Offline Reinforcement Learning for Portfolio Optimization and Trading

I. INTRODUCTION

A. Reinforcement Learning Overview

Optimal control and Reinforcement Learning (RL) are fields of study associated with the design and implementation of systems involved in sequential decision-making. In each, an agent attempts to learn a policy that selects optimal actions over a some distribution of states and rewards. Control problems generally specify the dynamics of the environment, providing a clear model of what factors and parameters might affect the performance of the agent. In the RL setting, however, these dynamics are usually unknown or at least only partially observable, and the agent must aim to construct a model of the environment while also attempting to "learn" accurately. This lack of initial knowledge about the environment is a primary factor that distinguishes RL from other modelling disciplines. In this setting, an agent optimizes the trade-off between "exploration," where the agent gathers more information about its setting and "exploitation," in which the agent attempts to take actions that maximize its accumulated rewards over time. Statistical tools and data analysis, combined with a mathematical formulation of sequential decision-making processes make RL suitable to tackle difficult problems in optimization.

B. On-policy vs. Off-policy

The two primary classes of RL algorithms are typically referred to as either:

1. on-policy, in which an agent attempts to learn an optimal policy that maps states to a set of actions. For a given state, the agent learns a function that selects the action for which its rewards over time are maximized.

2. off-policy, an agent learns a value function that represents the agent's accumulated rewards over the state and action spaces. The value function estimates the total discounted rewards of the agent assuming that it follows a greedy policy without using an explicit policy to search the space of actions.

C. Online vs. Offline

Another crucial part in the design of RL algorithms is the choice between online and offline algorithms. As data is made available, online algorithms update the parameters of the model. This data is then discarded, and the cycle repeats. In this setting, there must be some object (typically a stochastic policy) that generates data by exploring the environment in which the agent sits. Typically, samples are processed in batches. First, samples are stored in a replay buffer of a predetermined size, and the algorithm then runs the policy on those collected samples, updating its hyperparameters when the policy has finished searching. Incremental algorithms also exist and perform an update for every sample. Offline algorithms, on the other hand, do not have any control over the data that is available to them for processing. The set of states, actions, and rewards are statically defined, meaning the data itself is independent of the agent's policy or actions in a given state.

II. PROBLEM STATEMENT

The distinction between online and offline methods is crucial for an RL algorithm's applicability and use in complex or high-risk systems. In these systems, it is infeasible to run a policy without sufficient and rigorous testing of the model. The risk to the system's stakeholders is likely too great to deploy an algorithm whose basic mandate is to learn through experience, often taking sub-optimal actions to learn over time. Current research on this subject has focused on the design of simulators aimed at capturing specific and critical information about an agent's environment. Training on these simulators allows such algorithms to converge to a policy or value function that is theoretically similar to the environment the agent is likely to encounter in the real world. Yet under this paradigm, RL algorithms have been shown to be particularly unadapatable when tested in unfamiliar contexts and settings. A likely explanation is that data collection is relatively simple in simulated/structure settings; however, in real world environments, this task is much more costly, stunting an algorithm's ability to grow from diverse prior experiences. As a result, it is reasonable to assume that either the algorithms themselves are flawed in design or that the simulators do not capture substantive information about the agent's natural environment to allow for greater generalizability of current RL algorithms. While new methods for simulation are topics of burgeoning research, new efforts to investigate the advantage of offline RL methods in generalizing training/test data sets have surfaced. Although offline methods are not responsible for active data collection, they do utilize data generated from the real world, capturing many of the parameters and complexities void in simulated environments. Based on the disadvantages of simulators and the potential benefits of offline RL methods, this project proposal focuses on evaluating the latter in training policies that can not only succeed in more diverse and differing test environments but also service two potential problems relating to Portfolio Management.

A. Portfolio Management

Portfolio management is the process of allocating capital within a fixed range of asset classes or securities in way that reduces total risk while maximizing expected return. In this setting, testing the effectiveness of RL algorithms is particularly difficult. While historical data can provide some crucial insight into the future success of any algorithm, future prices and asset allocations are often wholly unpredictable and can diverge from historical trends in significant ways. An operational investment model that relies on RL can also mitigate and manage risks associated with liquidity, increase transparency for the end user, and allow for greater personalization in financial decision-making and resource allocation. Without a technology-based operational model in portfolio management, clients are often grouped into large, impersonal investment categories, often having to settle for a one-size fits all portfolio solution that rarely

