 상승 추세가 뚜렷합니다. RLTrader는 지속적으로 매수신호를 보내다가 3분기 초의 횡보 구간에서 매도세로 잠깐 돌아섭니다. 그리고 다시 매수하다가 4분기에 고점을 찍고 하락구간에서 잠깐식 매도하며 방어합니다.

사실 더 장기적인 관점에서 학습을 했으면 2분기 후반과 3분기 초반에서 매도하지않고 보유했을 수 있을 것입니다. 3분기 후반에서 크게 상승했기 때문에 그 편이 좋았을 것입니다.

장기적인 관점에서 학습하기 위해서 해 볼 수 있는 옵션은 지연 보상 기준을 높이는 것입니다. 여기서는 5% 이상의 수익과 손실이 발생했을 때 학습을 수행하는데 이 기준을 더 높이면 학습을 더 장기적인 관점에서 할 수 있습니다. 이때 과거의 특징들과 수행했던 행동들을 저장해 두는 메모리 크기(정책학습기 모듈에서 max_memory)를 더 크게 정하는 것이 좋을 것입니다.

7.6 투자 시뮬레이션 결과 6: KT(030200)

KT(030200)는 2016년 상승 추세, 2017년 박스권 종목입니다. 2016년 동안의 데이터를 학습하여 정책 신경망을 구축합니다. 여기서 학습시킨 정책 신경망을 활용하여, 8.2절에서는 2017년도 데이터를 활용해 투자 시뮬레이션을 해 봅니다.

7.6.1 종목의 개요

그림 7.46은 KT 종목의 2016년 차트를 보여줍니다. 차트는 주가를 보여주는 일봉차트와 거래량을 보여주는 막대 그래프로 구성됩니다. 차트의 Y축으로 왼쪽은 거래량을, 오른쪽은 주가를 보여줍니다.

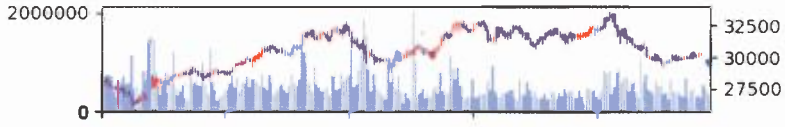


그림 7.46 KT(030200) 종목의 2016년 차트

2016년도 KT는 1분기와 2분기에서 큰폭의 주가 상승을 보이나 2분기 후반에서 조정받고 32,000원 부근에서 횡보하는 모습입니다.

그림 7.47은 KT의 2016년 최고가, 최저가, MDD를 보여줍니다.

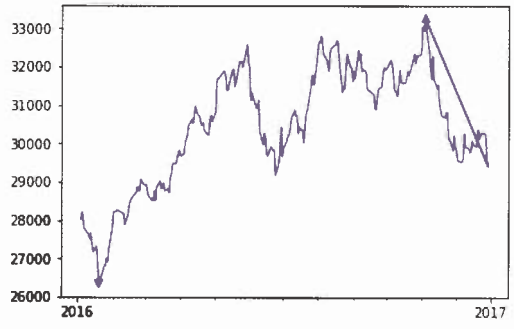


그림 7.47 KT(030200) 종목의 2016년 최고가, 최저가, MDD

최고가는 33,250원, 최저가는 26,350원, MDD는 약 11.6%입니다. 4분기에서 최고가 33,250원을 찍고 저항을 받아 11%이상이 하락하면서 MDD를 형성했습니다.

7.6.2 주식 데이터 전처리

KT(030200)의 차트 데이터 뒷부분(tail)은 그림 7.48과 같습니다.

	date	open	high	low	close	volume
4450	2016-12-23	30050	30400	30000	30250	346657
4451	2016-12-26	30250	30350	30150	30250	289098
4452	2016-12-27	30200	30300	30150	30200	544322
4453	2016-12-28	29800	29850	29600	29600	453869
4454	2016-12-29	29750	29750	29400	29400	372667

그림 7.48 KT(030200) 종목의 차트 데이터

차트 데이터는 일자(date), 시가(open), 고가(high), 저가(low), 종가(close), 거래량(volume)으로 구성됩니다. 이 데이터는 환경(environment) 모듈이 관리하며 매수금, 손익액 등을 계산할 때 사용합니다.

차트 데이터를 전처리하면 그림 7.49와 같이 이동평균 값들이 추가됩니다.

	close_ma5	volume_ma5	close_ma10	volume_ma10	close_ma20	volume_ma20	close_ma60	volume_ma60	close_ma120	volume_ma120
4450	30150.0	358412.0	30015.0	391770.8	29907.5	420024.20	31119.166667	468768.800000	31330.000000	462325.825000
4451	30220.0	361368.0	30055.0	381834.0	29930.0	413374.80	31091.666667	466701.016667	31331.666667	463308.941667
4452	30190.0	386109.6	30100.0	394448.5	29930.0	422636.75	31059.166667	469728.316667	31331.666667	464591.975000
4453	30090.0	380909.2	30080.0	406333.3	29917.5	413899.90	31018.333333	473163.783333	31326.666667	466140.075000
4454	29940.0	401322.6	30015.0	412766.0	29895.0	406572.70	30978.333333	473104.333333	31318.333333	467392.675000

그림 7.49 KT(030200) 종목의 이동평균값 전처리

전처리 결과, 종가의 5일 평균(close_ma5), 거래량의 5일 평균(volume_ma5), 종가의 10일 평균(close_ma10), 거래량의 10일 평균(volume_ma10), 종가의 20일 평균(close_ma20), 거래량의 20일 평균(volume_ma20), 종가의 60일 평균(close_ma60), 거래량의 60일 평균(volume_ma60), 종가의 120일 평균(close_ma120), 거래량의 120일 평균(volume_ma120)이 계산됩니다.

전처리 후의 데이터에 그림 7.50과 같이 학습 데이터 준비를 위해 필요한 값들을 추가합니다.

	open_lastclose_ratio	high_close_ratio	low_close_ratio	close_lastclose_ratio	volume_lastvolume_ratio
4450	-0.003317	0.004959	-0.008264	0.003317	0.281068
4451	0.000000	0.003306	-0.003306	0.000000	-0.166040
4452	-0.001653	0.003311	-0.001656	-0.001653	0.882829
4453	-0.013245	0.008446	0.000000	-0.019868	-0.166176
4454	0.005068	0.011905	0.000000	-0.006757	-0.178911

	close_ma5_ratio	volume_ma5_ratio	close_ma10_ratio	volume_ma10_ratio	close_ma20_ratio	volume_ma20_ratio
4450	0.003317	-0.032797	0.007829	-0.115154	0.011452	-0.174674
4451	0.000993	-0.199990	0.006488	-0.242870	0.010692	-0.300640
4452	0.000331	0.409760	0.003322	0.379957	0.009021	0.287919
4453	-0.016284	0.191541	-0.015957	0.116987	-0.010613	0.096567
4454	-0.018036	-0.071403	-0.020490	-0.097147	-0.016558	-0.083394

	close_ma60_ratio	volume_ma60_ratio	close_ma120_ratio	volume_ma120_ratio
4450	-0.027930	-0.260495	-0.034472	-0.250189
4451	-0.027070	-0.380550	-0.034523	-0.376015
4452	-0.027662	0.158802	-0.036119	0.171613
4453	-0.045726	-0.040778	-0.055118	-0.026325
4454	-0.050950	-0.212294	-0.061253	-0.202668

그림 7.50 KT(030200) 종목의 비율값 전처리

학습 데이터에는 전일 종가 대비 시가(open_lastclose_ratio), 종가 대비 고가(high_close_ratio), 종가 대비 저가(low_close_ratio), 전일 종가 대비 당일 종가(close_lastclose_ratio), 전일 거래량 대비 당일 거래량(volume_lastvolume_ratio), 5일 평균 종가 대비 당일 종가(close_ma5_ratio), 5일 평균 거래량 대비 당일 거래량(volume_ma5_ratio), 10일 평균 종가 대비 당일 종가(close_ma10_ratio), 10일 평균 거래량 대비 당일 거래량(volume_ma10_ratio), 20일 평균 종가 대비 당일 종가(close_ma20_ratio), 20일 평균 거래량 대비 당일 거래량(volume_ma20_ratio), 60일 평균 종가 대비 당일 종가(close_ma60_ratio), 60일 평균 거래량 대비 당일 거래량(volume_ma60_ratio), 120일 평균 종가 대비 당일 종가(close_ma120_ratio), 120일 평균 거래량 대비 당일 거래량(volume_ma120_ratio)이 계산되어 추가됩니다.

7.6.3 학습 파라미터 설정

KT(030200)를 대상으로 투자 시뮬레이션을 할 때에 설정한 학습 파라미터는 다음과 같습니다.

초기 자본금	10,000,000원
최소 투자 단위	20주
최대 투자 단위	20주
총 에포크 수	1,000회
초기 탐험률	0.5 (50%)
지연 보상 임계치	0.05 (5%)
학습 속도	0.0001
할인 요인	0% (미적용)

10,000,000원으로 주식투자 시뮬레이션을 합니다. 2016년 KT는 2~3만 원 대의 주가를 가지므로 투자 단위를 20주로 정했습니다. 1,000회 시뮬레이션을 반복하며 학습을 진행하고 초기 탐험률을 50%로 정하고 에포크가 지나가면서 동일 비율로 탐험률을 줄여나갑니다. 최종 에포크에서는 탐험률이 0이 되게 합니다.

이익 또는 손실이 기준점 대비 5% 초과로 발생했을 때 학습을 수행하도록 지연 보상 임계치를 0.05로 정하고 0.0001의 학습 속도로 학습을 진행합니다.

탐험률에 따라 무작위 투자를 진행하기 때문에 동일한 실험 환경을 갖추더라도 실험 결과가 다를 수 있습니다.

7.6.4 에포크 10일 때의 결과

KT 종목의 투자 시뮬레이션: 에포크 10의 결과는 그림 7.51과 같습니다. 정책 신경망이 아직 적합화되지 않은 상태입니다.

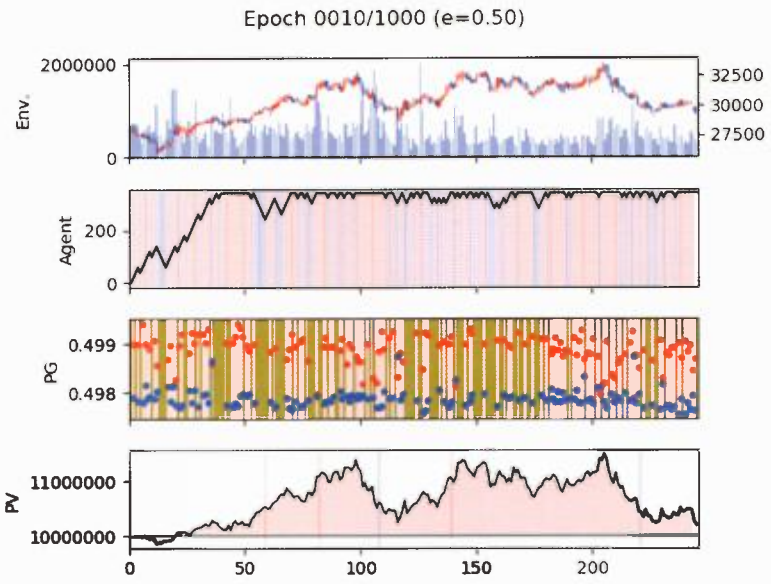


그림 7.51 KT(030200) 종목의 강화학습 투자 시뮬레이션: 에포크 10/1000

대부분의 구간에서 정책 신경망이 매수 신호를 보내고 있습니다. 상승 추세이기 때문에 무작위 투자에서 긍정적 보상을 받아 긍정적 학습을 많이 수행한 결과로 보입니다. 부정적 학습이 더 이루어지면서 정책 신경망이 더 적합화가 되어야 합니다.

에포크 10의 로그는 다음과 같습니다.

```
[Epoch 0010/1000] Epsilon:0.4955 #Expl.:114/246 #Buy:85 #Sell:68 #Hold:93 #Stocks:340
PV:₩10,201,000 POS:3 NEG:2 Loss: 0.722953
```

약 50%의 무작위 투자를 수행하였습니다. 총 246회의 투자 행동 중에서 114번을 무작위로 정했습니다. 85번 매수하고 68번 매도하고 93번 관망했습니다. KT 종목 투자 시뮬레이션에서는 한번 매수 또는 매도할 때의 투자 단위는 20주입니다. 최종적으로 340주를 보유합니다. 최종 포트폴리오 가치는 10,201,000원으로 약 2%의 얼마 되지 않는 수익이 발생했습니다.

3번의 긍정적 학습과 2번의 부정적 학습이 있었습니다. 학습 손실은 0.722953입니다.

7.6.5 에포크 200일 때의 결과

그림 7.52는 에포크 200일 때의 결과를 보여줍니다. 이제 정책 신경망이 에포크 10에 비해서 의미 있는 신호를 보내주고 있습니다.

