A Reinforcement Learning Approach to Optimal Execution

Ciamac C. Moallemi
Graduate School of Business
Columbia University
email: ciamac@gsb.columbia.edu

Muye Wang
Graduate School of Business
Columbia University
email: mw3144@gsb.columbia.edu

Current Revision: February 1, 2022

Abstract

We consider the problem of execution timing in optimal execution. Specifically, we formulate the optimal execution problem of an infinitesimal order as an optimal stopping problem. By using a novel neural network architecture, we develop two versions of data-driven approaches for this problem, one based on supervised learning, and the other based on reinforcement learning. Temporal difference learning can be applied and extends these two methods to many variants. Through numerical experiments on historical market data, we demonstrate significant cost reduction of these methods. Insights from numerical experiments reveals various tradeoffs in the use of temporal difference learning, including convergence rates, data efficiency, and a tradeoff between bias and variance.

1. Introduction

Optimal execution is a classic problem in finance that aims to optimize trading while balancing various tradeoffs. When trading a large order of stock, one of the most common tradeoffs is between market impact and price uncertainty. More specifically, if a large order is submitted as a single execution, the market would typically move in the adverse direction, worsening the average execution price. This phenomenon is commonly referred to as the "market impact". In order to minimize the market impact, the trader has an incentive to divide the large order into smaller child orders and execute them gradually over time. However, this strategy inevitably prolongs the execution horizon, exposing the trader to a greater degree of price uncertainty. Optimal execution problems seek to obtain an optimal trading schedule while balancing a specific tradeoff such as this.

We will refer to the execution problem mentioned above as the parent order problem, where an important issue is to divide a large parent order into smaller child orders to mitigate market impact. In this paper, we focus on the optimal execution of the child orders, that is, after the parent order is divided, the problem of executing each one of the child orders. The child orders are quite different in nature compared to the parent order. The child orders are typically much smaller in size, and the prescribed execution horizons are typically much shorter. In practice, a parent order is typically completed within hours or days while a child orders are typically completed within seconds or minutes. Because any further dividing of an order can be viewed as another parent

order problem, we will only consider the child order problem at the most atomic level. At this level, the child orders will not be further divided. In other words, each child order will be fulfilled in a single execution.

The execution of the child orders is an important problem and warrants its own consideration, apart from the parent order problem. These two problems focus on different aspects of execution at different time scales. In the parent order problem, the main tradeoff is between market impact and price risk, and the solution to the problem aims to find the trading rate schedule that optimally balances between the two on the time scale of hours to days. In the child order problem, as we consider it, the main consideration is simply getting the best price on the time scale of seconds to minutes. the price movement of the stock becomes the primary consideration. Therefore solution to the child order problem focuses on predicting the price movement and finding the optimal time for the execution.

More specifically, because the market impact is negligible for a child order and the order must be fulfilled in a single execution, the solution seeks to execute the child order at an optimal time within the prescribed execution horizon. In this paper, we will develop data-driven approach based on price prediction to solve the execution timing problem.

The main contributions of this paper are as follows.

- **Execution Timing Problem.** We formulate the execution timing problem as an optimal stopping problem, where prediction of the future prices is an important ingredient.

- **Data-Driven Approach.** Unlike the majority of work in this area, we make no model assumptions on the price dynamics. Instead, we construct a novel neural network architecture that forecasts future price dynamics based on current market conditions. Using the neural network predictions, the trader can develop an execution policy.

  In order to implement the data-driven approach, we develop two specific methods, one based on supervised learning (SL), and the other based on reinforcement learning (RL). There are also different ways to train the neural network for these two methods. Specifically, empirical Monte Carlo (MC) and temporal difference (TD) learning can be applied and provide different variants of the SL and RL methods.

- **Backtested Numerical Experiments.** The data-driven approach developed in this paper is tested using historical market data, and is shown to generate significant cost saving. More specifically, the data-driven approach can recover a price gain of 20% of the half-spread of a stock for each execution in average, significantly reduce transaction costs.

  The RL method is also shown to be superior than the SL method when the maximal achievable performance is compared. There are a few other interesting insights that are revealed in the numerical experiments. Specifically, the choice of TD learning and MC update method presents various tradeoffs including convergence rates, data efficiency, and a tradeoff between bias and variance.

Through numerical experiments, we also demonstrate a certain universality among stocks in the limit order book market. Specifically, a model trained with experiences from trading one stock can generate non-trivial performance on a different stock.

1.1. Literature Review

Earlier work in the area of optimal execution problem includes Almgren and Chriss [2000] and Bertsimas and Lo [1998]. These two papers lay the theoretical foundations for many further studies, including Coggins et al. [2003], Obizhaeva and Wang [2013], and El-Yaniv et al. [2001].

The paper that is perhaps most closely related to our work is Nevmyvaka et al. [2006]. They also apply reinforcement learning to the problem of optimal execution, but there are also many differences. They consider the dividing problem of the parent order and the goal is to obtain an optimal trading schedule, whereas we apply RL to solve the child order problem using a single execution. This allows us to simplify the child problem into an optimal stopping problem rather than an optimal scheduling problem. On a more technical level, they use a tabular representation to present the state variables, which force the state variables to be discretized. We allow continuous state variables by utilizing neural networks. Other differences include the action space, feature selections, and numerical experiment as well.

Another area in finance where optimal stopping is an important practical problem is pricing American options. Motivated by this application, Longstaff and Schwartz [2001] and Tsitsiklis and Van Roy [2001] have proposed using regression to estimate the value of continuation and thus to solve optimal stopping problems. Similarly to this work, at each time instance, the value of continuation is compared to the value of stopping, and the optimal action is the action with the higher value. The regression-based approach is also different in a number of ways. One difference is the choice of model. They use regression with linear model to estimate continuation values where as we use nonlinear neural networks. Another difference is that they fit a separate model for each time horizon using a backward induction process, which increases the remaining horizon one step at a time. By contrast, we fit a single neural network for all time horizons. Our approach can learn and extrapolate features across time horizons. This also leads to a straightforward formulation of temporal difference learning, which we will discuss in Section 3.4 and Section 4.3.

Deep learning has been applied to the study of optimal stopping problems. Notably, Becker et al. [2019] and Becker et al. [2020] use neural networks to learn an optimal stopping rule by parameterizing the stopping policy directly. Becker et al. [2021] employs a similar formulation but extends the problem to higher dimensions. These approaches are typically referred to as "policy-based approaches" as they characterize the stopping rule directly, typically as a sequence of (possibly randomized) binary decisions. In contrast, our approach is an example of a "value-based approach" as it learns the expected long-term future reward of taking each action and induces a stopping policy accordingly. Other work in this area includes that of Gaspar et al. [2020] and Herrera et al. [2021]. Gaspar et al. [2020] combine neural networks with Least-Squares Monte Carlo (LSMC) method to price American options. Herrera et al. [2021] use randomized neural

networks, where the weights in hidden layers are randomly generated and only the last layer is trained, to estimate continuation values. Our work is different in a few ways. First, we consider optimal execution problem as the main application of our proposed methods, and the numerical experiments reveal significant price gains. Second, we propose execution policies induced from both supervised learning and reinforcement learning approaches, and we discuss various tradeoffs between them when temporal difference learning is applied. Lastly, we design a specific neural network architecture, which captures the monotonic nature of the continuation value.

This work also joins the growing community of studies applying machine learning to tackle problems in financial markets. Sirignano [2019] uses neural networks to predict the direction of the next immediate price change and also reports the similar universality among stocks. Kim et al. [2002] utilize RL to learn profitable market-making strategies in a dynamic model. Park and Van Roy [2015] propose a method of simultaneous execution and learning for the purpose of optimal execution.

1.2. Organization of the paper

The rest of the paper is organized as follows. Section 2 introduces the mechanics of limit order book markets and outlines the optimal stopping formulation. Section 3 introduces the supervised learning method and its induced execution policy. TD learning is also introduced in this section. Section 4 introduces the reinforcement learning method and its induced execution policy. Section 5 outlines data source and the setup for the numerical experiments. Section 6 presents the numerical results and the various tradeoffs in training process introduced by TD learning. The aforementioned universality are also discussed in Section 6.

2. Limit Order Book and Optimal Stopping Formulation

2.1. Limit Order Book Mechanics

In modern electronic stock exchanges, limit order books are responsible for keeping track of resting limit orders at different price levels. Because investors' preferences and positions change over time, limit order books also need to be dynamics and changing over time. During trading hours, market orders and limit orders are constantly being submitted and traded. These events alter the amount of resting limit orders, consequently, the shape of the limit order book. There are other market events that alter the shape of the limit order book, such as order cancellation.

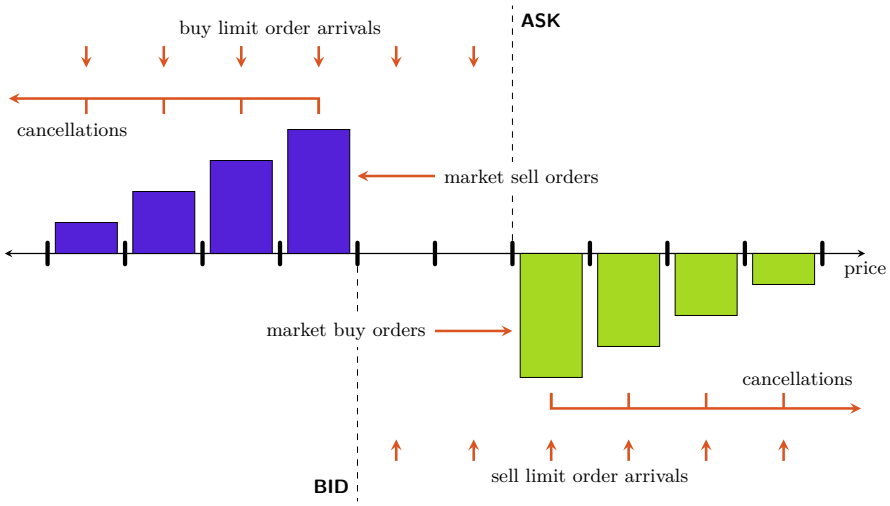

Figure 1: Limit orders are submitted at different price levels. The ask prices are higher than the bid prices. The difference between the lowest ask price and the highest bid price is the bid-ask spread. Mid-price is the average of the best ask price and the best bid price.

Limit order books are also paired with matching engines that match incoming market orders with resting limit orders to fulfill trades. The most common rule that the matching engine operates under is "price-time priority". When a new market order is submitted to buy, sell limit orders at the lowest ask price will be executed; when a new market order is submitted to sell, buy limit orders at the highest bid price will be executed. For limit orders at the same price, the matching engines follow a time priority — whichever order was submitted first gets executed first.

2.2. Price Predictability

Some theoretical models in the classic optimal execution literature treat future prices as unpredictable. However, this doesn't always reconcile with the reality. There is empirical evidence that stock prices can be predicted to a certain extent — Sirignano [2019] predicts the direction of price moves using a neural network and detects significant predictabilities.

Clearly, the ability to predict future prices would have major implications on stock executions. If a trader seeks to sell and predicts that the future price will move up, then the trader would have an incentive to wait. On the other hand, if the trader predicts that the future price will drop, then the trader would have an incentive to sell immediately. In short, at least at a conceptual level, price predictability improves execution quality. This motivates us to construct a data-driven solution incorporating price predictability to optimal execution problems.

2.3. Optimal Stopping Formulation

Our framework will be that of a discrete-time sequential decision problem over a finite execution horizon T. The set of discrete time instances within the execution horizon is $\mathcal{T} \triangleq \{0, 1, ..., T\}$. For

a particular stock, its relevant market conditions are represented by a discrete-time Markov chain with state $\{x_t\}_{t\in\mathcal{T}}$. We will assume that the transition kernel P for the states is time-invariant [1]. One of the state variables in the state that is of particular interest is the price of the stock, and we will denote this price process by $\{p_t\}_{t\in\mathcal{T}}$.

Consider the problem of selling one share of the stock, or equivalently, consider the order to be infinitesimal, that is, the order can't be further divided. This problem singles out the timing aspect of the execution and assumes that any action of the trader has no impact on the price process, the states, and the transitional kernel.

