

# Double Deep Q-Learning for Optimal Execution\*

Brian Ning<sup>†</sup>, Franco Ho Ting Ling<sup>‡</sup>, and Sebastian Jaimungal<sup>\*</sup>

## Abstract.

Optimal trade execution is an important problem faced by essentially all traders. Much research into optimal execution uses stringent model assumptions and applies continuous time stochastic control to solve them. Here, we instead take a model free approach and develop a variation of Deep Q-Learning to estimate the optimal actions of a trader. The model is a fully connected Neural Network trained using Experience Replay and Double DQN with input features given by the current state of the limit order book, other trading signals, and available execution actions, while the output is the Q-value function estimating the future rewards under an arbitrary action. We apply our model to nine different stocks and find that it outperforms the standard benchmark approach on most stocks using the measures of (i) mean and median out-performance, (ii) probability of out-performance, and (iii) gain-loss ratios.

**Key words.** DQN, Optimal Execution, Q-learning

**AMS subject classifications.**

**1. Introduction.** Financial markets are highly complex stochastic systems with significant heteroskedasticity. The problem of how to optimally execute large positions over a given trading horizon is an important problem faced by institutional investors, banks, and hedge funds. Naïvely rebalancing a portfolio could result in significant adverse price movements as other intelligent traders may read off the signal. Investors must balance trading quickly and obtaining poor execution prices, with trading slowly which exposes them to unknown market fluctuations.

Agents are often exposed to a plethora of information including prior stock prices and other market conditions. Determining the best trading policy in the presence of all of this information is an important part of the algorithmic trading literature. Traditionally, researchers propose a stochastic model based on empirical observations, such as an Ornstein-Uhlenbeck or stochastic volatility process, use historical data to estimate model parameters, such as the volatility, mean-reversion level and rate, propose a performance criteria which they aim to maximize, and then solve the problem analytically using methods in stochastic optimal control. One of the earliest works in this vein is the Almgren-Chriss [1] approach, where the authors assume prices are Brownian motion. There has been many generalizations including of this approach to account for a variety of market features, see, e.g., [6], [3], and [5], and the graduate textbook [4] for a modern treatment.

However, in the case of more complex price models without analytical solutions, or even non-parametric models, a different approach is necessary. Reinforcement learning [12] attempts to learn optimal policies for sequential decision problems by optimizing a cumulative

---

\*SJ would like to acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), [funding reference numbers RGPIN-2018-05705 and RGPAS-2018-522715]

<sup>†</sup>Department of Statistical Sciences, University of Toronto ([brian.ning@mail.utoronto.ca](mailto:brian.ning@mail.utoronto.ca), [sebastian.jaimungal@utoronto.ca](mailto:sebastian.jaimungal@utoronto.ca)).

<sup>‡</sup>Department of Computer Science, University of Toronto ([francohtlin@cs.toronto.edu](mailto:francohtlin@cs.toronto.edu)).

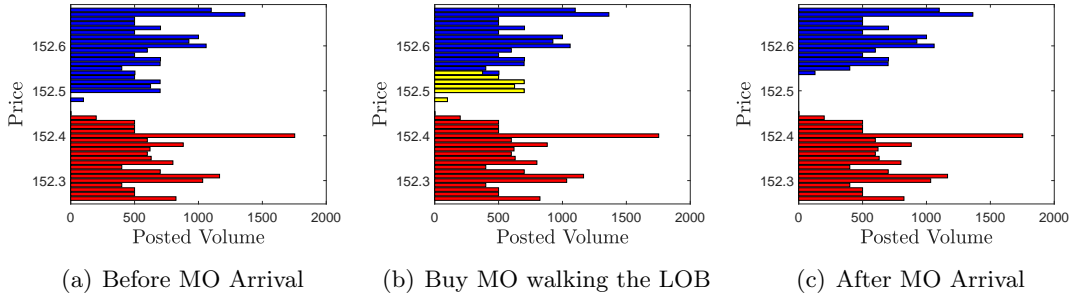
future reward function with few modeling assumptions (such as Markovian structure of the state space). The most popular reinforcement learning algorithm is Q-learning [14] which has been applied to the optimal execution problem in [10] and [7].

One of the major drawbacks to the Q-learning algorithm is that it estimates the optimal policy on a pointwise basis. This severely limits the ability to interpolate between possible actions, and therefore does not allow for effective generalization. Moreover, the algorithm's storage space scales exponentially with respect to the dimension of the state space. To address this issue, we modify recent advancements in Deep Q-Learning [9, 13] to make it useful for the solving the optimal execution problem. By estimating the Q-function used in Q-learning with a deep neural network, we can interpolate between actions and states as well as reduce the storage costs associated with classical Q-learning.

To the best of our knowledge this article is the first to adapt and modify the framework of Deep Q-learning to the optimal execution problem. We also provide numerical comparisons between our approach and classical methods, as well as discuss the financial interpretation of the results.

The remainder of this paper is structured as follows. Section 2 provides some background information on key concepts including a brief description of Q-learning and the optimal execution problem. Section 3 and 4 details the exact formulation of the optimal execution problem in a reinforcement learning setting and the adaption of Deep Q-learning. Section 5 explains how we train the network with a detailed algorithm. Section 6 and 7 presents our results on real data and the metrics we use to evaluate our findings.

## 2. Background.



**Figure 1.** The LOB of Facebook Inc. stock at 10:36 on 28 Mar, 2018 as a buy MO of size 3,000 arrives. The blue (red) bars represent available volume of sell (buy) orders at the corresponding price. The yellow bars in the middle panel indicate which LOs match the incoming MOs.

**2.1. Limit Order Books.** Traders may execute two main order types on modern electronic stock exchanges: *limit orders* and *market orders*. A limit order (LO) represents the intention to buy or sell at most a fixed number of shares at a specified price. If there are no shares available at that specified price (or better), then the limit order is collected in the *limit order book* (LOB) and remains there until the trader either cancels the order or it is filled by another trader. Therefore LOs have a price guarantee, but are not guaranteed to be executed. Traders who post limit orders are said to provide liquidity to the market as they provide a pool of

shares at different prices for other traders to either buy or sell. A market order(MO) represents a fixed number of shares to buy or sell, and is matched by the best available LOs remaining in the LOB. The lowest price of all sell LOs is known as the *ask price* and the highest price of all buy LOs is known as the *bid price*. The difference between the ask and bid is known as the *spread* and the average of the ask and bid price is known as the *mid-price*. The minimum change in the bid or ask is called the *tick size*. Figure fig:LOB-Snapshot illustrate a snapshot of the LOB for INTC as a buy MO walks the LOB. Before the MO arrives, the LOB has a spread of 1 tick, the bid and ask are \$48.40 and \$48.41, respectively. After the MO walks the LOB, the bid and ask are \$48.40 and \$48.46, respectively, and the spread widens to 6 ticks.

The share prices in our data are “small” relatively to the tick size, and hence the spread is typically 1 tick – such stocks are called *large tick stocks*. Therefore, we approximate all execution prices by the mid-price and ignore the spread. We avoid the cost associated with walking the LOB by applying a penalty on the size of any one order, thus the trader’s strategy only takes the liquidity posted at the best available price. The details are described in the paragraph before (3.1).

