Reinforcement Learning for Quantitative Trading

Sachiel Chuckrow and Johnathan Finizio

I. Introduction and Background

Quantitative trading is the process of using algorithms and data analysis to make stock trading decisions. The core goal of quant trading is to maximize returns while minimizing our risk, by making informed and safe decisions on historical and real-time data. Machine learning has emerged as the focal point of quant trading due to their abilities to create dynamic strategies that can adapt to the market in real time. Many ML algorithms have been developed and tested, including supervised and unsupervised learning, but the domain that produces the most favorable results is reinforcement learning. Reinforcement learning is a type of machine learning algorithm that trains an agent to make sequential decisions based on receiving rewards and penalties through interacting with an environment. Through deep RL, we are able to combine the perception and feature extraction capabilities of deep learning with the decision making abilities of reinforcement learning. This suits the nature of financial markets, as actions are based on many different features that help us to understand market activity.

Since data is the foundation of quant models, one must address what features to include in the trading environment. Markets can be influenced by a wide range of features, taking on a variety of different data formats, thus making trading difficult to simplify. The primary feature types to assess market trends are technical, fundamental, and macroeconomic. Technical features include metrics like volume and moving averages, that show the overall activity and trajectory of the market. Fundamental features include metrics like earnings per share (EPS) and dividend yield, describing the health of a company. Macroeconomic features affect entire markets or sectors, and can include factors such as interest rates, unemployment, and inflation. It is important to include aspects of each of these factors in any trading environment, due to the diverse market information that can be gained. While there are many numerical features that can help to understand market activity, quant trading lacks the ability to incorporate public sentiment and news into market predictions, leading to persisting challenges in an already complicated domain.

It has been established that the fluctuation in financial markets makes economic events very unpredictable. With a poor understanding of market behavior, trading behaviors are hard to generate without high volatility. The introduction of deep learning models improves our ability to generalize and predict market trends, with reinforcement learning allowing us to methodically take this information and create trading policies. Typically, good or bad policies can be determined by directly interacting with the environment, testing actions and observing their rewards. However, since our environment in this case is the stock market, testing our policy may result in extreme costs associated with poor rewards. To combat this issue, we propose the use of Offline Reinforcement Learning, that trains an agent to learn good and bad actions from an offline data set, allowing us to create well defined policies in a risk free way. While a finite dataset can create errors for "unseen" actions (actions that are not included in the dataset), offline RL algorithms have built in constraints and mechanisms that prevent the overestimation of reward values for these actions. Overall, this paper will illustrate how we created our stock trading environment, the benefits of using offline RL over other machine learning techniques, and the changes we made to enhance these algorithms.

II. Problem Statement and Motivation

Our goal is to experiment with CQL and BCQ algorithms in the context of reinforcement learning for quant. Ideally, we will create a model which is able to outperform the market in the long term. This idea is motivated by a lack of literature on using constrained learners in the context of offline stock trading environments. We feel that experimenting with this usage could prove to be effective as CQL and BCQ algorithms do a relatively good job of solving overestimation issues that plague offline learners, particularly in high risk environments like quant trading. In working with these models, hopefully we will get a better understanding of how constrained learners are able to combat overestimating in what is one of the most popular applications for not just RL but ML in general. Getting an idea for how constrained learners like CQL and BCQ act in these environments, namely how they test versus train can provide valuable insights.

III. Literature Review

Reinforcement learning has been experimented with for stock trading purposes since the early 2000s. Various models have been implemented with the common goal of predicting market trends to create trading policies. The shift from using supervised learning methods to reinforcement learning has opened a new door for the field of quant trading. In the 2001 paper “Stock Price Predictions Using Reinforcement Learning”, Jae Won Lee illustrates that traditional stock prediction models using supervised learning struggle with long-term goals with delayed rewards, making them suboptimal for trading environments. Lee attempts to combat this issue by incorporating RL, modeling stock prices as a Markov Decision Process and implementing Temporal Difference Learning to estimate state values. The state is represented by a vector of numerical stock features that allows the model to learn from more trends aside from just the price, including factors like volume or volatility. Features similar to Lee’s are ones that we will incorporate in our model as well. Due to the infinite number of states in this environment, Lee implements a neural network to estimate the value function. He grouped the predicted state values in 16 different grades, to determine if his model accurately predicts which states lead to higher rewards. As a result, higher predicted grades led to higher actual returns, showing that his model was picking up on the correct trends. However, most test examples fell into the mid-grade range, with the model lacking confidence to pick extreme positive or negative values. Another limitation was that the model only worked well in a window of about 5-10 days, with very short or very long time windows having weak performance. Despite these setbacks, Lee’s model showed promising signs that reinforcement learning could be applied to the stock market, giving us a very good starting point for our research.

While reinforcement learning is the clear choice in creating market policies, it is also important to consider how the properties of deep learning can be leveraged for market analysis. In *Forecasting S&P 500 Index Using Artificial Neural Networks and Design of Experiments,* Niaki and Hoseinzade experiment with artificial neural networks to predict the S&P 500’s value. This was done primarily to figure out what factors were most important to the S&P 500’s value and growth. The paper finds that exchange rates, particularly in the short term, greatly influence economic progress and stallation. The paper found that using an ANN can outperform the market, which serves to make our goal of outperforming the market more realistic, as ML methods have shown success when fed the right parameters. Since our model is so different, we are using different parameters, but experimenting with these could prove useful in the future.