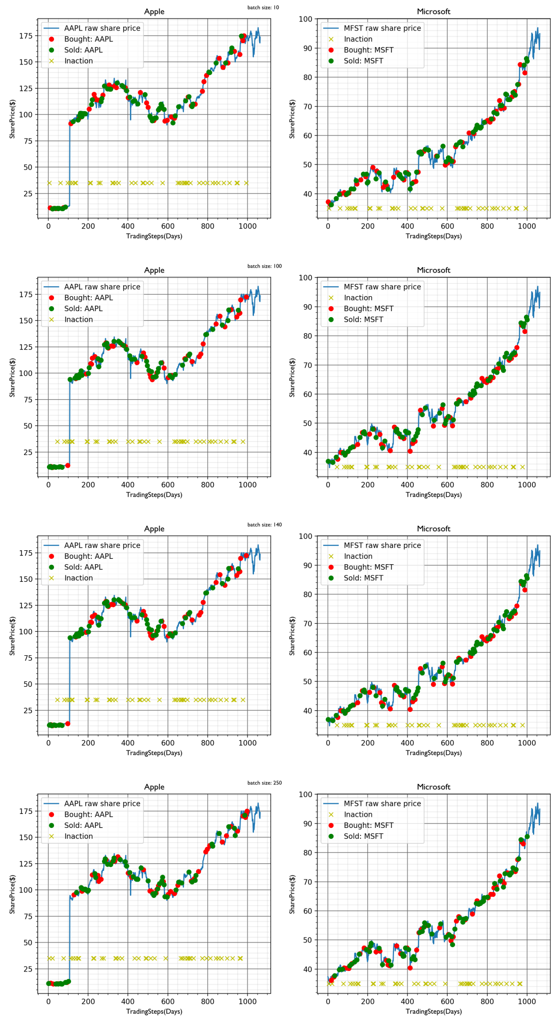

Fig. 1. Trading actions of BEAR agent for batch sizes ranging from 20 to 250 (Top to Bottom)

caters to their specific financial profile. Machine intelligence allows clients to have a transparent vision of their investments with associated lower costs.

III. METHODS

A. Data Acquisition

Stock price data ranging from January 1st 2014 to August 20th 2018 was collected for Apple (AAPL) and Microsoft (MFST) from Quandl. Both stocks had a general upward over the period. We removed the trends from the stock price data to ensure the agent would not simply just buy the stock at the beginning of the period and then sell at the end to gain a profit but engage with the environment to learn the structure of the stock price movements over time.

B. Trading Environment

In this project, we optimized a portfolio consisting of two risky assets and one riskless asset, cash. At every time step, the agent could either buy, sell, or hold AAPL or MFST. The trading environment was parametrized by the following:

(1) starting cash mean: the amount of cash the agent gets prior to trading
(2) max stride: number of timesteps until the agent retrieves the next closing price of the asset
(3) series length: the length of time the agent can trade
(4) starting point: the time at which the agent begins trading
(5) randomized cash standard deviation: standard deviation of the agent's initial cash holdings
(6) starting shares mean: the number of shares the agent gets of each stock prior to trading
(7) randomization of share deviation: standard deviation of the number of starting shares the agent gets
(8) inaction penalty: the penalty associated with not buy or selling either asset
(9) diversification bonus: bonus for having greater than 1 share invested in each asset.

The state space is a 8-dimensional space consisting of : (1) Number of shares of AAPL; (2) Number of shares of MFST; (3) Amount of Cash; (4) Current opening price of AAPL; (5) Current opening price of MFST; (6) Portfolio value; (7) Five-day window of AAPL prices; (8) Five day window of MFST prices

The action space is a 5-dimensional space consisting of: (1) 0: buy AAPL; (2) 1: sell AAPL; (3) Do nothing; (4) buy MFST; (5) Sell MFST

While this environment provides a good representation of the features an agent may need to assess when trading, it does not seem feasible to implement n number of assets, where n is large.

C. Algorithms

1) Actor Critic: Actor critic is a common RL algorithm that combines gradient-based methods with value functions. In policy gradient methods, the goal is to optimize the sum of the expected discounted long-term rewards for an episodic task:

$$v_\pi(\theta) = \sum_{t=0}^{T-1} \gamma^t R_{t+1}$$

We can express this as:

$$\sum_{t=0}^{T-1} \nabla_\theta log_{\pi_\theta} \pi_\theta(A_t|S_t) G_t$$

Gradient ascent follows a stochastic update rule:

$$\theta_{k+1} = \theta_k + \alpha_k \sum_{t=0}^{T-1} \nabla_\theta log_{\pi_\theta} \pi_\theta(A_t|S_t) G_t$$

where G_t represents the long-term discounted reward, π_θ is the policy, and α is the learning rate.