**2.2. Optimal Execution.** The optimal execution problem assumes studies a trader who at  $t = 0$  holds an inventory of  $q_0$  shares and must fully liquidated it by the end of the time period  $t = T$  (i.e.  $q_T = 0$ ). We study the problem in discrete times, so that trade orders  $x_t \geq 0$  are executed at evenly distributed time-steps  $t = \{0, 1, 2, \dots, T - 1\}$  where  $x_t$  represents the number of shares to be sold. The midprice  $p_t$  follows some unknown process which may or may not be impacted by the trader’s actions. In our case, however, we assume the trader’s actions do not directly effect the price process during training. This is a reasonable assumption to make as the volume of shares traded for most stocks of large companies heavily exceeds that of individual investors. However once the network is trained on historical data and used to perform online learning in real-time, it can learn what effect our trading action has on the price process and adapt the optimal policy as needed. We define  $s_t = [p_t, \bar{s}_t]^\top$  where  $\bar{s}_t$  is a vector of stochastic processes representing other features. Price and other states are jointly stochastic and may be affected by the actions taken at time  $t$  so that  $s_{t+1} = f(S_t, x_t)$  is random. The goal is to maximize the expected total profit obtained by selling the shares subject to transactions fees

$$(2.1) \quad \operatorname{argmax}_{x_0, x_1, \dots, x_{T-1}} \mathbb{E} \left[ \sum_{t=0}^{T-1} R(p_t, x_t) \right],$$

where  $R(s_t, x_t)$  is the profit obtained from selling  $x_t$  shares at mid-price  $p_t$ . Note, the only feature which feeds into  $R$  is the midprice. We work with the full state vector  $s_t$  from this point onwards and  $\bar{s}$  are various selected features as discussed in Section 6.

**2.3. Reinforcement Learning.** The goal in reinforcement learning is to learn a policy  $\pi \in \Pi$  to control a system with states  $s \in \mathcal{S}$  using actions  $x \in \mathcal{X}$  in order to maximize the total expected rewards based on some reward function  $R(s, x)$ . The system itself is defined by an initial state distribution  $P(s_0)$  and transition distribution  $P(s_t | s_{t-1}, x_{t-1})$ . The goal is to maximize the total expected discounted rewards defined as  $R = \sum_{t=1}^T \gamma^{t-1} R(s_t, x_t)$  over the space of all allowable policies  $\Pi$ . As optimal execution problems are often performed over short time periods, the discount factor is set close to one.

**2.4. Deep Q-Learning.** In Q-learning, the Q-function [14], given an optimal policy  $\pi \in \Pi$ , is defined as:

$$(2.2) \quad Q(s, x) = \mathbb{E} \left[ R(s, x) + \sum_{i=t+1}^T \gamma^{i-t} R(s_i^\pi, x_i^\pi) \right],$$

where the action at time  $t$  is arbitrary, but from  $t+1$  onwards it is optimal with  $s_i^\pi$  and  $x_i^\pi$  representing the state and action respectively at time  $i$  if the agent follows the optimal policy  $\pi$ . Under mild conditions, the Q-function satisfies the Bellman equation

$$(2.3) \quad Q(s, x) = \mathbb{E} \left[ R(s, x) + \gamma \max_{x' \in \mathcal{U}} Q(s^{s,x}, x') \right],$$

where  $s^{s,x}$  is the (random) state the system evolves to after taking action  $x$  when in state  $s$ , and  $\mathcal{U}$  is the admissible set of actions. It may be estimated in an online iterative fashion as follows (see [12])

$$(2.4) \quad Q^{(\ell+1)}(s, x) = Q^{(\ell)}(s, x) + \alpha_t \left[ R(s, x) + \gamma \max_{x' \in \mathcal{U}} Q^{(\ell)}(s^{s,x}, x') - Q^{(\ell)}(s, x) \right],$$

where  $Q^{(\ell)}(s, x)$  denotes the estimate of  $Q(s, x)$  at iteration  $\ell$ ,  $R(s, x)$  is a realized reward by taking action  $x$  in state  $s$ , and  $s^{s,x}$  is the (random) state the system evolves to after taking the action, as long as  $\sum_{t=0}^{\infty} \alpha_t = \infty$  and  $\sum_{t=0}^{\infty} \alpha_t^2 < +\infty$ .

When  $\mathcal{S}$  and  $\mathcal{X}$  are discrete and low dimensional, it is possible to represent the Q-function as a matrix. For larger dimensions or continuum, however, it is typically replaced with a model approximation. In Deep Q-Learning, this model is a fully connected neural network  $Q(s, x | \theta)$ , where  $\theta$  are the network parameters rather than using (2.4) to update the network.

At each iteration of the algorithm, the network parameters are updated by minimizing the squared loss between the Q-value using the previous network parameters  $\theta_\ell$  and the Q-value using updated network parameters. The loss function

$$(2.5) \quad L(\theta; \theta_\ell) = \left( \left[ R(s, x) + \gamma \max_{x' \in \mathcal{U}} Q(s^{s,x}, x' | \theta_\ell) \right] - Q(s, x | \theta) \right)^2$$

is minimized at each iteration and the new network parameters  $\theta_{\ell+1} = \arg\min_{\theta} L(\theta; \theta_\ell)$ .

**3. Optimal Execution in a Reinforcement Learning Setting.** In this work, we focus on demonstrating how to use our approach when the trader employs market orders only. An interesting follow up would be to incorporate an optimal mix of limit and market orders. The restriction to market orders is sub-optimal, as market orders incur a cost due to the bid-ask spread, as well as a cost due to walking the limit order book (as in Figure 1). Nonetheless, even with this restriction, we demonstrate that our approach provides gains over time weighted average price (TWAP) – which is optimal under the assumption that price is a Brownian motion, and more generally a martingale.

In the next subsections, we provide a description of the states, actions and rewards used in our reinforcement learning formulation of the optimal execution problem. The execution

time horizon is divided into  $T$  periods and execution decisions are made at the beginning of each period. Trades, however, are made each second.

Specifically, we denote the time periods where trading decisions can be made by  $T_0 < T_1 < T_2 < \dots < T_{N-1}$ . We also denote the end of the last trading period as  $T_N = T$ , however, the last decision is made at time  $T_{N-1}$ . The intra-period time, where trades occur, are indexed by  $\{\{t_{0,i}\}_{i \in \{0, \dots, M_0-1\}}, \dots, \{t_{N-1,i}\}_{i \in \{0, \dots, M_{N-1}-1\}}\}$ , with  $t_{k,i} = T_k + i\Delta t$ , where  $\Delta t = \frac{T_k - T_{k-1}}{M_k}$  so that period  $k$  has trading time indices  $T_k = t_{k,0} < t_{k,1} < \dots < t_{k,M_k-1} = T_{k+1} - \Delta t$  and is made up of  $M_k$  small time steps. In the numerical experiments we have  $T_k - T_{k-1} = c$ ,  $\forall k$  for some constant  $c$  and  $\Delta t = 1 \text{ sec}$ .

**3.1. States.** The state space (or feature space)  $\mathcal{S}$  contains the state of the LOB at the start of each period, as well as any prior information from previous periods. We make the (standard) assumption that given a state  $s_t \in \mathcal{S}$  and a trading action  $x_t \in \mathcal{X}$  at time  $t$ , the mapping  $(s_t, x_t) \mapsto s_{t+1}$  is Markovian. Time inhomogeneous processes may be incorporated by allowing one of the components of the state space to be time  $t$  itself. We explore a variety of features in the results section. The two states which play a crucial role in the optimal strategy are (i) the current time (or elapsed time)  $t$  and (ii) the remaining inventory  $q_t$  to execute, and they are always part of our state space.

**3.2. Actions.** A policy uniquely maps a state  $s \in \mathcal{A}$  to an action  $x \in \mathcal{X}$ . In particular, in the optimal execution problem, the only action is the amount of shares to sell via a market order. Naturally, the set of allowed actions is restricted such that  $x_t \in [0, q_t]$  for all  $t$ . We make a further restriction on the action space to take only integer values (or fixed multiples of integer values for larger inventories) in order to simplify the computation needed to find the optimal action at each time step. At each executable time step for actions, the total amount of shares sold are assumed to be evenly distributed over the time block on a second by second basis.