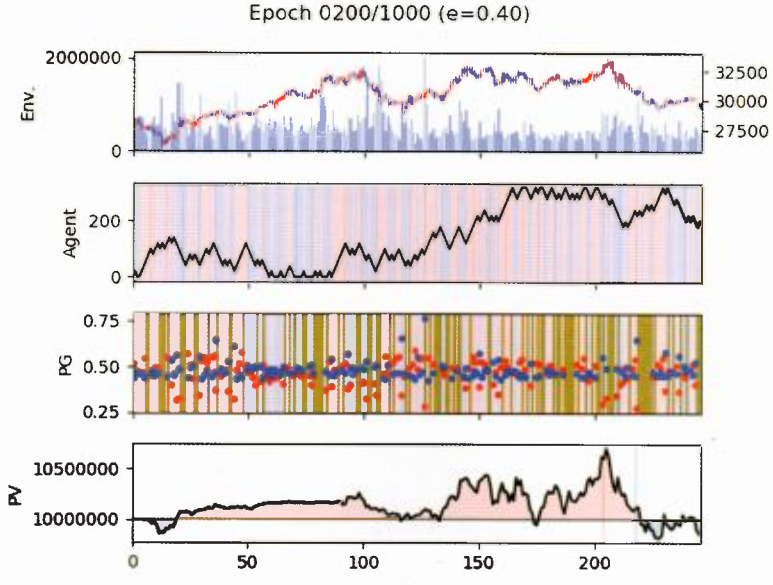


그림 7.52 KT(030200) 종목의 강화학습 투자 시뮬레이션: 에포크 200/1000

3분기에서 매수세를 보이며 보유 주식 수를 늘려가고 있습니다. 그러나 4분기 후반에서 MDD를 맞아서 큰 손실을 입고 있습니다.

에포크 200에서의 로그는 다음과 같습니다.

```
[Epoch 0200/1000] Epsilon:0.4004 #Expl.:94/246 #Buy:119 #Sell:109 #Hold:18 #Stocks:200
PV:₩9,860,000 POS:1 NEG:1 Loss: 0.800329
```

약 40%로 무작위 투자를 진행합니다. 총 246번의 투자 행동 중에서 94번을 무작위로 결정했습니다. 119번 매수하고 109번 매도하고 18번 관망했습니다. 최종 보유 주식 수는 200주입니다. 최종 포트폴리오 가치가 9,860,000원으로 약 1.4% 손실이 있었습니다.

한 번의 긍정적 학습과 한 번의 부정적 학습이 있었습니다. 학습 손실은 0.800329입니다.

7.6.6 에포크 600일 때의 결과

그림 7.53은 KT 종목의 투자 시뮬레이션: 에포크 600에서의 결과를 보여줍니다. 에포크 200과는 다르게 1분기에서 매수세를 보이고 있습니다.

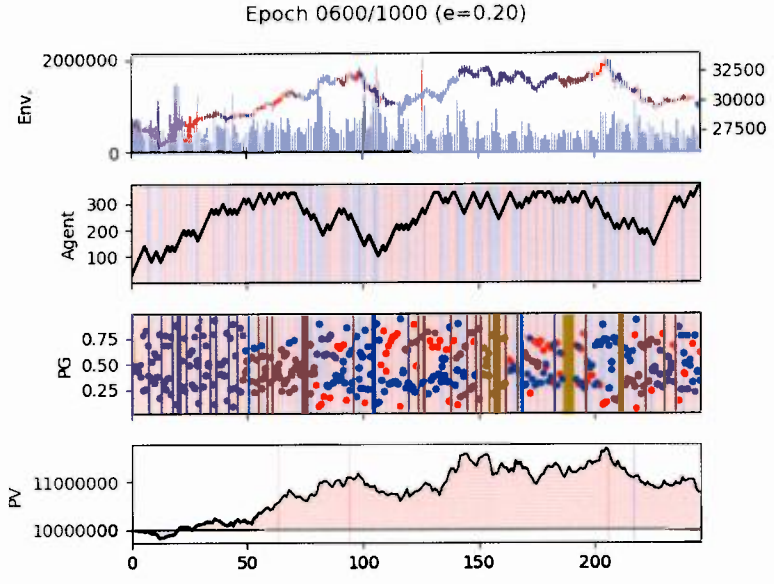


그림 7.53 KT(030200) 종목의 강화학습 투자 시뮬레이션: 에포크 600/1000

2분기 후반에 큰폭으로 하락할 때 보유 주식 수를 줄여줌으로써 큰 손실을 피하고 있습니다. 4분기에서도 큰폭의 하락에 보유 주식의 상당 부분을 매도하고 있습니다.

에포크 600의 로그는 다음과 같습니다.

```
[Epoch 0600/1000] Epsilon:0.2002 #Expl.:50/246 #Buy:127 #Sell:109 #Hold:10 #Stocks:360
PV:₩10,774,000 POS:3 NEG:1 Loss: 0.531850
```

약 20%로 무작위 투자를 진행했습니다. 246번의 투자 행동 중에서 50번을 무작위로 결정했습니다. 127번 매수하고 109번 매도하고 10번 관망했습니다. 최종 보유 주식 수는 360주이고 최종 포트폴리오 가치는 10,774,000원입니다.

3번의 긍정적 학습과 한 번의 부정적 학습을 수행했습니다. 학습 손실은 0.531850입니다.

7.6.7 에포크 1000일 때의 결과

그림 7.54는 에포크 1000의 결과를 보여줍니다. 상승구간에서 꾸준히 매수하여 보유 주식 수를 늘리고 있습니다.

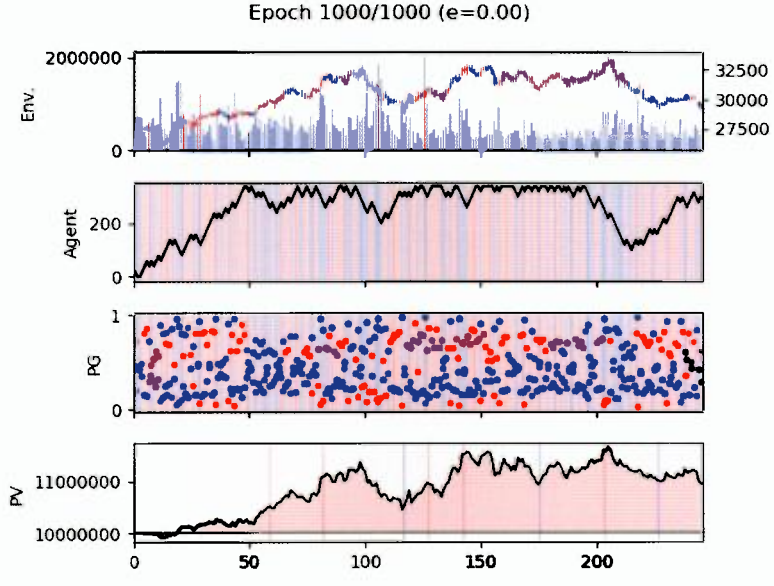


그림 7.54 KT(030200) 종목의 강화학습 투자 시뮬레이션: 에포크 1000/1000

4분기 후반에서는 매도하여 급락을 피하고 다시 매수하여 보유 주식 수를 늘리고 있습니다.

KT 종목의 투자 시뮬레이션: 에포크 1000의 로그는 다음과 같습니다.

```
[Epoch 1000/1000] Epsilon:0.0000 #Expl.:0/246 #Buy:117 #Sell:102 #Hold:27 #Stocks:300
PV:₩10,955,000 POS:5 NEG:3 Loss: 0.652624
```

무작위 투자는 하지 않고 정책 신경망의 신호만을 기준으로 투자 행동을 결정합니다. 117번 매수하고 102번 매도하고 27번 관망했습니다. 총 보유 주식 수는 300주이고 최종 포트폴리오 가치는 10,955,000원입니다.

이 에포크에서는 5번의 긍정적 학습과 3번의 부정적 학습을 수행했습니다. 학습 손실은 0.652624입니다.

7.6.8 총평

2016년 KT 종목은 1분기와 2분기에 뚜렷한 상승 추세를 보이며 3분기는 고점에서 횡보하며 4분에서 하락을 보입니다. 투자 시뮬레이션 결과 1분기에서 꾸준히 주식을 매수하여 수익을 보고 있으며 4분기에서 매도했다가 다시 매수하여 큰 하락을 방어합니다.

7.7 투자 시뮬레이션 결과 정리 및 원숭이 투자와의 비교

실제 주식 투자에서는 30여 개 종목으로 포트폴리오를 구성하고 주식 시장(코스피, 코스닥 등)의 수익률을 넘어서는지, 이른바 알파를 주로 확인하여 투자 전략의 유효성을 검증합니다.

이 책에서의 주식투자 시뮬레이션은 강화학습이 주식투자에 적용될 수 있는지를 확인하기 위해서 아주 기본적인 특징들만 사용하였고 한 종목만을 대상으로 정책 신경망을 학습하게 했습니다.

앞서 말한 제한 조건들에 맞춰 주식 시장과 RLTrader의 투자 시뮬레이션을 비교하는 것은 유의미하지 않습니다. 따라서 여기서는 10회의 원숭이 투자(monkey trading)[8]를 진행하여 그 수익률을 RLTrader의 투자 시뮬레이션과 비교합니다.

---

8 원숭이 투자란 무작위로 행동을 결정하는 주식 투자 방법을 의미합니다. 말도 안 된다고 생각할 수 있지만 많은 주식 투자 전략 관련 서적에서 펀드매니저들의 주식 투자 수익률보다 원숭이 투자의 수익률이 높다고 밝히고 있습니다.

이 책은 투자 전략을 제시하기보다는 개인들이 강화학습 주식투자 실험 환경을 구성하는 것을 돕는 데 초점을 맞추고 있습니다. 더 풍부한 특징(PBR, PER, ROE 등)을 고려하고 주식 시장의 전체 종목을 대상으로 포트폴리오를 구성하도록 RLTrader를 고도화해 나갈 것이며 관심 있으신 독자분들의 참여를 환영합니다. 개발 참여는 Github(https://github.com/quantylab/rltrader)에서 할 수 있습니다. 저자는 의미 있는 결과가 나오면 개정판을 출판할 계획입니다.

표 7.1은 시뮬레이션 대상 종목의 결과를 정리하여 보여줍니다.

표 7.1 원숭이 투자와 RLTrader 학습 시뮬레이션 결과 수익률 비교

구분	삼성전자	SK하이닉스	현대차	LG화학	NAVER	KT	평균
MT#01	22.41	3.29	-0.83	-4.50	9.66	11.57	6.93
MT#02	22.73	25.26	-4.28	1.25	17.67	7.35	11.66
MT#03	27.66	33.85	-1.05	-4.85	0.53	-3.68	8.74
MT#04	11.83	1.86	10.75	-5.84	6.46	-7.48	2.93
MT#05	19.10	0.37	-2.15	0.49	14.03	9.56	6.90
MT#06	16.06	11.15	12.78	-3.76	2.58	-5.88	5.49
MT#07	26.64	26.89	4.58	-8.37	27.47	1.69	13.15
MT#08	4.27	37.89	7.03	-1.90	3.17	-2.10	8.06
MT#09	26.01	15.38	3.45	3.45	2.97	1.27	8.76
MT#10	12.55	5.89	2.53	-3.47	5.93	0.72	4.03
MT avg.	18.93	16.18	3.28	-2.75	9.05	1.30	7.66
RLTrader	29.11	30.71	6.10	2.78	10.07	9.55	14.72
diff.	10.18	14.53	2.82	5.52	1.02	8.25	7.05

여기서 MT는 원숭이 투자를 의미합니다. 삼성전자와 SK하이닉스는 워낙 상승 추세의 종목이였기 때문에 원숭이 투자의 평균 수익률이 높습니다. 반면에 LG화학은 손실을 내는 경우가 많았습니다. 전체적으로 원숭이 투자의 평균 수익률은 7.66%입니다.

RLTrader는 평균 수익률이 14.72%이고 원숭이 투자와의 수익률 차이는 7.05% 입니다. 이 차이는 아주 기본적인 특징만으로도 정책 신경망 학습이 의미가 있었음을 보여줍니다.

7.8 이번 장의 요점

- 삼성전자, SK하이닉스, 현대차, LG화학, NAVER, KT 종목에 대한 강화학습 투자 시뮬레이션을 자세히 다뤘습니다.
- 각 종목의 주식 데이터를 전처리하는 과정을 살펴봤습니다.
- 2016년도 실제 주식 데이터로 투자 시뮬레이션을 진행하고 결과를 가시화된 차트와 로그로 살펴봤습니다.
- 각 투자 시뮬레이션에서 아쉬웠던 부분을 짚어보고 RLTrader를 개선시킬 방법을 모색해 봤습니다.
- 투자 시뮬레이션 결과를 원숭이 투자 결과와 비교하여 RLTrader의 유효성을 확인했습니다.

08장

모델 활용: 학습된 정책 신경망 모델을 사용한 투자 시뮬레이션

이번 장에서는 학습시킨 정책 신경망 모델을 불러와서 투자 행동 결정에 사용하는 방법을 다룹니다.

8.1 모델 학습과 모델 활용의 차이점

모델 학습은 많은 에포크(이 책에서는 1,000번)를 수행하면서 (무작위 또는 정책 신경망 신호를 근거로) 투자 행동을 수행하고 그로 인해 생긴 결과를 경험으로 삼아 포트폴리오 가치를 높일 수 있도록 학습합니다.