For a trader, the set of available actions at time t is $a_t \in \mathcal{A} = \{\text{HOLD, SELL}\}$. In other words, at any time instance, the trader can either hold the stock or sell the stock. Because the trader is endowed with only 1 share of the stock, once the trader sells, no further action can be taken. In essence, this is an optimal stopping problem — the trader holds the stock and picks an optimal time to sell. To generalize the notation, we will use {CONTINUE, STOP} to represent $\mathcal{A}$ for the rest of this paper.

Let τ be a stopping time. Then, the sequence of states and actions before stopping is as follows

$$\{x_0, a_0, x_1, a_1, ..., x_\tau, a_\tau\}, \tag{1}$$

where $a_\tau = \text{STOP}$ by the definition of the stopping time. The trader's goal is to maximize the expected total price difference between the execution price p_τ and the initial price, namely,

$$\max_{\tau} \quad \mathbb{E}[p_\tau - p_0]. \tag{2}$$

We will refer to this value as the total price gain and denote it by $\Delta P_\tau \triangleq p_\tau - p_0$. Maximizing the total price gain is equivalent to minimizing the implementation shortfall in this problem. Total price gain can be decomposed into the price gain between each time instance while the trader holds the stock. Let $\Delta p_t \triangleq p_t - p_{t-1}$. Then, the total price gain can be decomposed into per-period rewards

$$\Delta P_\tau = \sum_{t=1}^{\tau} \Delta p_t. \tag{3}$$

From a sequential decision problem standpoint, this is not the only way to decompose the total price gain across time. One can also design a framework where the traders only receive a terminal reward when they stops. This decomposition approach benefits a learning agent by giving per-period rewards as immediate feedback.

Define a σ-algebra $\mathcal{F}_t \triangleq \sigma(x_0, a_0, ..., , x_{t-1}, a_{t-1}, x_t)$ for each time t, and a filtration $\mathcal{F} \triangleq \{\mathcal{F}_t\}_{t\in\mathcal{T}}$. Let random variable π_t be a choice of action that is $\mathcal{F}_t$-measurable and takes values in $\mathcal{A}$, and let a policy π be a sequence of such choices, i.e. $\pi = \{\pi_t\}_{t\in\mathcal{T}}$, and is $\mathcal{F}$-adapted. As constrained by

[1] This assumption is justifiable in our setting as the execution horizon is typically quite short, and might be measured in seconds to minutes. Over such short time horizons, non-stationarity can be ignored. Beyond this, note that the time of the day is also included as a state variable so the price dynamics allow for time-of-day effects even though they are stationary.

the execution horizon, the last action must be STOP, i.e. $\pi_T \triangleq \text{STOP}$.

Let Π be the set of all such policies, and an optimal policy π^* is given by

$$\pi^* \triangleq \underset{\pi \in \Pi}{\text{argmax}} \ \mathbb{E}_\pi \left[\sum_{t=1}^{\tau_\pi} \Delta p_t \right], \tag{4}$$

where τ_π is the first stopping time associated with policy π, and the expectation is taken assuming the policy π is used. Learning an optimal policy from data is the main machine-learning task that will be discussed in the next two sections.

3. Supervised Learning Approach

Future stock prices are inherently stochastic, and this makes optimal execution a challenging problem. One way to simplify this problem is to replace the random distribution of future prices by a deterministic point estimates and thus reduce the stochastic problem into a deterministic one. The supervised learning approach, which will be introduced in this section, replaces the future price distribution with its conditional expectation. Assuming that prices will deterministically follow the prediction of the expected future price trajectory, an optimal execution policy can be readily derived. This is what we call the supervised learning (SL) method.

However, the SL method doesn't lead to an optimal execution policy because ignoring stochasticity also ignores the possibility that trader can take different sequence of actions on different trajectories. Section 3.6 illustrates this insufficiency further. This prompts us to develop the reinforcement learning (RL) method, which is the focus of Section 4. Section 4.5 provides more in-depth discussion regarding the differences and the similarities between the SL and RL methods.

3.1. Price Trajectory Prediction

Future prices have important implications on execution policies. If a selling trader can predict that the future price is higher than the current price, the trader would wait and execute at a later time. If the future price is predicted to be lower than the current price, the selling agent should sell immediately. In this section, we will formulate this intuition more formally and construct a price prediction approach to optimal execution via supervised learning.

Given a fixed execution horizon T, it's insufficient to only predict the immediate price change in the short term — even if the price goes down, it could still move back up and rise even higher before the end of the execution horizon. Therefore, to obtain an optimal execution policy, it's imperative to obtain a price prediction for the entire execution horizon. This can be achieved by predicting price changes at each time instances. More specifically, define a price change trajectory as follows,

$$\text{Price Change Trajectory} \triangleq [\Delta p_1, \Delta p_2, ..., \Delta p_T]. \tag{5}$$

This gives rise to p_t through

$$p_t = p_0 + \sum_{i=1}^{t} \Delta p_i.$$

In the rest of the section, we will construct supervised learning models to predict the price change trajectory.

3.2. Supervised Learning Method

Define an observation episode as a vector of states and price changes, ordered in time as (6). This is the data observation upon which we will construct supervised learning models.

$$\text{Observation Episode } \triangleq \{x_0, \Delta p_1, x_1, \Delta p_2, ..., \Delta p_T, x_T\}. \tag{6}$$

In order to take an action at time 0, the trader needs a price change trajectory prediction at time 0 when the only observable state is x_0. Given any current state x, in order to predict the subsequent price change trajectory, we construct a neural network as follows. The neural network takes a single state x as input and outputs a vector of T elements, corresponding to the price change at each of the subsequent time instance. This neural network is represented as follows in (7).

$$\text{Neural Network:} \quad NN^{\phi}(x) = [u_1^{\phi}(x), u_2^{\phi}(x), ..., u_T^{\phi}(x)]. \tag{7}$$

The neural network parameter is denoted by ϕ, and the output neuron $u_i^{\phi}(x)$ corresponds to the price change Δp_i for all $1 \leq i \leq T$.

Given an observation episode such as (6), the mean squared error (MSE) between predicted price changes and actual price changes can be used as a loss function. That is

$$\mathcal{L}(\phi; x_0) = \frac{1}{T} \sum_{i=1}^{T} \left[\Delta p_i - u_i^{\phi}(x_0)\right]^2. \tag{8}$$

The neural network can be trained by minimizing (8) averaged over many observation episodes. After the neural network is trained, it can be applied to all states, giving a price change trajectory prediction at each time instance.

3.3. Execution Policy

Given a state x, the output of the neural network is a prediction of the subsequent price change trajectory. Summing up the price changes provides an estimate of the cumulative price change. Let $W_{t:T}(x)$ be the estimated maximum cumulative price change over all remaining time when the current time is t. For all $t \in \mathcal{T} \setminus \{T\}$, $W_{t:T}(x)$ can be expressed as

$$W_{t:T}(x) \triangleq \max_{1 \leq h \leq T-t} \sum_{i=1}^{h} u_i^{\phi}(x). \tag{9}$$

Notice, because the transitional kernel P is assumed to be time-invariant (see Section 2.3), only the difference in indices $T-t$ matters in the value of $W_{t:T}(x)$, not the index t or T itself. At any time before T, if the future price trajectory rises higher than the current price, a selling trader would have an incentive to wait. Otherwise the trader should sell right away. This execution policy can be formally written as follows.

Supervised Learning Policy:
When the current time is t and the current state is x, define a choice of action π_t^{SL} as below.

$$\pi_t^{\mathrm{SL}}(x) \triangleq \begin{cases} \text{CONTINUE} & \text{if} \quad W_{t:T}(x) > 0 \\ \text{STOP} & \text{otherwise.} \end{cases}$$

The execution policy induced by the SL method is the sequence of all such choices, given by

$$\pi^{\mathrm{SL}}(\cdot) \triangleq \{\pi_t^{\mathrm{SL}}(\cdot)\}_{t\in\mathcal{T}}. \tag{10}$$

Note that this policy is a Markovian policy in that this decision at time t is a function of the current state x_t. This policy is dependent on the neural network through the value of $W_{t:T}(\cdot)$. To apply this policy, a trader would apply each action function sequentially at each state until STOP is taken. More specifically, given a sequence of states, the stopping time is given by

$$\tau_{\pi^{\mathrm{SL}}} \triangleq \min\{t \,|\, \pi_t^{\mathrm{SL}}(x_t) = \text{STOP}\}. \tag{11}$$

The total price gain induced by this policy on the specific observation episode is $\Delta P_{\tau_\pi^{\mathrm{SL}}} = p_{\tau_\pi^{\mathrm{SL}}} - p_0$. Once the trader stops, no further action can be taken.

3.4. Temporal Difference Learning

The method discussed in Section 3.2 is a straightforward supervised learning method. However it has a few drawbacks. From a practical perspective, given any observation episode such as (6), only $\{x_0, \Delta p_1, \Delta p_2, ..., \Delta p_T\}$ is being used to train the neural network and $\{x_1, x_2, ..., x_T\}$ isn't being utilized at all during the training process. This prompts us to turn to TD learning.

TD learning is one of the central ideas in RL (see Sutton and Barto [1998]) and it can be applied to supervised learning as well. Supervised learning uses empirical observations to train a prediction model, in this case, the price changes Δp_t. The price changes Δp_t are used as target values in the loss function (8). TD learning uses a different way to construct the loss function. In a neural network as in (7), offsetting outputs and state inputs correspondingly would result in the same prediction, at least in expectations. In other words, if the neural network is trained properly, the following is true for $0 \le k \le t-1$,

$$u_t^\phi(x_0) = \mathbb{E}\left[u_{t-k}^\phi(x_k)|x_0\right]. \tag{12}$$

In (12), the output $u_t^\phi(x_0)$ estimates the price change t time instances subsequent to the observation of the state x_0, namely, Δp_t. On the right side, the output $u_{t-k}^\phi(x_k)$ estimates of the price change $t-k$ time instances subsequent to the observation of the state x_k, and this also estimates Δp_t, coinciding with the left side.

This equivalence of shifting in time allows us to use current model estimates as target values to construct a loss function. This leads to a major advantage of TD learning, that is, TD learning updates a prediction model based in part on current model estimates, without needing an entire observation episode. To apply this more concretely in this case, the loss function for SL method can be reformulated as below for a specific observation episode.

$$\mathcal{L}(\phi; x_0) = \frac{1}{T}\left[\Big(\Delta p_1 - u_1^\phi(x_0)\Big)^2 + \sum_{i=2}^{T}\Big(u_{i-1}^\phi(x_1) - u_i^\phi(x_0)\Big)^2\right]. \tag{13}$$

Notice that $u_1^\phi(x_0)$ is still matched to the price change Δp_1. For $i \geq 2$, $u_i^\phi(x_0)$ is matched to the current model estimate with a time shift $u_{i-1}^\phi(x_1)$. In effect, instead of using the entire episode of price changes as the target values, TD uses $[\Delta p_1, u_1^\phi(x_1), u_2^\phi(x_1), ..., u_{T-1}^\phi(x_1)]$ as the target values, substituting all but the first element by current model estimates with x_1 as input. The loss function in (13) effectively reaffirms the equivalence in (12) using squared loss.

For every $1 \leq t \leq T$, (12) defines a martingale

$$\{u_t^\phi(x_0), u_{t-1}^\phi(x_1), ..., u_{t-k}^\phi(x_k), ..., u_1^\phi(x_{t-1})\}. \tag{14}$$

That is, conditioned on the current state, the expected value of future prediction k time instances ahead is equal to the current prediction of the same time instance. If the predictions exhibits predictable variability, in principle, the prediction model could be improved. TD learning with loss function in (13) can be viewed as a way of regularizing the prediction model to satisfy the martingale property in (12). This form of regularization also has the benefit of preventing overfitting, which will be discuss in Section 6.2.

The data required to compute (13) is $(x_0, \Delta p_1, x_1)$, which is a subset of the observation episode. Any other consecutive 3-tuple of the form $(x_t, \Delta p_{t+1}, x_{t+1})$ can be used to compute (13) as well. Because TD learning requires only partial observations to compute the loss function, it allows us to update the neural network on the go.

Compared to the conventional SL method in Section 3.2, TD learning uses data more efficiently. Given the same amount of data, it updates the neural network many more times without using repeated data. In fact, given any observation episode such as (6), the loss function in (13) can be computed T times using all 3-tuples within the observation episode, updating the neural network T times. On the other hand, the conventional SL uses the loss function in (8) and can update the neural network only once. This advantage in data efficiency resolves the aforementioned data-wasting issue — TD utilizes all the state variables and price changes in an observation episode during training.