An automated FX trading system using adaptive reinforcement learning by Dempster and Leemans introduces adaptive reinforcement learning into financial markets, namely foreign exchange. The paper focuses on a model based off of the RRL algorithm designed in 1999 by Moody and Saffell. The RRL algorithm is discretized, as only one unit of currency can be traded at a time, being bought or sold. This algorithm is simple and thus uses a gradient ascent optimizer, improving parameters via $w_{i,t} = w_{i,t-1} + \rho\Delta w_{i,t}$. The model proposed by Dempster and Leemans adjusts this model by layering it, using a three layer setup. The first layer is the RRL, which outputs preferred position in the model, while the second layer evaluates the recommendation in accordance with other factors. The second layer thus implements a trailing stop-loss function which is set so it is within a range of the best price ever reached during the lifetime of the position. This range is defined in the third layer, which searches for optimal values of parameters passed into the first two layers, maximizing the risk return profile of the model. This risk-averse, tunable model did work, and showed how while riskier policies often lead to better results, increasing risk-aversion did make sure that the model did not fall into any pits and lose lots of money, which is ideal for financial environments. This paper helped as a foundation for our online BCQ agent, which is a similar algorithm in that it has built in tunable risk aversion. In experimenting with this model, ideally we can further research into how building in risk-aversion to financial models can help produce a more stable model.

The incorporation of Deep Reinforcement Learning takes automated stock trading a step further. It incorporates the perception and feature extraction abilities of deep learning, while also including the decision making abilities of reinforcement learning. In the paper, “Application of Deep Reinforcement Learning in Stock Trading Strategies and Stock Forecasting”, Li et al. use Deep Q-Learning (DQN) to approach stock trading strategies in a new way. 10 stocks were randomly selected for training, and three deep RL methods, DQN, Double DQN (DDQN), and Dueling DDQN were applied. The goal of DQN is to approximate the Q-function (expected reward for a state/action pair) using a neural network. It stores previous experiences in a replay buffer that is sampled during model updates to improve the policy. DDQN fixes overestimation bias in DQN, that is caused by the same network selecting and evaluating actions. This is accomplished through separate selection and evaluation networks, leading to more stable and accurate estimates. Lastly, Dueling DDQN splits up the DDQN Q-network into two streams: value and advantage. The value stream tells us how good it is to be in a certain state, whereas the advantage stream tells us how good each action is. This helps to learn which states are valuable, even when actions don't differ much. The performance was evaluated through profit metrics, reward functions, price forecasting, and other technical indicators. Dueling DDQN performed best out of all three methods, generating the highest profit across the randomly selected stocks. This is most likely due to the fact that the architecture is suitable for a stock trading environment, with many different states and similar actions (buy/sell). Overall, Dueling DDQN further enhances the progress of applying RL to stock trading. The pitfall of DQN variants is the fact that they lead to "action distribution shift", meaning that the Q-values of actions not represented in the dataset are often overestimated, leading to suboptimal policies in the online environment.

This brings us to the main challenge of offline RL for quant trading: There are infinitely many state action pairs, but we have finite data. As a result, we cannot verify estimated Q-values if the state-action pair is absent in the dataset, leading to the action distribution shift. This is where Conservative Q-Learning (CQL) and Batch Constrained Q-Learning (BCQ) come into play. In the paper, "Conservative Q-Learning for Offline Reinforcement Learning", Kumar et al illustrates this concept of action distribution shift. In standard RL, this overestimation can be corrected by interacting with the environment, whereas in offline RL, this correction is impossible due to the lack of data from the environment for every state and action. Conservative Q-learning constrains the learned policy to stay close to the behavior policy, to improve stability and reduce overestimation. The CQL framework penalizes actions not supported by the dataset by adding the regularization term, lower bounding the Q-value. Also, instead of fully evaluating each policy iteration, the algorithm uses the Q function to approximate the best value at each step. This method is suitable for our project, since directly interacting with the stock market has very severe consequences. We cannot take on the risk of overestimating values for specific actions, because in reality these actions could cost us a lot of money.

An alternative approach to take in this project is applying BCQ. The primary difference between BCQ and CQL is their method of approach. CQL is a direct method, in which it uses the Q-value plus a penalty to directly adjust the Q-values. BCQ is an indirect method, in which it uses a VAE or GAN to generate likely actions from the dataset and uses the Q-network to pick the best candidate among all of the actions. This constrains the policy to stay near the dataset, allowing us to estimate the Q-values with better accuracy. We will test both methods to see which performs best on stock data. Overall, these techniques for offline RL will be the primary tool for creating our stock trading agent.

IV. Methods and Implementation

A. Environment

Our environment is initialized with a list of randomly selected stocks, their prices, which are updated at each timestep, dividends for each stock, which at each timestep is assigned to the last reported dividend, volume of each stock, interest rate for 10 year treasury bonds, held cash, and S&P 500 data. We normalized all features to ensure stable training and accurate performance. We also experimented with only using dividends and stock prices, which proved to give worse results fairly uniformly. Notably, the environment performed poorly when dividend values were not updated to be the most recent value at the end of each timestep. In cases where there were significant zero or NA values, both of our models diverged with regularity. Our state is then initialized with a starting amount of cash, all held.