**3.3. Rewards.** The reward equals the total reward over each period, and each period reward is made of rewards for trading each second. Over each period, we assume the agent sends orders at a constant rate, so that when the trader makes a decision at time  $T_k$  to send  $x_{T_k}$  orders over the next period, the actual trades are made of  $\frac{x_{T_k}}{M_k}$  equal trades every second. To account for transaction costs and the potential that these shares may walk the LOB (as in Figure 1), we also incorporate a quadratic penalty on the number of shares executed during each second. This penalty term can be adjusted with a hyperparameter to represent the liquidity of the asset in question. One may also replace it with any other non-linear penalty that reflects the agent's utility associated with taking on impact risk. Specifically, for each intra-period timestep  $[t_{k,i}, t_{k,i+1})$  the reward is

$$(3.1) \quad \check{R}_{k,i} = q_{t_{k,i}} (p_{t_{k,i+1}} - p_{t_{k,i}}) - a \left( \frac{x_{T_k}}{M_k} \right)^2,$$

where  $q_{t_{k,i}}$  and  $p_{t_{k,i}}$  are the remaining inventory and price at time  $t_{k,i}$ , respectively. The total reward over the period  $[T_k, T_{k+1})$  is the sum of each intra-period reward,

$$(3.2) \quad R_k = \sum_{i=0}^{M_k-1} \check{R}_{k,i},$$

and the total reward is the sum of each period reward

$$(3.3) \quad \sum_{k=0}^{N-1} R_k = \sum_{k=0}^{N-1} \sum_{i=0}^{M_k-1} \tilde{R}_{k,i} = \sum_{k=0}^{N-1} \sum_{i=0}^{M_k-1} \left\{ q_{t_{k,i}}(p_{t_{k,i+1}} - p_{t_{k,i}}) - a \left( \frac{x_{T_k}}{M_k} \right)^2 \right\}$$

$$(3.4) \quad = -q_0 p_0 + \sum_{k=0}^{N-1} \sum_{i=0}^{M_k-1} \left\{ p_{t_{k,i+1}} \frac{x_{T_k}}{M_k} - a \left( \frac{x_{T_k}}{M_k} \right)^2 \right\}$$

where the last equality follows as  $q_T = 0$ ,  $q_t - q_{t-1} = x_t$ , and by using a summation by parts formula. The total reward is the amount sold at each time period less a penalty based on the amount sold.

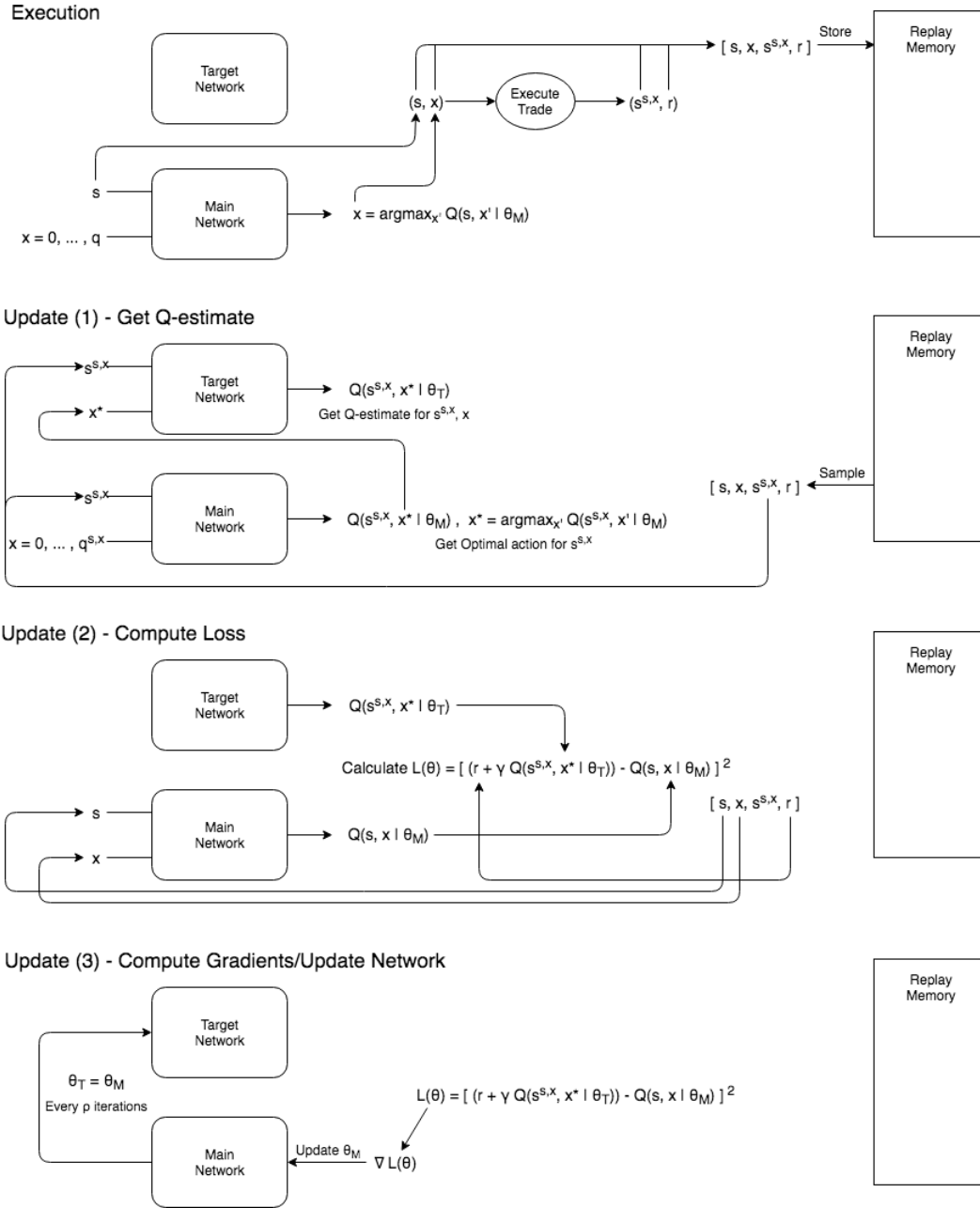
**4. Network Architecture.** In this section we describe our network architecture and the update procedure we use to train the networks. Figure 2 illustrates the various steps in the training procedure and how the training and target networks used in our double deep Q-learning framework relate to one another.

**4.1. Basic Network Architecture.** As actions in the optimal execution problem are (in principle) continuous, our network architecture uses both states and action as inputs and the Q-value function as output. If instead, we discretise or bucket actions, it would be possible to use states only as inputs, and have a network for each action; however, as the allowed actions depends on the state (recall that  $x_t \in [0, q_t]$ ) making this approach less tractable. Another possible architecture is to have multiple Q-value function as outputs, with each corresponding to the specific action taken. Since we have varying action spaces at each time step, the logical approach is to use a network that takes both a state and action as input.

The network architecture has dimension  $s + 1$ , where  $s \in \{2, 3, 4, 5\}$  corresponds to the number of features and the extra dimension is the action. The collection of features we use are: time, inventory, price, quadratic variation, and price. They are discussed further in Section 7.3. As the input dimension is low, we use a fully connected feed-forward neural network with 6 layers and 20 nodes in each layer. At each layer, we use a ReLU activation function to prevent vanishing gradients and provide sparsity, and we use the RMSprop optimizer (see [9]).

