resource links
https://github.com/deepcrypto/Reinforcement-learning-in-portfolio-management-
https://github.com/tr8dr/Adversarial-PG
https://lilianweng.github.io/posts/2018-02-19-rl-overview/
http://incompleteideas.net/book/RLbook2020.pdf

Background

https://medium.com/towards-data-science/an-intuitive-introduction-to-reinforcement-learning-part-i-d81512f5e25c

- covers high-level concepts, (value/policy, etc.) incl. Q-learning
- i wrote it :)
  https://spinningup.openai.com/en/latest/algorithms/ddpg.html
- DDPG explanation
  https://en.wikipedia.org/wiki/Convex_optimization
  https://www.cvxpy.org
- CVXPY provides a library for solving convex optimization problems
- needs investigation
  https://en.wikipedia.org/wiki/Black–Scholes_model

Other useful algorithms: VPG, REINFORCE, Actor-Critic, PPO, TRPO

Policy Gradient Methods

For the sake of brevity, I won't go over basic RL concepts/terminology here - I'll focus on covering what isn't already here

In the paper "Adversarial Deep RL in Portfolio Management", Liang et. al suggests the usage of a basic policy gradient method (coupled with adversarial training) as an alternative and potential improvement over methods like DDPG and PPO. For this reason, we'll spend some time on the intuition and math behind basic policy gradient methods in RL, which will also serve as a foundation for other algorithms like DDPG and PPO.

Define the following:

S_t	State at specific time step t.
A_t	Action taken at specific time step t.
R_t	Reward received after taking a specific action a_t at a specific state s_t.
$\pi(a\|s)$	Policy function, which gives the probability of taking action a given state s.
γ	Discount factor, which determines the weight of future rewards.
R_t	Return, the total accumulated reward from time step t onward: $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$
$V^\pi(s)$	Value function, the expected return from state s following policy π : $V^\pi(s) = \mathbb{E}_\pi[R_t \mid s_t = s]$
$V^\pi(s')$	Value function for the next state s'.
$Q^\pi(s,a)$	Action-value function, the expected return from state s after taking action a and then following policy π : $Q^\pi(s,a) = \mathbb{E}_\pi[R_t \mid s_t = s, a_t = a]$
θ	Parameters of the policy or value function used in gradient-based learning.
α	Learning rate used in gradient-based learning algorithms.
ϵ	Exploration parameter used in ϵ-greedy policies.

Let's also define some simple notation - capital letters will be used to denote specifics in time, ie a specific state at some timestep. Lowercase letters will be used to denote generalities, such as any state. We'll use the subscript t for uppercase letters, such as $S_t, A_t, R_t, S_{t+1}, A_{t+1}\ldots$ and we'll use the "prime" notation to denote subsequent states for lowercase letters, eg $s', a', r', s'', a'', \ldots$
Formally:

s	General notation for state (could refer to any timestep).
a	General notation for action (could refer to any timestep).
r	General notation for state (could refer to any timestep).
s'	Next state (subsequent state) after some action a at state s.
a'	Next action taken after arriving at state s'.
r'	Reward received after taking action a' at state s'.

Policy gradient methods focus on updating the policy parameters, aiming to converge to the optimal policy after some number of iterations. From Bellman's equations for value and action-value, we understand that the value function can be defined both recursively and in terms of the action-value using the following two identities:

$$\begin{aligned} V(s) &= \mathbb{E}[G_t|S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma V(S_{t+1})|S_t = s] \end{aligned}$$

$$V(s) = \sum_a \pi(a|s) q_\pi(s,a)$$

Policy gradient methods aim to learn the optimal parameterization θ that will yield the best approximation of the optimal value function $V^*(s)$. In other words, we want the maximum possible return from any given state. This equates to maximizing the value function for some policy π_θ. For this purpose, we aim to maximize the objective function, which in this case will be defined as the value function for all states s.

$$J(\theta) = V(s) = \mathbb{E}[G_t]$$

Fundamentally, the optimization of the objective function can be determined through gradient ascent. Computing the gradient of the objective function allows us to find the optimal parameters. However, the main issue is that the gradient also depends on the transitional probability distribution intrinsic to the environment, which in many cases is unknown. *Policy gradient theorem* gives us a way to define the policy gradient independent of such a distribution. I derive this here.

Policy gradient theorem states:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) Q^{\pi_\theta}(s,a)\right]$$

Given the gradient, we can now perform gradient ascent on this function. This is the fundamental algorithm behind policy gradients in reinforcement learning - though they may differ slightly in their usage of policy gradient theorem, simply policy gradient algorithms like REINFORCE and VPG build on this theorem. You can find my implementation of REINFORCE here

DPG

Deterministic policy gradient (DPG), used in the paper "A Deep Reinforcement Learning Framework for the Financial Portfolio Management Problem" by Jiang et. al is a precursor to DDPG, which we will discuss later.

General Observations

Problem Statement

Financial portfolio management is the continuous reallocation of funds into a set of assets, using various instruments to achieve an optimal growth rate (rate of return). Experienced investors will base their trading decisions on a variety of metrics and relevant news. Numerical features such as trading volume and price history can be used in training a ML model to make investment decisions. Previous iterations of the portfolio management problem have used neural networks in an attempt to predict stock prices to varying degrees of success. More algorithmic strategies such as "follow-the-winner", "follow-the-loser", and "pattern-matching" have also historically been used for portfolio management. However, strategies based on deterministic algorithms lack robustness, and training neural networks for stock prediction also have fundamental problems making them less robust and less reliable, especially when applied to larger markets.

