CS 182 Final Project Report
Automated Trading: Model-Free Reinforcement Learning Approaches

Ben Barrett, Rangel Milushev and Rahul Naidoo

December 8, 2017

1 Introduction

We have created three reinforcement learning (RL) trading agents which trade a single asset. Our guiding objective has been to develop a trading agent which at a minimum exhibits rational behavior,[1] and more aspirationally generates returns on some initial investment that exceed the returns that would be attained by a human investor pursuing a buy-and-hold strategy (that is, purchasing and subsequently holding an asset over some period). Trading is a field rife with opportunity for automation given the rapid pace of financial developments, which sometimes exceed the attention capabilities of humans, and given the potential for productivity improvements from attaining human-equivalent or improved investment results with less human involvement.

Our trading agents are based on model-free approaches and thus do not require any estimation: we build a tabular SARSA agent, a tabular Q-learning agent, and an approximate Q-learning agent. All of these are derived from Bellman's dynamic programming equations, but each estimates and updates the value of a state-action pair in slightly different ways. In general, all three algorithms iteratively receive a sample of the transition and associated reward, and using the sample update their estimation of an action's value in a particular state, $Q(s, a)$, through linear interpolation [7]. As time passes and more samples are considered, under certain conditions the estimates of $Q(s, a)$ generated by tabular SARSA and tabular Q-Learning provably converge to true value $Q^*(s, a)$ [6], while in the case of approximate Q-Learning given sufficiently informative features the true value $Q^*(s, a)$ is approximated. Thus at least in theory, the algorithms we use allow us to find the true values of state-action pairs such that our trading agents can select objective-maximizing actions in any given state.

The algorithms we use in our project lend themselves particularly to our chosen application because the complexity of the financial markets means that model-based reinforcement learning methods quickly become intractable. In most cases the probability of arriving in one state from another state, for example the probability of an asset appreciating at a particular moment in time, cannot be accurately predicted, and moreover even if such probabilities were possible to predict

[1] How we might define rational behavior in this context is itself a complex question, but one might think for example that selling an asset before it depreciates further or buying an asset before it appreciates is rational.

the size of the state space in the financial markets would often prohibit storage for space complexity reasons. Thus RL, and in particular model-free RL, in which agents make no attempt to estimate reward functions or transition probabilities but simply learn an optimal policy by doing, is suited to our purposes.

The remainder of this report proceeds as follows. In section 2 we outline related work that informed our implementation and testing. In section 3 we precisely specify the trading problem we try to solve. In section 4 we describe our approaches and responses to various difficulties we encountered. In section 5 we present the results of tests we subject our algorithms to, and provide a comparison of the performance of our agents. Section 6 concludes with a discussion of results and outlines possible improvements and extensions.

2 Background and Related Work

The bulk of our algorithms are derived from the course or from the accompanying textbook. Our tabular and approximate Q-Learning algorithms are adapted from the variants introduced by Professor Kuindersma in lecture. Our tabular SARSA agent is based on Russell and Norvig's description of the SARSA Q-update rule [6].

As background rather than as a guide to implementation, we also consulted descriptions of previous algorithmic trading implementations. For instance, Lee et al. use multiple Q-Learning agents to trade stocks, but split the tasks of stock pricing and stock selection between the agents [1]. While we initially considered this approach, we opted for a single Q-Learning agent for simplicity.

In a different paper Lee et al. stress the central difficulty in algorithmic trading as being the representation of states [2]; Bertoluzzo and Corazza echo this, noting the importance of their discretization of the state space for the success of their trading agent. While we did not adopt either of these authors' state representations, their discussion of different possibilities led us to spend a significant amount of time considering the best encoding.

Separately, two papers by Moody and Saffell and Wang et al. confirmed our instincts about the most appropriate benchmark for our trading agents' performance. Both sets of authors used the returns yielded by a buy-and-hold strategy as the metric for assessing the performance of their single asset-trading systems [5] [3], and so we too let this be our benchmark in testing.

Finally, in constructing our reward function, we were informed by Wang et al.'s description of previous approaches [3], and elected to make our agents' rewards the returns implied by the product of their portfolio position and the day-on-day percentage change in the asset's price. This approach allows us to ensure that our agents are rewarded for buying or holding when the traded asset appreciates, and penalized for not selling when the traded asset depreciates.

3 Problem Specification

Our overarching goal, as stated previously, was to devise an algorithm that trades a single asset optimally. By this we mean an algorithm that generates total returns as close as possible to some theoretical limit determined by the performance of the underlying asset. In testing, this meant generating total returns as close as possible to the limit determined by an asset's historical performance over the testing period, while in general it means generating maximal returns over an unbounded horizon (in the sense that our trading agents could continue trading indefinitely and thus indefinitely generate a return on some initial investment).

For our implementation, we specified the problem as choosing the return-maximizing action on every trading day and in a particular state over all corresponding state-action pairs, where the action set contained the actions *buy*, *sell* and *hold*. To be able to choose the return-maximizing action we naturally also needed to determine the anticipated returns for taking an action in a particular state, meaning that our problem constituted both learning state-action values (observation) and subsequently selecting actions (action). In particular, given an initial estimate of the value of a state-action pair, we needed to update our estimate of a pair's value using a transition and reward sample, and then determine the return-maximizing action. Thus the objective of our trading agents was to learn a policy, online, that maximized our agents' returns.