However, gradients have higher than optimal variance that can adversely affect the convergence of RL algorithms. Often, a Q function is used as an unbiased estimator to reduce the variance of the gradient estimates such that the update rule becomes:

$$\theta_{k+1} = \theta_k + \alpha_k \sum_{t=0}^{T-1} \nabla_\theta log_{\pi_\theta} \pi_\theta(A_t|S_t)(G_t - Q(s,a))$$

Here, we use actor-critic as a baseline against the bootstrapped error accumulation reduction (BEAR) algorithm.

2) Deep Q Learning: Deep Q-learning is a standard model-free RL algorithm. Given A Q-function: $SxA \longrightarrow R$, we can construct a policy that maximizes our long-term rewards:

$$\pi = argmax_a Q(s,a)$$

For large state or action spaces, a non-linear function approximator like a neural network can be useful. Then. we establish an update rule based on the Bellman equation:

$$Q(s,a) = R_{t+1}(s,a) + \gamma * max_a Q(s_{t+1}, a_{t+1})$$

However, the use of non-linear function approximators to represent the Q-function introduces instabilities in the model. These instabilities stem from correlations between the target value and the actual Q-function estimates. To arrive at an optimal policy, we aim to minimize the L^1 loss between the q-function estimate and the target Q value $y(s,a)$:

$$loss = Q_(s,a) - y(s,a)$$

To reduce these instabilities, methods like experience replay introduce a replay buffer. With a replay buffer, it is then possible to collect prior transitions $\{s, a_i, s', R(s,a)\}$ and sample from prior transitions before any updates are made. We use DQN here as a baseline to compare against the performance of BEAR.

3) Bootstrapped Error Accumulation Reduction (BEAR): BEAR is a practical off-policy algorithm that aims to reduce the instabilities present in traditional Deep Q-Learning algorithms. Q-Learning methods often fail in offline settings as a result of errors accumulating in the Bellman backup. While the optimization step constitutes a simple supervised regression problem, the Q targets themselves come from the current estimate of the Q-function. Attempting to maximize the value with these errors can lead to selecting actions for Q-estimates that lie outside of the training distribution. To solve this issue, a few other algorithm, namely BCQ, have attempted to constrain the distributional distance between the learned policy and the behavior policy. BEAR notes that this constraint is too harsh and can lead to poor performance. Instead, the author proposes matching the support of the distributions such that the learned policy $\pi(a|s)$ only has positive density where the behavior policy has density above some threshold ϵ. In theory, this should restrict out-of-distribution (OOD) actions while allowing the learned policy to still perform well in cases in which the behavior policy follows a uniform density. The primary insight of this method is to search for a policy from a set of policies Π_e that remains in the support of the behavior policy $\beta(a|s)$ to prevent accidental error accumulation in the Bellman backup. Accordingly, it uses k Q functions and the minimum value for Q-value for the policy improvement step:

$$\pi(s) = max_{\pi \in \Pi_e} E_{a \quad \pi(.|s)} [min_{j=1...k} Q_j(s,a)]$$

Then, it constrains the set of policies to those that are in the same support as the behavior policy.An MMD loss function is used to calculate deviations of the learned policy from the support of $\beta(a|s)$. If $\{x_1, \ldots, x_n\}$ represent samples from a distribution P and $\{t_1, \ldots, y_n\}$:

$$MMD(P,Q) = \frac{1}{n^2}\sum_{i,i'} k(x_i, x_i') - \frac{2}{nm}\sum_{i,j} k(x_i, y_j) + \frac{1}{m^2}\sum_{j,j'} k(y_j, y_j')$$

where k is a kernel function.

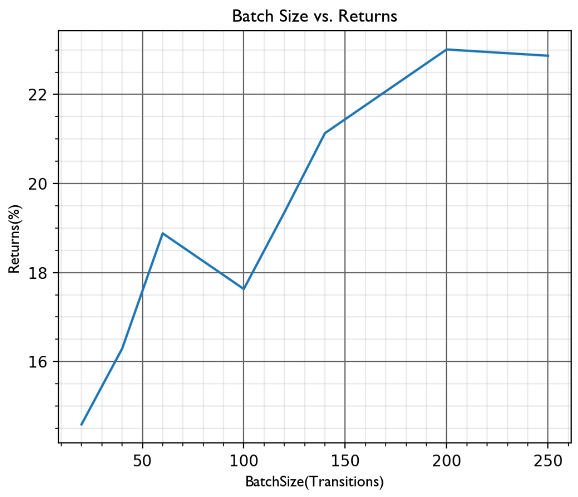

Fig. 2. Portfolio returns computed using BEAR for batch sizes=20, 40, 60, 100, 120, 140, 200 ,250