**4.2. Experience Replay.** In order to improve network stability in the case of extreme values in recently visited states, a replay memory is used to provide a database of previously visited combinations of state and action to sample from in order to perform the network weight updates. [8] Specifically, at each time step the tuple  $d = (s, x, r, s^{s,x})$  is stored in the replay memory buffer  $\mathcal{D}$ . Where  $s^{s,x}$  denotes the state we arrive at from  $s$  when action  $x$  is taken. Batches are then sampled uniformly from  $\mathcal{D}$  and used to compute the gradient of the loss function in (2.5) in order to update the network. Once the replay memory has reached a certain size  $K$ , we randomly remove a transition from the first  $\frac{K}{2}$  tuples. This is done to ensure that we do not remove cases that were immediately placed into the buffer which may not have been selected by random sampling to update the network yet.

**4.3. Main and Target Networks.** As noted in [13], the original form of DQN may suffer from oscillations in the gradient updates, as the same network  $Q(s, x|\theta)$  generates both the



**Figure 2.** Execution procedure and full update procedure used when training the network. Note that when we are training our network,  $\epsilon$ -Greedy is added to the Execution step. In the update procedure, we split the process into 3 steps, (1) Getting the Q-estimate given the next state  $s^{s,x}$ , (2) Computing the Loss after we have obtained the Q-estimate and (3) Computing the gradients and updating our network weights.



next state's target Q-values and updates the current state's Q-values. To reduce this instability, we use two neural networks: a main network  $Q(s, x | \theta_M)$  and a target network  $Q(s, x | \vartheta_T)$ . We update the main network at the end of each decision period  $t = T_{k+1}$ ,  $k \in 0, 1, \dots, N-1$ , while the target network  $\vartheta_T$  remains fixed and is replaced by  $\theta_M$  only after several periods of trading. As well, we compute gradients from samples in our replay memory  $\mathcal{D}$  instead of the immediate transitions observed. Thus the loss function is written as

$$(4.1) \quad L(\theta; \vartheta_T) = \sum_{j=1}^J \left( \left[ r_{(j)} + \gamma \max_{x' \in [0, q_j - x_j]} Q(s_{(j)}^{s,x}, x' | \vartheta_T) \right] - Q(s_{(j)}, x_{(j)} | \theta) \right)^2,$$

and  $\theta_M = \operatorname{argmin}_{\theta} L(\theta; \vartheta_T)$  where  $(s_{(j)}, x_{(j)}, r_{(j)}, s_{(j)}^{s,x})$  is sampled from the memory replay  $\mathcal{D}$  for all  $j = 1 \dots J$  described in 4.2.

**4.4. Double DQN.** Main-target networks often overestimate the Q-values and we correct this by using Double DQN [13]. We decouple the first term from our loss function in (4.1) into action selection and action evaluation by using our main network  $Q(s, x | \theta_M)$  to select the best action, and our target network  $Q(s, x | \vartheta_T)$  to generate our Q-value estimate. This is in contrast to using the target network for both selecting the best action and generating the Q-value. Our modified estimate of the Q function is

$$(4.2) \quad Q(s_{(j)}^{s,x}, x^*(s_{(j)}^{s,x} | \theta_M) | \vartheta_T), \quad \text{where} \quad x^*(s_{(j)}^{s,x} | \theta_M) = \operatorname{argmax}_{x' \in [0, q_{(j)}]} Q(s_{(j)}^{s,x}, x' | \theta_M),$$

our modified loss function is

$$(4.3) \quad L(\theta; \vartheta_T) = \sum_{j=1}^J \left( \left[ r_{(j)} + \gamma Q(s_{(j)}^{s,x}, x^*(s_{(j)}^{s,x} | \theta_M) | \vartheta_T) \right] - Q(s_{(j)}, x_{(j)} | \theta) \right)^2,$$

the updated main network minimizes the loss function  $\theta_M = \operatorname{argmin}_{\theta} L(\theta; \vartheta_T)$ , and  $(s_{(j)}, x_{(j)}, r_{(j)}, s_{(j)}^{s,x})$  is sampled from the memory replay  $\mathcal{D}$  for all  $j = 1 \dots J$ .

**4.5. Treatment of the Zero Ending Inventory Constraint.** In DQN, the network structure estimates the Q-function at all time periods, including the last period. In the optimal execution problem, however, the last period has the additional constraint that inventory must be drawn down to zero by the end of the period. In classical Q-Learning, with a standard matrix representation of the Q-function, such constraints are imposed analytically and the terminal period reward determines the terminal Q-function

$$(4.4) \quad Q(s_{T_{N-1}}, x_{T_{N-1}}) = R_{N-1}.$$

However, as discussed in Section 3 each period is made of one-second intervals, and an action taken at time  $T_k$  results in trades at each second  $\{t_{k,i}\}_{i=0, \dots, M_k-1}$  in that period. At the start of the last period  $T_{N-1}$ , the reward  $R_{N-1}$  in Equation (3.2) depends on the price path  $\{s_{T_{N-1},i}\}_{i=0, \dots, M_{N-1}-1}$  which are not measurable (except for  $i=0$ ) at time  $T_{N-1}$  when the action is to be taken. Thus there is no deterministic value for the last timestep as the reward at this timestep depends on how the price process behaves in the intra-period intervals from the action-executable time to the end of the period.



There are several ways to address this issue. One approach is to assume a model for the intra-period price process  $s_{t_{N-1},i} \sim \pi_0 \left( \cdot \middle| s_{t_{N-1},i-1}, \frac{q_{T_{N-1}}}{M_{N-1}} \right)$  during the last period, which enforces the constraint  $q_{T_N} = 0$  and implies that  $x_{T_{N-1}} = q_{T_{N-1}}$ , and therefore

$$(4.5) \quad Q(s_{T_{N-1}}, x_{T_{N-1}}) = \mathbb{E} \left[ \sum_{i=0}^{M_{N-1}-1} \check{R}_{N-1,i}^{\pi_0, x_{T_{N-1}}} \middle| s_{T_{N-1}} \right],$$

where  $\check{R}_{N-1,i}^{\pi_0, x_{T_{N-1}}}$  is the intra-period reward (as defined in (3.1)) obtained when the state evolves according to the law of  $\pi_0$  after executing action  $x_{T_{N-1}}$  at time  $T_{N-1}$ .

A second approach is to estimate the Q-value of this last interval using the neural network  $Q(s_{T-1}, x_{T-1}) \approx Q(s_{T-1}, x_{T-1} | \theta)$  and enforce the restriction that the only action allowable at the last action-executable time equals the remaining inventory.

We take an alternate approach that places less burden on the network to estimate terminal Q-values. We add a single one second time step at the end of the trading horizon over which all remaining shares are liquidated. To incorporate this terminal liquidation into the learning procedure, whenever a selected state from the replay buffer corresponds to the last time period, we replace the correspond term in the loss in Equation (4.1) with

$$(4.6) \quad \left( \left[ r_{(j)} + \gamma \mathfrak{R}(s_{(j)}^{s,x}, q_{(j)} - x_{(j)}) + \gamma Q \left( s_{(j)}^{s,x}, x^* \left( s_{(j)}^{s,x} \middle| \theta_M \right) \middle| \vartheta_T \right) \right] - Q(s_{(j)}, x_{(j)} | \theta) \right)^2$$

where the terminal reward (from liquidating all remaining shares) is

$$(4.7) \quad \mathfrak{R}(s, q) = q(p' - p) - a q^2,$$

and  $p'$  is the price at time  $T + \Delta T$  reached from the state  $s_{(j)}^{s,x}$  at time  $T_N$ . This specific loss function is only applied to the experiences sampled from the replay memory where the initial state corresponds to time  $t = T_{N-1}$ . All other sampled experiences uses the original loss function defined in (4.1). The total batch loss the sum of the individual losses of each experience sampled.