모델 활용은 모델 학습으로 구축된 정책 신경망 모델을 사용하여 (학습과정 없이) 정책 신경망의 신호만을 근거로 투자 행동을 수행합니다.

8.1.1 시뮬레이션 과정 차이점

학습된 모델로 투자 행동을 결정하는 방식(8장에서 다루는 내용)이 학습을 하면서 투자 시뮬레이션을 하는 방식(7장에서 다룬 내용)과 다른 점은 다음과 같습니다.

- **학습된 정책 신경망 모델(HDF5 파일) 사용**

  이미 학습된 정책 신경망 모델을 불러와서 정책 신경망에 적용하고 정책 신경망의 출력을 근거로 투자 행동을 결정합니다.

- **학습 기능을 비활성화**

  이미 학습된 모델을 사용하므로 학습 기능을 비활성화합니다. 학습 기능을 비활성화하면 무작위 투자 비율 조정, 배치 학습 데이터 구성, 정책 신경망 배치 학습 등을 수행하지 않습니다.

- **초기 무작위 투자 비율 0%**

  무작위 투자는 강화학습을 위한 장치이므로 학습된 모델을 사용할 때는 무작위 투자를 하지 않습니다. 그러므로 초기 무작위 투자 비율을 0%로 정해 줍니다.

- 에포크 1회 수행

  에포크를 여러 번 수행하면서 정책 신경망을 학습해 나가는데, 에포크마다 정책 신경망이 학습되어가면서 투자 시뮬레이션 결과가 달라집니다. 그러나 학습된 모델을 사용하고 학습 기능을 비활성화하면 모든 몇 번을 수행하든지 그 결과는 같습니다. 그러므로 에포크를 1로 정하여 한 번만 투자 시뮬레이션을 하도록 합니다.

물론 학습된 모델을 불러오고 학습 기능을 활성화해서 추가적으로 학습을 진행할 수도 있습니다.

8.1.2 소스코드의 차이점

학습된 정책 신경망은 RLTrader 프로젝트 폴더 내의 models 폴더에 저장됩니다. 먼저 HDF5 파일이 존재하는지 확인합니다.

삼성전자(005930) 종목의 경우 '〈RLTrader〉/models/005930/model_〈버전명〉.h5' 경로에 저장됩니다. 버전명은 '20180101000000'과 같은 형식으로 지정되어 있을 것입니다.

예제 8.1은 정책 신경망 모델 파일을 읽어오는 코드를 보여줍니다. 뒤에서 기존의 main 모듈과 다른 부분을 주로 설명합니다.

예제 8.1 비학습 주식투자 시뮬레이션

https://github.com/quantylab/rltrader/blob/master/main_notraining.py

```
8    if __name__ == '__main__':
9        stock_code = '005930'  # 삼성전자
10       model_ver = '20180202000545'

51       # 비학습 투자 시뮬레이션 시작
52       policy_learner = PolicyLearner(
53           stock_code=stock_code, chart_data=chart_data, training_data=training_data,
54           min_trading_unit=1, max_trading_unit=3)
55       policy_learner.trade(balance=10000000,
56                            model_path=os.path.join(
57                                settings.BASE_DIR,
58                                'models/{}/model_{}.h5'.format(stock_code, model_ver)))
```

10번 줄에서 정책 신경망 모델의 버전명을 지정해 줍니다. 모델의 전체 파일명은 'model_〈버전명〉.h5'과 같은 형식이므로 모델에 따라 달라지는 부분인 버전명만 model_ver 변수에 지정해 줬습니다.

52번 줄에서 종목 코드, 차트 데이터, 학습 데이터, 최소 및 최대 투자 단위를 지정해 줍니다. 기존 main 모듈에서 지정해 줬던 지연 보상 기준(delayed_reward_threshold)과 학습 속도(lr)는 입력하지 않아도 됩니다. 비(非)학습 투자 시뮬레이션에서는 사용하지 않는 인자들이기 때문입니다.

55번 줄에서 준비한 정책 학습기 객체의 trade() 함수를 호출합니다. trade() 함수는 비학습 투자 시뮬레이션을 수행하기 위해 인자들을 적절히 설정하여 fit() 함수를 호출하는 함수입니다. trade() 함수에서는 fit() 함수를 호출할 때 수행할 에포크 수인 num_epoches를 1로, learning 인자에 False를 줍니다.

기존 main 모듈 마지막 부분의 '정책 신경망을 파일로 저장' 코드 블록은 제거해 줍니다. 이미 저장된 정책 신경망 모듈을 사용했고 추가적으로 학습을 하지 않았기 때문입니다. 만약 추가적인 학습을 수행하여 모델을 새로 저장하고 싶다면 코드 블록을 제거하지 않고 그대로 두면 됩니다.

8.2 학습된 정책 신경망 모델을 사용한 투자 시뮬레이션

7장에서 학습한 각각의 종목의 투자를 위한 정책 신경망 모델을 사용하여 2017년도 주식 데이터에 적용해 봅니다.

8.2.1 학습된 모델 적용 1: 삼성전자(005930)

2017년도 삼성전자 종목은 뚜렷한 상승 추세를 보이고 있으나 중간중간 조정을 받는 모습입니다.

그림 8.1은 2016년도 삼성전자 주식 데이터로 학습한 정책 신경망 모델을 2017년도 주식 데이터에 적용한 결과입니다.

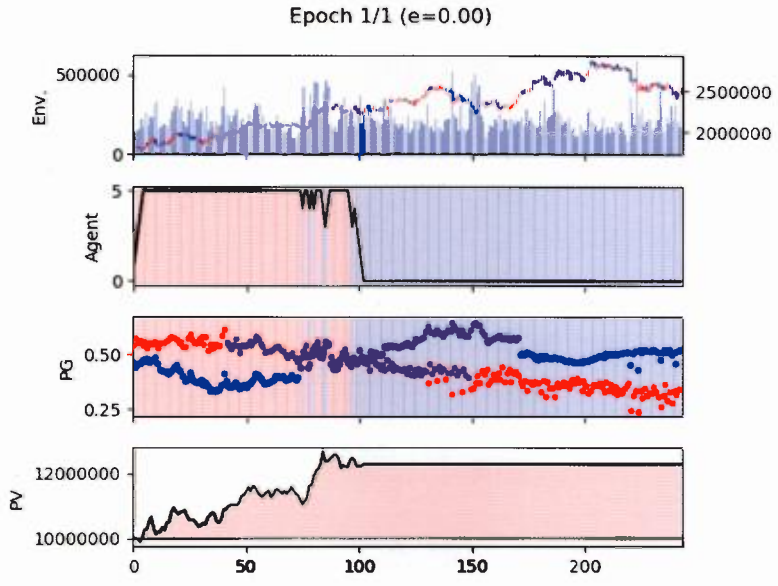


그림 8.1 학습한 삼성전자(005930) 종목 정책 신경망 모델 적용

정책 신경망이 1분기와 2분기 초반까지 매수 신호를 주고 있으며 그 이후는 매도 신호를 보내고 있습니다. 여기서 사용하는 정책 신경망은 삼성전자의 2016년 주식 데이터만을 학습한 것으로 학습 데이터를 더 풍부하게 준비하여 학습하면 결과가 더 좋을 수 있습니다. (다만 학습 시간이 더 많이 필요할 것입니다.)

이 결과의 로그는 다음과 같습니다.

```
[Epoch 1/1] Epsilon:0.0000 #Expl.:0/243 #Buy:11 #Sell:11 #Hold:221 #Stocks:0
PV:₩12,278,000 POS:0 NEG:0 Loss: 0.000000
```

무작위 투자를 하지 않으므로 탐험률은 0%이고 총 243번의 투자 행동을 모두 정책 신경망으로만 결정합니다. 11번 매수하고 11번 매도하고 221번 관망합니다. 최종 보유 주식 수는 없고 포트폴리오 가치는 12,278,000원으로 약 23%의 수익을 냈습니다.

이미 학습된 정책 신경망을 사용하고 추가 학습은 수행하지 않았습니다.

8.2.2 학습된 모델 적용 2: SK하이닉스(000660)

2017년도 SK하이닉스 종목은 삼성전자 종목과 유사한 패턴을 보이며 뚜렷한 상승 추세를 보이고 있습니다.

그림 8.2는 2016년도 SK하이닉스 종목의 주식 데이터로 학습한 정책 신경망 모델을 2017년도 주식 데이터에 적용한 결과입니다.

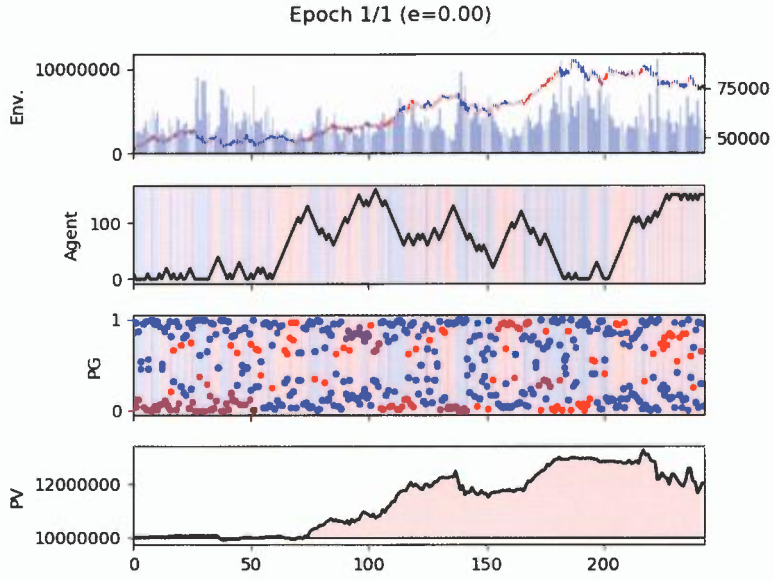


그림 8.2 학습한 SK하이닉스(000660) 종목 정책 신경망 모델 적용

정책 신경망이 3분기 후반과 4분기 초반에서 매도 신호를 보내면서 고점에서 주식을 매도하고 있습니다. 그 이후 고점 횡보 구간에서 다시 매수하고 있는 모양새입니다.

이 결과의 로그는 다음과 같습니다.

```
[Epoch 1/1]  Epsilon:0.0000  #Expl.:0/243  #Buy:114  #Sell:99  #Hold:30  #Stocks:150
PV:₩12,032,500  POS:0  NEG:0  Loss:  0.000000
```

무작위 투자를 하지 않으므로 탐험률은 0%이고 총 243번의 투자 행동을 모두 정책 신경망으로만 결정합니다. 114번 매수하고 99번 매도하고 30번 관망합니다. 최종적으로 150주를 보유합니다. 포트폴리오 가치는 12,032,500원으로 약 20%의 수익을 냈습니다.

이미 학습된 정책 신경망을 사용하고 추가 학습은 수행하지 않았습니다.

8.2.3 학습된 모델 적용 3: 현대차(005380)

현대차의 2017년도 주가는 박스권에서 등락을 반복하고 있습니다.

그림 8.3은 2016년도 현대차 종목의 주식 데이터로 학습한 정책 신경망 모델을 2017년도 주식 데이터에 적용한 결과입니다.

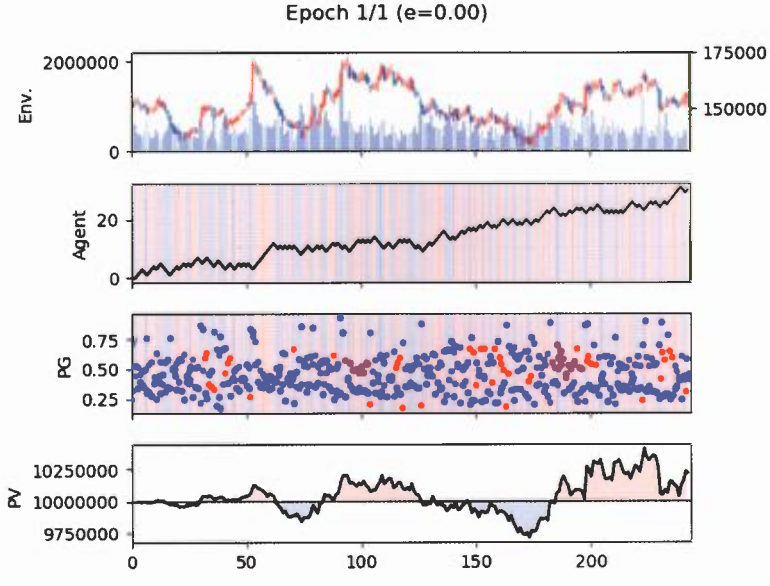


그림 8.3 학습한 현대차(005380) 종목 정책 신경망 모델 적용

정책 신경망이 꾸준하게 매수 신호를 보내면서 보유 주식 수를 늘려가고 있습니다. 박스권에서 등락을 반복하고 있어서 주가와 거래량만을 특징으로 학습하기 어려워 보입니다.