4 Approach

4.1 Design Decisions

We let states be defined as tuples of $(\mathit{portfolio\ position},\ \Delta\%\ \mathit{in\ asset\ price})$ (where the percentage change is computed day-on-day) because changes in the price of an asset have differential meanings for financial agents depending on whether they have invested in that asset or not. We ruled out defining states using prices because many assets will continually appreciate or depreciate and thus only revisit price points a handful of times, if at all. By contrast percentage changes in an asset's price are frequently recurring, and thus incorporating percentage changes into our agents' states has the additional benefit of increasing the likelihood that the time series over which our agents are trading is stationary (thereby increasing the likelihood of an agent's Q-values or feature weights converging).

However, because percentage changes $\Delta p \in \mathbb{R}$ and similarly because the percentage f of a financial agent's capital and portfolio value invested in an asset $f \in \mathbb{R}$ $(0 \leq f \leq 100)$, directly using the tuple-based state definition above would mean that we could not rule out an unbounded state space, that is a state space with an infinite number of states. For this reason, and heeding previous authors' emphasis on the importance of state-space discretization for model-free reinforcement learning, we decided to round day-on-day percentage changes in our traded asset's price to the nearest integer. Hence, our agent treats a 0.4% price change as a 0% price change and a 0.5% price change as a 1% appreciation. For the same reason we also determined to make our agents' portfolio positions binary, such that they always either invest or divest the entirety of their portfolio in each action, and at any given time either hold 100% of their portfolio value in the traded asset or 0% in the asset (and 100% in cash). This leads to a state space

S with $|S| = 2 \times$ *number of distinct percentage changes in price*, where the second factor in this product can be bounded in normal financial conditions. Since we have three possible actions for every state, *buy*, *sell* or *hold*, we now know our state-action space Q has size $|Q| = 2 \times 3 \times$ *number of distinct percentage changes in price*, which allows us to implement tabular Q-Learning and tabular SARSA when otherwise in a financial environment only approximate approaches would be feasible.

Aside from state-space discretization, we also faced the issue of balancing exploration with exploitation, that is valuing discovering new states sufficiently to ensure we do not operate in a local optimum ad infinitum, but also maximizing our agents' returns in the short-run by ensuring our agents take actions that are optimal given the agent's current estimation of a state's Q-values. To address this issue, we implemented an ϵ-greedy action selection policy, which involves acting randomly with some small probability ϵ and selecting the greedy action with some large probability $(1 - \epsilon)$. But because simple ϵ-greedy policies ensure that agents continue to take random actions even once the true Q-values have been learned, inspired by simulated annealing we implemented an inverse exponential 'temperature function' which decreases ϵ as the number of trading days t increases.[2] Specifically, our temperature function has the form $\epsilon = e^{-\beta t}$, where t is the number of trading days that have passed and $\beta > 0$ is some parameter (which we set experimentally).

A final design decision involved determining how to generate successor states. Because our simulation environment does not involve transaction costs, our current state can only be related to the successor state through the portfolio position implied by our agent's actions (that is, we know that if our agent sells in the current time period our agent's portfolio position in the next time period will necessarily be 0). However, because our agent's purchasing and selling decisions do not impact the traded asset's subsequent percentage change in price (according to economic theory, unless our agent's starting capital is artificially inflated) the successor state, as a tuple of our agent's portfolio position and the day-on-day percentage change in asset price, is not fully specified by our agent's action in the present state. As a simple solution we therefore decided to let our agents assume that a successor state equals the portfolio position implied by a particular action in the present state, and the current day-on-day percentage change in asset price projected into the next day (that is, we decided to project a trend).[3]

4.2 Implementation Components

We built our agents using algorithmic trading platform Quantopian's API, which faciliated the sourcing of data and allowed for easy testing in our experiments. Our implementations of our trading algorithms (tabular SARSA, tabular Q-Learning and approximate Q-Learning) share the following common elements:

1. *A main driver function*. The main function has four main parts. First, it calls functions which update our price data variables and calculates daily price percentage changes. Next, it generates a new state as dependent on the agent's portfolio position and the price percentage change and uses that information to determine the day's action (buy, sell or hold). It then

[2] We found experimentally that an inverse exponential temperature function worked better than other inverse functions.

[3] Our reasoning was that on average this trend projection would hold true.

runs a temperature function (for ϵ-greedy behavior) and determines whether to execute the calculated day's action, or whether to execute a random action. Finally, the learning agent's *update* function is called in order to update the agent's set of current $Q(s, a)$ values or feature weights w_i.

2. *An initialization function*. The initialize function is required by Quantopian, the trading platform we use to source data and run our simulations. Within the initialize function, so-called 'schedule functions' are initialized, and 'context' is filled. Schedule functions are Quantopian artifacts that execute according to a predetermined schedule. The main function is a schedule function that executes every day when the market opens. 'Context' is the Quantopian equivalent of a global variable within most standard programming languages. Variables and data structures can be initialized as elements of 'context' and then accessed throughout our code. We store various price data, along with useful data structures, within 'context'. Our learning agents, which we define as classes, are also initialized within 'context'.

3. *A daily data function*. We include a daily data function which executes once a day when the market opens. This function is an example of a Quantopian 'schedule function.' Our daily data function queries Quantopian's data interface and updates our price variables, such as the traded asset's current price.