At each step, which equates to a real life day, the model takes an action. Our action space is continuous, as our model is able to buy and sell stocks at will, as long as it has the funds to allocate. The model reallocates the stocks in its portfolio at day end, by selling and purchasing. After the action is taken, prices are updated with the value of the stocks at the end of the day (so close prices), and passed into our environment. Holdings are updated to reflect the new prices of each stock, calculated as *holdings* × *prices*. Additionally, trades are counted, calculated as

|*new holdings - current holdings*| × *prices*

done in order to help the algorithm learn not to trade when it does not need to. The trade count is then stored in a cost value (C), which is calculated by total trades times some constant defined as a hyperparameter. Our Portfolio value (P) is calculated as total holdings plus total cash held. While typically the reward function is simply the increase in portfolio value, we define our function in a slightly different way to improve stability. Reward is calculated as a function of portfolio value and S&P 500 Value (B):

$$\left(\frac{P_n - C_n}{P_{n-1} - C_{n-1}} - 1\right) - \left(\frac{B_n}{B_{n-1}} - 1\right)$$

This equation represents the percentage change in our portfolio value, subtracted by the percentage change in the S&P 500. This definition helps to make sure that the model is performing with the goal of consistently out-performing the market. This is useful because it makes sure that gains and losses are weighed against market trends: a 1% gain during a period of decline is often worth more than a 1% gain when the market is on an upswing.

B. Models

i. Linear Regression

A linear regression can be used as a baseline analysis for determining the complexity of financial markets. We construct our regression model using the features price, volatility, previous dividend, and interest rate, to try and predict the stock price for the next day. Through our regression results, one is able to observe whether or not a more complex, deep learning model is needed to analyze market trends. It is important to always consider simple models first, due to the concept of Occam's razor: simple models generalize better than complex ones with the same performance.

ii. Random Forest

Random forests are additionally going to be used as a baseline in order to show how fundamental methods do at predicting stock prices. We will use the random forest to learn trends from a stock, and attempt to predict whether it will go up or down on the next day. This will also serve as a simplistic model, to make sure more complicated models are actually necessary in the first place.

iii. DQN

A Deep Q-Network combines Q-Learning with deep learning, to learn good actions in high dimensional spaces. The DQN will serve as a baseline for comparison with our offline RL models. It is a type of online reinforcement learning algorithm constructed of a neural network to estimate Q-values. A Q-value is the cumulative reward the agent can expect to receive if it follows a certain action, a, in state, s, and follows the optimal policy for subsequent actions. This value is learned using the Bellman equation:

$$Q(s,a) + \alpha \left(R + \gamma \max_{a'} Q(s',a') - Q(s,a) \right)$$

We take the current Q-value, $Q(s,a)$, and add the temporal difference error, scaled by our learning rate α. The temporal difference error is the difference between the target and the current Q-value, ensuring that our adjustment updates the Q-value correctly. Our Q-network gets updated through the use of experience replay, which samples batches of previous experiences to stabilize training and remove correlations between consecutive experiences. Lastly, actions are selected using ϵ-greedy decay, prioritizing exploration in the beginning of training, and exploitation by the end. Actions must be discrete for a DQN, so we discretized our action space for this model.

iv. Conservative Q-Learning:

The Conservative Q-Learning (CQL) algorithm is an offline reinforcement learning algorithm, based on Deep Q-Learning (DQN), that penalizes out of distribution actions to reduce overestimation. There are two main types of CQL algorithms that we implemented: Actor-Free, and Actor-Critic.

Actor-Free

The Actor-Free variant of CQL has a similar network architecture to DDQN, containing two Q-networks to reduce overestimation bias and directly learns from an offline dataset of expert actions. We collected expert data by implementing a simple momentum trading strategy for historical data, that prioritizes stocks with good recent performance. While Q-networks typically require the use of a discrete action space, we combat this issue by using a separate behavioral cloning neural network to generate the actions for our network to evaluate, feeding both the state and action into our Q-network. So, the policy is derived by guided Q-learning, using actions sampled from behavioral cloning. As previously mentioned, the purpose of the CQL algorithm is to learn policies from static datasets while preventing the overestimation of actions out of our distribution. This is accomplished by penalizing Q-values of state-action pairs not supported by the dataset, during the update of our network.

$$\underbrace{\text{Bellman Error}}_{\text{standard Q-learning}} + \underbrace{\alpha \cdot \left(\mathbb{E}_{a\sim\pi(\cdot|s)} \left[Q(s,a) \right] - \mathbb{E}_{a\sim\mathcal{D}} \left[Q(s,a) \right] \right)}_{\text{conservatism penalty}}$$

The loss is described above, using the conventional Bellman Error and our regularization term. The regularization term encourages the Q-function to assign lower values to state-action pairs that are unlikely under the dataset. Our hyper parameter α, controls the strength of the penalty for unlikely actions. A higher α controls overestimation for out of distribution actions, whereas a lower α allows the model to generalize better. Therefore, tuning this hyperparameter is critical to performance.

Actor Critic

The Actor-Critic structure contains both a selection and evaluation network for actions. Due to this structure, we are able to incorporate our desired continuous action space.