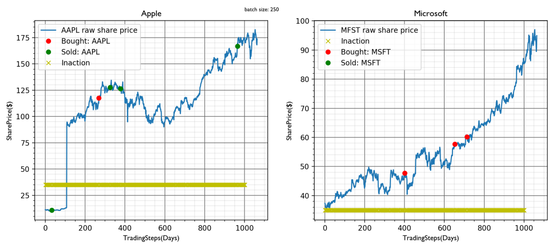

Fig. 3. Trading actions of DQN agent: Multi-Layer Perceptron with hidden layer dimension of for batch size of 250

IV. RESULTS AND DISCUSSION

*Note: each model was trained for 100 episodes. The initial parameters of the test trading environment were the following: (1) max stride = 4; (2) series length = 250; (3) starting cash mean = 100; (4) randomize cash standard deviation = 10; (5) starting shares mean = 100; (6) randomize shares standard deviation = 10.

Figure 2 shows the performance of BEAR for batch sizes ranging from 20 to 250 transitions. As the batch size increases, the portfolio returns trends upward linearly. This might indicate that as the agent works in a more offline setting, its performance increases.

Table 2 shows the portfolio allocation metrics for the Actor-Critic

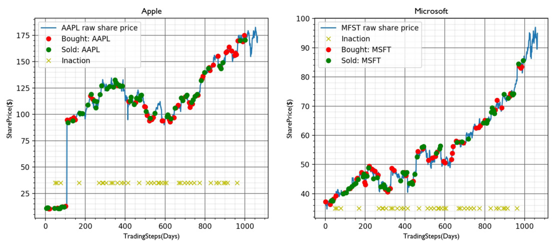

Fig. 4. Trading actions of actor-critic agent: Gated Recurrent Unit with 5-step lookback trained on trading environment with series length 250.

TABLE 1. Portfolio allocation metrics for BEAR.

Batch Size	Return (%)	End Value	Shares (AAPL)	Shares (MSFT)	Cash ($)
20	14.59	7,518	66	83	456.81
40	16.29	8,625.04	79	94	405.68
60	18.88	8702.23	86	81	582.93
100	17.63	8922.10	80	94	650.44
120	19.35	9355.81	90	97	431.09
140	21.13	9835.50	98	90	797.44
200	23.01	7811.84	93	55	558.44
250	22.87	8750.46	98	69	626.44

TABLE 2. Portfolio allocation metrics for DQN and Actor Critic baselines.

Model	Return (%)	End Value	Shares (AAPL)	Shares (MSFT)	Cash ($)
Actor-Critic	10.56	2291.51	25	22	27.63
DQN	26.24	9660.82	92		109.26

and DQN architectures. The Actor-Critic algorithm performed generally well over the time horizon, reaching a return of approximately 10.86% over the time horizon but did not perform as well as DQN when trained on a batch size of 250. During the test trading run, the agent sold most of its shares of APPL and MFST by the end and spent of its cash buying these assets in earlier time steps. DQN generated an estimated return of 33.81% over the 4-year time period. The DQN agent did not buy or sell any assets for a large portion of the test trading run. Instead, it traded very few assets at times for which the share price experienced high volatility.

While BEAR performed better than actor-critic for a series length of 250 it under-performed DQN in an offline setting for a batch size of 250. In the purely offline setting, the portfolio return reached 22.87%. In this particular environment, it does not seem that the performance of Q-Learning methods is enhanced in the offline setting by error reduction through support matching. While Figure 2 indicates that the performance of the portfolio is positively correlated with batch size, the DQN baseline still seems to perform better. It is unclear whether DQN's increased performance is a result of over-fitting or the result of the failure of support matching in BEAR to engender better performance.

V. CONCLUSION

In this paper, we explored the use of BEAR as an offline RL method in a portfolio optimization trading environment consisting of cash and AAPL and MFST shares. The support matching constraint of BEAR and the ability to prevent accidental error accumulation did not yield better performance than the standard Deep Q Learning algorithm. To further evaluate the presence of overfitting among models tested in this paper, further studies may include more rigorous validation methods in trading environments with more diverse parameters. Additionally, further hyperparameter optimization of the BEAR model architecture is necessary to verify if BEAR can in fact provide better portfolio returns than the DQN model.

REFERENCES

[1] Kumar, A., Fu, J., Soh, M., Tucker, G., Levine, S. (2019). Stabilizing off-policy q-learning via bootstrapping error reduction. In Advances in Neural Information Processing Systems (pp. 11761-11771).
[2] Fujimoto, S., Meger, D., Precup, D. (2018). Off-policy deep reinforcement learning without exploration. arXiv preprint arXiv:1812.02900.
[3] Tom Grek's GitHub repo: https://github.com/tomgrek/RL-stocktrading.
[4] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M. (2013). Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602.