4. *An agent.* We construct our trading agents as classes with methods that execute the appropriate reinforcement learning algorithm. Thus, each of our implementations, whether tabular SARSA, tabular Q-Learning, or approximate Q-Learning, has a corresponding agent class. These classes primarily serve to maintain and update dictionaries that store either Q-values or feature weights (with state-action or feature keys) .

4.3 Algorithms Specification

Each of our trading agents operates according to the corresponding pseudocode presented in Algorithms 1, 2 and 3 below. Fundamentally our algorithms are similar; the most major difference is that for SARSA we update our agent's estimate of the Q-values after observing the value of the successor $Q(s', a')$ (where the successor is chosen through a predetermined policy and we observe state-action-reward-state-action), whereas in Q-Learning we assume a greedy successor-selection policy.

5 Experiments

We perform extensive testing of our trading agents using Quantopian's integrated simulation suite. The simulation suite lets our trading agent execute trades over specified historical intervals as if it were trading in real time in the present; all data is accessed and provided through Quantopian. Every testing episode is separate, meaning that what our algorithms learn in one testing episode is not carried over into another. We standardize our testing by assuming for each test that our agents have $\$1,000,000$ in starting capital. We also run all simulations over the same period, from December 1, 2011 to December 1, 2017 (meaning that our agents trade on approximately 1500 days in a single simulation).[4] In total, we ran over 400 trials of our algorithms.

[4]There are approximately 252 trading days in a calendar year, and we run our historical simulations over six years.

Algorithm 1 Tabular SARSA

procedure TABULAR SARSA($state, action$)
  Initialize $Q(s, a)$ arbitrarily
  state $\leftarrow$ (portfolio position, price percentage change)
  $s \leftarrow$ state
  choose action $\in \{buy, sell, hold\}$ from s using policy derived from Q
  $a \leftarrow$ action
  for each day **do**
    execute a, observe $R(s, a) \in [-100, 100]$, s'
    choose a' from s' using predetermined policy
    $Q(s, a) \leftarrow Q(s, a) + \alpha(R(s, a) + \gamma Q(s', a') - Q(s, a))$
    $s \leftarrow s'$
    $a \leftarrow a'$
  end for
end procedure

Algorithm 2 Tabular Q-Learning

procedure TABULAR Q-LEARNING($state, action$)
  Initialize $Q(s, a)$ arbitrarily
  state $\leftarrow$ (portfolio position, price percentage change)
  $s \leftarrow$ state
  for each day **do**
    Choose action $\in \{buy, sell, hold\}$ which maximizes $Q(s, a)$ for given state s
    $a \leftarrow$ action with probability $(1 - \epsilon)$, random action with probability ϵ
    execute a, observe $R(s, a) \in [-100, 100]$, s'
    $Q(s, a) \leftarrow Q(s, a) + \alpha(R(s, a) + \gamma \max_{a'} Q(s', a') - Q(s, a))$
    $s \leftarrow s'$
  end for
end procedure

Algorithm 3 Approximate Q-Learning

procedure APPROXIMATE Q-LEARNING($state, action$)
  Initialize weights w_i arbitrarily uniformly
  Initialize state s
  for each day **do**
    Choose action $\in \{buy, sell, hold\}$ which maximizes $Q(s, a)$ for given state s
    $a \leftarrow$ action with probability $(1 - \epsilon)$, random action with probability ϵ
    Predict (or assume) $R(s, a) \in [-100, 100]$ and s', and compute $f_i(s, a)$
    Update weights according to $w_i \leftarrow w_i + \alpha(R(s, a) + \gamma \max_{a'} Q(s', a') - Q(s, a)) f_i(s, a)$
    $s \leftarrow s'$
  end for
end procedure

We first tune each of our trading agents by finding the optimal tuple of hyperparameters (α, γ) for a specified asset, where α is the learning rate and determines by how much our algorithms update their current estimation of the value of a state-action pair using the most recent transition and reward sample, and γ is the discount factor applied to future returns and determines how much our agent values future returns relative to immediate returns.[5] We execute 5 test runs for every combination of α and γ at 0.05 increments, where α and γ are bounded such that $0.05 \leq \alpha \leq 0.35$ and $0.85 \leq \gamma \leq 1$ (these bounds were determined through quick experimentation). We record the total returns generated as well as the standard deviation of the total returns for each parameter pair, because we wish to value consistency (this is particularly important in a trading context where real funds are at stake and trades cannot be executed multiple times in simulation). We ultimately selected values of α and γ that maximized the difference $average\ total\ returns - standard\ deviation\ of\ total\ returns$ (where this criterion was selected to trade off high average performance against consistency).[6]

Next, having found the hyperparameters that maximize average total returns for each trading agent, we report each trading agent's returns performance as well as the consistency of each agent's results for an asset that depreciates over the testing period and would deliver an investor adhering to a buy-and-hold strategy, that is an investor who purchases an asset and then does nothing with it, -5.93% returns. In an attempt to generalize from the differences in the performance of our agents trading our original asset, we also simulate trading on ten different equities with the largest market capitalizations on the S&P 500 for which data is also continuously available over the course of our standard testing period.[7] We also provide the benchmark performances of the chosen equities for comparison with data computed from Yahoo Finance.

Finally, to ensure that our algorithms are functioning as we would expect, we report and interpret a sample set of weights assigned to our features by our approximate Q-Learning agent at the end of testing. We also provide the average number of distinct states our agents explore, to provide a measure of the effectiveness of our state-space discretization and to demonstrate further that our algorithms explore the state space appropriately.