This approach makes no assumption on the price process, unlike the first approach. It also places much less burden on the network to correctly estimate terminal Q-values, as would be the case using the second approach. One shortcoming is that the constraint of liquidating all shares by  $T_N$  is not strictly enforced, although it is enforced by  $T_N + \Delta T$ . For any reasonable value of  $a$  (the coefficient of the quadratic penalty term on size of execution in the rewards function), however, the optimal action results in selling all (or near all) remaining inventory and thus satisfying the restriction in the original problem.

**5. Training Method.** In this section, we specify how we train the network as outlined in Algorithm 1.

**5.1.  $\epsilon$ -Greedy.** Reinforcement Learning require trading off exploration versus exploitation. [12] Exploration allows the system to evolve into regions in state space which have not yet been sampled. These regions may have larger rewards than what the model would otherwise tell us. Once state space is sufficiently explored, we can be more certain what actions

```

Initialize replay memory  $\mathcal{D}$  of size  $\mathcal{N}$ ;
Initialize action-value function  $Q(\cdot|\theta_M)$  with random weights. Pre-train  $Q(\cdot|\theta_M)$  on
boundary cases and make a copy  $Q(\cdot|\vartheta_T)$ ;
for trading episode  $b \in B$  do
  for  $i \leftarrow 0$  to  $N - 1$  do
    With probability  $\epsilon$  select random action  $x_i \sim \text{Binomial}\left(q_i, \frac{1}{T_i}\right)$ ;
    Otherwise select  $x_i = \max_{x' \in [0, q_{T_i}]} Q(s_{T_i}, x' | \theta_M)$  (optimal action);
    Execute the action  $x_{T_i}$  and observe the reward  $r_{T_i}$  and next state  $s_{T_{i+1}}^{s_{T_i}, x_{T_i}}$ ;
    Store transition  $(s_{T_i}, x_{T_i}, r_{T_i}, s_{T_{i+1}}^{s_{T_i}, x_{T_i}})$  in replay buffer  $\mathcal{D}$ ;
    for  $j \leftarrow 1$  to  $J$  do
      Sample random minibatch of transitions  $(s_{(j)}, x_{(j)}, r_{(j)}, s_{(j)}^{s,x})$  from  $\mathcal{D}$ ;

      Set  $y_{(j)}(\vartheta_T) = \begin{cases} r_{(j)}, & \text{for } t_{(j)}^{s,x} = T_N \\ r_{(j)} + \gamma \mathfrak{R}(s_{(j)}^{s,x}, q_{(j)}), & \text{for } t_{(j)}^{s,x} = T_{N-1} \\ r_{(j)} + \gamma Q(s_{(j)}^{s,x}, x_{(j)}^* | \vartheta_T), & \text{otherwise} \end{cases}$ 

      where  $q_{(j)}$  is the inventory remaining corresponding to the state  $s_{(j)}$  and  $t_{(j)}^{s,x}$ 
      is the time remaining corresponding to the state  $s_{(j)}^{s,x}$ 

      
$$x_{(j)}^* = \operatorname{argmax}_{x' \in [0, q_{(j)}]} Q(s_{(j)}^{s,x}, x' | \theta_M) ;$$


    end
    Obtain new network parameters  $\theta_M$  by minimizing

    
$$L(\theta; \vartheta_T) = \sum_{j=1}^J [y_{(j)}(\vartheta_T) - Q(s_{(j)}, x_{(j)} | \theta)]^2$$


    using gradient descent to obtain  $\theta_M = \operatorname{argmin}_{\theta} L(\theta; \vartheta_T)$ ;
  end
  Decay  $\epsilon = \tau\epsilon$ ;
  After  $\rho$  iterations update  $\theta_M = \vartheta_T$ ;
end

```

**Algorithm 1:** Double DQN Optimal Execution Training Method

are better than others, and we can exploit this knowledge by greedily selecting the best action given the state we are in.

For the exploration/exploitation trade off we use  $\epsilon$ -greedy [12] actions. Specifically, we set

$\epsilon \in (0, 1)$ , let  $\xi \sim \text{Unif}(0, 1)$ , and use the policy

$$(5.1) \quad x_{T_k} = \begin{cases} \text{Binomial} \left( q_{T_k}, \frac{1}{T_N - T_k} \right), & \text{if } \xi < \epsilon \\ \max_{x \in [0, q_{T_k}]} Q(s_{T_k}, x \mid \vartheta_T), & \text{otherwise.} \end{cases}$$

This selects the current estimate of the optimal action an average of  $(1 - \epsilon)$  of the time and a random action sampled from a Binomial the other  $\epsilon$  of the time. We decay  $\epsilon \leftarrow \tau\epsilon$ , with  $\tau < 1$ , because as we observe more data, the network estimate of  $Q$  is more accurate, and we wish to exploit more often. The  $\epsilon$ -Greedy approach relies on random exploration as opposed to other distribution based approaches like Boltzmann Exploration or by using a Bayesian Neural Network.[12] For the  $\epsilon$ -Greedy action, we choose a binomial distribution so that (i) the action respects the constraint  $x \in [0, q]$ , (ii) on average, the selected action equals  $\frac{q_{T_k}}{T_N - T_k}$  which is a TWAP strategy for the remaining inventory, and (iii) the action space is sufficiently explored.

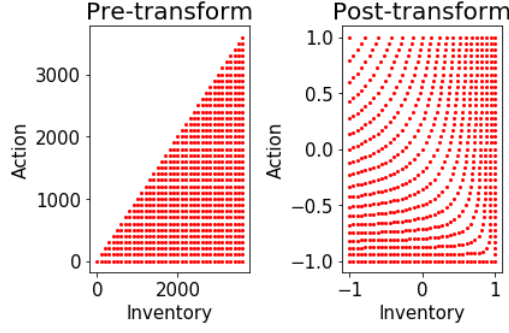
**5.2. Hyperparameters.** As seen in Algorithm 1, there are a number of hyperparameters aside from the usual neural network hyperparameters. Additionally, there is (i) the replay memory size  $\mathcal{N}$ , (ii) the rate of decay of  $\epsilon$  given as  $\tau$ , and (iii) the rate at which to update the target and main network  $\rho$ . These hyperparameters cannot be tuned with cross-validation due to the training procedure for reinforcement learning algorithms and are therefore carefully selected. Another important hyperparameter specific to the optimal execution problem is the quadratic penalty coefficient  $a$ . The parameter  $a$  is selected based on the liquidity of the stock and the trader’s estimated transaction costs.

**5.3. Pre-training.** In order to increase network stability during training, the network is first trained on a set of boundary action cases using randomly selected price intervals from the full data set. These boundary cases are: hold the full inventory then sell all shares at the last time step, and sell all shares at the at the first time step.

**6. Features.** The state space is defined by a set of features we chose to observe from the market and used as inputs to our neural network. In the following section, we will explain each input variable and the transformations applied to project them into the range  $[-1, 1]$ .

The two key features are time and inventory. It is vital to track how much time remains to execute trades and how much inventory remains to be executed. We transform time by an affine transformation so that it lies in the interval  $[-1, 1]$ . As the domain of allowable actions is restricted to be less than the inventory remaining at the start of the same period, the original domain of inventory/action pairs lie in a triangular region in inventory/action space (see the left panel of Figure 3). To increase stability of the algorithm, we transform the triangular region into the domain  $\mathcal{K} = [-1, 1] \times [-1, 1]$ . The individual points may be seen in Figure 3 and details of the transformation can be found in Appendix A.1.