Solution Methodology

Developing an asset-agnostic trading strategy is essential to ensuring scalability across large and diverse markets. Though training a network on a single stock may provide a cheap and effective solution for small use cases, it's simply not effective or memory-efficient at a larger scale.

> Using the proposed EIIE architecture, we should aim to evaluate its performance in a larger market with different assets. Replicating the architecture from the paper could allow us to a) evaluate its performance in-house b) experiment with hyperparams and making other changes, eg feature selection, IIE implementation c) serve as a good first exercise for the design team

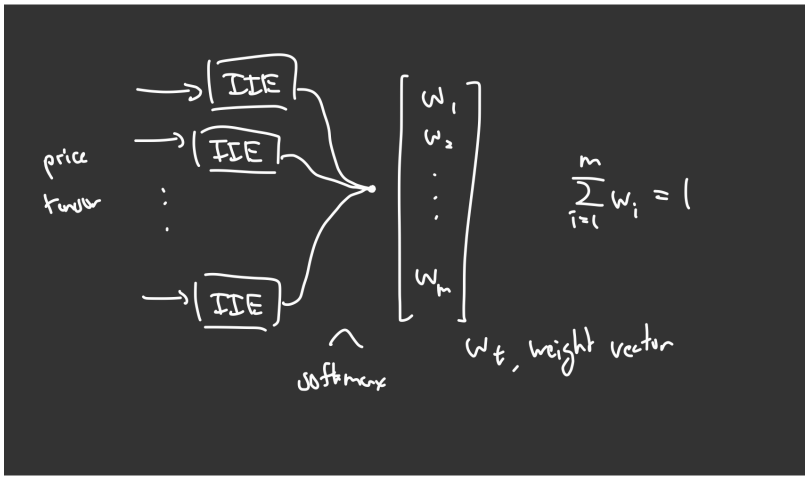
The core of this architecture lies in the paper by Jiang et. al ("A Deep Reinforcement Learning Framework for the Financial Portfolio Management Problem") which proposes an ensemble method of evaluating the viability of m selected assets, and updating its weight in the portfolio.

- trading on a market with limited assets and diversification potential is also likely not the most optimal strategy - the papers generally limit their asset selection, probably for simplicity and efficiency though this could potentially be an area to improve on
- we should aim to simulate the behaviour of an actual experienced trader as much as possible by providing a similar range of assets and financial products to select from for optimal results - ideally we should beat the market by some not insignificant percentage
    - the downside of this being the introduction of (a lot) more variables to manage, it is still possible to achieve similar results with less assets, depending on how asset preselection is done
- feature selection includes closing/high/low/opening prices, price-to-earnings ratio
- metrics to evaluate on include rate of return (ROR), Sharpe ratio,

A Deep Reinforcement Learning Framework for the Financial Portfolio Management Problem

- traditional portfolio management strategies include "follow-the-winner", "follow-the-loser", "pattern-matching", "meta-learning" - could potentially draw inspiration from existing strategies in addition to serving as a benchmark

- Meta-learning is an interesting ML strategy that could be worth exploring further
- EIIE (ensemble of independent evaluators) allows us to perform asset-independent price prediction using a shared set of parameters (ie we train a single model class aka an independent evaluator across m different assets and copy this model to create an ensemble that can perform prediction across multiple assets simultaneously, aggregating predictions in a softmax layer to generate portfolio weights)



- We can further simplify this structure in the case of CNNs by expanding the scope to 2D...

Definitions

- the *portfolio weight vector* ω is defined as the relative weight distribution of assets in the portfolio, expressed as a nonnegative value between 0 and 1
- the sum of values in the vector should naturally add to 1
- initialization: $\omega_0 = (1, 0, \ldots, 0)^T$ where the first index represents liquid cash
- we can assume that for any timestep t the portfolio holdings can be represented by the weight vector $\omega_t = (\omega_0, \omega_1, \ldots, \omega_m)^T$ where m is the number of unique assets in the portfolio
- the value of the ith asset at time t is denoted by $v_{i,t}$
- we generally express the prices in normalized form as a relative change compared to the previous timestep: this is a ratio defined as $v_{i,t}/v_{i,t-1}$
- the prices of all assets at time t is expressed using the *relative price vector*

$$y_t := v_t \oslash v_{t-1} = (1, \frac{v_{1,t}}{v_{1,t-1}}, \frac{v_{2,t}}{v_{2,t-1}}, \ldots, \frac{v_{m,t}}{v_{m,t-1}})^T$$

- note that since the first index in the relative price vector is dedicated to cash (in this case, Bitcoin) the price will never change since it represents the baseline - cash will not change in value relative to itself
- the total portfolio value is then expressed as $p_t = p_{t-1}\, y_t \cdot \omega_{t-1}$ where y_t represents the relative price vector at time t, ω_{t-1} is the weight vector at time $t-1$, and p_{t-1} is the previous portfolio value

- we justify the recursive definition later on but it should be noted that it yields the same result as multiplying assets prices by quantity, but is more efficient and mathematically makes more sense

Deterministic Policy Gradient

Deep Reinforcement Learning for Portfolio Management

- builds on the previous paper and opts for DDPG instead of just DPG (adds deep learning)
- adds arbitrage and shorting mechanisms

Adversarial Deep Reinforcement Learning in Portfolio Management

- adversarial learning involves augmenting data with intentionally poor examples, eg introducing noise into the training set, which is the implementation in this paper
- adversarial learning improves robustness of models though obviously too much noise in training set will decrease efficiency and make it more difficult for the model to learn
- this paper builds on the previous one and serves as a counterpoint, demonstrating potential shortcomings of applying complex algorithms like DDPG or PPO and instead opts for a (vanilla?) policy gradient approach, augmented by adversarial training
-