TD(m-step) Prediction:

We will refer to the updating method used in the conventional SL method outlined in Section 3.2 as the "empirical Monte Carlo (MC)" [2] update method. The MC update method trains a prediction model exclusively using samples from historical data observations. It turns out that there is a full spectrum of algorithms between TD and MC.

In (13), TD substitutes all but the first target value by current model estimates. This can be generalized to a family of TD methods by substituting fewer target values and keeping more observations. Specifically, we can construct a TD(m-step) method that uses m price changes and $T-m$ model estimates as target values. The loss function of TD(m-step) for a specific observation episode is

$$\mathcal{L}(\phi; x_0) = \frac{1}{T}\left[\sum_{i=1}^{m}\left(\Delta p_i - u_i^{\phi}(x_0)\right)^2 + \sum_{i=m+1}^{T}\left(u_{i-1}^{\phi}(x_m) - u_i^{\phi}(x_0)\right)^2\right]; \qquad m = 1, ..., T. \tag{15}$$

The data required to compute the above loss function is a $(m+2)$-tuple, given by

$$(x_0, \Delta p_1, \Delta p_2, ..., \Delta p_m, x_m), \tag{16}$$

and this can also be generalized to any $(m+2)$-tuple within the observation episode. TD(m-step) updates the neural network $T+1-m$ times using one observation episode.

Notice, when $m = T$, (15) becomes the same as (8). In other words, TD(T-step) is the same as the MC update method. When $m = 1$, TD(1-step) has the loss function in (13). The TD step size m is a hyper-parameter that controls the degree of TD when training the neural network. We will discuss the effect of the TD step size m in greater detail in Section 6.2.

Target Network:

Neural networks are typically trained using stochastic gradient descent (SGD). However, when SGD is applied to (13) and (15), the changes in the parameter ϕ would causes changes in both the prediction model and the target values. This links the model prediction and the target values, introducing instabilities into the training process. A popular way of this issues is by using a second "target" network that provides the target values during the training and is only updated periodically. This idea of using "double Q-learning" was first introduced by **?** and the usage of a target network is introduced by van Hasselt et al. [2016]. Adopting this idea, instead of a single neural network, we maintain two neural networks. These two neural networks need to have identical

[2] In this paper, our Monte Carlo updates utilize empirical samples, and do not require a generative model as in typical Monte Carlo simulations.

architectures and we denote their parameters by ϕ and ϕ', respectively,

$$\begin{aligned}\text{Train-Net:} \quad & NN^{\phi}(x) = [u_1^{\phi}(x), u_2^{\phi}(x), ..., u_T^{\phi}(x)] \\ \text{Target-Net:} \quad & NN^{\phi'}(x) = [u_1^{\phi'}(x), u_2^{\phi'}(x), ..., u_T^{\phi'}(x)].\end{aligned}$$

The train-net's parameter ϕ is the model that SGD changes during each iteration and the target-net is used exclusively for producing target values. The loss function can be written as

$$\mathcal{L}(\phi; x_0) = \frac{1}{T}\left[\sum_{i=1}^{m}\left(\Delta p_i - u_i^{\phi}(x_0)\right)^2 + \sum_{i=m+1}^{T}\left(u_{i-1}^{\phi'}(x_m) - u_i^{\phi}(x_0)\right)^2\right]; \qquad m = 1, ..., T. \tag{17}$$

The target-net also needs to be updated during the training so that it always provides accurate target values. Therefore, the train-net needs to be copied to the target-net periodically throughout the training procedure. The entire algorithm is outlined below in Section 3.5.

3.5. Algorithm

To summarize, the complete algorithm using supervised learning with TD(m-step) is displayed below. This algorithm will be referred to as the SL-TD(m-step) algorithm in the rest of this paper.

Algorithm 1: SL-TD(m-step)

Initialize ϕ and ϕ' randomly and identically;

while *not converged* **do**

1. From a random episode, select a random starting time t, sample a sub-episode $(x_t, \Delta p_{t+1}, ..., \Delta p_{t+m}, x_{t+m})$ for $0 \le t \le T - m$;
2. Repeat step 1 to collect a mini-batch of sub-episodes;
3. Compute the average loss value over the mini-batch using (17);
4. Take a gradient step on ϕ to minimize the average loss value;
5. Copy target-net with train-net ($\phi' \leftarrow \phi$) periodically;

end

To monitor the training progression, in-sample and out-of-sample MSE can be computed and monitored. Each iteration of neural network parameter ϕ induces a corresponding execution policy. Applying this execution policy to observation episodes either in sample or out of sample gives the price gains on these episodes. This measure of average price gains on observation episodes can also be used to monitor the training progression.

3.6. Insufficiency

We will use a hypothetical example to illustrate the insufficiency of the SL method outlined above. Let there be two possible future scenarios A and B for the price of a particular stock. Under these

two scenarios, price change trajectories over the next two time instances are

$$\Delta P_A = [\Delta p_1^A, \Delta p_2^A] = [+1, -4]; \quad \Delta P_B = [\Delta p_1^B, \Delta p_2^B] = [-2, +3].$$

Assume that these two scenarios occur with equal probability given all current information, namely,

$$P(A|x_0) = P(B|x_0) = 0.5.$$

Given this information, the ex-post optimal execution would be to sell at $t = 1$ under scenario A and sell at $t = 2$ under scenario B. This execution plan would yield an execution price of $+1$ under either scenario.

Now consider applying the SL method when only the state x_0 is observable. The neural network is trained using MSE and it's well known that the mean minimizes MSE. In other words, the optimal prediction would be

$$NN^{\phi}(x_0) = [u_1^*(x_0), u_2^*(x_0)] = P(A|x_0) \cdot \Delta P_A + P(B|x_0) \cdot \Delta P_B = [-0.5, -0.5].$$

This prediction indicates that future price changes will always be negative and therefore the trader should sell at $t = 0$ and induce an execution price of 0.

It's not a surprise that the ex-ante execution is inferior compared to the ex-post execution. However, this example also reveals a rather unsatisfactory aspect of the SL method — even with "optimal prediction", the SL method fails to capture the optimal execution. The trader stops too early and misses out on future opportunities.

4. Reinforcement Learning Approach

The SL method outlined above predicts the future price change trajectory for each state using neural networks. The predicted price change trajectory induces an execution policy, which can be applied sequentially to solve the optimal execution problem. However, the SL method doesn't lead to an optimal policy, which prompts us to turn to reinforcement learning (RL).

The insufficiency of the SL method discussed in Section 3.6 is mainly caused by the way SL averages predictions. SL produces an average prediction by simply averaging the price change trajectories under all possible scenarios, disregarding the fact that a trader might take different sequence of actions under each scenario. If the trader predicts a future price downturn, then the trader would stop and sell early. However, this price downturn, even though it can be predicted and avoided by the trader, is still accounted for in the SL prediction. In the example outlined in Section 3.6, Δp_2^A is one such price downturn. Including price downturns that can be predicted and avoided in the model predictions lead to a suboptimal policy.

RL averages trajectories from future scenarios differently. Instead of averaging the trajectories directly, RL allows the trader to take different sequence of actions under each scenario, and averages

the resulting rewards. This way, if a price downturn can be predicted and avoided by the trader, it won't be accounted for in the RL prediction. This leads to an improved execution policy compared to the SL method.

However, RL adds more complexity to the algorithms, especially during the training process. SL predicts price change trajectories, which are exogenous to the trader's policy. During training, as SL prediction becomes more accurate, the induced policy improves. On the other hand, because RL predicts future rewards, which are dependent on the execution policy, the target values of the prediction are no longer exogenous. While the RL model is being trained, the induced policy changes accordingly, which in turns also affects the future rewards. We will discuss how this difference complicates the training procedure in the rest of this section.

The procedure of applying RL to the sequential decision problem isn't all that different compared to the SL method. RL also uses neural networks to evaluate the "value" or "quality" of each state, which leads to an execution policy that can be applied sequentially. The main difference is that instead of predicting price change trajectories, RL predicts what's called continuation value.

4.1. Continuation Value

Continuation value is defined as the expected maximum reward over all remaining time instances when the immediate action is CONTINUE. Specifically, we write $C_{t:T}(x)$ to denote the continuation value when the current time is t and the current state is x. For all $t \in \mathcal{T} \setminus \{T\}$, this is defined as

$$C_{t:T}(x) \triangleq \sup_{\pi \in \Pi^0_{t:T}} \mathbb{E}_\pi \left[\sum_{i=t}^{\tau_\pi} \Delta p_i \middle| x_t = x \right]. \tag{18}$$

The set $\Pi^0_{t:T}$ contains all policies starting from time t that don't immediately stop at time t, i.e.

$$\Pi^0_{t:T} = \{(\pi_t, \pi_{t+1}, ..., \pi_T) \,|\, \pi_t = \text{CONTINUE}\}. \tag{19}$$

The stopping time τ_π is the stopping time associated with policy π and the expectation is taken assuming the policy π is used. Notice that for any fixed x, the value of $C_{t:T}(x)$ depends on the pair (t, T) only through $T - t$. By convention, $C_{t:t}(x) \triangleq 0$ for all states x and times t.

Optimal Policy:

Because the future price gain of STOP is always 0, the definition of the continuation value leads to a very simple execution policy — the trader should continue if and only if the continuation value is strictly larger than 0. At time t, if the current state is x, define an action function as

$$\pi^{\text{RL}}_t(x) \triangleq \begin{cases} \text{CONTINUE} & \text{if} \quad C_{t:T}(x) > 0 \\ \text{STOP} & \text{otherwise.} \end{cases}$$

The execution policy induced by the RL method is the sequence of such action functions defined at all time instances,

$$\pi^{\mathrm{RL}}(\cdot) = \{\pi_t^{\mathrm{RL}}(\cdot)\}_{t\in\mathcal{T}}. \tag{20}$$

When applying this policy sequentially to a sequence of states, the associated stopping time and the total price gain is given by

$$\tau_{\pi^{\mathrm{RL}}} \triangleq \min\{t \,|\, \pi_t^{\mathrm{RL}}(x_t) = \mathrm{STOP}\}; \quad \Delta P_{\tau_{\pi}^{\mathrm{RL}}} = p_{\tau_{\pi}^{\mathrm{RL}}} - p_0. \tag{21}$$

Bellman Equation:

The above definition of the continuation value leads to the following properties.

1. If the current time is $T-1$, there is only one time instance left and the continuation value is the expectation of the next price change,

$$C_{T-1:T}(x) = \mathbb{E}[\Delta p_T | x_{T-1} = x]. \tag{22}$$

2. If there is more than one time instance left, the continuation value is the sum of the next price change and the maximum rewards achievable over all remaining time. This leads to a Bellman equation given by

$$C_{t:T}(x) = \mathbb{E}\left[\Delta p_{t+1} + \max\{0, C_{t+1:T}(x_{t+1})\} | x_t = x\right]. \tag{23}$$

   If the trader follows the optimal policy starting at time $t+1$, the total reward accumulated after $t+1$ is precisely $\max\{0, C_{t+1}(x_{t+1})\}$. This is how (23) implicitly incorporated the execution policy.

Monotonicity:

At any time, the continuation value can be decomposed into a sum of increments of continuation values of stopping problems of increasing horizon. Because increasing time horizon allows more flexibility in trader's actions and can only increase the value of a stopping problem, these increments are non-negative. In other words, continuation values have a certain monotonicity.

At any time t, for any $i \geq 1$, define the continuation value increment as

$$\delta_i(x) \triangleq C_{t:t+i}(x) - C_{t:t+i-1}(x). \tag{24}$$

This is the difference in continuation values when the time horizon is $i-1$ time steps away and one time step is added. Then, for $i > 1$,

$$\delta_i(x) \geq 0. \tag{25}$$

When $i = 1$, the continuation value increment $\delta_i(x) = C_{t:t+1}(x) - C_{t:t}(x) = \mathbb{E}[\Delta p_{t+1} | x_t = x]$ can

be negative. Summing up these increments recovers the continuation value, namely,

$$C_{t:T}(x) = \sum_{i=1}^{T-t} \delta_i(x). \tag{26}$$

4.2. Learning Task

Unlike the price trajectory, the continuation value isn't directly observable from the data. Furthermore, the continuation value is dependent on the induced policy and the induced policy evolves as the continuation value changes. For these reasons, learning the continuation value is not a conventional supervised learning task.