5.1 Results

In Figures 1-3, we present the performance of our trading agents as a function of the hyperparameter tuple (α, γ) (based on trading our baseline asset, General Electric, from December 1, 2011 to December 1, 2017). For our tabular SARSA algorithm, the parameter pair maximizing the difference between the algorithm's total returns and the standard deviation of its returns is $(\alpha = 0.3, \gamma = 1)$. For our tabular Q-Learning algorithm, the optimal parameter pair is $(\alpha = 0.2, \gamma = 1)$, while for

[5] Note we actually have three hyperparameters, α, γ and β, the parameter we use to determine how quickly our temperature function for ϵ goes to 0. Because representing a four-dimensional space graphically is difficult, we elected to omit the results of testing β here. However, we found $\beta = 0.01$ led to positive results.

[6] Note that advantage of this criterion is that it is a linear combination of average total returns and the standard deviation of total returns; thus someone who for example cared more about high average returns than variability could easily find a suitable hyperparameter tuple by applying a larger weight to an algorithm's average total returns.

[7] Please note that Quantopian's simulation suite breaks if one attempts to trade an asset that is not continuously tradable over the specified testing period. This is a flaw in Quantopian's platform rather than in our algorithms.

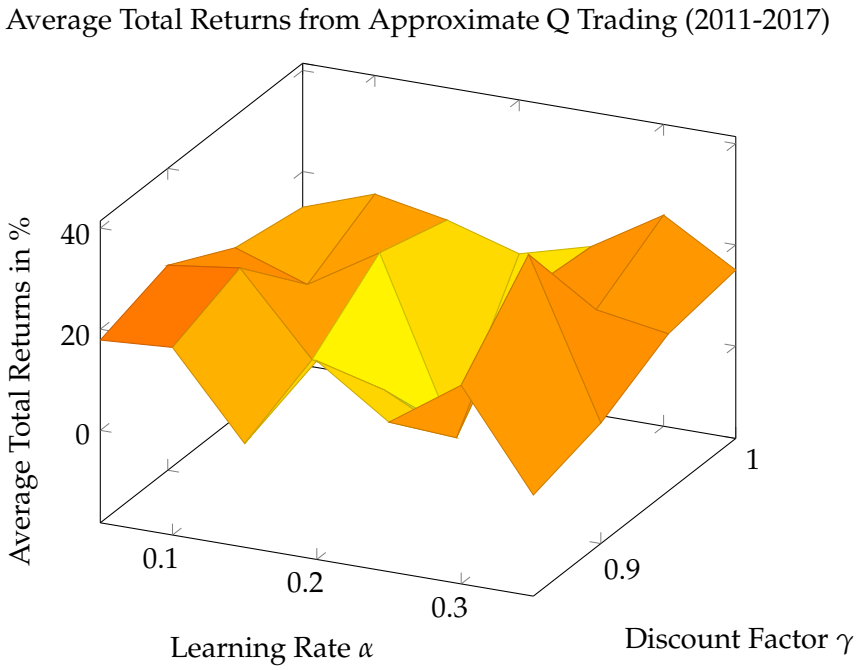

Figure 1: Tuning our Approximate Q-Learning Agent. Averages computed from 5 iterations. Based on simulated trading of General Electric shares with a starting capital of $\$1,000,000$ *over the period December 1, 2011-December 1, 2017.*

our approximate Q-Learning agent the parameter pair is $(\alpha = 0.3, \gamma = 0.9)$.

No obvious relationship between average total returns and the hyperparameters is discernible, although we speculate that a relationship would become more readily apparent if we increased the number of test iterations and if we tuned our algorithms on a wider variety of assets. It nonetheless seems that a low α (where low is loosely defined) is inadequate for our purposes because it prohibits our trading agents from rapidly updating their estimates of $Q(s, a)$ values when the financial markets or the underlying asset being traded are volatile or behave in a way that does not match previous behavior; by contrast a high α (where high is also loosely defined) means that our agents insufficiently retain their estimation of a $Q(s, a)$ value from previous experience and thus fail to generalize sufficiently. Similarly, it seems on the whole that higher rather than lower values of γ are advantageous for our trading agents' performance because higher values lead our algorithm to act as if with greater foresight and project from the present more intelligently.

In Table 1, we report the average total returns and standard deviation of total returns attained by our trading agents while trading our baseline asset, General Electric (GE), using the parameters identified as optimal in the previous step. We see that our tabular Q-Learning agent outperforms the other two agents, yielding average total returns of 63.1% against tabular SARSA's 44.6% and approximate Q-Learning's 36.4%. That said, given the relatively small number of test iterations we were able to complete (five runs per agent) it is difficult to say without further analysis whether these differences are statistically significant. All three of our agents exhibit approximately equal