이 결과의 로그는 다음과 같습니다.

```
[Epoch 1/1] Epsilon:0.0000 #Expl.:0/243 #Buy:135 #Sell:105 #Hold:3 #Stocks:30
PV:₩10,222,000 POS:0 NEG:0 Loss: 0.000000
```

무작위 투자를 하지 않으므로 탐험률은 0%이고 총 243번의 투자 행동을 모두 정책 신경망으로만 결정합니다. 135번 매수하고 105번 매도하고 3번 관망합니다. 최종 보유 주식 수는 30주입니다. 최종 포트폴리오 가치는 10,222,000원으로 거의 보합입니다.

이미 학습된 정책 신경망을 사용하고 추가 학습은 수행하지 않았습니다.

8.2.4 학습된 모델 적용 4: LG화학(051910)

LG화학 종목의 2017년도 주가는 3분기와 4분기에서 상승 추세를 보입니다.

그림 8.4는 2016년도 LG화학 종목의 주식 데이터로 학습한 정책 신경망 모델을 2017년도 주식 데이터에 적용한 결과입니다.

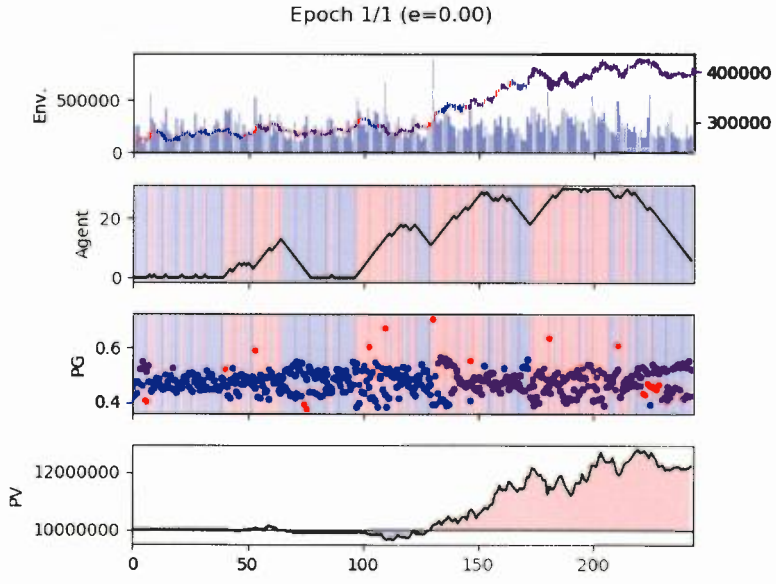


그림 8.4 학습한 LG화학(051910) 종목 정책 신경망 모델 적용

정책 신경망이 매수세와 매도세를 번갈아가며 포트폴리오 가치를 높이고 있습니다. 주로 3분기와 4분기에서 보유 주식 수를 늘리고 4분기 후반의 고점에서 매도세를 유지하는 모습입니다.

이 결과의 로그는 다음과 같습니다.

```
[Epoch 1/1]  Epsilon:0.0000  #Expl.:0/243  #Buy:94  #Sell:88  #Hold:61  #Stocks:6
PV:₩12,222,000  POS:0  NEG:0  Loss:  0.000000
```

무작위 투자를 하지 않으므로 탐험률은 0%이고 총 243번의 투자 행동을 모두 정책 신경망으로만 결정합니다. 94번 매수하고 88번 매도하고 61번 관망합니다. 최종적으로 6주를 보유합니다. 최종 포트폴리오 가치는 12,222,000원으로 약 22.2%의 수익률을 보입니다.

이미 학습된 정책 신경망을 사용하고 추가 학습은 수행하지 않았습니다.

8.2.5 학습된 모델 적용 5: NAVER(035420)

NAVER 종목은 2017년에 급등과 급락을 반복했습니다. 주가가 70만 원대에서 90만 원대까지 오르내렸습니다. 특히 3분기에 깊은 하락세를 보입니다.

그림 8.5는 2016년도 NAVER 종목의 주식 데이터로 학습한 정책 신경망 모델을 2017년도 주식 데이터에 적용한 결과입니다.

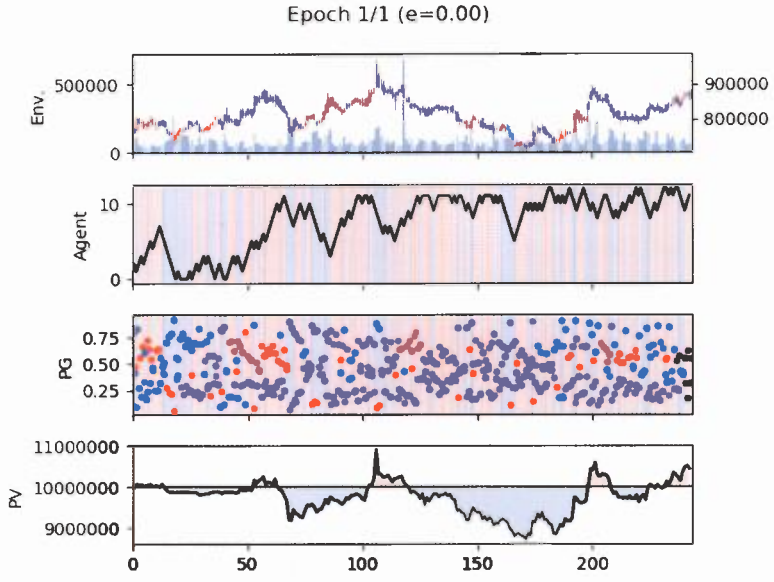


그림 8.5 학습한 NAVER(035420) 종목 정책 신경망 모델 적용

첫 번째 차트의 중간 부분에 아주 많은 거래량이 표시되는데, 이는 2017년 6월 27일에 있었던 NAVER와 미래에셋대우의 5천억 원 규모의 상호 지분 투자에 따른 것입니다.

정책 신경망이 매수세와 매도세를 번갈아가고 있지만 포트폴리오 가치를 아주 높이지는 못하고 있습니다.

이 결과의 로그는 다음과 같습니다.

```
[Epoch 1/1] Epsilon:0.0000 #Expl.:0/243 #Buy:116 #Sell:105 #Hold:22 #Stocks:11
PV:₩10,445,000 POS:0 NEG:0 Loss: 0.000000
```

무작위 투자를 하지 않으므로 탐험률은 0%이고 총 243번의 투자 행동을 모두 정책 신경망으로만 결정합니다. 116번 매수하고 105번 매도하고 22번 관망합니다. 11주를 최종적으로 보유합니다. 포트폴리오 가치는 10,445,000원으로 약 4.5%의 적은 수익률을 보입니다.

이미 학습된 정책 신경망을 사용하고 추가 학습은 수행하지 않았습니다.

8.2.6 학습된 모델 적용 6: KT(030200)

KT 종목의 2017년도 주가는 3분기 초반까지는 상승 추세를 보이다가 3분기 후반에 큰 폭의 하락을 보입니다.

그림 8.6은 2016년도 KT 종목의 주식 데이터로 학습한 정책 신경망 모델을 2017년도 주식 데이터에 적용한 결과입니다.

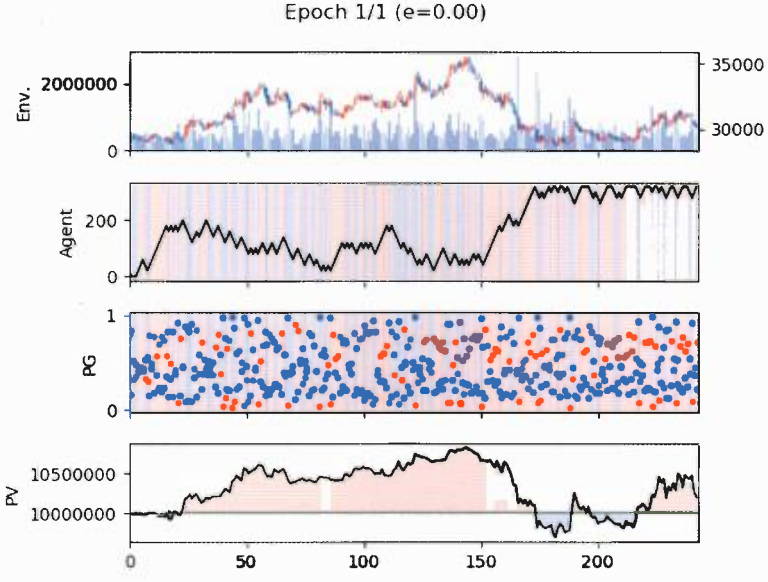


그림 8.6 학습한 KT(030200) 종목 정책 신경망 모델 적용

특이점으로는 정책 신경망이 4분기의 하락구간에서 매집했다는 점입니다. 사실 2018년도 1분기에도 주가가 하락한 점을 감안하면 좋은 선택은 아니었습니다.

이 결과의 로그는 다음과 같습니다.

```
[Epoch 1/1] Epsilon:0.0000 #Expl.:0/243 #Buy:120 #Sell:104 #Hold:19 #Stocks:320
PV:₩10,209,000 POS:0 NEG:0 Loss: 0.000000
```

무작위 투자를 하지 않으므로 탐험률은 0%이고 총 243번의 투자 행동을 모두 정책 신경망으로만 결정합니다. 120번 매수하고 104번 매도하고 19번 관망합니다. 최종 보유 주식 수는 320주입니다. 포트폴리오 가치는 10,209,000원으로 약 2.1%의 적은 수익률을 냈습니다.

이미 학습된 정책 신경망을 사용하고 추가 학습은 수행하지 않았습니다.

8.2.7 총평

2016년도 주식 데이터로 학습한 정책 신경망 모델을 2017년에 적용해도 어느 정도 투자 행동 결정을 잘 해 주고 있음을 알 수 있습니다.

삼성전자와 SK하이닉스의 경우 2016년에 상승 추세였고 2017년에도 상승 추세여서 학습한 정책 신경망 모델이 어느 정도 적용되었습니다.

현대차 주가의 경우 2016년, 2017년에 박스권에서 오르내렸는데, 2016년 1분기에서 큰폭의 상승을 경험한 때문인지 매수 신호가 매도 신호보다 조금 더 강하게 발생하여 보유 주식 수가 늘어가는 모습입니다. 특히 현대차의 경우는 더 풍부한 학습 데이터가 필요해 보입니다.

LG화학의 경우 하락 추세인 2016년도 주식 데이터를 학습하였는데도 정반대로 상승 추세를 보이는 2017년도 주식 데이터에도 잘 적용된 것을 볼 수 있었습니다.

NAVER와 KT의 경우 2016년에 상승 추세였다가 2017년에 박스권에서 등락을 반복하고 있는데, 학습한 정책 신경망 모델이 비교적 하락 구간을 방어하는 모습입니다. 그러나 이들 종목 역시 학습 데이터를 더 풍부하게 하여 박스권에서의 투자 결정을 조금 더 잘 할 필요가 있습니다.

물론 2016년도에 해당하는 1년치 주식 데이터를 학습 데이터로 삼기에는 충분하지 않습니다. 주식 투자 정책 신경망이 더 다양한 케이스에서 잘 대응하려면 더 오랜 기간의 주식 데이터와 더 다양한 특징들을 학습해야 할 것입니다.

또한 실전 투자에서는 거래 수수료와 세금을 고려하면 매일 투자를 수행하는 것은 현명한 선택이 아닐 것입니다. 일봉이 아닌 주봉으로 투자행동을 결정하여 주 1회 투자를 시뮬레이션 해보는 것도 좋은 시도일 것입니다.

그리고 한 종목만으로 학습하지 않고 같은 업종의 여러 종목을 모아서 하나의 정책 신경망이 학습하게 해 보는 것도 해 볼 수 있습니다. 이 경우 좀 더 일반화된 정책 신경망 모델을 구축할 수 있을 것입니다.

8.3 투자 시뮬레이션 결과 정리 및 원숭이 투자와의 비교

7.7절에서의 방식과 동일하게 학습된 정책신경망으로 2017년도 데이터를 적용한 투자 시뮬레이션 결과와 원숭이 투자의 결과를 비교합니다.

표 8.1은 시뮬레이션 대상 종목의 결과를 정리하여 보여줍니다.

표 8.1 원숭이 투자와 RLTrader 비학습 시뮬레이션 결과 수익률 비교

구분	삼성전자	SK하이닉스	현대차	LG화학	NAVER	KT	평균
MT#01	14.34	33.95	-6.23	8.50	5.75	-5.42	8.48
MT#02	6.36	10.76	0.65	12.64	-0.21	5.16	5.89
MT#03	17.79	17.59	7.25	24.99	9.81	-2.59	12.47
MT#04	25.44	27.28	0.73	20.38	0.06	-2.94	11.82
MT#05	7.15	6.53	3.65	3.56	6.01	5.04	5.32
MT#06	7.40	26.22	-2.50	8.41	5.29	2.18	7.83
MT#07	11.62	2.68	4.55	7.43	1.31	-0.85	4.46
MT#08	13.26	15.64	-12.68	15.42	-0.20	0.11	5.26
MT#09	7.21	13.44	1.63	7.77	8.07	-3.34	5.80
MT#10	12.23	7.30	3.15	17.65	0.86	3.47	7.44
MT avg.	12.28	16.14	0.02	12.67	3.68	0.08	7.48
RLTrader	27.80	20.33	2.22	22.22	4.45	2.09	13.18
diff.	15.52	4.19	2.20	9.55	0.78	2.01	5.71

여기서 MT는 원숭이 투자를 의미합니다. 삼성전자와 SK하이닉스의 경우 2017년에도 상승 추세입니다. 원숭이 투자에서의 평균 수익률이 각각 12.28%, 16.14%로 높은 편입니다. LG화학은 2016년과 다르게 2017년에서는 뚜렷한 상승 추세를 보이며 원숭이 투자에서의 평균 수익률도 12.67%입니다.