In order for the induced policy to apply to each time instance in the sequential decision problem, the continuation value $C_{t:T}(x)$ needs to be estimated for each $t = 0, ..., T-1$. We design a neural network to learn the continuation value from data. Because the parameter t is discrete and have a fixed range, we incorporate this parameter directly into the neural network architecture. More specifically, the neural network takes a state x as an input and outputs a vector of T elements, representing each of the continuation value increment δ_i. The neural network can be represented as

$$\text{Neural Network:} \quad NN^{\phi}(x) = [u_1^{\phi}(x), u_2^{\phi}(x), ..., u_T^{\phi}(x)]. \tag{27}$$

This neural network contains T neurons on the output layer, and each neuron $u_i^{\phi}(x)$ is meant to approximate $\delta_i(x)$. As a result of this construction, the estimated continuation value is the summation of the neuron outputs, given by

$$\hat{C}_{t:T}^{\phi}(x) = \sum_{i=1}^{T-t} u_i^{\phi}(x). \tag{28}$$

There are two benefits of this neural network construction. One is that by incorporating time t as part of the architecture, it captures the commonality among continuation values for the entire time horizon. Training neural networks with auxiliary tasks, such as continuation values with different horizons, has shown benefits in other applications. This idea is referred to as "multi-task learning." Secondly, due to the monotonicity of the continuation values, the increments $\delta_i(x)$ should always be non-negative for $i > 1$. This implies that the true value of $u_i^{\phi}(x)$ is non-negative for $i > 1$. Using this architecture, we can also easily enforce this positivity on the output layer by applying the softplus activation function. This way the neural network output is consistent with the monotonicity property.

In order to train the neural network, we need to construct target values from observation episodes as in (6). We can compute the "empirical continuation value" at time t when the current state is x_t, given by

$$\tilde{C}_{t:T}(x_t) = \Delta p_{t+1} + \sum_{i=2}^{T-t} \Delta p_{t+i} \cdot \Pi_{j=1}^{i-1} \mathbb{1}\{\hat{C}_{t+j:T}^{\phi}(x_{t+j}) > 0\}, \tag{29}$$

where $\hat{C}^{\phi}$ is the continuation value estimated from the current neural network using (28). The right side of (29) includes the immediate price change Δp_{t+1} and other price changes conditionally. The price change Δp_{t+i} is only accounted for if the trader reaches time $t+i$. Because the trader follows the execution policy induced by the current model, this condition is expressed as $\Pi_{j=1}^{i-1}\mathbb{1}\{\hat{C}^{\phi}_{t+j:T}(x_{t+j}) > 0\}$.

The difference in the empirical continuation values is the empirical increments, given by

$$\tilde{\delta}_i(x) = \tilde{C}_{t:t+i}(x) - \tilde{C}_{t:t+i-1}(x). \tag{30}$$

This is the target value for $u_i^{\phi}(x)$. Now that the target values for the neural network outputs are in place, we can compute the mean squared error (MSE) loss function according to (31) and apply SGD to train the network parameters.

$$\mathcal{L}(\phi; x) = \frac{1}{T}\sum_{i=1}^{T}\left[u_i^{\phi}(x) - \tilde{\delta}_i(x))\right]^2. \tag{31}$$

4.3. TD Learning

The empirical continuation values can be obtained through TD learning as well. Instead of using empirical observations of price changes as in (29), the current model estimates of continuation values can be used to compute the empirical continuation values.

As described in Section 3.4, two neural networks are used, one for training and one for evaluating target values, given as follow,

$$\begin{aligned}
&\text{Train-Net:} \quad NN^{\phi}(x) = [u_1^{\phi}(x), u_2^{\phi}(x), ..., u_T^{\phi}(x)] \\
&\text{Target-Net:} \quad NN^{\phi'}(x) = [u_1^{\phi'}(x), u_2^{\phi'}(x), ..., u_T^{\phi'}(x)].
\end{aligned}$$

According to TD(1-step), the empirical continuation value $\tilde{C}_{t:T}(x_t)$ is the sum of the immediate price change Δp_{t+1} and the estimated continuation value at evaluated at state x_{t+1}, conditional on the trader reaching time $t+1$. This is given by

$$\tilde{C}_{t:T}(x_t) = \Delta p_{t+1} + \hat{C}^{\phi'}_{t+1:T}(x_{t+1}) \cdot \mathbb{1}\{\hat{C}^{\phi}_{t+1:T}(x_{t+1}) > 0\}; \quad \forall t \leq T-1. \tag{32}$$

Notice that on the right side of (32), the policy is induced using the train-net and the continuation value accumulated is evaluated from the target-net. This idea is commonly referred to as "Double Q-Learning" which was introduced by **?** and applied in DQN by van Hasselt et al. [2016]. The use of separate policies for control and continuation value estimation mitigates "error maximization", or bias that is introduced by the optimization over statistically estimated quantities. In this context, estimating the continuation value by maximizing over estimates of future continuation values under the same policy results in estimates that are systematically overestimated. This effect is mitigated by using different policies for the estimation of furture continuation values and the decision to stop in (32). The data used in (32) is $\{x_t, \Delta p_{t+1}, x_{t+1}\}$, which naturally extends to any

3-tuple of the same form. As discussed in Section 3.4, in the SL setting, TD learning can be viewed as a form of regularization to enforce that future price predictions satisfy the martingale property. In the RL setting here, TD learning enforces the Bellman equation, which can also be viewed as a form of regularization to enforce the time consistency of continuation value estimates.

The TD(m-step) can be applied as well, which expresses the empirical continuation value as

$$\tilde{C}_{t:T}(x_t) = \Delta p_{t+1} + \sum_{i=2}^{m} \Delta p_{t+i} \cdot \Pi_{j=1}^{i-1} \mathbb{1}\{\hat{C}^{\phi}_{t+j:T}(x_{t+j}) > 0\} + \hat{C}^{\phi'}_{t+m:T}(x_{t+m}) \Pi_{j=1}^{m} \mathbb{1}\{\hat{C}^{\phi}_{t+j:T}(x_{t+j}) > 0\}; \quad \forall t \leq T-m. \tag{33}$$

The data used in (33) is a $(2m+1)-$tuple

$$\{x_t, \Delta p_{t+1}, x_{t+1}, ..., \Delta p_{t+m}, x_{t+m}\}, \tag{34}$$

which can be generalized to any $(2m+1)-$tuple of the same form.

For TD(m-step), if the current time t is larger than $T-m$, then the current model estimates are no longer used as an additive terms in the computation of the target value. The target value of the continuation value is simply given by (29).

These TD methods computes the empirical continuation values, which leads to empirical increments. These increments help train the network networks as target values through the loss function in (31).

4.4. Algorithm

To summarize, the complete algorithm using reinforcement learning with TD(m-step) is displayed below. This algorithm will be referred to as the RL-TD(m-step) algorithm in the rest of this paper.

Algorithm 2: RL-TD(m-step)

Initialize ϕ and ϕ' randomly and identically;

while *not converged* **do**

1. From a random episode, select a random starting time t, and sample a sub-episode $(x_t, \Delta p_{t+1}, x_{t+1}.\Delta p_{t+2}, x_{t+2}, ..., \Delta p_{t+m}, x_{t+m})$ for $0 \leq t \leq T-m$;
2. Repeat step 1 to collect a mini-batch of sub-episodes;
3. Compute empirical continuation value increments and the average loss values using (31);
3. Take a gradient step on ϕ to minimize the average loss value;
3. Copy target-net with train-net ($\phi' \leftarrow \phi$) periodically;

end

When compared to the SL method, one critical difference in the RL method is that the target values for training the neural network is dependent on the induced policy. Therefore, the target value also changes during the training process. As a result, it's more difficult and perhaps less meaningful to monitor the MSE loss value during training. In order to monitor the training progress,

the induced policy can be applied to observation episodes either in sample or out of sample to produce price gains.

4.5. Discussion

The optimal stopping problem is challenging because the future prices are stochastic. A simplification would be to make the problem deterministic. The simplest deterministic model consistent with the stochastic dynamics is to replace random quantities with their expectations. In particular, at each time t, we replace the stochastic future price trajectory with its mean. The general idea of resolving a new, deterministic control problem at each instance of time falls under the rubric of Model Predictive Control (MPC). See Akesson and Toivonen [2006], for example, an application of neural networks in MPC.

In this context, the continuation value becomes

$$C_{t:T}^{MPC}(x) = \max_{1\leq h\leq T-t} \mathbb{E}\left[\sum_{i=1}^{h} \Delta p_{t+i} \middle| x_t\right]. \tag{35}$$

This motivates the SL method and the execution policy based upon (9). Notice that $C_{t:T}^{MPC}(x)$ is an underestimate of the true continuation value defined in (18), because it only optimizes over deterministic stopping times, while the true continuation value allows random stopping times. In other words,

$$C_{t:T}^{MPC}(x) \leq C_{t:T}(x).$$

Another simplification of the stochastic dynamics of the future price would be to use information relaxation, i.e., giving the trader the perfect knowledge of future price changes. There is a literature of information relaxation applied to stopping problems, which was pioneered by Rogers [2003] and Haugh and Kogan [2004]. In our context, one information relaxation would be to reveal all future prices to the decision maker. This would make the problem deterministic, and result in

$$C_{t:T}^{IR}(x) = \mathbb{E}\left[\max_{1\leq h\leq T-t} \sum_{i=1}^{h} \Delta p_{t+i} \middle| x_t\right] \tag{36}$$

as the value of continuation. This is clearly an overestimation of the true continuation because it optimizes with access to future information, while the true continuation value expressed in (18) only optimizes with access to information that is currently available.

$$C_{t:T}(x) \leq C_{t:T}^{IR}(x).$$

A regression approach can formulated to estimate this maximum value, and this is in the spirit of Desai et al. [2012]. We don't pursue this idea in this paper.

We can compare our method with the earlier work on optimal stopping problems in Longstaff and Schwartz [2001] and Tsitsiklis and Van Roy [2001]. In these regression-based methods, by

using a backward induction process that increases horizon one step at a time, a separate regression model is fitted for each time horizon. In our method, we fit a nonlinear neural network that predicts the continuation value for all time horizons. By this neural network architecture, the model is able to capture common features across time horizons. Additionally, in the RL method, due to the monotonicity, we know the incremental values δ_i are positive, as in (25). We apply softplus activation function on the output layer to enforce the positivity of the neural network outputs. This way, the estimated continuation values produced by the neural network also possess the desired monotonicity.

The idea of TD learning also manifest in the regression-based methods as well. Longstaff and Schwartz [2001] approximates the continuation value when the horizon is t using continuation values when the horizon is $t-1$. This is similar to the idea of RL-TD(1-step). Tsitsiklis and Van Roy [2001] approximates the continuation value when the horizon is t using future rewards under policies determined when the horizon was $t-1$. This is similar to the spirit of Monte Carlo update or RL-TD(T-step).

5. Numerical Experiment: Setup

The following two sections discuss the numerical experiments that test the SL and RL methods discussed above. The data source from NASDAQ used in the experiments is outlined in detail in Appendix A. This section will outline the setup of these experiments including the features, and neural network architectures.

5.1. Experiment Setup

Stock Selection:

The dataset we use is over the entire year of 2013, which contains 252 trading days. A set of 50 high-liquidity stocks are selected for this study. The summary statistics for these 50 stocks can be seen in the Appendix B (see Table 6).

For each stock, 100 observation episodes are sampled within each trading day, with the starting time uniformly sampled between 10am and 3:30pm New York time. Each episode consists of 60 one-second intervals. In other words, the time horizon is one minute and $T = 60$.

Train-Test Split:

Due to the complexity of the neural network and the ease of overfitting, it's imperative to separate the data for training and testing so that the reported results aren't overly optimistic. Specifically, the dataset of observation episodes is randomized into three categories, a training dataset (60%), a validation dataset (20%), and a testing dataset (20%). The randomization occurs at the level of a trading day. In other words, no two episodes sampled from the same day would belong to two

different categories. This is to avoid using future episodes to predict past episodes within the same day, as it introduces look-ahead bias and violates causality.

This randomization setup allows the possibility of using future days' episodes to predict past days' price trajectories. However, because the execution horizon is as short as a minute and the features selected mostly capture market microstructure, we deem the predictabilities between different days as negligible.