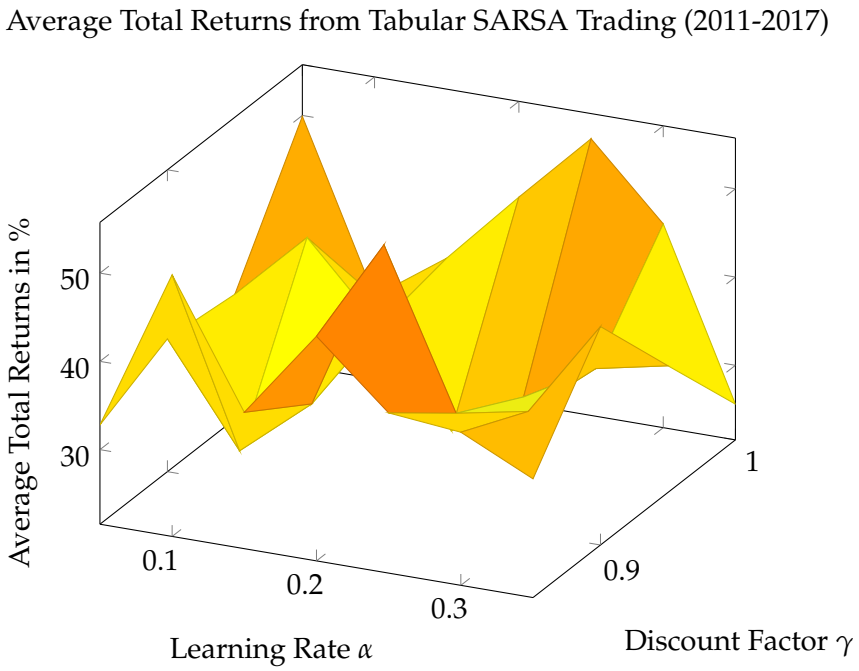

Figure 2: Tuning our Tabular SARSA Agent. Averages computed from 5 iterations. Based on simulated trading of General Electric shares with a starting capital of $1,000,000 *over the period December 1, 2011-December 1, 2017.*

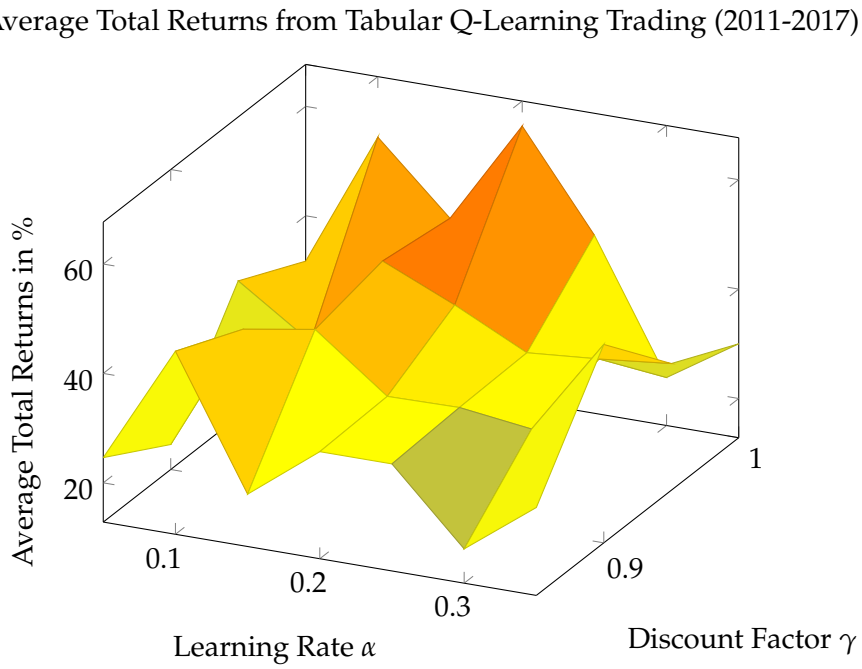

Figure 3: Tuning our Tabular Q-Learning Agent. Averages computed from 5 iterations. Based on simulated trading of General Electric shares with a starting capital of $1,000,000 *over the period December 1, 2011-December 1, 2017.*

	Best Average Returns (%)	S.D. (pp)
Tabular SARSA	44.6	20.6
Tabular Q-Learning	63.1	26.9
Approximate Q-Learning	36.4	20.4
Benchmark	-5.93	-

Table 1: Averaged over 5 iterations. Based on simulated trading of General Electric shares with a starting capital of $\$1,000,000$ *over the period December 1, 2011-December 1, 2017. Hyperparameters* α *and* γ *optimized for each algorithm. All figures to 3 s.f.. Benchmark figure computed from Yahoo Finance data.*

variation in their performance, but this variation, while significant, is not especially concerning because all returns within a standard deviation of the mean continue to outperform the benchmark of a buy-and-hold strategy.

In Figure 4 and Table 2, we exhibit our agents' performance on a wider variety of assets. On average the ranking of our agents is preserved; averaging over 5 iterations and ten stocks, our tabular Q-Learning agent generates total returns of 181%, while our tabular SARSA agent yields 168% and 118%. Our agents yield these returns against a benchmark average of 199%, which at face value suggests that our agents underperform, that is our agents perform worse than an agent in expectation acting at random in trading the same assets. However, it is important to consider the distortionary effect that Amazon's growth has on our computed benchmark value; omitting Amazon's growth our benchmark equals 161% (to 3 s.f.). Thus excluding Amazon our tabular Q-Learning and tabular SARSA agents outperform the benchmark. We are inclined to use 161% total returns as our benchmark both because Amazon's growth is exceptional and because the relatively small number of test iterations we were able to complete (five runs per agent per stock) means our agents' underperformance trading Amazon might be regarded as an outlier.