Actor: $\pi_\phi(a \mid s)$ - A policy network that outputs a softmax distribution for actions

Critic: $Q_\theta(s,a)$ - A Q-network that estimates the value for a state/action pair

Instead of directly learning on expert data, we learn from a dataset of random actions to remove the need for a prior trading policy. Our algorithm is also free to explore other actions that may not be a part of the "expert's" behavior, due to the fact that the actor directly improves its policy solely based on maximizing rewards, guided by the critic. The critic performs updates in the same way as the Actor-Free model, using the Bellman error with a conservatism penalty

$$\underbrace{\text{Bellman Error}}_{\text{standard Q-learning}} + \underbrace{\alpha \cdot \left(\mathbb{E}_{a\sim\pi(\cdot|s)}\left[Q(s,a)\right] - \mathbb{E}_{a\sim\mathcal{D}}\left[Q(s,a)\right]\right)}_{\text{conservatism penalty}}$$

However, the actor is updated strictly using policy gradients. Our change to the typical Actor-Critique structure is a modified objective function, now incorporating an entropy bonus. Our proposed modification further mitigates investment risk, promoting the creation of a diverse portfolio. If we rely heavily on an out of distribution action (or even a supported one) and "put all of our eggs in one basket", this could be costly if the action was suboptimal. In the equation below, we directly add an entropy term to our actor objective, to adjust how much our model prioritizes diversification.

$$\max_{\phi}\ \mathbb{E}_{s\sim\mathcal{D}}\left[Q_\theta(s,\pi_\phi(s)) + \beta\,\mathcal{H}(\pi_\phi(\cdot\mid s))\right]$$

As a result, our model is able to select more stable actions that reduce overall risk.

v. Batch-Constrained Q-Learning:

Batch Constrained Q-Learning (BCQ) is defined as a Q learner which has a batch of actions it can choose from, rather than an infinite set. This batch of actions is generated by sampling likely outcomes from a pre-generated set of actions. In our case, we progressively generate our action set, where known actions are 'cached' and stored as potential actions to take. A BCQ then at each step, decides on a next action by judging the most likely actions, and selecting a batch of them. Then, a perturbation network randomly shifts the actions to create new actions so the agent can learn. The BCQ algorithm contains four neural modules:

Conditional Variational Auto Encoder (VAE): The VAE models the distribution of actions in our state space. From state s and action a in the dataset, the VAE produces parameters μ and σ, which are used with a decoder in order to reconstruct $\hat{a} = \text{decode}(s,z)$ from latent $z \sim \mathcal{N}(\mu,\sigma^2)$. VAE thus helps to minimize reconstruction loss plus KL divergence. This gives us loss function :

$$L_{VAE} = E_{(s,a)[\sim\mathcal{D}\ ?]}\left[\|a-\hat{a}\|^2\right] + \beta D_{KL}(\mathcal{N}(\mu,\sigma^2)\|\mathcal{N}(0,I))$$

Where β is our weighting factor. Once our training is complete, the decoder,

$$\hat{a} = \text{VAE.decode}(s,z) \text{ with } z \sim \mathcal{N}(0,I)$$

can sample what are essentially likely (or feasible) actions from historical policy. So if only a VAE is used, we can only take actions previously seen. In order to break this we come to our next neural module.

Perturbation Model (actor): The perturbation model is essentially our actor. The BCQ trains a deterministic policy matrix first, which 'jitters' or perturbs the VAE, allowing it to sample outside of known states and actions. The actor, given state s and a decoded action, will output

$$\tilde{a} = a + \xi_\phi(s,a) \text{ with } |\xi_\phi|\infty \le \phi,$$

Where the action is determined from likely, possible actions already in the dataset. These actions are then adjusted to form the new actions which the actor can take. The actor is trained by the deterministic policy gradient, maximizing $Q_1(s,\tilde{a})$. This model will thus gently nudge our model towards better solutions without making strange leaps which are prone to overestimation.

Q Networks (Critics): BCQ, notably uses two Q functions to evaluate state-action values, following a clipped double learning update. Given a batch *(s,a,r,s')*, the target value is computed by first sampling candidate next actions, a'_j, produced from the VAE and Perturbation actor. The target value is given by:

$$Q^j_{targ} = \lambda\min(Q'_1(s',a'_j), Q'_2(s',a'_j)) + (1-\lambda)\max(Q'_1(s',a'_j), Q'_2(s',a'_j))$$

where $0 \le \lambda \le 1$ biases towards the minimum. The algorithm then takes the maximum over candidate actions to form the Bellman target. Critics are then updated by minimizing:

$$L_Q = \tfrac{1}{2}\left[(Q_1(s,a)-y)^2 + (Q_2(s,a)-y)^2\right].$$

This update is only done with actions from the batch for current values, restricting bootstrapping to the VAE/Perturbator for next actions. This form of double critics helps to reduce overestimation.

When evaluating, with new state *s,* the BCQ samples a batch of N vectors to generate new actions via the decoder from the VAE. After this, it perturbs each action to $\tilde{a}_i = \pi(s, a_i)$, selecting the best action via $\tilde{a}^* = \arg\max_i Q_1(s, \tilde{a}_i)$. This then gives us the best possible action similar to actions in or similar to the dataset. Training consists of sampling our batch from replay, using VAE to generate actions and compute Bellman target, then updating the critics. After all this, the actor policy is updated to maximize Q_1. Target networks are updated:

$$Q_i' \leftarrow \tau Q_i + (1-\tau) Q_i'$$
$$\pi' \leftarrow \tau \pi + (1-\tau)\pi'$$

Ideally, over many iterations we may converge to a high yield policy while staying within bounds.