Testing Regimes:

We consider two regimes under which the models can be trained and tested. One is the "stock-specific" regime where a model is trained on a stock and tested on the same stock. The other is the "universal" regime where all the data of 50 stocks is aggregated before training and testing. This regime presumes that there is certain universality in terms of the price formation process across stocks. Specifically, the experiences learned from trading one stock can be generalized to another stock.

5.2. State Variables and Rewards

State Variables:

In a limit order book market, the current condition of the market represents the collective preferences of all the investors, and therefore can have predictive power for the immediate future. In order to capture this predictability, we have extracted a set of features from the order book to capture market conditions. This set of features can be sampled at any given time during the trading hours. The complete set of features and their descriptions can be found in the Appendix C (see Table 7).

In addition to the current market condition, the past dynamics of the market can have strong indictions of the future evolution as well. To better capture this temporal predictability, the same set of features is collected not only at the current time, but also at each second for the past 9 seconds. This entire collection of 10 sets of features collectively represent the market state variable. More specifically, let s_t be the set of features collected at time t. Then the state variable defined by $x_t = (s_{t-9}, s_{t-8}, ..., s_t)$ is a time series of recent values of 1these features, available at time t.

Normalized Price Changes/Rewards:

We selected a diverse range of stocks with an average spread ranging from 1 tick to more than 54 ticks. The magnitudes of the price changes of these stocks also varied widely. As a result, it's inappropriate to use price changes directly as rewards when comparing different stocks. Instead, we normalized the price changes by the average half-spread, and use these quantities as rewards. In effect, the price gains are computed in units of percentage of the half-spread. If the price gain is exactly the half-spread, then the trade is executed at the mid-price. Thus, if the normalized price gain achieves 100%, then the trader is effectively trading frictionlessly. In the implementation, the average half-spread is taken to be the time-averaged half-spread in the previous 5 trading days.

Recurrent Neural Network (RNN):

RNN is specifically designed to process time series of inputs (see Figure 2). Sets of features are ordered temporally and RNN units connect them them horizontally. The output layer is of dimension 60, matching the time horizon T. For the RL method, the monotonicity of the continuation value implies that the output neurons are non-negative except the $u_1^{\phi}(x)$. To enforce this positivity, the softplus activation function is applied to the output layer in the RL settings. A more detailed description of the neural network architecture can be found in the Appendix D.

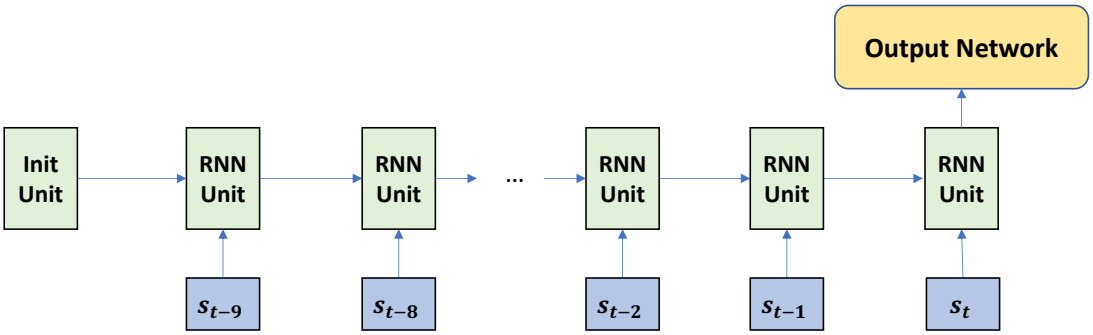

Figure 2: Recurrent Neural Network (RNN) Architecture

6. Numerical Experiment: Results

This section presents the results of the numerical experiments and discusses the interpretation of these results.

6.1. Best Performances

TD learning is applied to both the SL and RL method, with various update step m (see Section 3.4). These algorithms, SL-TD(m-step) and RL-TD(m-step), are trained using the training data, tuned with the validation data, and performances are reported using the testing data. Neural network architecture, learning rate, update step m, and other hyper-parameters are tuned to maximize the performance. The best performances using SL and RL are reported in Table 1. These figures are price gains per episode averaged over all 50 stocks. The price gain is reported in percentage of half-spread. The detailed performance for each stock can be found in Appendix (see Table 9).

Given sufficient data and time, the RL method outperforms the SL method. This is true under both the stock-specific regime and the universal regime. The models trained under the universal regime generally outperform the models trained under the stock-specific regime as well.

Price Gain (% Half-Spread)	**SL** (s.e.)	**RL** (s.e.)
Stock-Specific	21.40 (0.15)	24.82 (0.16)
Universal	22.34 (0.15)	25.47 (0.16)

Table 1: The universal model outperforms the stock-specific models with both SL and RL by 4.4% and 2.6%, respectively. RL outperforms SL under the stock-specific and universal regime, by 16% and 14%, respectively. The figures reported are in units of percentage of half-spread (% half-spread), and are computed out of sample on the testing dataset.

6.2. Comparative Results

Both SL and RL method are specified by TD learning with various update step m (see Section 3.4). These TD specifications extend SL and RL method to two families of algorithms, SL-TD(m-step) and RL-TD(m-step). The update step m controls the target values of the neuron network during training. Specifically, among T neurons in the output layer, m of them are matched to the empirical observations and $T - m$ are matched to the current model estimates. Different values of m and the difference between SL and RL presents various tradeoff in algorithm performance, which we will discuss shortly.

We will evaluate these algorithms using a few metrics, including their rate of convergence with respect to gradient steps, running time, their data efficiencies, and bias-variance tradeoff.

Rate of Convergence (Gradient Steps):

Figure 3 plots the price gain progression with respect to the number of gradient steps taken. As we can see from this figure, after controlling for the learning rate, batch size, neural network architecture, and other contributing factors, the RL method requires more gradient steps in SGD to converge compared to the SL method. It's also apparent that the convergence is slow when the update step m is small.

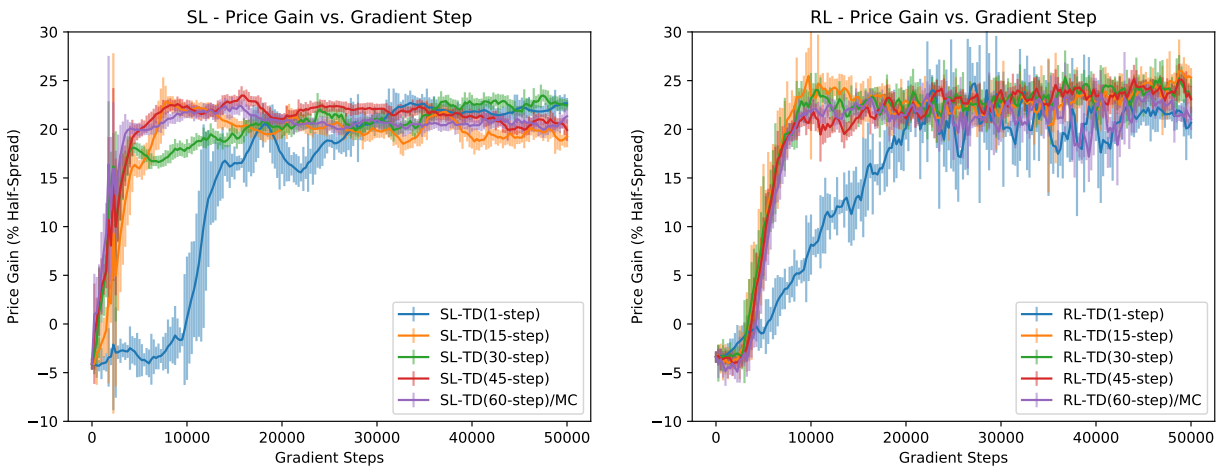

Figure 3: Price Gain vs. Gradient Steps. The trajectories are averaged over 10 random starts, the shaded regions correspond to the standard error of each trajectory.

Running Time:

Training neural networks can be time-consuming. Perhaps the most time-consuming part is iteratively taking gradient steps as part of the SGD procedure. Because a neural network typically takes thousands of steps to train, the time it takes to perform a single gradient step is an important measurement to evaluate the running time of an algorithm. We will refer to this time as the gradient step time.

We have measured the average gradient step time over 50,000 gradient steps in SL-TD(m-step) and RL-TD(m-step). The result is plotted in Figure 4.

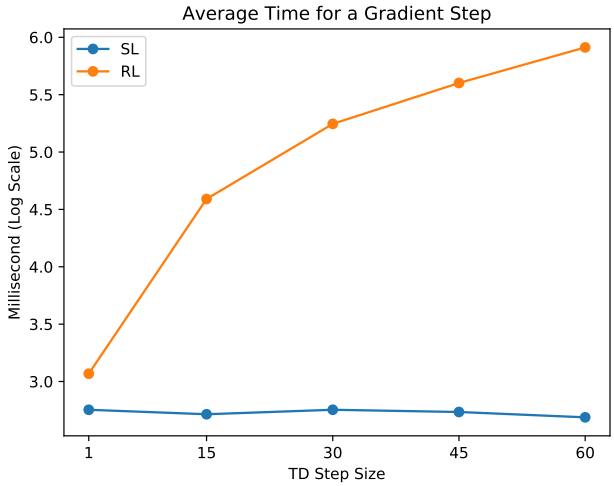

Figure 4: Average gradient step time (in log scale) over 50k gradients steps. The standard errors are negligible and thus not shown.

There are many factors that contributes to the gradient step time, such as the power of the CPU, the implementation choices and others. We have controlled all these factors so that the differences displayed in Figure 4 is solely due to the differences in TD step size m [3] The actual values of the gradient step time are not important, but it is clear that the gradient step time increases as the step size m increases in RL method, but stays flat in SL method.

This difference between SL and RL method comes down to the difference in the loss functions. In the SL method, in order to compute the the loss function for a specific data observation, it requires two neural network evaluations, namely $u^{\phi}(x_0)$ and $u^{\phi'}(x_m)$. This is true for all SL-TD(m-step), except for SL-TD(60-step). In SL-TD(60-step), (8) only evaluates the train-net once. This explains why gradient step time is relatively constant for different values of m in SL-TD(m-step).

On the other hand, in RL-TD(m-step), computing the loss function requires m neural network evaluations, which scales linearly with m. This can be seen in (33). This explains why gradient step time roughly scales proportionally with the m in RL-TD(m-step).

Figure 5 plots the price gains progression with respect to elapsed running time. Among RL-TD(m-step), RL-TD(1-step) converges slowest with respect to gradient steps (see right figure in Figure 3). However, because each gradient step takes much less time, RL-TD(1-step) actually converges fastest in term of running time among all RL methods. In other words, given a fixed limited amount of time, RL-TD(1-step) achieves the best performance within all RL methods.

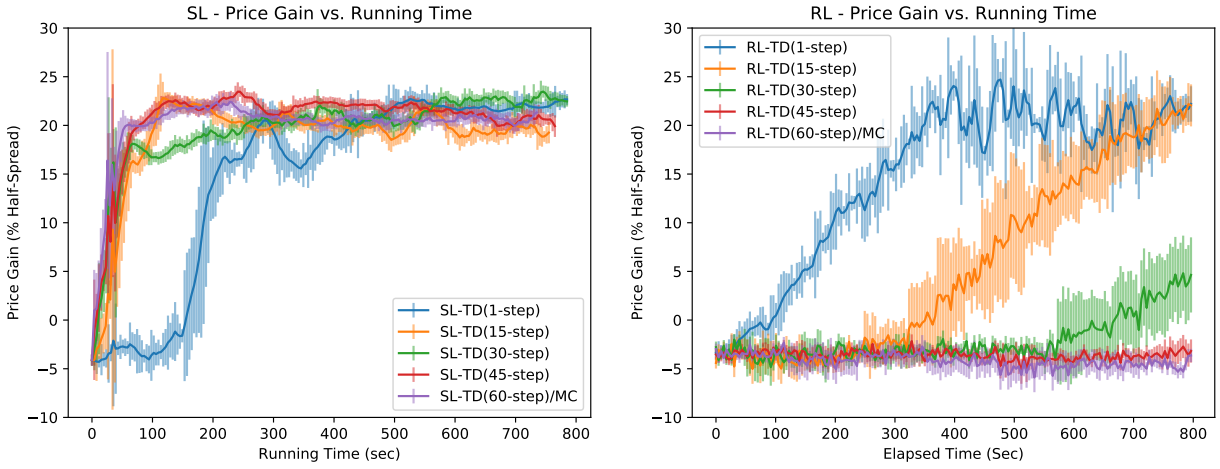

Figure 5: Price Gain vs. Running Time. The trajectories are averaged over 10 random starts, the shaded regions correspond to the standard error of each trajectory.