We ascribe the relatively poor performance of our approximate Q-Learning agent to the sparsity of the features we use for trading; we hypothesize that using more and more rich features would improve our approximate Q-Learning agent's performance. We also caution against extrapolating from the relative performance of the agents in testing that tabular SARSA will underperform tabular Q-Learning on average. In theory and under the right conditions, both SARSA and tabular Q-Learning should converge to the true Q-values, meaning there is no fundamental theoretical basis for the difference in performance. Indeed, we see that for some assets tabular SARSA outperforms tabular Q-Learning, which suggests the difference in average total returns might again be a reflection of the relatively small number of tests we execute. In general, the differences in performance we observe might be exaggerated in our tests because by the nature of our financial environment small differences in returns from an agent's first few actions can lead to significantly different action trajectories in the later stages of a simulation.

In Table 3, we present example weights from the conclusion of a test run of our approximation Q-Learning agent. All of the weights can be reconciled with economic interpretations, particularly if we recall that features in approximate Q-Learning can be likened to regressors in ordinary least squares regression. The negative weight assigned to the all-time high indicator likely re-

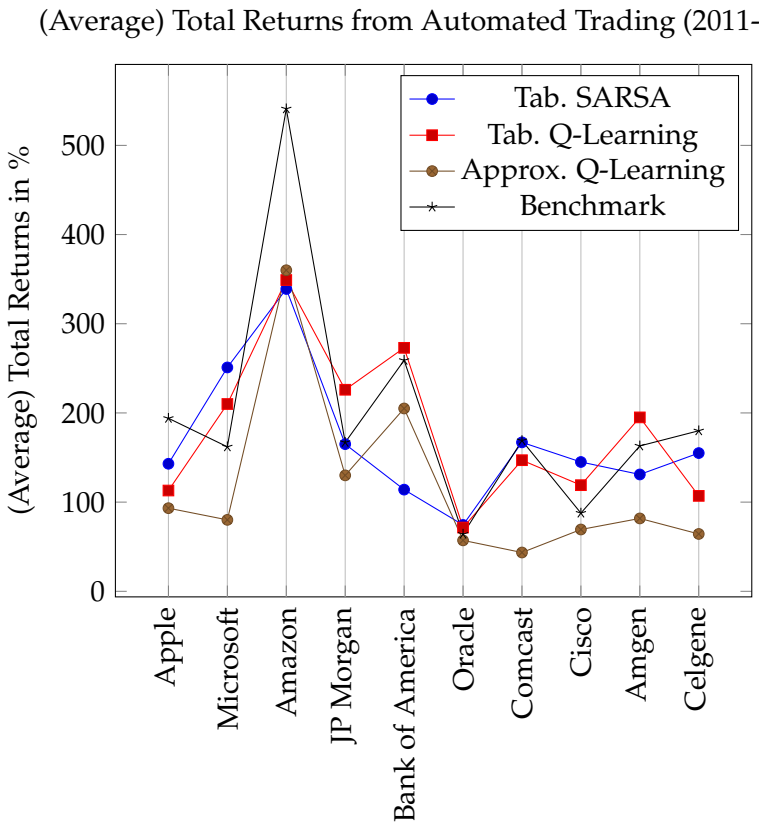

Figure 4: Total returns from trading the specified equities, averaged over 5 iterations. Based on simulated trading with a starting capital of $\$1,000,000$ *over the period December 1, 2011-December 1, 2017. Hyper-parameters* α *and* γ *optimized for each algorithm. All figures are percentages (unless stated otherwise) to 3 s.f.. Benchmarks computed from Yahoo Finance data.*

Stock	Tab. SARSA	Tab. Q-Learning	Approx. Q-Learning	Benchmark
Apple	143	113	93.2	194
Microsoft	251	210	80.1	162
Amazon	339	349	360	541
JP Morgan	165	226	130	167
Bank of America	114	273	205	259
Oracle	74.4	71.5	57.1	64.1
Comcast	167	147	43.5	169
Cisco	145	119	69.3	87.7
Amgen	131	195	81.7	163
Celgene	155	107	64.3	180
Average Total Returns	168	181	118	199
Average S.D. (pp)	64.3	67.7	68.7	-

Table 2: Total returns from trading the specified equities, averaged over 5 iterations. Based on simulated trading with a starting capital of $\$1,000,000$ *over the period December 1, 2011-December 1, 2017. Hyperparameters* α *and* γ *optimized for each algorithm. All figures are percentages (unless stated otherwise) to 3 s.f.. Benchmarks computed from Yahoo Finance data.*

flects the fact that GE reached an all-time high share price shortly before depreciating significantly towards the end of our simulation period; our agent likely internalized the resulting losses as a penalty. The 1.0 weight assigned to the all-time low indicator (the weight all features were assigned upon initialization) reflects the fact that GE never reached its all-time low price over the course of the simulation. The positive weight assigned to the Above Opening Price Indicator (Buy) likely reflects that if GE's shares were trading above their opening price they typically rallied further throughout the course of the trading day. By contrast, the modest negative weight assigned to the Below Opening Price Indicator (Buy) likely reflects the fact that our agent made losses by buying while GE shares were on a negative swing, but also made gains by buying while GE shares were trading at favorable prices (and later appreciated). The weight assigned to the Above Opening Price Indicator (Hold) suggests that our agent rarely held its position at this price level, while the weight assigned to the Below Opening Price Indicator (Hold) suggests that positive and negative swings cancelled each other out. We can also interpret the negative weights assigned to the Above Opening Price Indicator (Sell) as reflecting that by selling our agent typically missed out on positive swings, while the negative weights assigned to the Below Opening Price Indicator (Sell) indicate that our agent made losses by selling shares in a price trough.