Other important features include price, quadratic variation and price trend. Price  $p_{t_k, i}$  is taken as the midprice at the end of each second. We subtract price at the beginning of each hour from the midprice, and perform an affine transformation so that only outliers in the historical prices lie outside the range  $[-1, 1]$ . We denote this feature as the transformed price  $(\tilde{P})$ . Quadratic Variation (QV) [2] is a measure of the volatility of the asset prize and



**Figure 3.** Inventory and Action Transformation. For explicit formula see Appendix A.1.

we anticipate that volatility affects the optimal strategy. To incorporate this effect, We use QV from the previous period as a feature, and estimate it as

$$(6.1) \quad QV_{T_k} = \sum_{i=0}^{M_{k-1}-1} (p_{t_{k-1,i}} - p_{t_{k-1,i-1}})^2, \quad \forall k \in \{1, \dots, N-1\}$$

The initial QV value,  $QV_{T_0}$ , is defined using price data from the period before trading begins. To ensure the feature corresponding to QV lies in the interval  $[-1, 1]$ , we transform the QV by subtracting the mean and scaling by twice the standard deviation.

## 7. Results.

**7.1. Data.** We test our approach on data using all trading days from 2-January-2017 to 30-March-2018 and stocks Apple (AAPL), Amazon (AMZN), Facebook (FB), Google (GOOG), Intel (INTC), Microsoft (MSFT), NetApp Inc. (NTAP), Market Vectors Semiconductor ETF (SMH) and Vodophone (VOD). We use the full limit order book information to extract the midprice at the end of each second. As well, we exclude the most volatile times of the day and are able to avoid the diurnal patterns by analyzing the hours 11am-12pm, 12pm-1pm, 1pm-2pm of each day separately. These trading hours are referred to as hours 11, 12 and 13.

**7.2. Evaluation Metrics.** To evaluate the performance of our solution, we use Profit and Loss with Transaction Cost (P&L) computed for each trading hour  $b \in \mathcal{B} = \{11, 12, 13\}$  as follows

$$(7.1) \quad P\&L_b = \sum_{k=0}^{N-1} \sum_{i=0}^{M_k-1} \left\{ x_{t_{k,i}} p_{t_{k,i}} - a \left( \frac{x_{T_k}}{M_k} \right)^2 \right\},$$

where  $\frac{x_{T_k}}{M_k}$  is the number of shares executed,  $p_{t_{k,i}}$  the price at the start of the second, and the term  $-a \left( \frac{x_{T_k}}{M_k} \right)^2$  accounts for the penalty when executing large trades.

P&L is computed for each hour of trading and compared to a Time-Weighted Average Price (TWAP) strategy (i.e. selling the same number of shares at each action-executable timestep). TWAP is often used a fair price of the asset of the duration of a trader, and is

the resulting optimal strategy when agents believe the asset price process is a martingale [1]. As a measure of relative performance, we use the basis point improvement of P&L relative to TWAP, defined as

$$(7.2) \quad \Delta P\&L_b := \frac{P\&L_b^{\text{model}} - P\&L_b^{\text{TWAP}}}{P\&L_b^{\text{TWAP}}} \times 10^4, \quad \forall b \in \mathcal{B}.$$

Given the relative performance for each day, and each trading hour, we report its mean, median, standard-deviation, gain-loss ratio

$$GLR := \frac{\mathbb{E}[\Delta P\&L \mid \Delta P\&L > 0]}{\mathbb{E}[-\Delta P\&L \mid \Delta P\&L < 0]},$$

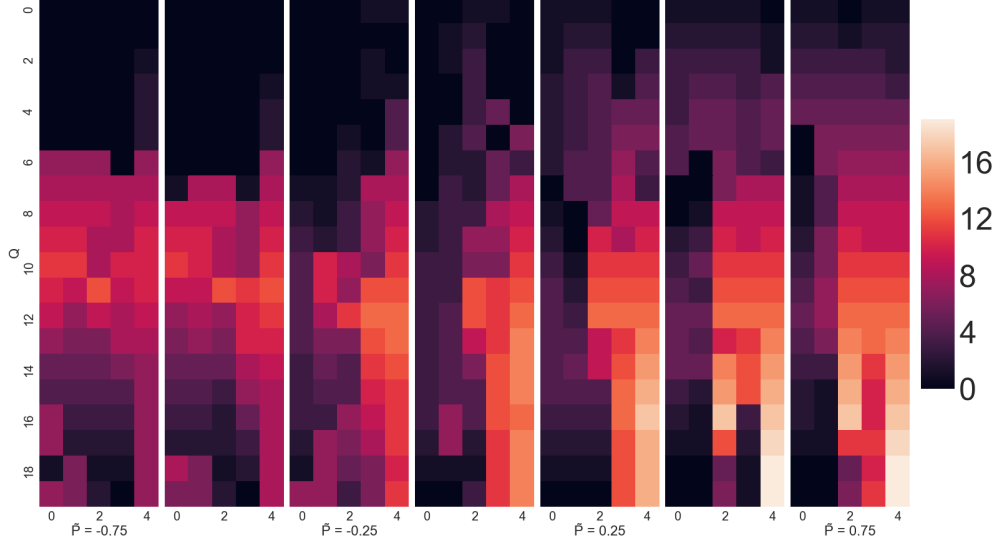
and the positive probability  $\mathbb{P}(\Delta P\&L > 0)$ .

**7.3. Experiments.** For the experiments, we set  $Q_0 = 20$  lots (1 lot equals 100 shares) over a time horizon of  $T = 1$  hour. We split the hour evenly into  $N = 5$  periods. At the beginning of each interval we compute the number of shares we wish to execute. The shares are then executed each second evenly across the entire interval. We test the neural network architecture specified in Section 4.1 with a quadratic penalty  $a = 0.01$ , main-target network update after  $\rho = 15$  iterations, and a replay memory of size  $\mathcal{N} = 10,000$ . We fix our discount factor  $\gamma = 0.99$  for all tests.

**7.3.1. Time, Inventory.** The two key features that affect the rate of trading are Time and Inventory. Our results show that the optimal policy in this case is purely deterministic. This is consistent with the information filtration available to a trader who makes trades using only their current inventory and time. This is also consistent with the continuous time stochastic control approaches, see, e.g., [1][4].

**7.3.2. Time, Inventory, Price.** Once a network model is trained on data, we iterate through all the time steps, inventory levels and price levels and compute the optimal actions traced out in every state. Those optimal actions are displayed in the heatmaps in Figure 4. As the figure shows, for the same level of inventory and time period, as we move from the left panel (lower prices) to the right panel (higher prices), the optimal strategy is to send more shares. As well, for any given panel, as we move from earlier times to later times, with the inventory remaining held fixed, the optimal strategy is to increase the shares executed. At the very last time period, all remaining shares are executed, regardless of inventory remaining or price. Finally, for a fixed time period, as the inventory remaining decreases, the optimal strategy is to execute less shares. All of these observed features of the optimal strategy are consistent with the trader aiming to execute all shares by the end of the time horizon, taking advantage of price improvements, all while managing their inventory risk.