Data Efficiency:

The central idea of TD learning is to use current model estimates instead of actual data observations to train models. Naturally, with different step sizes, TD method uses data differently. In the

[3]The results in Figure 4 use the aforementioned RNN neural network architecture described in Appendix D. But qualitatively, the results hold for general neural network architectures as well, as the differences in running time is caused by the complexity in the loss function evaluation.

SL method, TD(m) uses $(x_t, \Delta p_{t+1}, \Delta p_{t+2}..., \Delta p_{t+m}, x_{t+m})$ to update the neural network once. This is counted as m time instances worth of data. Similarly, in the RL method, TD(m) uses $(x_t, \Delta p_{t+1}, x_{t+1}.\Delta p_{t+2}, x_{t+2}, ..., \Delta p_{t+m}, x_{t+m})$ to update the neural network once. We also regard this as m time instances worth of data. Notice, however, for each intermediate time instance $t+k$, RL method uses both the state variable x_{t+k} and the price change Δp_{t+k}, whereas the SL method only uses the price change Δp_{t+k}.

For SL-TD(m-step) and RL-TD(m-step), it takes m time instances worth of data observations to perform a gradient step. In other words, for a larger m, each gradient step in TD(m-step) is more informed as the data length is larger. However, given the same amount of data, TD(m-step) with a larger m can not perform as many gradient steps than TD(m-step) of a smaller m without reusing data.

This has important implication to the choice of algorithms, especially in finance. Because (over longer time horizons such as months or years) financial market is time-varying and non-stationary, only the recent historical data can be used to train models to predict the future. In situations like this where the duration of the usable historical data is relatively limited, a more data-efficient algorithm can potentially produce a more complex model with the same duration of historical data.

One way to evaluate the data efficiency of an algorithm is to evaluate its performance based on how much data it has accessed, measured in time instances. In our implementations, at any given time, the price p_t and the state variable x_t can be used to train the models. Using either the price, the state variable, or both all count as accessing one time instance of data. Figure 6 plots the price gain progression with respect to quantity of data accessed. It shows that TD(1-step) is the most data-efficient, in either SL or RL method. In other words, when data is limited, TD(1), the method that uses the least information to perform a gradient step, performs the best because it can take more gradient steps with the same amount of data.

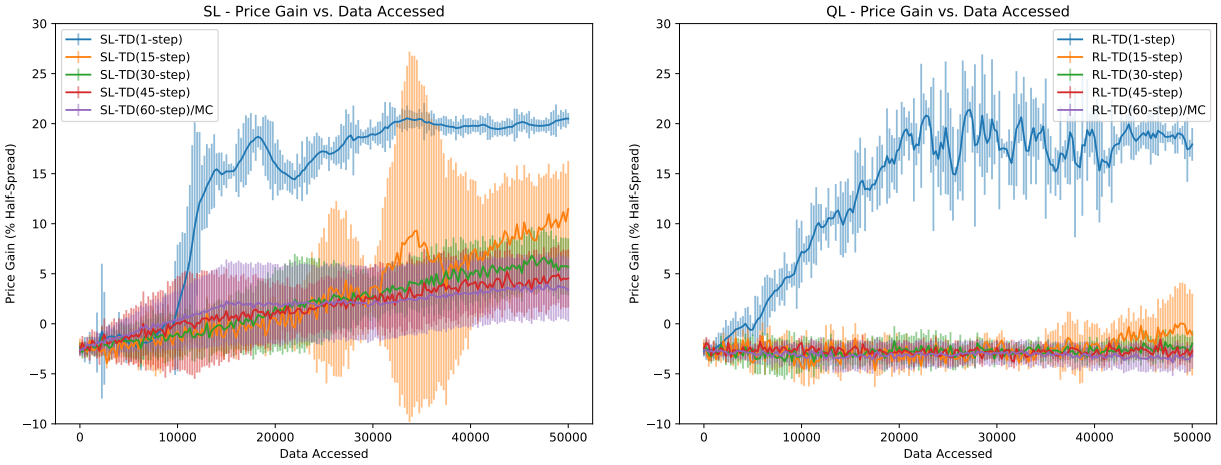

Figure 6: Price Gain vs. Data Points Accessed. The trajectories are averaged over 10 random starts, the shaded regions correspond to the standard error of each trajectory.

Bias–Variance Tradeoff:

The bias–variance tradeoff has been a recurring theme in machine learning, and it's especially relevant in a discussion of TD learning. Previous studies have reported that TD update generally leads to higher bias and lower variance compared to Monte Carlo update when applied to the same prediction model (see Kearns and Singh [2000] and Francois-Lavet et al. [2019]). We observe a similar pattern in our experiment.

As part of the SL method, the neural network is used to predict price change trajectories given an observable state variable. Consider a particular state x_0, and let $f_i(x_0)$ be the true price change at the ith time instances ahead. Then the price change trajectory can be represented as a vector of price changes $f(x_0) = [f_1(x_0), f_2(x_0), ..., f_{60}(x_0)]$. Let y_i be the observable price change at the ith time instance. Then $y_i = f_i(x_0) + \epsilon_i$, and the observable price change trajectory is $y = [y_1, y_2, ..., y_{60}]$.

Consider a set of training datasets, $\mathcal{D} = \{D_1, D_2, ..., D_n\}$. A neural network can be trained on each training dataset and produce a predicted trajectory in the SL method, denoted by $\hat{f}(x_0; D) = [\hat{u}_1(x_0; D), \hat{u}_2(x_0; D), ..., \hat{u}_{60}(x_0; D)]$. Averaging all these predictions from each dataset give the average ith price change prediction $\bar{u}_i(x_0) = \frac{1}{n}\sum_{i=1}^{n} \hat{u}_i(x_0; D_i)$ and the average price change trajectory prediction $\bar{f}(x_0) = \frac{1}{n}\sum_{i=1}^{n} \hat{f}(x_0; D_i)$. We now arrive at the following bias variance decomposition for the prediction of the ith interval:

$$\begin{aligned}
\text{MSE}_i(x_0) &= \mathbb{E}_{D\in\mathcal{D}}\left[(y_i - \hat{u}_i(x_0; D))^2\right] && (37)\\
&= \epsilon_i^2 + [f_i(x_0) - \bar{u}_i(x_0)]^2 + \mathbb{E}_{D\in\mathcal{D}}\left[(\bar{u}_i(x_0) - \hat{u}_i(x_0; D))^2\right] && (38)\\
&= [y_i - \bar{u}_i(x_0)]^2 + \mathbb{E}_{D\in\mathcal{D}}\left[(\bar{u}_i(x_0) - \hat{u}_i(x_0; D))^2\right]. && (39)
\end{aligned}$$

Equation (38) is the common bias–variance decomposition, where ϵ_i^2 is the irreducible noise variance, $[f(x_0) - \bar{u}(x_0)]^2$ is the squared bias term, and $\mathbb{E}_{D\in\mathcal{D}}\left[(\bar{u}(x_0) - \hat{u}(x_0; D))^2\right]$ is the prediction variance. This decomposition can be reformulated as (39). Each term in (39) is observable and thus can be measured empirically. We will refer to $[y_i - \bar{u}_i(x_0)]^2$ as noise variance squared bias (NVSB).

A set of 100 training datasets are used, each producing a unique neural network. Testing these neural networks on the same testing dataset produces MSE, prediction variances, and NVSB for each time instance. A square root is taken of these values to obtain root mean squared error (RMSE), the prediction standard deviations (pred. std.), and the noise standard deviation bias (NSDB). These values are averaged across all time instances and plotted in Figure 7.

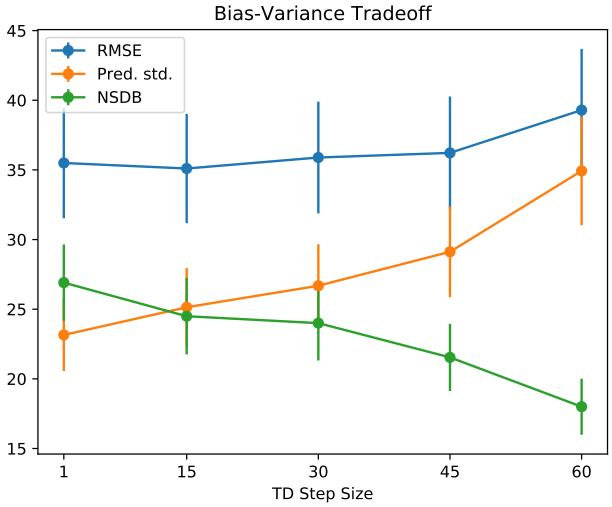

Figure 7: Bias–Variance Tradeoff vs. Step Size. Standard errors bars are plotted.

It's clear that there is a bias-variance tradeoff — TD with a smaller step size reduces variance and increases bias, and TD with a larger step size increases variance and reduces bias. A large prediction variance typically leads to overfitting. Indeed, this can also be observed empirically. When training the SL method using a small training dataset, the in-sample RMSE of TD(60-step) decreases quickly while its out-of-sample RMSE increases (see Figure 8). This is because TD(60-step) fits to the noisy patterns in the training data that don't generalize out of sample. Using the same training and testing data, TD(1-step) and TD(30-step) don't overfit nearly as much as TD(60-step).

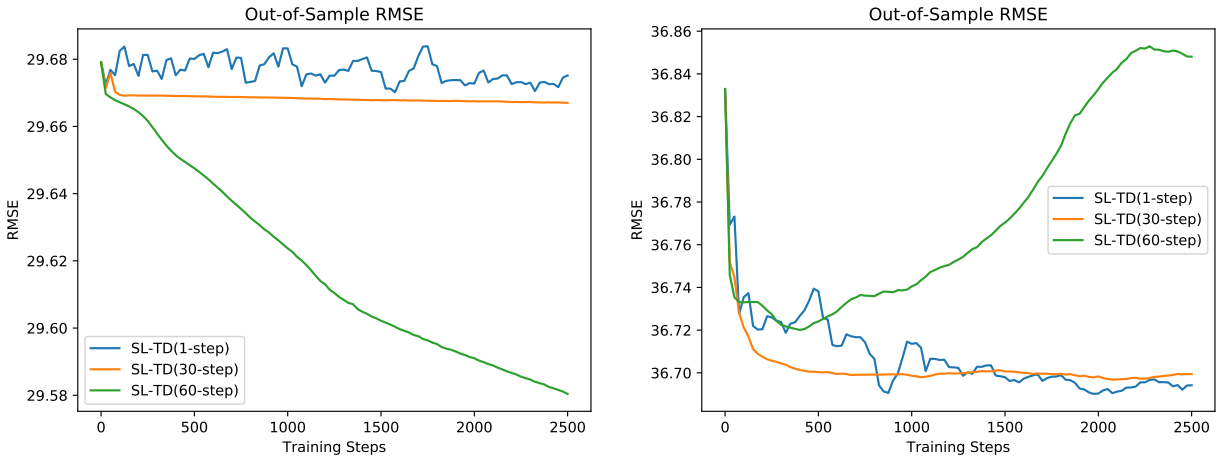

Figure 8: Left: In-sample RMSE; Right: Out-of-sample RMSE.

6.3. Universality

In Section 6.1, the universal models outperforms the stock-specific models. This reveals certain universality across stocks, that is, the experience learned from one stock can be generalized to a different stock. To further reinforce the evidence of this universality, we conduct another experiment under the "take-one-out" regime. Under the "take-one-out" regime, a model is trained on 49 stocks and tested on the stock that has been left out of the training. This way, the reported testing performance is out of sample in the conventional machine-learning sense and also on a stock that isn't part of the training data.

Table 2 displays the average performance of models trained under all three regimes. The detailed performance for each model can be found in Appendix (see Table 9). The take-one-out models performance comparable to the stock-specific models, indicating evidence of universality across stocks. However, the best performing model is still the universal model. This implies that there are still values in specific stocks.

Price Gain (% Half-Spread)	**SL** (s.e.)	**RL** (s.e.)
Stock-Specific	21.40 (0.15)	24.82 (0.16)
Take-one-out	21.55 (0.15)	24.85 (0.16)
Universal	22.34 (0.15)	25.47 (0.16)

Table 2: Performance comparison among models trained under all three regimes.