Finally, we present the average number of distinct states that each of our trading agents explored in simulation. If we recall that the size of our discretized state space S is given by $|S| = 2 \times$ *number of distinct percentage changes in price*, the figures in Table 4 support the view that our agents adequately and successfully explore large proportions of the state space. Using Yahoo Finance data we find over our simulation period that there were 14 distinct percentage changes in the day-on-day price of GE; thus we see see our tabular SARSA agent explored on average 85% of states, our tabular Q-Learning agent explored on average $\approx$ 80% of states, and our approximate Q-Learning agent explored $\approx$ 78% of states. The slightly lower number of states explored

Feature	Sample Weights
All-Time High Indicator	-4.93
All-Time Low Indicator	1.00
Above Opening Price Indicator (Buy)	6.03
Below Opening Price Indicator (Buy)	-0.0714
Above Opening Price Indicator (Hold)	1.01
Below Opening Price Indicator (Hold)	0.00439
Above Opening Price Indicator (Sell)	-2.00
Below Opening Price Indicator (Sell)	-4.26

Table 3: Sample weights obtained at the conclusion of simulated trading of General Electric shares with a starting capital of $1,000,000 *over the period December 1, 2011-December 1, 2017. All figures to 3 s.f..*

	Average # of Distinct States Explored	Proportion of States Explored (%)
Tab. SARSA	23.8	85
Tab. Q-Learning	22.5	80
Approx. Q-Learning	21.7	78

Table 4: Averaged over 10 iterations (3 s.f.). Proportions to 2 s.f.. Based on simulated trading of General Electric shares with a starting capital of $1,000,000 *over the period December 1, 2011-December 1, 2017.*

by our approximate Q-Learning agent corroborates its underperformance relative to our two tabular trading agents (by exploring less our approximate Q-Learning agent discovers fewer high return state-action pairs). It also reflects our observation during simulation that our approximate Q-Learning agent sometimes seemed to get stuck in local optima, that is after some period of trading our Q-Learning agent converged to a single preferred action. This convergence to a preferred action can likely be attributed to the sparsity and generality of our features, which we have already addressed previously.

6 Discussion

Overall, our results indicate the viability of model-free reinforcement learning algorithms for automated trading purposes. For a benchmark asset on which a buy-and-hold investor would make -5.93% returns on, our trading agents average returns in excess of 36%. Trading a larger pool of assets, and excluding an outlying stock, our tabular SARSA and tabular Q-Learning agents both outperform the buy-and-hold investor, generating average returns on 168% and 181% against a 161% return benchmark. Thus, in a large set of tests, our algorithms perform better than an agent acting at random. The broad success of our algorithms seems to relate significantly to our success in discretizing the state space for trading a single asset, which makes tabular approaches feasible. While the performance of our approximate Q-Learning agent is somewhat disappointing, it is also understandable in view of the simplicity of the features we constructed (which were intended to be applicable for every stock, but to be effective seem to require specificity to the asset being

traded).

Over the course of the design and implementation of our system, we have come to fully appreciate the complexity of the financial markets, and the scope of decisions that need to be made in order to build functioning and intelligent automated trading agents. Aside from the possible extensions and improvements we discuss below, in the future we would like to experiment with policy search methods, which might be an improvement on our current methods because instead of attempting to accurately find Q-values and therefrom maximize returns the immediate objective with policy search methods is maximizing returns through lookaheads.

6.1 Possible Extensions and Improvements

Out of concern for the feasibility of our project we made several simplifying assumptions in the implementation of our agents, such as making our agents' portfolio positions binary. In future work we would hope to relax some of the simplifying assumptions we made and extend our project in the following ways:

1. *Add features.* As stated previously, we believe the performance of our approximate Q-Learning agent was hampered by our exclusive use of general own-price features, rather than features that relate to other assets' prices, general macroeconomic variables, or asset fundamentals. We would hope to construct features that offer a more accurate and rich characterization of different states and thus allow our approximate Q-Learning agent to better estimate state-action pairs' Q-values. Additional general features, like a 20-day or 50-day moving price average, could also improve our algorithm's performance.

2. *Modify α. Modify ϵ.* While we experimented with different values for our learning hyperparameter α, setting α as a constant assumes the underlying system is stationary. Even though we worked with percentage changes in asset price, these time series were likely still not stationary. There has been some discussion of constructing a dynamic α [4], perhaps as a function of the volatility of the traded asset, and we would hope to consider such an implementation in the future. Similarly, one might also wonder whether we could improve our agents' performance by relating ϵ in our action selection policy to the underlying asset's behavior.

3. *Experiment with discretization.* We currently discretize our state space such that our agents only distinguish between unit percentage changes in price. It would be interesting to see whether a more granular representation of the state space would allow our agents to make more nuanced investment decisions. Our current method of discretization rules out trading more frequently than once a day (because with more frequent trading too many percentage changes in price are rounded to 0); thus a more granular discretization could also allow us to trade more volatile assets than the equities our agents have traded in testing (where this is currently impossible because for volatile assets the day-on-day percentage changes are too great and thus states too dispersed for our trading agents to be effective). On the other hand, a possible consequence might be that our state-space expands to the point that tabular methods begin to perform poorly .