**7.3.3. Time, Inventory, Price, Quadratic Variation.** Next, we add QV to assess how volatility alters the optimal solution. To illustrate how the strategy is affected by time, price, inventory, and QV, in Figure 5 we show heatmaps of the optimal strategy split across different price and QV levels, as a function of time and inventory. As QV increases, the number of executions increases regardless of the price. Periods of high volatility implies increased



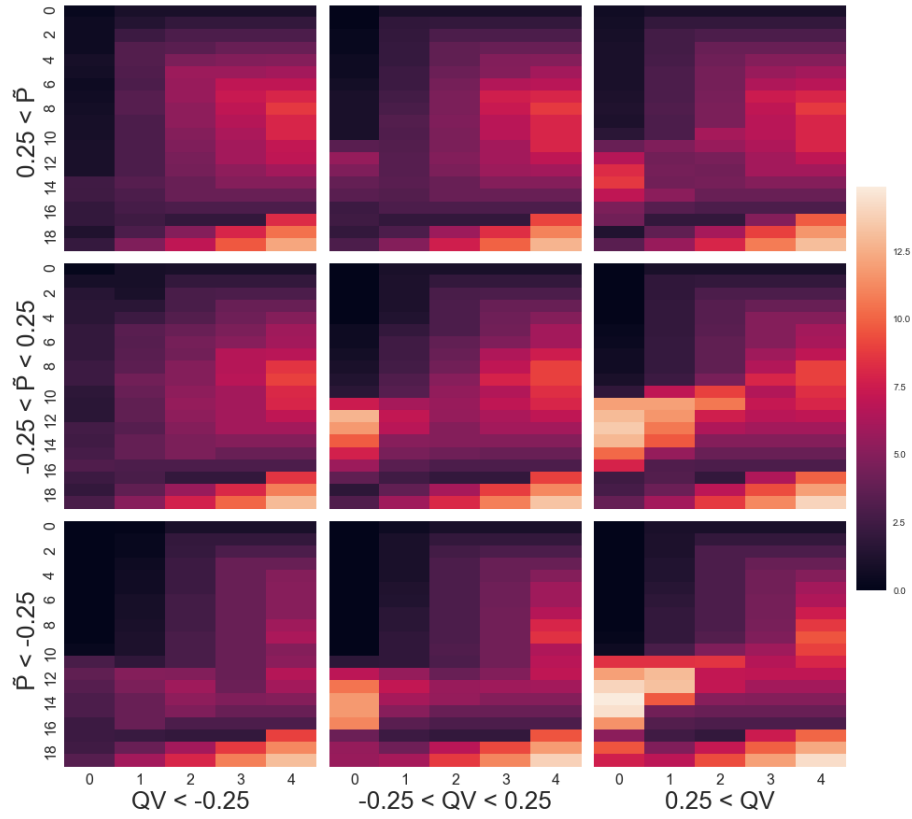
**Figure 4.** NTAP – Optimal Actions for All Possible States with Features: Time, Inventory, Price — Each plot from left to right denotes a change from low to high (normalized) price. Within each panel, the  $x$  and  $y$  axis denote time and inventory remaining, respectively. The color of individual squares denote the amount of shares sold, lighter being more shares, darker being less.

fluctuations in price, and therefore the agent wishes to get rid of their shares rather than be exposed to this risk. As prices move from low to high, there is a visible increase in the shares executed, which is consistent with the results in Section 7.3.2 and Figure 4.

**7.4. Statistical results.** Table 1 shows various comparative statistics of the learned strategies using TIP and TIPQV for a collection of assets.

While the results vary from one asset to the next, generally the relative P&L has positive Mean and Median, as well, the gain-loss ratio is at least 1, and the probability of outperforming TWAP is high. Only Google and Amazon appear to have no significant improvement beyond TWAP. Moreover, adding quadratic variation generally improves the performance of the strategy relative to TWAP.

The corresponding histograms of the relative P&L are shown in Figures 6 and 7.



**Figure 5.** NTAP – Average Optimal Actions Split on QV and Price with Features: Time, Inventory, Price, QV. — For each individual plot, the x-axis denotes time intervals and y-axis inventory remaining. Moving across graphs, the x-axis denote changes from low to high QV, whereas the changes in the y-axis denote changes in price from low to high.



**Table 1***Relative P&L performance for all stocks with respect to TWAP strategy.*

Ticker	Features	$\Delta P\&L$			GLR	$\mathbb{P}(\Delta P\&L > 0)$
		Median	Mean	Std.Dev.		
AAPL	TIP	2.92	2.85	6.1	1.0	77.4%
	TIPQV	2.68	2.68	5.3	1.2	76.2%
AMZN	TIP	0.06	-0.28	11.3	0.9	50.1%
	TIPQV	-0.02	-0.15	11.7	1.0	49.9%
FB	TIP	2.61	2.52	7.6	1.2	68.1%
	TIPQV	2.29	2.26	8.3	1.1	64.3%
GOOG	TIP	-0.59	0.21	7.4	1.3	45.7%
	TIPQV	-0.40	0.05	11.2	1.1	47.2%
INTC	TIP	11.63	11.08	5.6	2.5	95.8%
	TIPQV	11.96	11.40	4.0	3.6	97.8%
MSFT	TIP	5.97	5.93	3.0	3.1	97.4%
	TIPQV	5.89	5.95	3.8	2.8	94.8%
NTAP	TIP	10.13	9.62	5.0	1.8	96.4%
	TIPQV	10.05	9.64	5.0	2.1	96.2%
SMH	TIP	3.80	2.67	6.7	1.0	73.5%
	TIPQV	4.90	4.86	3.9	2.1	91.8%
VOD	TIP	14.68	13.99	5.2	4.1	97.8%
	TIPQV	15.72	15.43	3.1	18.6	99.2%