종목에 따라 정도의 차이는 있으나 모든 종목에서 RLTrader의 수익률이 원숭이 투자에 비해 높았습니다. 원숭이 투자의 전체 평균 수익률은 7.48%, RLTrader의 전체 평균 수익률은 13.18%로, 5.71%의 차이가 있습니다. 2016년의 데이터로 학습한 정책 신경망을 학습 데이터로 쓰이지 않았던 2017년의 데이터에 적용했을 때도 의미 있는 결과를 보여줍니다.

또한, 향후 연구(future work)를 수행할 만한 가치가 있음을 확인했습니다. 1) 학습에 사용할 특징을 더욱 풍부하게 하고, 2) 더 장기간의 데이터를 학습에 사용하고, 3) 한 종목이 아니라 여러 종목(즉, 포트폴리오)을 기준으로 학습하고, 4) 거래 수수료와 세금을 감안하여 매일이 아니라 월 1회 또는 연 1회와 같이 투자를 수행하는 등 더 다양한 실험을 해볼 가치가 있습니다. 저자는 이러한 실험을 관심 있는 독자들과 함께 진행할 것이고 향후 개정판(또는 별도의 책)으로 그 결과를 공개할 것입니다.

8.4 이번 장의 요점

- 2016년도 주식 데이터로 학습한 정책 신경망을 2017년도 주식 데이터에 적용해 보았습니다.
- 삼성전자, SK하이닉스, 현대차, LG화학, NAVER, KT 종목에 대해서 학습된 정책 신경망 모델을 사용한 결과를 살펴봤습니다.
- 각 종목의 결과를 가시화된 차트와 로그로 살펴봤습니다.
- 더 좋은(투자를 잘하는) 정책 신경망 모델을 만들기 위해 시도해 볼 수 있는 사항들을 정리해 봤습니다.
- 투자 시뮬레이션 결과를 원숭이 투자 결과와 비교하여 RLTrader의 유효성을 확인했습니다.

부록 A

기본 용어 정리

A.1 파이썬 프로그래밍 기본 용어 정리

파이썬 프로그래밍에서의 기본적인 용어를 다음과 같이 간단히 정리합니다.

- **패키지(package)**

  패키지는 관련 있는 모듈들의 묶음으로 폴더(Folder, Directory)의 형태로 존재합니다.

- **모듈(module)**

  모듈은 관련 있는 클래스, 함수, 변수 등을 포함하는 묶음으로 파이썬 파일 형태(확장자가 py)로 존재합니다.

- **클래스(class)**

  클래스는 관련이 있는 변수, 함수 등을 포함하는 묶음으로 모듈에 포함됩니다. 파이썬에서 클래스는 class 키워드로 정의할 수 있습니다.

- **함수(function, method)**

  함수는 특정 기능을 수행하는 코드 묶음으로 파라미터를 입력받아 특정한 처리를 하고 결과를 반환합니다. 파이썬에서 def 키워드로 함수를 정의할 수 있습니다.

- **파라미터(parameter)**

  파라미터는 함수에 주어주는 입력입니다. 파이썬에서는 튜플 형식의 입력과 키워드 형식의 입력을 지원합니다. 튜플 형식은 순서를 맞춰서 전달하는 입력이고 키워드 형식은 키워드로 전달하는 입력입니다. 예제 A.1과 같이 튜플 형식과 키워드 형식의 파라미터를 전달할 수 있습니다.

예제 A.1 튜플 형식과 키워드 형식 파라미터

```
1  def fnc(param_tup1, param_tup2, param_key1='', param_key2=''):
2      print(param_tup1, param_tup2)
3      print(param_key1, param_key2)
4
5  fnc('param_tup1', 'param_tup2', param_key1='param_key1', param_key2='param_key2')
```

튜플 형식의 파라미터의 경우 꼭 모두 전달해야 하며 키워드 형식의 파라미터의 경우 기본값이 존재하기 때문에 필요에 따라 전달하면 됩니다.

- **메인(main)**

  프로그램이 시작될 때 불리는 코드 영역(파이썬은 스크립트 인터프리터로 스크립트를 처음부터 읽어나가는데, 메인 영역은 직접적으로 파이썬 스크립트를 실행했을 때만 실행합니다.)

A.2 머신러닝 기본 용어 정리

머신러닝(machine learning)에서 일반적으로 쓰이는 용어들을 다음과 같이 정리합니다.

- **특징(feature), 속성(attribute)**

  특징이란 학습모델로 정답을 도출하기 위해 고려할 데이터를 의미합니다. 의미 있는 특징이 많으면 그만큼 학습이 용이합니다. 일련의 특징을 특징벡터(feature vector)라고 합니다. 특징을 속성(attribute)이라고도 부릅니다.

- **레이블(label), 클래스(class)**

  레이블이란 특징벡터를 머신러닝 모델에 통과시켰을 때 도출되기를 기대하는 정답입니다. 클래스도 레이블과 같은 의미를 가집니다. 머신러닝 분야에서는 표준 용어가 정립되어 있지 않고 주로 레이블이나 클래스라는 용어를 혼용합니다.

- **인스턴스(instance), 사례(example, case), 샘플(sample)**

  인스턴스란 학습 데이터에 포함된 하나의 특징벡터를 의미합니다. 지도학습의 경우 레이블이 부여된 특징벡터가 될 것입니다. 같은 의미로 사례, 샘플이라고도 부릅니다.

- **학습 데이터(training data)**

  학습 데이터는 학습모델을 적합화(fitting)하거나 훈련(training)하는 데 쓸 재료가 되는 데이터입니다. 학습 데이터의 품질이 학습의 성패를 좌우하는 경우가 많습니다. 지도학습에서의 학습 데이터는 정답(레이블, 클래스)을 부여한 특징벡터의 집합으로 볼 수 있습니다.

- **평가 데이터(test data)**

  평가 데이터란 정답을 아는 특징벡터의 집합으로, 학습된 머신러닝 모델에 이 특징벡터를 통과시켰을 때 정답이 도출되는지 확인하여 학습의 성패를 검증합니다.

  그림 A.1에서 학습/평가 데이터의 예를 보여줍니다.

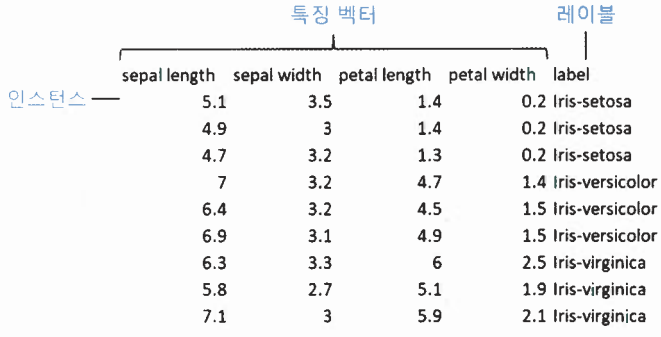


그림 A.1 학습/평가 데이터 예제에서의 특징, 레이블, 인스턴스

- **손실(loss), 비용(cost), 오차(error)**

  학습 데이터 또는 평가 데이터를 학습 모델에 통과시켰을 때 정답과의 괴리 정도를 손실이라고라고 합니다. 학습을 진행함에 따라 손실이 줄어들면서 데이터에 적합화됩니다. 비용과 오차도 같은 의미로 쓰이는 용어입니다.

- **과적합(overfitting)**

  학습 데이터에 과도하게 적합화하여 학습 데이터에 포함된 샘플을 학습모델에 통과시켰을 때는 정답이 많이 도출되지만 평가 데이터의 샘플에서는 오답이 많이 발생하는 문제를 과적합이라 합니다.

A.3 주식 거래 기본 용어 정리

증권 분야에서 쓰이는 기본적인 용어들을 다음과 같이 정리합니다.

- **봉 차트(candle chart)**

  봉 차트 또는 캔들차트는 주가의 시가, 저가, 고가, 종가의 흐름을 쉽게 파악할 수 있는 차트입니다.

- **시가(open)**

  주식시장이 열렸을 때 형성된 주가를 의미합니다. 전일 종가보다 당일 시가가 더 높으면 갭 상승이라 하고 그 반대의 경우 갭하락이라 합니다.

- **고가(high)**

  봉의 단위(일반적으로 일일 단위) 동안 형성한 가장 높은 주가를 의미합니다.

- **저가(low)**

  봉의 단위(일반적으로 일일 단위) 동안 형성한 가장 낮은 주가를 의미합니다.

- **종가(close)**

  주식시장을 마감할 때 마지막으로 형성된 주가를 의미합니다. 일반적으로 전일 종가보다 당일 종가가 더 높으면 주가가 상승했다고 하고 그 반대의 경우 주가가 하락했다고 합니다. 전일 종가와 당일 종가의 차이를 비율로 구한 것을 주가의 전일대비등락률이라 합니다.

- **거래량(volume)**

  거래량은 투자자들의 매수 및 매도 주문이 체결된 주식량을 의미합니다.

- **포트폴리오 가치(portfolio value)**

  포트폴리오 가치란 여러 주식을 사고 팔면서 이루어낸 금전적 가치로 여기에는 보유 현금과 주식 평가금이 포함됩니다. 이 책에서는 한 종목을 강화학습 모델로 투자 시뮬레이션을 하기 때문에 포트폴리오 가치가 한 종목의 주식 평가금을 의미합니다.

- **수익률(profit/loss)**

  수익률은 초기 자본금 대비 현재의 포트폴리오 가치를 비율로 구한 것입니다. 초기 자본금이 10,000,000원이고 포트폴리오 가치가 11,000,000원이면 수익률은 10%입니다. 이 포트폴리오 가치가 변동될 수 있기 때문에 수익률은 확정된 것이 아닙니다. 보유 주식을 모두 팔아 현금화했을 경우 수익률이 실현수익률이 됩니다.

- **수정주가(adjusted close)**

  수정주가는 증자, 액면분할 등에 의해 주가가 변했을 때 현재의 주가를 기준으로 과거의 주가를 조정해 준 가격입니다.

- **주가수익비율(PER)**

  주가를 주당순이익으로 나눈 값으로 순이익을 기준으로 주가가 높은지 낮은지를 판단하는 지표입니다.

- **주가순자산비율(PBR)**

  주가를 주당순자산으로 나눈 값으로 순자산을 기준으로 주가가 높은지 낮은지를 판단하는 지표입니다.

부록 B

RLTrader 커스터마이징

이번 장에서는 여러 가지 실험을 해 볼 수 있도록 RLTrader의 에이전트, 정책 신경망, 주식 데이터를 수정하는 방법을 다룹니다.

이전 장에서 여러 종목의 투자 시뮬레이션 결과를 확인했고 개선 여지에 대해서 언급했습니다. 에이전트, 정책 신경망, 주식 데이터 등을 수정해 가면서 다방면으로 실험을 해 볼 가치가 있습니다.

B.1 에이전트 모듈 커스터마이징

행동을 수행하는 주체인 에이전트 모듈에서 커스터마이징할 대상과 각 대상들을 커스터마이징하는 방법을 소스코드 수준에서 다룹니다.

B.1.1 커스터마이징 대상

에이전트 모듈은 정책 신경망의 출력을 기준으로 투자 행동을 정하거나 무작위로 행동을 정합니다. 그리고 행동에 따른 즉시 보상과 지연 보상을 결정합니다.

여기서 매매 수수료 및 세금, 행동 결정 방법, 보상 결정 방법을 변경할 수 있습니다.

B.1.2 매매수수료 및 세금 커스터마이징 방법

예제 B.1은 매매 수수료 및 세금을 커스터마이징하는 방법을 보여줍니다.

예제 B.1 매매 수수료 및 세금 커스터마이징

https://github.com/quantylab/rltrader/blob/master/agent.py

```
9       TRADING_CHARGE = 0  # 거래 수수료 미고려 (일반적으로 0.015%)
10      TRADING_TAX = 0  # 거래세 미고려 (실제 0.3%)
```

https://github.com/quantylab/rltrader/blob/master/agent_custom.py

```
9       TRADING_CHARGE = 0.00015  # 매수 또는 매도 수수료 0.015%
10      TRADING_TAX = 0.003  # 거래세 0.3%
```

주식투자 시뮬레이션에서 수수료 및 거래세를 고려하고자 한다면 TRADING_CHARGE 및 TRADING_TAX 값을 적절히 설정해 주면 됩니다. 사용하는 증권사의 수수료에 맞게 수정하고자 한다면 TRADING_CHARGE 값만 알맞게 정해 주면 됩니다. 거래세는 0.3%로 정해져 있으니 이 값을 입력해 주면 됩니다. 여기서 주의할 사항은 백분율(%)로 주로 언급되는 수수료 및 거래세에 100을 나눈 값을 설정해 주어야 한다는 것입니다.