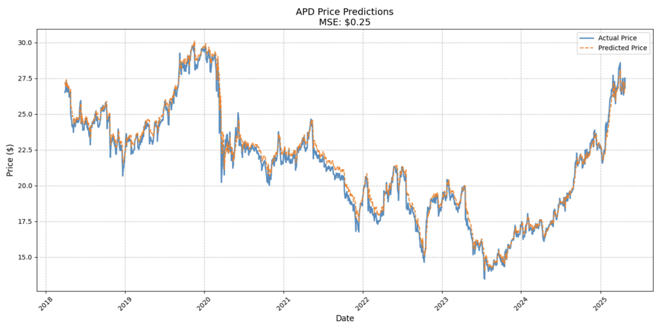
Our primary implementation of BCQ involves training a policy over an episode, then updating our agent to learn after the episode finishes. This then continues, with the agent generating new actions which are stored within the replay buffer at the end of each iteration. In this way, the model becomes its own expert. This application is not offline learning, which is traditional of BCQs, but also is different from much off-policy online learning in that the model is only updated at the end of each episode, rather than learning as it explores. Using a BCQ in this format allows us to generate unique actions that only an independent learner could create, while working within boundaries that keep the model from overestimating while training and producing failing results during testing. The model serves to maintain conservative actions, avoiding ‘crashes’ that other models face, often losing much if not all of their money when they take a poor action. In this way, we have a slow-updating online BCQ learner, which allows us to leverage the benefits of exploration that a DQN has while also maintaining the safety and conservation of a BCQ. This implementation, which we will refer to as ‘partially online’ was effective and will be discussed further in results. Additionally, we trained and tested a pure offline BCQ learner for comparison. This agent learns solely from a generated batch of random actions.

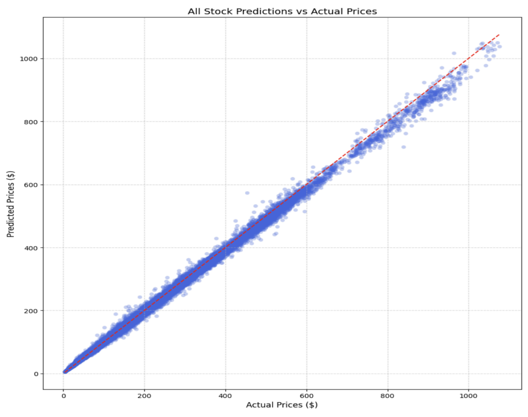
V. Experimental Results

A. Supervised Learning

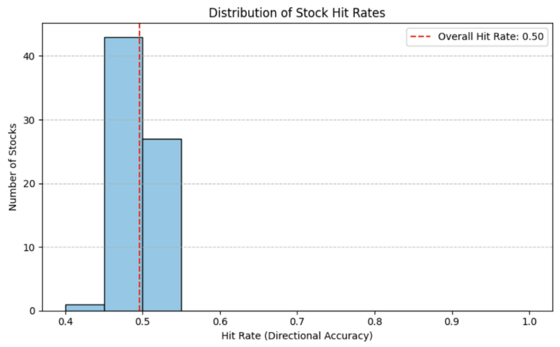
i. Linear Regression

At first glance, our linear regression model seemed to output strong results, with predicted prices following similar trends as the actual stock prices. The following graphs show the comparison of actual versus predicted prices for a randomly selected stock (APD), and the correlation between all actual and predicted prices in our testing set.





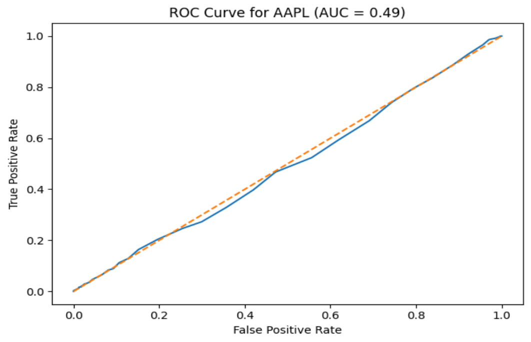
While these plots show strong similarities between predictions and actual values, we must examine our “hit rate”, which is the probability that we correctly identify a price increase or decrease. This metric is critical when decision when to invest in a stock.



This graph demonstrates that our model does not do a good job of predicting the direction of a stock price, with the result being slightly worse than a 50/50 gamble. Therefore, we should turn to deep learning for capturing market trends.

ii. Random Forests

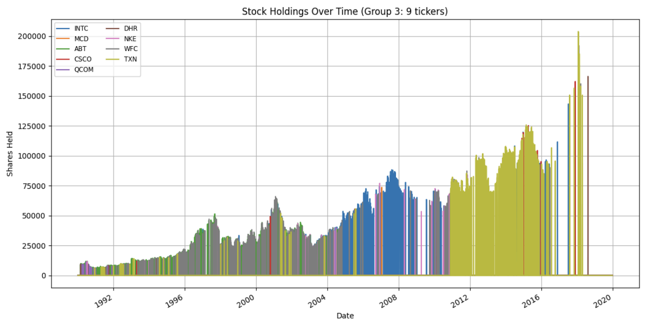
Random forests performed comparably to linear regressions, although without the feigned benefits.



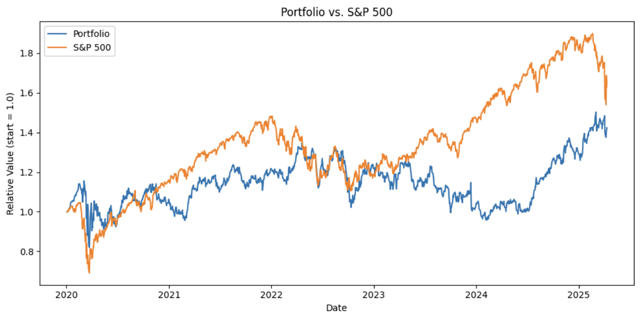
As shown in the graph, the regressor was right less than half of the time. The forest actually just predicted that the stock would always decrease, artificially inflating its accuracy. Again, we move on to more complex methods seeking better results.