6.4. Result Summary

There isn't a single algorithm that is the most superior in all aspects. Rather, different algorithms might be preferable under different situations. The following lists some of these insights determined through the numerical results:

- Max Performance:
  - The RL method outperforms the SL method.
  - Universal model outperforms stock-specific model.

  If data and time aren't binding constraints and the goal is to maximize the performance, the universal RL model performs the best and is recommended for this situation.

- Time Limitation:
  - SL Method: Monte Carlo update method converges fastest.
  - RL Method: TD(1-step) update method converges fastest.

  If time is the binding constraint, then a fast algorithm is preferable. For the SL method, Monte Carlo update method (SL-TD(T-step)) is fastest with respect to running time. For the RL method, TD(1-step) provides the fastest convergence with respect to running time.

- Data Limitation:
  - SL Method: TD(1-step) update method is most data-efficient.
  - RL Method: TD(1-step) update method is most data-efficient.

  If the amount of data is the binding constraint, then a data-efficient algorithm is preferable. TD(1-step) provides the most data-efficient algorithms, for both SL method and the RL method.

- Prevent Overfitting:
  Monte Carlo update method leads to a high-variance and low-bias prediction model, which is prone to overfitting. TD learning leads to a low-variance and high-bias prediction, which provides the benefit of preventing overfitting.

References

B. M. Akesson and H. T. Toivonen. A neural network model predictive controller. *Journal of Process Control*, 16:937–946, 2006.

R. Almgren and N. Chriss. Optimal execution of portfolio transactions. *Journal of Risk*, 3(2):5–39, 2000.

S. Becker, P. Cheridito, and A. Jentzen. Deep optimal stopping. *Journal of Machine Learning Research*, 20, 2019.

S. Becker, P. Cheridito, and A. Jentzen. Pricing and hedging american-style options with deep learning. *Journal of Risk and Financial Management*, 13(7), 2020.

S. Becker, P. Cheridito, A. Jentzen, and T. Welti. Solving high-dimensional optimal stopping problems using deep learning. *European Journal of Applied Mathematics*, 32(3):470–514, 2021.

D. Bertsimas and A. W. Lo. Optimal control of execution costs. *Journal of Financial Markets*, 1: 1–50, 1998.

R. Coggins, A. Blazejewski, and M. Aitken. Optimal trade execution of equities in a limit order market. *IEEE International Conference on Computational Intelligence for Financial Engineering*, 10(1109), 2003.

V. Desai, V. Farias, and C. Moallemi. Pathwise optimization for optimal stopping problems. *Management Science*, 58(12):2292–2308, 2012.

R. El-Yaniv, A. Fiat, R. M. Karp, and G. Turpin. Optimal search and one-way trading online algorithms. *Algorithmica*, pages 101–139, 2001.

V. Francois-Lavet, G. Rabusseau, J. Pineau, D. Ernst, and R. Fonteneau. On overfitting and asymptotic bias in batch reinforcement learning with partial observability. *Journal of Artificial Intelligence Research*, 65, 2019.

R. Gaspar, S. Lopes, and B. Sequeira. Neural network pricing of american put options. *Risks*, 8 (3), 2020.

M. Haugh and L. Kogan. Pricing american options: A duality approach. *Operations Research*, 52 (2), 2004.

C. Herrera, F. Krach, P. Ruyssen, and J. Teichmann. Optimal stopping via randomized neural networks. 2021.

M. Kearns and S. Singh. Bias-variance error bounds for temporal difference updates. *COLT: Proceedings of the Thirteenth Annual Conference on Computational Learning Theory*, pages 142–147, 2000.

A. Kim, C. Shelton, and T. Poggio. Modeling stock order flows and learning market-making from data. *AI Memo*, 2002.

F. Longstaff and E. Schwartz. Valuing american options by simulation: A simple least-squares approach. *The Review of Financial Studies*, 14(1):113–147, 2001.

NASDAQ TotalView-ITCH 4.1. NASDAQ Stock Exchange, 2010. URL `http://www.nasdaqtrader.com/content/technicalsupport/specifications/dataproducts/nqtv-itch-v4_1.pdf`.

Y. Nevmyvaka, Y. Feng, and M. Kearns. Reinforcement learning for optimazed trade execution. *International Conference on Machine Learning*, pages 673–680, 2006.

A. Obizhaeva and J. Wang. Optimal trading strategy and supply/demand dynamics. *Journal of Financial Markets*, 16(1):1–32, 2013.

B. Park and B. Van Roy. Adaptive execution: Exploration and learning of price impact. *Operations Research*, 63(5):1058–1076, 2015.

L.C.G. Rogers. Monte carlo valuation of american options. *Mathematical Finance*, 2003.

J. A. Sirignano. Deep learning for limit order books. *Quantitative Finance*, 19(4):549–570, 2019. URL `https://doi.org/10.1080/14697688.2018.1546053`.

R. Sutton and A. Barto. *Reinforcement Learning*. ISBN 978-0-585-0244-5. MIT Press, 1998.

J. Tsitsiklis and B. Van Roy. Regression methods for pricing complex american-style options. *IEEE Transactions on Neural Networks*, 12(4), 2001.

H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. *AAAI'16: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pages 2094–2100, 2016.

A. NASDAQ Data Source

The NASDAQ ITCH dataset provides level III market data from the NASDAQ stock exchange [NAS, 2010]. This dataset contains event messages for every event that has transpired at the exchange. Common market events include "add order", "order executed", and "order cancelled". These market events occur throughout the trading hours and constantly change the limit order book (LOB). An example of an "add order" event message is shown below in Table 3.

time	ticker	side	shares	price	event
9:30:00.4704337	BAC	B	2000	12.02	"A"

Table 3: The event reads: A bid limit order of 2000 shares of BAC stock is added to the LOB at price level $12.02 at 9:30:00.4704337.

From these event messages, a limit order book can be constructed to display the prices and numbers of resting shares (depth) at each price level. This system is dynamic and it changes every time a new event occurs in the market.

time	b.prc 5	b.prc 4	b.prc 3	b.prc 2	b.prc 1	a.prc 1	a.prc 2	a.prc 3	a.prc 4	a.prc 5
9:30:00.4704337	12.01	12.02	12.03	12.04	12.05	12.06	12.07	12.08	12.09	12.10
9:30:00.8582938	12.01	12.02	12.03	12.04	12.05	12.06	12.07	12.08	12.09	12.10

Table 4: The above table is a snapshot of the LOB displaying the prices of the top 5 price levels on both sides of the market before and after the event from Table 3. The event from Table 3 doesn't change the prices at each level.

time	b.prc 5	b.prc 4	b.prc 3	b.prc 2	b.prc 1	a.prc 1	a.prc 2	a.prc 3	a.prc 4	a.prc 5
9:30:00.4704337	10000	43700	13100	12100	7500	5200	15300	15900	17000	22200
9:30:00.8582938	10000	41700	13100	12100	7500	5200	15300	15900	17000	22200

Table 5: The above table is a snapshot of the LOB displaying the number of shares on the top 5 price levels on both sides of the market before and after the event from Table 3. The event from Table 3 reduces 2000 shares at price $12.02.

The limit order book reflects the market condition at any given moment and this provides the environment of the optimal execution problem.

B. Summary Statistics of Selected Stocks

Stock	Volume ($M)	Avg. Prices ($)	Price Vol.($)	Return Vol.	One Tick (%)	Spread
AAPL	94768.13	472.28	44.56	29%	0%	13.60
ADBE	3999.91	46.73	5.98	25%	81%	1.39
ADI	2441.46	46.57	2.16	20%	72%	3.26
ADP	3572.48	69.66	6.01	14%	59%	4.53

ADSK	2754.37	39.11	3.53	27%	70%	3.00
AMAT	3664.65	15.25	1.76	28%	99%	1.14
AMD	535.55	3.38	0.60	52%	98%	1.02
AMGN	9096.26	103.89	10.12	27%	18%	4.61
AMZN	17102.94	297.89	41.59	26%	0%	16.40
ATVI	2298.19	15.39	1.90	38%	98%	2.01
AVGO	2155.93	38.50	5.02	31%	61%	4.50
BAC	11935.64	13.44	1.31	23%	99%	1.01
BRK.B	3578.57	110.20	7.20	15%	11%	6.71
CHTR	1985.65	113.96	19.16	28%	3%	17.75
CMCSA	10030.18	43.23	3.40	20%	95%	1.27
COST	4996.78	111.86	7.00	14%	14%	4.58
CSCO	14958.94	22.69	1.83	34%	99%	1.02
CSX	2049.05	24.78	1.84	19%	96%	1.47
DIS	5904.70	62.83	5.71	18%	76%	1.81
EBAY	11696.10	53.45	1.88	25%	84%	1.46
F	4821.01	15.34	1.72	24%	98%	1.02
FB	32453.19	34.59	10.56	48%	93%	0.99
FISV	1481.28	91.68	11.36	53%	7%	8.31
GE	7809.10	23.99	1.62	17%	98%	1.04
GILD	11996.61	58.60	10.77	58%	66%	2.20
GS	7129.39	156.51	9.46	21%	2%	8.48
ILMN	1790.24	72.74	16.52	33%	7%	10.72
INTC	13742.28	23.05	1.34	20%	98%	0.99
INTU	3664.81	65.12	4.79	20%	47%	5.28
ISRG	4161.03	450.20	69.49	35%	0%	54.36
JNJ	10063.27	85.73	6.67	13%	75%	3.15
JPM	15719.47	51.85	3.35	19%	93%	1.71
LRCX	2413.90	47.00	4.62	25%	48%	3.19
MDLZ	6152.37	30.67	2.14	20%	97%	1.89
MELI	1224.31	109.33	16.36	37%	1%	27.96
MRK	7717.00	46.40	2.31	17%	93%	2.09
MSFT	27291.37	32.47	3.44	25%	98%	1.08
MU	9123.92	13.36	4.43	38%	98%	1.07
NFLX	15554.60	246.42	73.44	65%	0%	21.53
NVDA	2325.16	14.18	1.25	21%	98%	1.27
PEP	6836.76	80.35	4.14	14%	73%	3.14
QCOM	15814.58	66.36	3.38	18%	88%	2.37
SBUX	7015.92	67.38	9.18	19%	62%	1.46
T	8735.23	35.43	1.25	15%	97%	1.25
TXN	5857.73	37.61	3.27	18%	95%	1.42
UPS	4350.56	88.62	6.75	14%	42%	3.35
V	7143.93	180.35	16.65	21%	3%	15.47
VRTX	2983.46	68.01	13.45	70%	9%	10.04
VZ	7297.25	48.66	2.66	17%	92%	1.92
WFC	10620.15	40.17	3.27	16%	97%	1.11

Table 6: Descriptive statistics for the selected 50 stocks over 2013. Average price and (annualized) volatility are calculated using daily closing price. Volume ($M) is the average daily trading volume in million dollars. One tick (%) is the percentage of time during trading hours that the spread is one tick. Spread is the time-averaged difference between best bid price and best ask price, in ticks.

C. Features Used in State Variables

The set of features that make up the state s_t at any given time t is listed in Table 7. These features are designed to be symmetric between buying and selling. In order words, the side of the market (bid or ask) is only distinguishable relative to the intended trade direction as near-side and far-side. Near-side is the side at which the execution seeks to fulfill an order, and the far-side is the opposite side. Namely, for a buying order, the near-side is the bid-side and the far-side is the ask-side. For a selling order, the near-side is the ask-side and the far-side is the bid-side.

Category	**Features**
General Information	Time of day, Price normalized by average spread
Spread	Spread, Spread normalized by return volatility, Spread normalized by price volatility
Depth (top 5 price levels)	Queue imbalances, Near depths normalized by average daily volume, Far depths normalized by average daily volume
Trading Flow	Number of trades in near-side and in far-side within the last second, Number of price changes towards near-side and far-side within the last second
Intensity Measures	Intensity measure for trades at near-side and far-side, price changes at near-side and far-side

Table 7: State variable features.

- For a sell order, the price p_t of the stock is taken to be the best bid price; for a buy order, the price p_t of the stock is taken to be the best ask price. The price is normalized by average spread, which is taken to be the trailing 5 day time-averaged spread.
- Return volatility and price volatility are computed using the adjusted closing daily price for the previous 21 days.
- Near depths and far depths refer to the number of outstanding shares at each of the top 5 price levels at respective sides. These values are normalized by the trailing 21-day average daily trading volumes in shares.
- Queue imbalance is defined as

$$QI = \frac{\text{near depth} - \text{far depth}}{\text{near depth} + \text{far depth}}.$$

This is a value between -1 and 1 and represents the imbalance of the supply and demand of the stock at the current price level. This can be calculated using depths at the top price levels and aggregated depth at the top 5 price levels. We compute the QI for the each of the top 5 price levels to be used as features.