에이전트는 정책 신경망의 출력에서 매수가 매도보다 높은 값을 가지면 매수를, 그 반대의 경우 매도를 선택했습니다. 물론 탐험률에 따라 무작위로 행동을 결정하기도 했습니다. 무작위로 행동을 결정하는 로직은 변경할 일이 없을 것이지만 정책 신경망 값을 기준으로 행동을 결정하는 방법은 변경할 여지가 있습니다.

B.1.3 행동 결정 로직 커스터마이징 방법

예제 B.2는 정책 신경망 출력 값을 기준으로 행동을 결정하는 로직을 커스터마이징하는 방법을 보여줍니다.

예제 B.2 **행동 결정 방법 커스터마이징**

https://github.com/quantylab/rltrader/blob/master/agent.py

```
75          else:
76              exploration = False
77              probs = policy_network.predict(sample)  # 각 행동에 대한 확률
78              action = np.argmax(probs)
79              confidence = 1 + probs[action]
80          return action, confidence, exploration
```

https://github.com/quantylab/rltrader/blob/master/agent_custom.py

```
75          else:
76              exploration = False
77              probs = policy_network.predict(sample)  # 각 행동에 대한 확률
78              action = np.argmax(probs) if np.max(probs) >= 0.1 else Agent.ACTION_HOLD
79              confidence = 1 + probs[action]
80          return action, confidence, exploration
```

정책 신경망은 매수와 매도에 대해 각각 출력 값을 주는데, RLTrader는 그 두 값 중에서 높은 값을 선택하여 투자 행동 결정을 내립니다.

이 경우, 정책 신경망 출력 값이 매우 낮았는데도 매수 또는 매도를 수행할 수 있습니다. 정책 신경망의 출력 값이 어느 정도의 값(임계값, threshold) 이상일 경우에만 결정된 행동을 결정하도록 할 수도 있습니다.

78번 줄의 코드를 변경하여 정책 신경망의 출력 값이 0.1 이상일 경우에만 행동을 수행하고 0.1 미만일 경우 관망할 수 있습니다.

B.2 정책 신경망 모듈 커스터마이징

행동을 결정할 머리 역할을 하는 정책 신경망 모듈에서 커스터마이징할 대상과 각 대상들을 커스터마이징하는 방법을 소스코드 수준에서 다룹니다.

B.2 1 커스터마이징 대상

앞서 설명에서 RLTrader에서 LSTM 신경망을 사용한다고 했습니다. LSTM 신경망의 레이어 구성을 변경할 수 있고 LSTM 외의 다양한 신경망을 사용할 수도 있습니다.

B.2.2 레이어 차원이나 드롭아웃 확률 커스터마이징 방법

예제 B.3과 같이 간단한 커스터마이징으로 LSTM의 레이어 차원, 드롭아웃 확률 등을 변경할 수 있습니다.

예제 B.3 정책 신경망 레이어 구성 커스터마이징

https://github.com/quantylab/rltrader/blob/master/policy_network.py

```
13          self.model = Sequential()
14
15          self.model.add(LSTM(256, input_shape=(1, input_dim),
16                              return_sequences=True, stateful=False, dropout=0.5))
17          self.model.add(BatchNormalization())
18          self.model.add(LSTM(256, return_sequences=True, stateful=False, dropout=0.5))
19          self.model.add(BatchNormalization())
20          self.model.add(LSTM(256, return_sequences=False, stateful=False, dropout=0.5))
21          self.model.add(BatchNormalization())
22          self.model.add(Dense(output_dim))
23          self.model.add(Activation('sigmoid'))
24
25          self.model.compile(optimizer=sgd(lr=lr), loss='mse')
```

https://github.com/quantylab/rltrader/blob/master/policy_network_custom.py

```
13          self.model = Sequential()
14
15          self.model.add(LSTM(256, input_shape=(1, input_dim),
16                              return_sequences=True, stateful=False, dropout=0.3))
17          self.model.add(BatchNormalization())
18          self.model.add(LSTM(128, return_sequences=True, stateful=False, dropout=0.3))
19          self.model.add(BatchNormalization())
20          self.model.add(LSTM(64, return_sequences=False, stateful=False, dropout=0.5))
21          self.model.add(BatchNormalization())
22          self.model.add(Dense(output_dim))
23          self.model.add(Activation('sigmoid'))
24
25          self.model.compile(optimizer=sgd(lr=lr), loss='mse')
```

B.2.3 신경망 유형 커스터마이징 사례

신경망 유형 자체를 변경할 수도 있습니다. LSTM에서 심층 신경망으로 변경해 보겠습니다. 예제 B.4는 LSTM 신경망에서 심층 신경망으로 커스터마이징을 한 결과를 보여줍니다.

예제 B.4 정책 신경망 레이어 유형 커스터마이징

https://github.com/quantylab/rltrader/blob/master/policy_network.py

```
13          self.model = Sequential()
14
15          self.model.add(LSTM(256, input_shape=(1, input_dim),
16                              return_sequences=True, stateful=False, dropout=0.5))
17          self.model.add(BatchNormalization())
18          self.model.add(LSTM(256, return_sequences=True, stateful=False, dropout=0.5))
19          self.model.add(BatchNormalization())
20          self.model.add(LSTM(256, return_sequences=False, stateful=False, dropout=0.5))
21          self.model.add(BatchNormalization())
22          self.model.add(Dense(output_dim))
23          self.model.add(Activation('sigmoid'))
24
25          self.model.compile(optimizer=sgd(lr=lr), loss='mse')
```

https://github.com/quantylab/rltrader/blob/master/policy_network_dnn.py

```
13          self.model = Sequential()
14
15          self.model.add(Dense(128, input_shape=(1, input_dim)))
16          self.model.add(Dropout(0.5))
17          self.model.add(BatchNormalization())
18          self.model.add(Dense(128))
19          self.model.add(Dropout(0.5))
20          self.model.add(BatchNormalization())
21          self.model.add(Dense(128))
22          self.model.add(Dropout(0.5))
23          self.model.add(BatchNormalization())
24          self.model.add(Dense(output_dim))
25          self.model.add(Flatten())
26          self.model.add(Activation('sigmoid'))
27
28          self.model.compile(optimizer=sgd(lr=lr), loss='mse')
```

LSTM 레이어를 Dense 레이어로 교체하고 Dropout을 별도로 추가하면 됩니다. LSTM은 2차원 데이터의 배치를 입력을 받기 때문에 일관성을 맞춰 주기 위해서 Dense 레이어로 교체하고 입력 차원을 2차원으로 정해줬습니다. 이 때문에 신경망의 마지막에 Flatten을 추가해서 1차원 출력으로 재조정해야 합니다.

B.3 학습 데이터 커스터마이징

학습 데이터에서 커스터마이징할 대상과 각 대상들을 커스터마이징하는 방법을 소스코드 수준에서 다룹니다.

B.3.1 커스터마이징 대상

학습 데이터의 특징으로 주가와 거래량 관련 값을 고려했습니다. '기관순매수 거래량'과 '외국인순매수 거래량'을 추가해 보겠습니다.

B.3.2 '기관순매수' 및 '외국인순매수' 데이터 획득

우선 그림 B.1과 같이 키움증권 HTS에서 '키움종합차트' 창을 띄웁니다.

그림 B.1 주식 데이터 특징 추가

그리고 좌측 사이드바에서 추가하고 싶은 데이터를 선택합니다. 여기서는 '기관순매수'와 '외국인순매수' 데이터를 추가했습니다.

이제 차트에 '기관순매수'와 '외국인순매수' 데이터가 추가되었고 차트에 마우스 오른쪽 버튼을 클릭을 해서 '텍스트창'을 띄웁니다. 그러면 그림 B.2와 같이 '기관순매수' 및 '외국인순매수' 열이 추가되어 있는 것을 확인할 수 있습니다.

그림 B.2 특징 추가 이후의 주식 데이터

이렇게 주식 데이터에 특징을 추가하고 CSV 파일로 만들어 놓은 뒤 RLTrader에서 읽어옵니다. 추가한 특징을 사용하기 위해서는 data_manager 모듈을 대폭 수정해야 합니다.

B.3.3 코드 조각 1: 주식 데이터 읽기 커스터마이징

예제 B.5와 같이 data_manager 모듈의 load_chart_data() 함수를 수정하여 추가한 두 열을 가져올 수 있도록 합니다. '기관 순매수'를 'inst', '외국인 순매수'를 'frgn'으로 명명했습니다.

예제 B.5 주식 데이터 읽기 커스터마이징

https://github.com/quantylab/rltrader/blob/master/data_manager.py

```
5     def load_chart_data(fpath):
6         chart_data = pd.read_csv(fpath, thousands=',', header=None)
7         chart_data.columns = ['date', 'open', 'high', 'low', 'close', 'volume']
8         return chart_data
```

https://github.com/quantylab/rltrader/blob/master/data_manager_custom.py

```
5     def load_chart_data(fpath):
6         chart_data = pd.read_csv(fpath, thousands=',', header=None)
7         chart_data.columns = ['date', 'open', 'high', 'low', 'close', 'volume', 'inst', 'frgn']
8         chart_data['inst'] = pd.to_numeric(chart_data['inst'].str.replace(',', ''), errors='coerce')
9         chart_data['frgn'] = pd.to_numeric(chart_data['frgn'].str.replace(',', ''), errors='coerce')
10        return chart_data
```

'inst'와 'frgn' 데이터는 2015년 9월 7일 날짜부터 존재하기 때문에 그 이전 날짜에서는 빈 문자열(empty string)이 채워져 있습니다. 그래서 수정 후의 8번 줄과 9번 줄에서 빈 문자열을 Not A Number(nan) 값으로 바꾸어 줍니다.

B.3.4 코드 조각 2: 주식 데이터 전처리 커스터마이징

이제 예제 B.6과 같이 전처리 과정에서 'inst'와 'frgn' 데이터를 처리하도록 수정합니다.

예제 B.6 주식 데이터 전처리 커스터마이징

https://github.com/quantylab/rltrader/blob/master/data_manager.py

```
11    def preprocess(chart_data):
12        prep_data = chart_data
13        windows = [5, 10, 20, 60, 120]
14        for window in windows:
15            prep_data['close_ma{}'.format(window)] = prep_data['close'].rolling(window).mean()
16            prep_data['volume_ma{}'.format(window)] = (
17                prep_data['volume'].rolling(window).mean())
18        return prep_data
```

https://github.com/quantylab/rltrader/blob/master/data_manager_custom.py

```
13    def preprocess(chart_data):
14        prep_data = chart_data
15        windows = [5, 10, 20, 60, 120]
16        for window in windows:
17            prep_data['close_ma{}'.format(window)] = prep_data['close'].rolling(window).mean()
18            prep_data['volume_ma{}'.format(window)] = (
19                prep_data['volume'].rolling(window).mean())
20            prep_data['inst_ma{}'.format(window)] = prep_data['inst'].rolling(window).mean()
21            prep_data['frgn_ma{}'.format(window)] = prep_data['frgn'].rolling(window).mean()
22        return prep_data
```

'기관 순매수 거래량'과 '외국인 순매수 거래량'의 이동평균 값을 계산하는 코드가 20번, 21번 줄에 추가되었습니다.

B.3.5 코드 조각 3: 학습 데이터 생성 커스터마이징

이제 학습 데이터에도 추가한 특징을 반영할 차례입니다. 예제 B.7은 추가한 특징을 반영하기 위해 커스터마이징한 학습 데이터 준비 코드를 보여줍니다.

예제 B.7 학습 데이터 생성 커스터마이징

https://github.com/quantylab/rltrader/blob/master/data_manager_custom.py

```
48        training_data['inst_lastinst_ratio'] = np.zeros(len(training_data))
49        training_data.loc[1:, 'inst_lastinst_ratio'] = \
50            (training_data['inst'][1:].values - training_data['inst'][:-1].values) / \
51            training_data['inst'][:-1]\
52                .replace(to_replace=0, method='ffill') \
53                .replace(to_replace=0, method='bfill').values
54        training_data['frgn_lastfrgn_ratio'] = np.zeros(len(training_data))
55        training_data.loc[1:, 'frgn_lastfrgn_ratio'] = \
56            (training_data['frgn'][1:].values - training_data['frgn'][:-1].values) / \
57            training_data['frgn'][:-1]\
58                .replace(to_replace=0, method='ffill') \
59                .replace(to_replace=0, method='bfill').values
```

```
69          training_data['inst_ma%d_ratio' % window] = \
70              (training_data['inst'] - training_data['inst_ma%d' % window]) / \
71              training_data['inst_ma%d' % window]
72          training_data['frgn_ma%d_ratio' % window] = \
73              (training_data['frgn'] - training_data['frgn_ma%d' % window]) / \
74              training_data['frgn_ma%d' % window]
```

기존 코드 수정 없이 기관과 외국인의 순매수 거래량을 처리하는 코드가 추가됩니다. 48번 줄에서 59번 줄까지는 기관과 외국인 순매수 거래량의 전일 대비 비율을 구하는 부분입니다.

B.3.6 코드 조각 4: 메인 모듈 커스터마이징 1

이렇게 준비한 학습 데이터로 정책 학습기 모듈이 학습을 진행할 수 있게 해 주는 main 모듈 역시 수정해 줘야 합니다.

예제 B.8에서는 메인 모듈의 임포트 부분에서 달라지는 점을 보여줍니다.