B. DQN

The DQN produces suboptimal results, failing to converge well in training and incorporating risky policies. The figure below shows the stock holdings for nine of our stocks throughout the period.



The graph shows how our policy allocates the majority of portfolio value to a single stock, leading to risky investments and poor performance. This policy did not perform well on the test data, underperforming compared to the S&P 500.



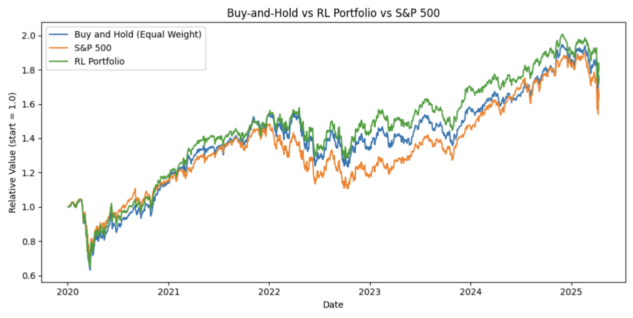
The graph shows how our DQN portfolio is consistently worse than the S&P 500 values. This error can be attributed to the overestimation of Q-values and limitations of a discrete action space, that we hope to combat with our offline RL methods.

C. CQL

All test data was calculated using 71 randomly selected stocks from the S&P 500. This was to maintain our original state shape, while also testing model generality.

i. Actor Free

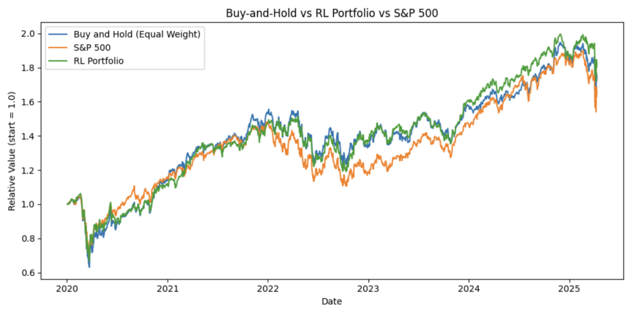
The Actor-Free model produced decent results, with our model's portfolio (80.7% return) outperforming both a buy and hold strategy (74.7% return) and investing in the S&P 500 (65.8% return).



The performance relies heavily on what comprises our expert data. In this case, we use a risk-adjusted momentum strategy, that selects the top 10 stocks with the highest sharpe ratio and allocates 95% of the portfolio to the stocks, with the rest allocated to cash holdings. Despite strong performances, our expert's policy puts a constraint on our maximum profit, since the model cannot deviate too far from expert actions.

ii. Actor Critic

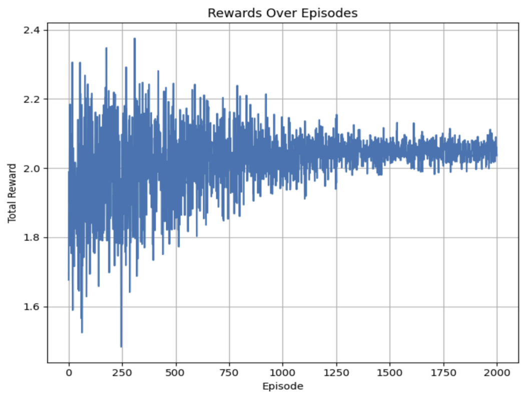
The Actor-Critic model produces slightly better results, achieving a return on investment of 81.7% compared to the 74.7% return for a buy and hold strategy and the 65.8% return of investing in the S&P 500.



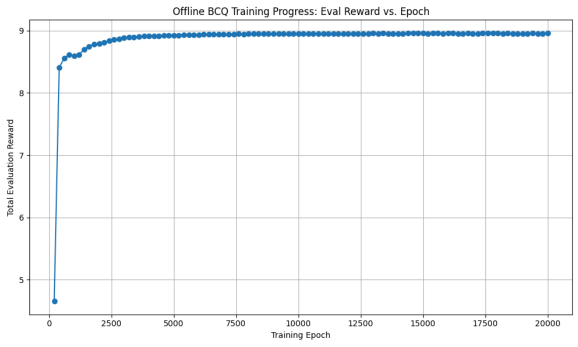
The Actor-Critic results in a 1% higher return compared to the Actor-Free model, which may be attributed to the fact that it is not restricted to an expert dataset, but rather a random set of actions in the space. This allows for the agent to deviate from the dataset (with the conservatism penalty) in search of higher rewards.

D. BCQ

The BCQ agent was trained on data from 1990 to 2019 and tested from 2020 to 2025. The model was trained on 71 randomly selected stocks, all from the S&P 500. The BCQ trains slower than CQL implementations, but learns quickly. Training performance was high, but of note the portfolio value does widely follow market trends, so the model did not learn to bulk cash out when the market is in recession, even in training. Training was done with epsilon decay, which worked best when the model had time to learn with simulate with both high epsilon values and low epsilon values, so decay took around half of the time for the learner while the rest was done with very low epsilon values. Learning curve can be seen here for the partially online and fully models:



Partially Online Model

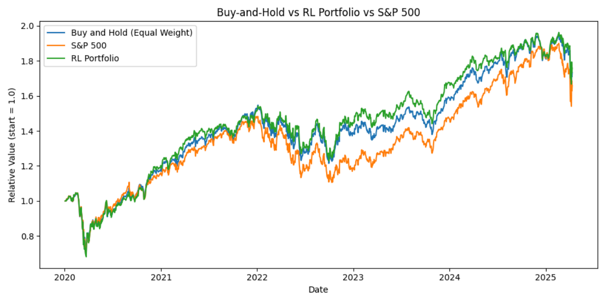


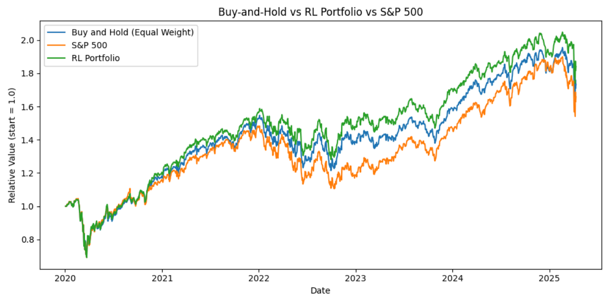
Fully Offline Model

Of note, the two models are graphed differently: The partially online model graphs by training reward while the offline model is evaluated at intervals and graphed that way. This is done because the offline model trains and tests much faster than the offline model, allowing for this graph above.

In order to test, we set up our own online environment where the model can act and learn on new data it has not seen before. We experimented with testing on the pre-trained stocks as well as new stocks for both the partially online and fully offline models, to varying degrees of success. When testing on the stocks which were initially passed into training, the model performed well, outperforming the market usually, as well as usually outperforming buy and hold strategies. This was true for both the partially online model as well as the fully offline model, with similar results when testing on original training stocks:

Partially Online



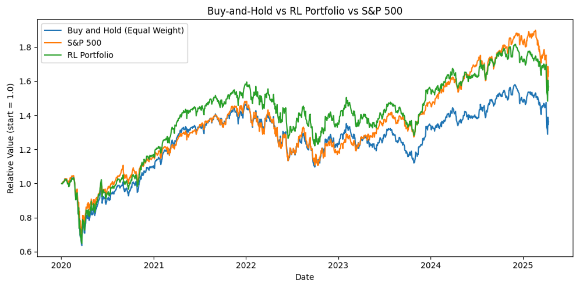


Fully Offline

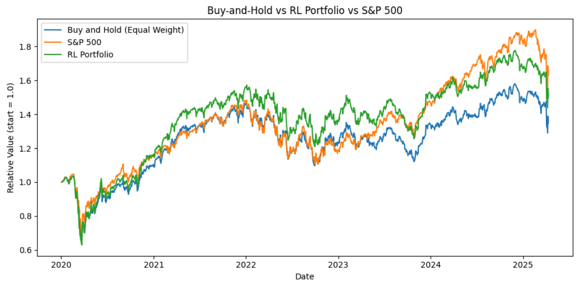
These results do seem promising, with both models looking similar. This would seem to favor the offline learner since it is much faster and requires less compute, but testing on different stocks than training revealed interesting insights. Both learners were sensitive to quality of input stocks, where the online model was more volatile, having higher highs and lower lows, although both models were relatively steady, at most falling 9% behind the market over five years, with sub-optimal stocks. On our testing sets, both models perform comparably well, with the offline model slightly outperforming both the buy and hold strategies, as well as the online model. We saw maximum returns over this period as 116.45% for the online model, with the offline model clocking in at around 120.20%, both being 1.45%

and 5.65% better than the market. In this sense, the offline model seems to be able to take advantage of high performers better, although this benefit may be explained by a much faster runtime and thus many, many more episodes possible for training.

One experiment which seemed particularly interesting and highlights the benefits of a BCQ in stock trading, is the ability to mitigate poor performers. While the BCQ usually performs better than the buy and hold strategy, it performs much better when poor performers are selected. This highlights the benefit of BCQ algorithms, being conservative, able to mitigate the effects of strong downwards trends in the market. Since the BCQ usually slightly outperforms the market as well as buy and hold strategies when good performers are selected, and it mitigates poor performers, we find it likely that the BCQ is able to outperform the market as well as its selected stocks in the long run. In this case, we see the online model outperform the buy and hold case by around 25%, while underperforming the market by around 10%, while the offline model outperformed buy and hold strategies by closer to 20%.



Online: Poor Performing Stocks



Offline: Poor Performing Stocks

The performance of the online BCQ in this environment is promising and shows why implementing BCQ into high risk, online environments can be a useful application of the technology. While compute remains much more difficult, taking orders of magnitude longer to train compared to the offline model, results show that the added exploration of the online learner is beneficial in reducing risk from stock market crashes, and could be useful in other high risk environments when you cannot afford to take risky decisions. The offline learner, however, performs comparably and is easier to train, and may provide greater gains in the long term. In the implementation, we see how the conservative nature of a BCQ, and reliance on past actions make it much less likely to fall into pitfalls of poor performing stocks, making it a very low risk option. More testing would be needed in order to fully evaluate the potential risks associated with the BCQ, but we feel confident in saying that there is a very low risk of the model crashing and suffering financial losses beyond 15% over a five year period. Exploring how the model acts over longer periods of time could also be enlightening, as the recent market crash associated with the Trump Administration's tariffs has led to our model crashing significantly, and it remains to be seen how our model reacts to crashes that are this rapid. More research could also be done into why stock choice matters, and what types of stocks perform the best in regards to the model.

VI. Analysis and Discussion

Our experiment results clearly show that offline methods outperform traditional supervised learning methods and DQN. Linear regression and random forests seemed to capture market variability, when in reality they struggled to capture market direction. Both of their hit rates were roughly random, serving as a bad foundation for making financial decisions.