- Intensity measure of any event is modeled as an exponentially decaying function with increments only at occurrences of such an event. Let S_t be the magnitude of the event at any given time t, $S_t = 0$ if there is no occurrence of such event at time t. The intensity measure $X(t)$ can be modeled as

$$X(t + \Delta t) = X(t) \cdot \exp(-\Delta t/T) + S_{t+\Delta t}.$$

At any time t and for any duration Δt, if there is no event occurrence between t and $t + \Delta t$, then the intensity measure decays exponentially. The time constat T controls the rate of the decay.

In our implementation, we measure the intensity measure for trades and price changes on the near-side and far-side. The magnitude of a trade is the size of the trade in dollars, normalized by average daily volume. The magnitude of a price change is normalized by the average spread.

D. Neural Network Architecture

The RNN architecture is designed to process the state variables described in Section 5.2 and Appendix C. The state variable consists of 10 sets of features, thus, the neural network has 10 RNN units, each takes a set of feature as input, as illustrated by Figure 2. The RNN Units are implemented as LSTM units with dimension 64. The output of the LSTM units go through another 5 layers of fully-connected network, which is denoted as "Output Network" in Figure 2.

In our implementation, because the execution horizon is made of 60 time intervals, the output of the neural network also has dimension of 60. In the SL method, the last layer of the output network is a linear layer with dimension 60. In the RL method, due to the monotonicity of the continuation value, all the continuation value increments are non-negative with the exception of the first one. This is described by equation (25). To enforce this, in the RL method, softplus activation function is applied to all units in the last layer except the first one. This is the enforce the positivity of the neural network output. The Python Tensorflow implementation of the neural network in the RL method is provided in Table 8.

```python
import tensorflow as tf
import tensorflow.contrib.layers as layers

def rnn_model(inpt, rnn_states, rnn_seq, output_dim, scope, phase):
    # inpt: an array that contains the data within each batch in the SGD algorithm
    # inpt.shape = [batch_size, rnn_seq, rnn_state]
    # rnn_states: the number of features in the state variables
    # rnn_seq: the length of the sequence in the input.

    def dense_linear(x, size):
        return layers.fully_connected(x, size, activation_fn=None)

    def dense_batch_relu(x, size):
        h1 = layers.fully_connected(x, num_outputs = size, activation_fn=None)
        h2 = layers.batch_norm(h1, center=True, scale=True, is_training=phase)
        return tf.nn.relu(h2)

    inpt_rnn = tf.reshape(inpt, [tf.shape(inpt)[0], rnn_seq, rnn_states])

    with tf.variable_scope(scope):
        inputs_series = tf.split(inpt_rnn, rnn_seq, 1)
        inputs_series = [tf.squeeze(ts,axis = 1) for ts in inputs_series]
        cell = tf.contrib.rnn.LSTMCell(64)
        states_series, current_state = tf.contrib.rnn \
            .static_rnn(cell, inputs_series, dtype =tf.float32 )

        out = current_state
        for i in range(5):
            out = dense_batch_relu(out, 64)
        out = dense_linear(out, output_dim)

        # apply softplus to all units except the first one
        out = tf.concat([tf.expand_dims(out[:, 0], axis = 1),
                         tf.nn.softplus(out[:, 1:])], axis = 1)
    return out

```

Table 8: The Python Tensorflow implementation of the RNN architecture.

Hyperparameter Tuning:

A few neural network architectures were been tried and the reported architecture had the best performance on the validation datasets. Hyperparameters are also tuned on the validation datasets. For each of the following hyperparameters, a few values were tried and the best values are reported below:

- Learning rate: 3×10^{-5}
- Batch size: 1024
- Target network copy frequency: every 1000 gradient steps
- TD step size m: 15 (both SL and RL)

E. Algorithm Performance

Stock	SL (Specific)	RL (Specific)	SL (Out)	RL (Out)	SL (Universal)	RL (Universal)
AAPL	37.36 (1.21)	44.8 (1.24)	38.57 (1.21)	43.23 (1.23)	38.9 (1.22)	44.4 (1.23)
ADBE	27.6 (0.80)	30.4 (0.81)	27.27 (0.81)	30.15 (0.81)	27.36 (0.80)	30.4 (0.81)
ADI	17.68 (1.09)	20.2 (1.10)	17.34 (1.08)	19.88 (1.09)	18.2 (1.08)	20 (1.09)
ADP	11.38 (1.07)	12.4 (1.08)	11.40 (1.07)	12.41 (1.09)	11.74 (1.08)	12.40 (1.10)
ADSK	30.58 (1.10)	34.20 (1.12)	29.57 (1.10)	33.67 (1.12)	29.48 (1.11)	33.40 (1.13)
AMAT	13.52 (0.71)	14.60 (0.72)	13.94 (0.71)	15.13 (0.72)	13.62 (0.72)	15.00 (0.73)
AMD	22.36 (0.72)	24.20 (0.73)	21.15 (0.72)	22.95 (0.73)	22.32 (0.73)	25.20 (0.74)
AMGN	37.98 (1.21)	44.60 (1.23)	38.89 (1.21)	45.67 (1.23)	41.96 (1.22)	46.80 (1.24)
AMZN	25.80 (1.32)	29.40 (1.35)	23.96 (1.32)	25.75 (1.35)	25.54 (1.33)	28.40 (1.35)
ATVI	18.58 (0.89)	20.60 (0.91)	21.35 (0.89)	22.68 (0.91)	22.58 (0.90)	22.60 (0.91)
AVGO	17.38 (1.05)	18.40 (1.07)	17.58 (1.05)	18.92 (1.07)	18.82 (1.06)	19.02 (1.08)
BAC	22.94 (0.71)	27.40 (0.72)	23.63 (0.71)	27.67 (0.72)	23.76 (0.72)	27.80 (0.73)
BRK.B	32.90 (1.25)	36.60 (1.28)	33.28 (1.25)	37.23 (1.28)	35.08 (1.26)	37.80 (1.28)
CHTR	12.74 (1.23)	16.20 (1.25)	12.72 (1.23)	16.16 (1.25)	13.82 (1.24)	17.40 (1.26)
CMCSA	17.16 (1.09)	21.00 (1.11)	16.64 (1.09)	20.17 (1.11)	17.70 (1.10)	21.00 (1.12)
COST	32.82 (1.31)	38.20 (1.34)	34.13 (1.31)	39.17 (1.34)	36.48 (1.32)	41.60 (1.34)
CSCO	15.16 (0.68)	17.60 (0.69)	14.99 (0.68)	16.93 (0.69)	15.60 (0.69)	17.40 (0.70)
CSX	14.74 (1.03)	16.20 (1.05)	15.09 (1.03)	16.49 (1.05)	15.22 (1.04)	17.40 (1.06)
DIS	18.44 (1.21)	21.40 (1.23)	19.54 (1.21)	22.89 (1.23)	20.62 (1.22)	23.40 (1.24)
EBAY	14.86 (1.19)	18.20 (1.22)	14.93 (1.19)	18.30 (1.22)	15.04 (1.20)	18.80 (1.22)
F	22.56 (0.89)	27.40 (0.91)	24.00 (0.89)	28.36 (0.91)	24.66 (0.90)	28.20 (0.91)
FB	15.68 (1.43)	16.20 (1.46)	15.91 (1.43)	16.46 (1.46)	16.04 (1.44)	16.60 (1.47)
FISV	21.36 (1.20)	24.20 (1.22)	21.76 (1.20)	24.65 (1.22)	22.96 (1.21)	24.40 (1.23)
GE	22.26 (0.68)	26.40 (0.69)	22.02 (0.68)	26.49 (0.69)	22.40 (0.69)	26.60 (0.70)
GILD	24.44 (0.90)	32.40 (0.92)	23.38 (0.90)	28.84 (0.92)	23.66 (0.91)	29.80 (0.92)
GS	28.38 (1.19)	34.40 (1.21)	26.80 (1.19)	31.82 (1.21)	27.24 (1.20)	32.20 (1.22)
ILMN	19.44 (1.22)	24.80 (1.24)	19.62 (1.22)	25.13 (1.24)	21.12 (1.23)	25.60 (1.25)
INTC	27.42 (0.75)	29.80 (0.77)	26.69 (0.75)	29.26 (0.77)	26.96 (0.76)	29.90 (0.77)
INTU	15.04 (1.11)	18.60 (1.13)	15.85 (1.11)	19.78 (1.13)	16.98 (1.12)	20.00 (1.14)
ISRG	15.92 (1.50)	19.00 (1.53)	17.39 (1.50)	21.05 (1.53)	19.50 (1.52)	21.80 (1.54)
JNJ	15.00 (1.09)	18.20 (1.11)	14.76 (1.09)	17.97 (1.11)	14.98 (1.10)	19.00 (1.12)
JPM	24.96 (0.80)	30.60 (0.82)	25.32 (0.80)	30.71 (0.82)	26.50 (0.81)	31.40 (0.82)
LRCX	17.04 (1.12)	20.20 (1.14)	16.81 (1.12)	21.08 (1.14)	17.36 (1.13)	21.80 (1.15)
MDLZ	11.92 (1.02)	14.20 (1.04)	12.66 (1.02)	14.97 (1.04)	12.74 (1.03)	14.40 (1.05)
MELI	13.90 (1.25)	15.20 (1.28)	14.42 (1.25)	15.72 (1.28)	15.14 (1.26)	17.00 (1.28)
MRK	27.74 (0.98)	34.20 (1.00)	28.33 (0.98)	34.96 (1.00)	29.38 (0.99)	36.40 (1.00)
MSFT	28.04 (0.81)	32.80 (0.83)	28.20 (0.81)	32.95 (0.83)	29.04 (0.82)	33.60 (0.83)
MU	36.30 (0.98)	36.60 (1.00)	34.86 (0.98)	35.87 (1.00)	35.06 (0.99)	36.40 (1.00)
NFLX	18.06 (1.39)	20.80 (1.42)	18.75 (1.39)	21.63 (1.42)	19.98 (1.40)	23.60 (1.42)
NVDA	16.64 (0.69)	18.00 (0.70)	16.96 (0.69)	18.48 (0.70)	16.82 (0.70)	19.00 (0.71)
PEP	13.78 (1.10)	18.40 (1.12)	13.52 (1.10)	18.12 (1.12)	14.42 (1.11)	18.80 (1.13)
QCOM	27.52 (0.77)	35.80 (0.78)	28.47 (0.77)	37.09 (0.78)	29.24 (0.77)	36.80 (0.79)
SBUX	38.26 (1.09)	41.40 (1.11)	37.05 (1.09)	39.84 (1.11)	37.94 (1.10)	39.60 (1.12)
T	18.06 (1.01)	20.40 (1.03)	17.10 (1.01)	19.65 (1.03)	17.92 (1.02)	20.60 (1.04)
TXN	11.22 (1.05)	12.40 (1.07)	11.66 (1.05)	12.72 (1.07)	12.16 (1.06)	13.40 (1.08)

UPS	15.54 (1.08)	16.20 (1.10)	16.38 (1.08)	17.37 (1.10)	18.84 (1.09)	19.20 (1.11)
V	24.46 (1.31)	29.60 (1.34)	25.15 (1.31)	29.47 (1.34)	25.90 (1.32)	29.80 (1.34)
VRTX	26.32 (1.19)	27.80 (1.21)	26.67 (1.19)	27.64 (1.21)	26.66 (1.20)	27.60 (1.22)
VZ	14.78 (0.93)	18.00 (0.95)	14.19 (0.93)	17.60 (0.95)	14.48 (0.94)	18.60 (0.96)
WFC	16.06 (1.05)	20.00 (1.07)	16.68 (1.05)	20.20 (1.07)	16.96 (1.06)	21.00 (1.08)
Avg.	21.40 (0.15)	24.82 (0.16)	21.55 (0.15)	24.85 (0.16)	22.34 (0.15)	25.47 (0.16)

Table 9: These price gains are out-of-sample performances reported on the testing dataset. The numbers displayed are in percentage of the half-spread (% Half-Spread). The numbers in parenthesis are standard errors.