예제 B.8 메인 모듈 커스터마이징 (1)

https://github.com/quantylab/rltrader/blob/master/main.py

```
1    import logging
2    import os
3    import settings
4    import data_manager
5    from policy_learner import PolicyLearner
```

https://github.com/quantylab/rltrader/blob/master/main_custom.py

```
1    import logging
2    import os
3    import settings
4    import data_manager_custom as data_manager
5    from policy_learner import PolicyLearner
```

위에서 data_manager 모듈을 수정하여 data_manager_custom 모듈을 만들었습니다. 메인 모듈에서 data_manager를 임포트 하는 대신 data_manager_custom을 임포트하고 이를 data_manager로 명명하여 사용합니다.

B.3.7 코드 조각 5: 메인 모듈 커스터마이징 2

예제 B.9에서는 학습 데이터의 특징을 리스팅한 feature_training_data 변수를 수정하는 부분을 보여줍니다.

예제 B.9 메인 모듈 커스터마이징 (2)

https://github.com/quantylab/rltrader/blob/master/main.py

```
38      # 학습 데이터 분리
39      features_training_data = [
40          'open_lastclose_ratio', 'high_close_ratio', 'low_close_ratio',
41          'close_lastclose_ratio', 'volume_lastvolume_ratio',
42          'close_ma5_ratio', 'volume_ma5_ratio',
43          'close_ma10_ratio', 'volume_ma10_ratio',
44          'close_ma20_ratio', 'volume_ma20_ratio',
45          'close_ma60_ratio', 'volume_ma60_ratio',
46          'close_ma120_ratio', 'volume_ma120_ratio'
47      ]
```

https://github.com/quantylab/rltrader/blob/master/main_custom.py

```
38      # 학습 데이터 분리
39      features_training_data = [
40          'open_lastclose_ratio', 'high_close_ratio', 'low_close_ratio',
41          'close_lastclose_ratio', 'volume_lastvolume_ratio',
42          'inst_lastinst_ratio', 'frgn_lastfrgn_ratio',
43          'close_ma5_ratio', 'volume_ma5_ratio',
44          'inst_ma5_ratio', 'frgn_ma5_ratio',
45          'close_ma10_ratio', 'volume_ma10_ratio',
46          'inst_ma10_ratio', 'frgn_ma10_ratio',
47          'close_ma20_ratio', 'volume_ma20_ratio',
48          'inst_ma20_ratio', 'frgn_ma20_ratio',
49          'close_ma60_ratio', 'volume_ma60_ratio',
```

```
        'inst_ma60_ratio', 'frgn_ma60_ratio',
        'close_ma120_ratio', 'volume_ma120_ratio',
        'inst_ma120_ratio', 'frgn_ma120_ratio',
    ]
```

feature_training_data에 새롭게 추가한 특징인 'inst_lastinst_ratio', 'frgn_lastfrgn_ratio', 'inst_ma5_ratio', 'frgn_ma5_ratio', 'inst_ma10_ratio', 'frgn_ma10_ratio', 'inst_ma20_ratio', 'frgn_ma20_ratio', 'inst_ma60_ratio', 'frgn_ma60_ratio', 'inst_ma120_ratio', 'frgn_ma120_ratio'를 추가해 줍니다.

이런 식으로 원하는 데이터를 특징으로 정하여 학습 데이터를 수정하고 주식투자 강화학습에 직접 반영할 수 있습니다.

부록 C

딥러닝에서 GPU 사용하기

케라스의 백엔드(back-end) 라이브러리, 즉 케라스의 기반 라이브러리인 텐서플로는 별도의 추가 환경 설정을 해 주지 않으면 기본적으로 CPU(central processing unit)로 연산을 수행합니다. 딥러닝은 수많은 다중 연산을 수행을 필요로 하는데 CPU는 순차 연산에 효과적이라 딥러닝에서는 효과가 떨어집니다. 반면에 GPU(graphics processing unit)는 코어가 네 개 안팎인 CPU에 비해 더 많은 수십 개의 코어가 있어서 병렬처리에 효과적입니다. 따라서 딥러닝에서 GPU를 쓰면 CPU를 쓰는 것에 비해 효율을 높일 수 있고, 결과적으로 학습 시간을 크게 줄일 수 있습니다.

이번 장에서는 텐서플로에서 GPU가 사용되도록 설정하는 방법을 다룹니다.

C.1 GPU 사용을 위한 하드웨어 준비

모든 PC에서 GPU로 딥러닝을 할 수 있는 것은 아닙니다. GPU를 사용하기 위한 조건으로 가장 중요한 것은 그래픽카드입니다. 수많은 그래픽카드 중에서 텐서플로에서 딥러닝에 사용할 수 있는 그래픽카드는 한정적입니다. 이 책을 저술하는 시점에서, 텐서플로에서는 엔비디아(Nvidia)에서 제조해 출시한 그래픽카드 중에서 어느 정도 성능이 높은 제품만을 지원합니다. 그렇다고 초고사양 그래픽카드가 필요하다는 의미는 아닙니다.

이번 절에서는 현재 장착된 그래픽카드를 확인하고 사용하고 있는 그래픽카드가 텐서플로에서 딥러닝에 사용할 수 있는 카드인지 확인하는 방법을 다룹니다.

C.1.1 그래픽카드 인식 확인

먼저 장착된 그래픽카드를 확인해야 합니다. 키보드의 '윈도우키 + X' 조합을 누르면 화면 좌측 하단에 그림 C.1과 같은 메뉴가 뜹니다.

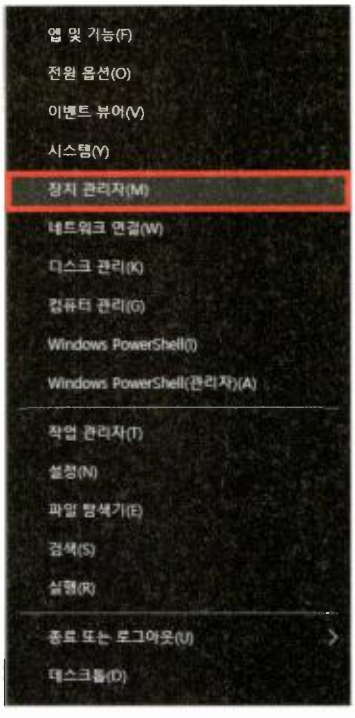


그림 C.1 장치 관리자 메뉴

여기서 '장치 관리자'를 클릭하면 그림 C.2와 같이 '장치 관리자' 창이 뜹니다. 여기서 '디스플레이 어댑터' 항목을 펼쳐보면 장착된 그래픽카드를 확인할 수 있습니다.

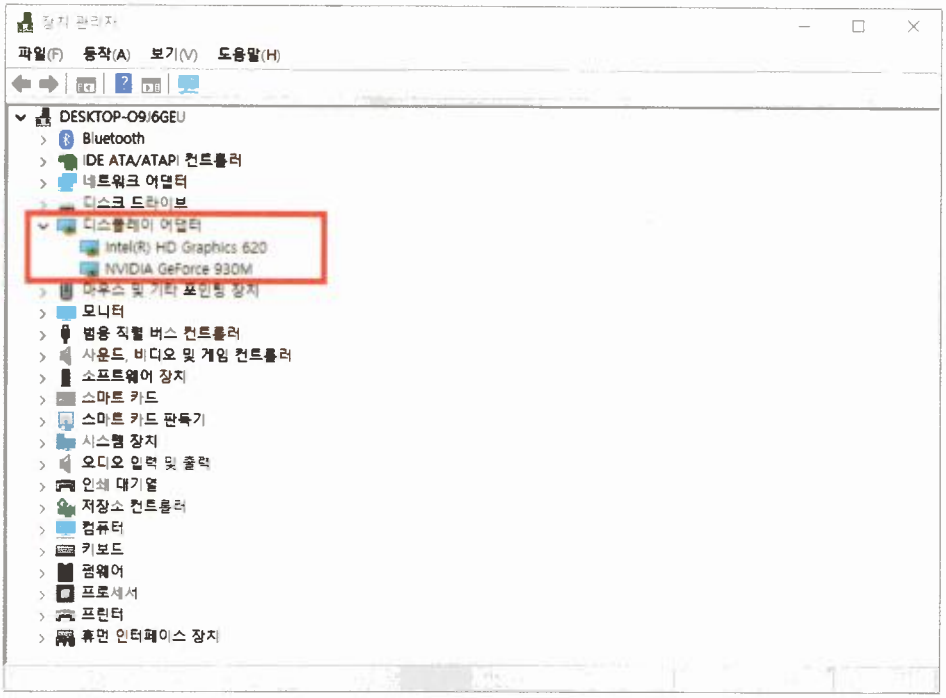


그림 C.2 장치 관리자의 디스플레이 어댑터

여기서는 'NVIDIA GeForce 930M' 장치가 인식되어 있음을 확인할 수 있습니다. 만약 그래픽카드가 인식되어 있지 않다면 윈도우 업데이트를 시도해 보거나 그래픽카드에 맞는 드라이버(driver) 소프트웨어를 다시 설치해 보시길 바랍니다.

C.1.2 호환되는 그래픽카드 확인

장착된 그래픽카드를 확인했으면 이 그래픽카드를 딥러닝에서 사용할 수 있는 그래픽카드인지 확인해야 합니다.

딥러닝에서 그래픽카드를 활용할 때 NVIDIA의 CUDA 툴킷을 활용하는데, CUDA 툴킷을 활용할 수 있는 그래픽카드인지 확인해야 합니다.

구글에 'cuda gpu'를 검색하여 'CUDA GPUs | NVIDIA Developer'(https://developer.nvidia.com/cuda-gpus) 웹페이지에 접속합니다.

그림 C.3과 같이 'CUDA-Enabled GeForce Products' 메뉴를 확인할 수 있습니다.

그림 C.3 CUDA 호환 제품군

이 메뉴를 선택하면 그림 C.4와 같이 CUDA 툴킷을 사용할 수 있는 그래픽카드의 리스트가 나옵니다.

GeForce Desktop Products

GPU	Compute Capability
NVIDIA TITAN Xp	6.1
NVIDIA TITAN X	6.1
GeForce GTX 1080 Ti	6.1
GeForce GTX 1080	6.1
GeForce GTX 1070	6.1
GeForce GTX 1060	6.1
GeForce GTX 1050	6.1
GeForce GTX TITAN X	5.2
GeForce GTX TITAN Z	3.5
GeForce GTX TITAN Black	3.5
GeForce GTX TITAN	3.5
GeForce GTX 980 Ti	5.2
GeForce GTX 980	5.2
GeForce GTX 970	5.2
GeForce GTX 960	5.2
GeForce GTX 950	5.2

GeForce Notebook Products

GPU	Compute Capability
GeForce GTX 1080	6.1
GeForce GTX 1070	6.1
GeForce GTX 1060	6.1
GeForce GTX 980	5.2
GeForce GTX 980M	5.2
GeForce GTX 970M	5.2
GeForce GTX 965M	5.2
GeForce GTX 960M	5.0
GeForce GTX 950M	5.0
GeForce 940M	5.0
GeForce 930M	5.0
GeForce 920M	3.5
GeForce 910M	5.2
GeForce GTX 880M	3.0
GeForce GTX 870M	3.0
GeForce GTX 860M	3.0/5.0(**)

그림 C.4 CUDA 호환 그래픽 카드 (중/고급 사양 제품)

여기에서 장착되어 있는 그래픽카드를 찾습니다. 저자가 사용하고 있는 그래픽카드인 'NVIDIA GeForce 930M'이 이 목록에 들어 있다는 점을 확인할 수 있습니다. 이 그래픽카드는 노트북 제품군에 속하며 계산 능력(compute capability)이 5.0점인 것을 알 수 있습니다.

이 리스트에 포함되어 있더라도 계산능력이 3.0점 미만이면 딥러닝에서 (정확히는 텐서플로에서) 사용할 수 없습니다.

딥러닝에서 활용 가능한 그래픽카드가 없더라도 CPU로 충분히 RLTrader를 구동할 수 있으니 안심하시길 바랍니다.

C.2 GPU 사용을 위한 소프트웨어 준비

장착하고 있는 그래픽카드가 딥러닝에서 활용 가능하다면 이제 필요한 소프트웨어를 설치합니다. 설치할 소프트웨어는 CUDA 툴킷과와 cuDNN 라이브러리입니다.

C.2.1 CUDA 툴킷 설치

구글에 'cuda toolkit'을 검색하면 'CUDA Toolkit | NVIDIA Developer'(https://developer.nvidia.com/cuda-toolkit) 웹페이지를 찾을 수 있습니다. 이 페이지에서 'Download Now' 버튼을 클릭하면 그림 C.5와 같은 화면을 볼 수 있습니다.