Naive DQN underperformed due to many compounding factors. First, the discretized actions space constricted the policy, forcing the agent to create risky portfolio decisions through rigid allocation decisions. DQN is also prone to Q-value overestimation, even in an online environment, due to the high dimensional space and sparse rewards that exist in our environment. As a result, the learned policy was extremely volatile, and easily out performed by a "buy and hold strategy" or by investing in the S&P 500.

As for our offline methods, both the CQL and BCQ showed success in adapting to market trends, creating safe, profitable policies. The Actor-Critic framework for the CQL slightly outperformed the Actor-Free variant, with the difference potentially being attributed to less confined action selection. By penalizing out of distribution actions, we can clearly see the control in overestimation, stable policies. BCQ is notable for just how safe it is as an investment tool, being able to mitigate losses from underperforming stocks. Regarding the online versus offline versions, the online learner is slightly more conservative, gaining less but also losing less. The offline learner makes more aggressive actions, which often leads to greater gains, and lower losses. This makes both options viable depending on compute capabilities. Overall, both the CQL and BCQ create stable yet profitable policies across all variations, making them suitable for quantitative trading.

VII. Conclusion and Future Work

Overall, we see how using offline methods for financial environments makes sense for a number of reasons. Financial markets often cannot be trained on in real time, and exploring can be dangerous. In addition, the natural tendency for both CQL and BCQ to be conservative in their policy leads to an ideal matching for environments where taking large amounts of risk could prove to be dangerous. More testing will need to be done in order to evaluate which of our two offline models is better, as both have their benefits and drawbacks. The CQL is much faster to run, and provides an implementation which is fully able to exploit high value stocks. The BCQ environment is more conservative, which leads to a more stable portfolio, with the associated benefits and drawbacks. In the future, more testing would be beneficial in order to fully understand the full extent of how these algorithms could mitigate risk financially. The incorporation of more features and recent data in the training set could also lead to better results, especially considering the current economic climate. Overall, offline reinforcement learning contains many possibilities in the world of quantitative training.

VIII. Appendix

Here are stock tickers used for training. Selection was based on data availability from 1990-2020.

AAPL, MSFT, UNH, LLY, JPM, JNJ, XOM, WMT, PG, CVX, HD, MRK, COST, PEP, ADBE, KO, BAC, ORCL, INTC, MCD, ABT, CSCO, QCOM, DHR, NKE, WFC, TXN, AMD, NEE, AMGN, PM, HON, UNP, UPS, MS, LOW, BA, IBM, CAT, MDT, GS, GE, DE, T, LRCX, ADI, CI, SYK, MU, SCHW, ADP, MMC, BDX, PFE, ADSK, SO, PGR, TGT, AXP, AON, SLB, CL, APD, AEP, CSX, F, GM, FDX, DG, NSC, ITW

IX. References

BY571. “CQL/CQL-DQN/Agent.py at Main · BY571/CQL.” *GitHub*, 2021, github.com/BY571/CQL/blob/main/CQL-DQN/agent.py. Accessed 1 May 2025.

Deng, Yue, et al. “Deep Direct Reinforcement Learning for Financial Signal Representation and Trading.” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, no. 3, Mar. 2017, pp. 653–664, https://doi.org/10.1109/tnnls.2016.2522401.

Gao, Yuanqi, et al. “Batch-Constrained Reinforcement Learning for Dynamic Distribution Network Reconfiguration.” *IEEE Transactions on Smart Grid*, vol. 11, no. 6, 1 Nov. 2020, pp. 5357–5369, ieeexplore.ieee.org/document/9126832/, https://doi.org/10.1109/TSG.2020.3005270. Accessed 13 Aug. 2022.

Jae Won Lee. “Stock Price Prediction Using Reinforcement Learning.” *ISIE 2001. 2001 IEEE International Symposium on Industrial Electronics Proceedings (Cat. No.01TH8570)*, 2020, ieeexplore.ieee.org/abstract/document/931880, https://doi.org/10.1109/isie.2001.931880. Accessed 9 Apr. 2020.

Jiang, Zhengyao, et al. “A Deep Reinforcement Learning Framework for the Financial Portfolio Management Problem.” *ArXiv.org*, 2017, arxiv.org/abs/1706.10059.

Kumar, Aviral, et al. “Conservative Q-Learning for Offline Reinforcement Learning.” *ArXiv.org*, 19 Aug. 2020, arxiv.org/abs/2006.04779.

Li, Yuming, et al. “Application of Deep Reinforcement Learning in Stock Trading Strategies and Stock Forecasting.” *Computing*, vol. 102, no. 6, 23 Dec. 2019, pp. 1305–1322, https://doi.org/10.1007/s00607-019-00773-w.

Mnih, Volodymyr, et al. *Playing Atari with Deep Reinforcement Learning*. 19 Dec. 2013.

sfujim. “BCQ/Continuous_BCQ/BCQ.py at Master · Sfujim/BCQ.” *GitHub*, 2018, github.com/sfujim/BCQ/blob/master/continuous_BCQ/BCQ.py. Accessed 1 May 2025.

Xu, Maochun, et al. “Deep Reinforcement Learning for Quantitative Trading.” *ArXiv.org*, 25 Dec. 2023, arxiv.org/abs/2312.15730.

Yasin, Alhassan, and Prabdeep Gill. “Reinforcement Learning Framework for Quantitative Trading.” *Arxiv.org*, 2019, arxiv.org/html/2411.07585v1#bib.bib3. Accessed 1 May 2025.