4. *Add transaction costs.* Although the simulation environment we assume, without transaction costs, is not totally unrealistic because there now exist trading platforms such as Robinhood

which allow free transactions, it would be of interest to see how our agents' behavior changes with the incorporation of transaction rigidities. We anticipate that with transaction costs our agents would trade less frequently because in many scenarios the positive returns from making a transaction would be outweighed by the brokerage cost.

5. *Trade multiple assets. Execute more complex transactions.* A natural next step for our trading agents would be to purchase and sell multiple assets, and by implication manage a portfolio. We would also hope to relax the constraint we placed on our agents' actions, and allow our agents to buy, sell or short more complex fractions of our portfolio value. But here again we might find the resulting increase in the size of the state space prohibitive.

6. *Internalize price effects from agent activity.* As previously stated, we currently we make the assumption, broadly supported by economic theory, that our agents' purchasing decisions have no impact on the market price of the asset they are trading. However, with less deep assets than the ones we have tested our agents on, that is for scarcer assets, we might expect our agents' actions to impact the future price of they asset they trade. In the future we would hope to build into our agents a pricing model through which our agents internalize the effects of their actions on the price of the assets they are trading, and adjust their estimates for Q-values accordingly.

A Appendix: System Description

The first step to using our system is to clone our GitHub repository, which can be found here. To run our algorithms and use our system, which is integrated into algorithmic trading platform Quantopian's development environment, please visit Quantopian's homepage and create a free account. Then enter the platform's IDE, which is accessible here. To run our tabular SARSA agent, copy-paste the code we provide in `sarsa.py` into the IDE; to run our tabular Q-learning agent, copy-paste the code we provide in `tabularQ.py` into the IDE; and to run our approximate Q-Learning agent, copy-paste the code provided in `approximateQ.py` into the IDE (in all three cases please completely replace the default algorithm you will find in the Quantopian IDE).

With the code for an agent copied into the IDE, run a 'full backtest' (still on Quantopian) by specifying the start and end dates of the period you wish to trade in, setting a reasonable amount of starting capital $\$1,000,000$, and then clicking `Run Full Backtest`. Since our agents only execute trades on a day-by-day rather than interday basis, to see reasonable results we recommend testing our agents over a period of at least a few years. If you wish, you may change the asset our agents trade by altering the code within the IDE; specifically, modify `context.security = sid( )` where the (stock) symbol for the asset to be traded should be written within `sid( )`.[8] As the backtest runs (and once it has concluded), you will be able to see three graphs that reflect our agent's performance for that particular simulation. Quantopian automatically charts our agents' total returns, daily or weekly returns, and transactions. We obtained the results included in this report by repeatedly running backtests and recording the resulting data. We also inserted print statements into our code and reviewed the backtest logs; for instance, this is how we checked and obtained the concluding weights for our approximate Q-Learning agent.

Although using our system is fairly simple and (we hope) adequately explained here, should you have any outstanding questions please feel free to contact us at {`rahulnaidoo, rmilushev, fabianbarrett`}`@college.harvard.edu`.

[8] Note typically Quantopian will provide the correct security ID (SID) by autocomplete once the first letters of a stock symbol have been written.

B Appendix: Group Makeup

- Ben Barrett led the early-stage research for our trading agents, looking into the most appropriate algorithms to implement and conceptualizing modifications to fit a financial environment. He also undertook the testing of our agents and contributed to all three agent implementations.
- Rangel Milushev led the implementation of our tabular Q-Learning and tabular SARSA agents. He also contributed to our initial research, assisted with debugging efforts, and located key papers which guided our approach.
- Rahul Naidoo led the implementation of our approximate Q-Learning agent, contributed to the remaining two implementations, and conceptualized the design of our features. He also discovered Quantopian as a simulation platform, assisted with testing, and led an extensive debugging effort.

This division of work and distribution of contributions corresponds to the equal division we described in our project proposal.

References

[1] J.W. Lee et al. "A Multiagent Approach to Q-Learning for Daily Stock Trading". In: *IEEE Transactions on Systems, Man and Cybernetics* 37 (2007), pp. 864–877. DOI: 10.1109/TSMCA.2007.904825.

[2] J.W. Lee et al. "A Q-Learning Based Approach to the Design of Intelligent Stock Trading Agents". In: *Proceedings of the IEEE International Engineering Management Conference* 3 (2004), pp. 1289–1292. DOI: 10.1109/IEMC.2004.1408902.

[3] Yang Wang et al. *Deep Q-Trading*. URL: http://cslt.riit.tsinghua.edu.cn/mediawiki/images/5/5f/Dtq.pdf.

[4] Francesco Bertoluzzo and Marco Corazza. "Reinforcement Learning for Automated Financial Trading: Basics and Applications". In: *Recent Advances of Neural Network Models and Applications* 26 (2014). DOI: 10.1007/978-3-319-04129-2_20.

[5] John Moody and Matthew Saffell. "Reinforcement Learning for Trading Systems and Portfolios". In: *American Association for Artificial Intelligence* (1998). DOI: https://pdfs.semanticscholar.org/10f3/4407d0f7766cfb887334de4ce105d5aa8aae.pdf.

[6] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2010.

[7] Christopher Watkins and Peter Dayan. "Technical Note: Q-Learning". In: *Machine Learning* 8 (1992), pp. 279–292. DOI: https://link.springer.com/content/pdf/10.1007%2FBF00992698.pdf.