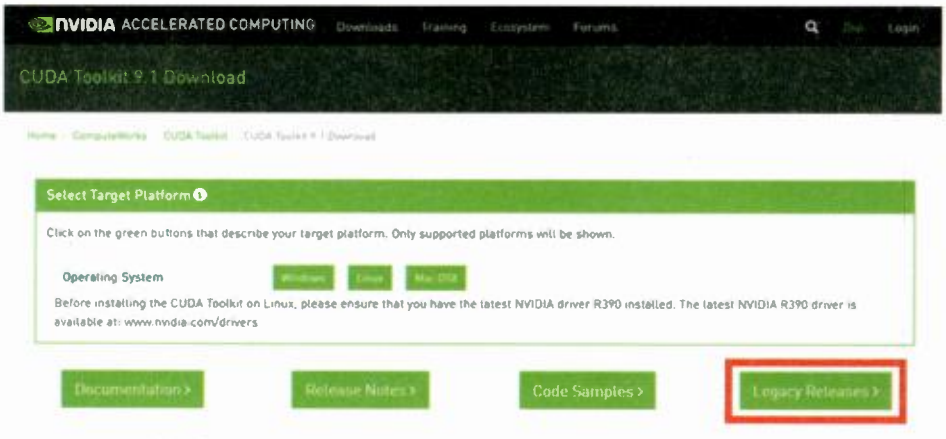


그림 C.5 CUDA 툴킷 다운로드 화면

여기서 'Legacy Releases' 버튼을 클릭해서 'CUDA Toolkit 9.0'을 내려받습니다. 내려받은 설치파일 'cuda_9.0.176_win10.exe'을 실행하여 그림 C.6과 같이 설치를 진행합니다.

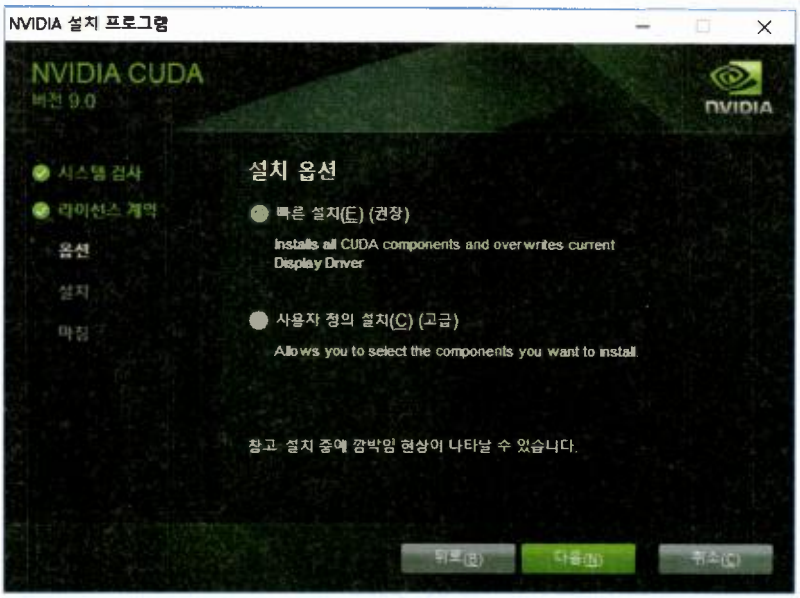


그림 C.6 CUDA 툴킷 설치 진행 화면

NVIDIA 설치 프로그램에서 권장으로 표시하듯이 '빠른 설치'를 선택하여 설치하면 됩니다.

C.2.2 cuDNN 라이브러리 설치

CUDA 툴킷을 설치했으면 이제 cuDNN 라이브러리를 설치할 차례입니다.

구글에 'cudnn'을 검색하면 'NVIDIA cuDNN | NVIDIA Developer'(https://developer.nvidia.com/cudnn) 웹페이지를 찾을 수 있습니다. 이 페이지에서 'Download' 버튼으로 cuDNN 라이브러리를 다운로드할 수 있는데, NVIDIA Developer의 회원가입을 필요로 합니다.

그림 C.7과 같은 회원가입 폼(Form)을 적절히 채워서 회원가입을 진행하시길 바랍니다.

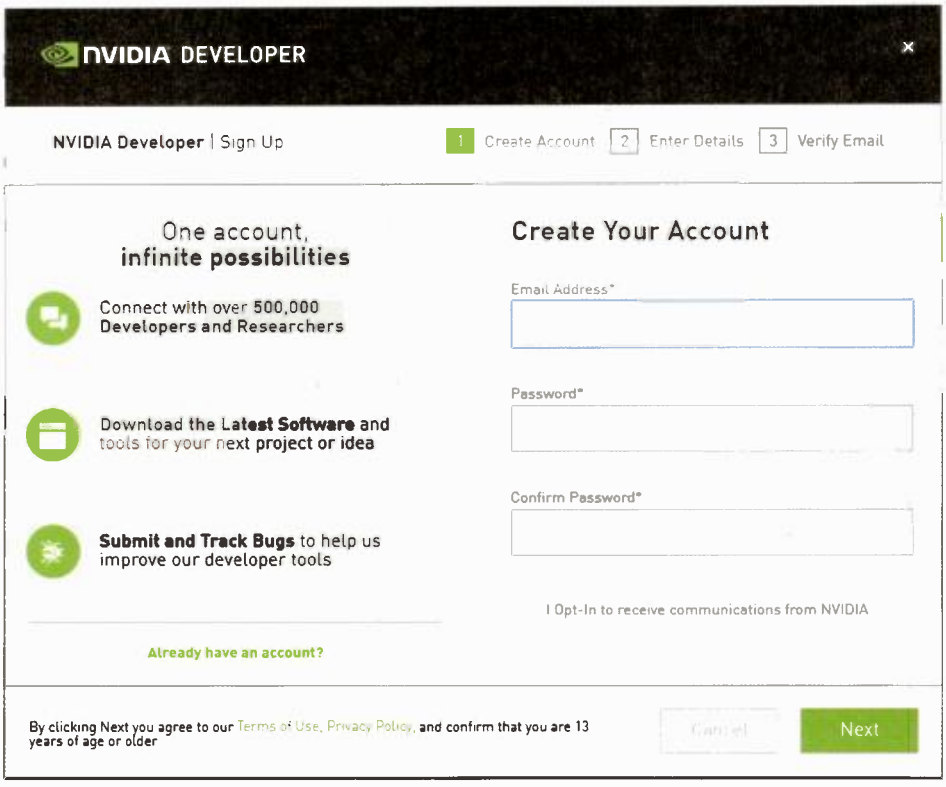


그림 C.7 NVIDIA Developer 회원가입 화면

로그인하고 다시 'Download' 버튼을 클릭하면 cuDNN 라이브러리를 버전별로 선택하여 다운로드할 수 있는 화면이 연결됩니다. 그림 C.8과 같이 'cuDNN v7.0.5 for CUDA 9.0'을 선택하여 운영체제(OS)에 맞게 다운로드하시길 바랍니다.

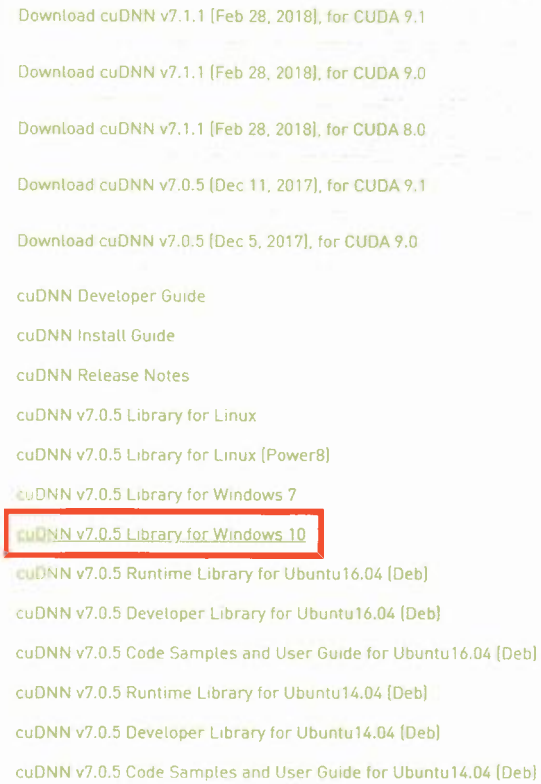


그림 C.8 cuDNN 라이브러리 다운로드 화면

저자는 윈도우 10에서 'cuDNN v7.0.5 Library for Windows 10'을 선택하여 내려받았습니다. 내려받은 파일은 zip으로 압축되어 있을 텐데, 압축을 풀면 'cuda' 폴더가 있습니다. 이 폴더 안의 'bin', 'include', 'lib' 세 폴더를 CUDA 폴더에 덮어써야 합니다.

먼저 그림 C.9와 같이 cuDNN 라이브러리의 'bin', 'include', 'lib' 폴더를 복사합니다.

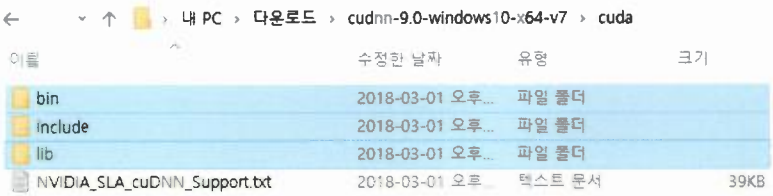


그림 C.9 cuDNN 라이브러리 파일

그림 C.10과 같이 CUDA 툴킷이 설치된 폴더로 이동합니다. C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v9.0와 같은 경로에 CUDA가 설치되어 있을 것입니다.

그림 C.10 설치된 CUDA 툴킷 경로

여기에 복사한 세 폴더 'bin', 'include', 'lib'을 붙여 넣습니다. 그림 C.11과 같이 덮어 쓸 것인지를 물어보는 대화상자가 뜨면 '대상 폴더의 파일 덮어쓰기'를 선택하면 됩니다.

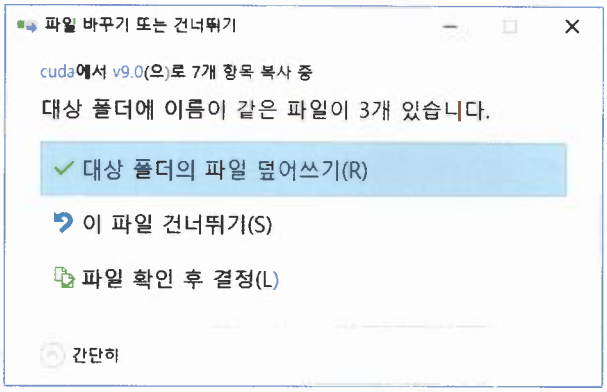


그림 C.11 cuDNN 라이브러리 파일 적용

여기까지 하면 CUDA 툴킷과 cuDNN 라이브러리 설치가 완료된 것입니다.

C.2.3 텐서플로의 GPU 사용 최종 확인

이제 텐서플로가 그래픽카드를 사용하는지 확인하면 됩니다. 기본적으로 텐서플로는 사용 가능한 그래픽카드가 있으면 CPU보다 우선적으로 GPU를 사용하게 되어 있습니다.

예제 C.1과 같이 텐서플로를 임포트하고 세션을 실행하면 GPU를 사용하는지 출력된 로그를 보고 확인할 수 있습니다.

예제 C.1 텐서플로가 GPU를 사용하는지 확인

```
1    import tensorflow as tf
2    hello = tf.constant('Hello, TensorFlow!')
3    sess = tf.Session()
4    print(sess.run(hello))
```

그림 C.12와 같이 'TensorFlow device'에 '/gpu:0'과 같이 출력되면 텐서플로가 GPU를 사용하는 것입니다.

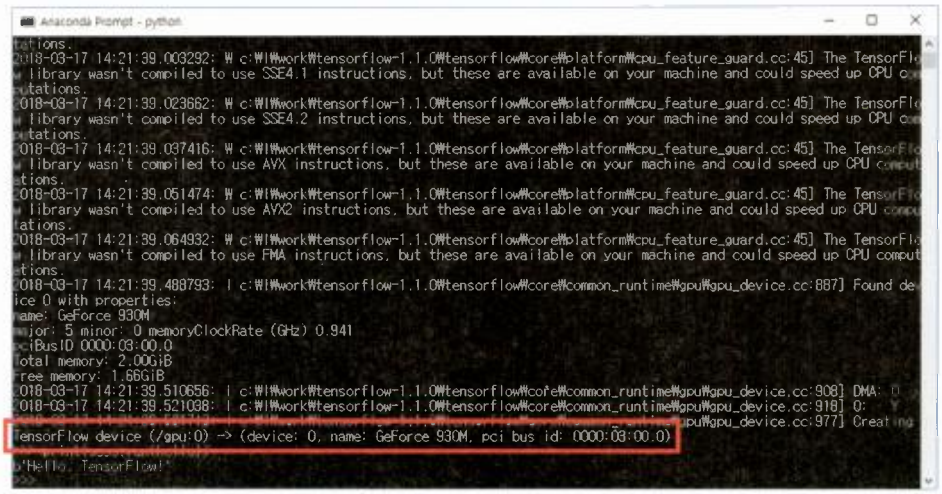


그림 C.12 텐서플로의 GPU 활용 확인

꼭 GPU를 사용해야 딥러닝을 할 수 있는 것은 아닙니다. CPU로도 충분히 딥러닝을 공부해 볼 수 있으니 사용할 수 있는 GPU가 없더라도 실망하지 마시기 바랍니다.

A – D

E – J

K – O

P – Z

ㄱ – ㅁ

ㅂ – ㅅ

ㅇ – ㅈ

ㅊ – ㅎ