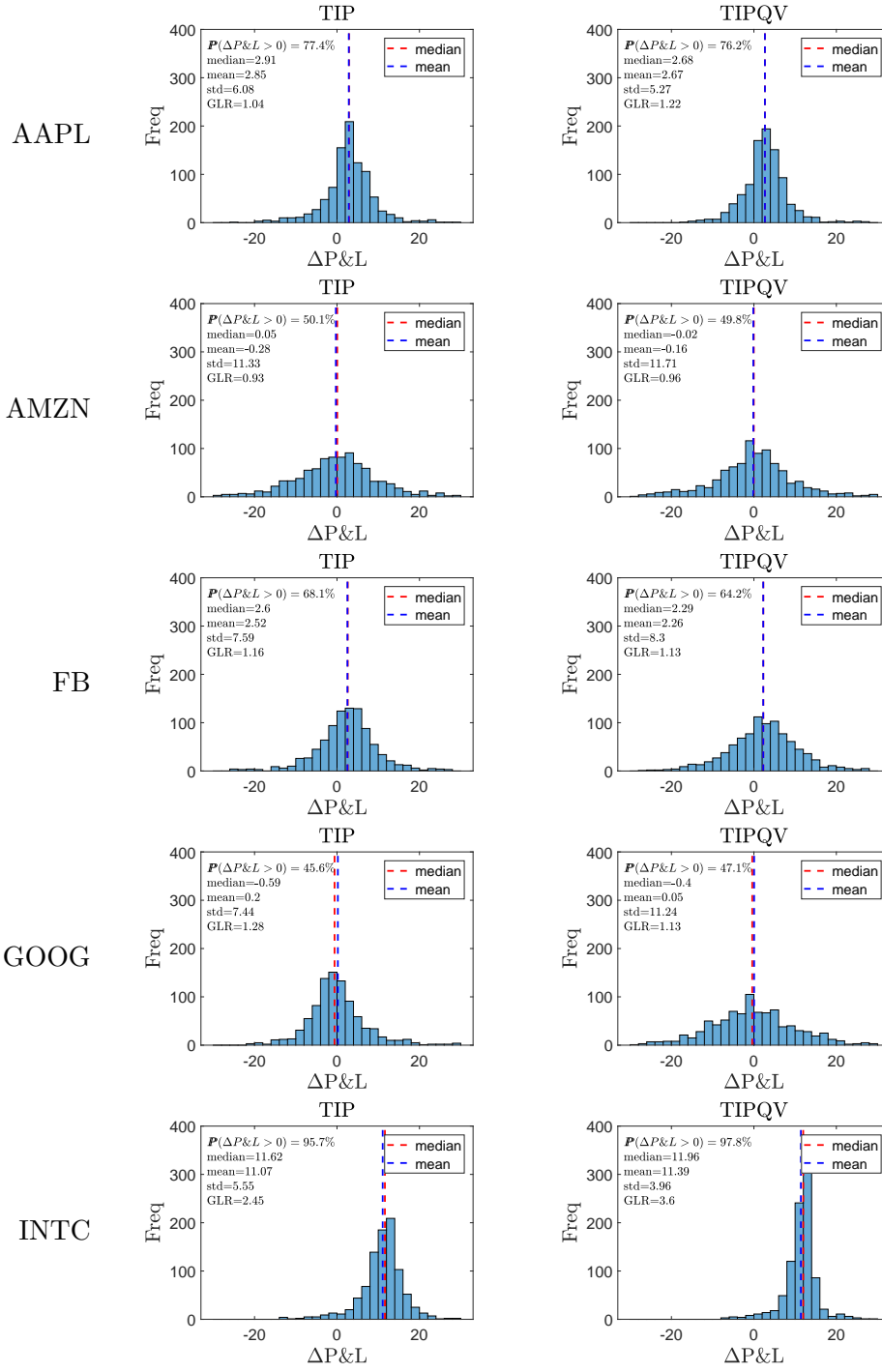


Figure 6. Distribution of the relative P&L (basis points) with various features.

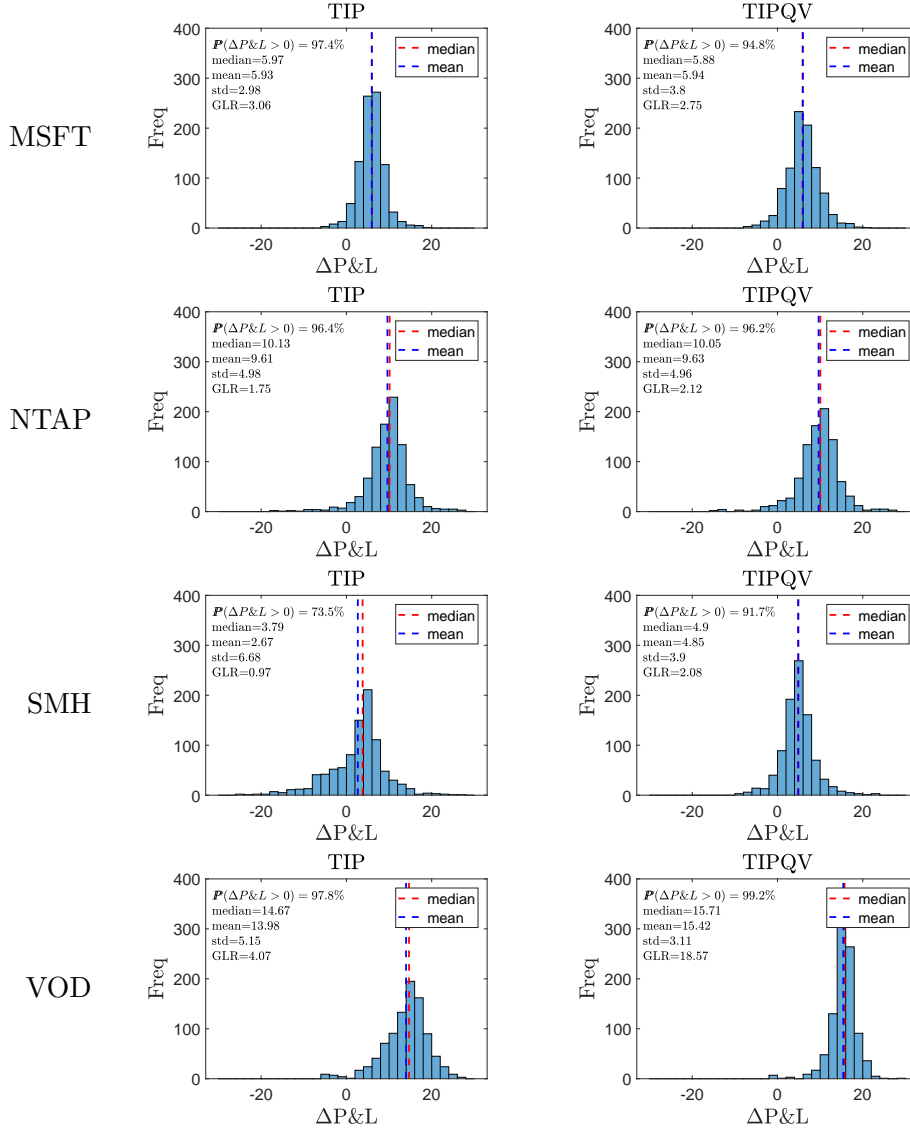


Figure 7. Distribution of the relative  $P\&L$  (basis points) with various features.

**8. Conclusions.** In this paper we formulated the optimal execution problem as a reinforcement learning problem. We developed a deep reinforcement learning technique that requires a number of modifications to the double deep Q-learning approach to account for the hard constraints as well as the . The results show the approach outperforms TWAP on seven out of the nine stocks and for the two under performing stocks, one is statistically insignificant. There are a number of directions still left open for investigation. Two obvious direction are to increase the number of assets analysed and to include a number of additional features, such as, price history and limit order book history. Another direction is to combine the analytical approaches, e.g., [3] and [5], or data-driven modeling as in [11], together with deep Q-learning by using the analytical optimal trading strategy as the starting point for reinforcement learning.

## REFERENCES

- [1] R. ALMGREN AND N. CHRISS, *Optimal execution of portfolio transactions*, Journal of Risk, 3 (2001), pp. 5–40.
- [2] O. E. BARNDORFF-NIELSEN AND N. SHEPHARD, *Estimating quadratic variation using realized variance*, Journal of Applied Econometrics, 17 (2002), pp. 457–477, <https://doi.org/10.1002/jae.691>, <https://doi.org/10.1002/jae.691>.
- [3] Á. CARTEA AND S. JAIMUNGAL, *Incorporating order-flow into optimal execution*, Mathematics and Financial Economics, 10 (2016), pp. 339–364.
- [4] Á. CARTEA, S. JAIMUNGAL, AND J. PENALVA, *Algorithmic and high-frequency trading*, Cambridge University Press, 2015.
- [5] P. CASGRAIN AND S. JAIMUNGAL, *Trading algorithms with learning in latent alpha models*, Mathematical Finance, Forthcoming. <https://doi.org/10.1111/mafi.12194>, (2018).
- [6] O. GUÉANT AND C.-A. LEHALLE, *General intensity shapes in optimal liquidation*, Mathematical Finance, 25 (2015), pp. 457–495.
- [7] D. HENDRICKS AND D. WILCOX, *A reinforcement learning extension to the almgren-chriss framework for optimal trade execution*, in 2014 IEEE Conference on Computational Intelligence for Financial Engineering Economics (CIFEr), March 2014, pp. 457–464, <https://doi.org/10.1109/CIFEr.2014.6924109>.
- [8] L.-J. LIN, *Reinforcement Learning for Robots Using Neural Networks*, PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1992. UMI Order No. GAX93-22750.
- [9] V. MNIH, K. KAVUKCUOGLU, D. SILVER, A. GRAVES, I. ANTONOGLOU, D. WIERSTRA, AND M. A. RIEDMILLER, *Playing atari with deep reinforcement learning*, CoRR, abs/1312.5602 (2013), <http://arxiv.org/abs/1312.5602>, <https://arxiv.org/abs/1312.5602>.
- [10] Y. NEVMYVAKA, Y. FENG, AND M. KEARNS, *Reinforcement learning for optimized trade execution*, in Proceedings of the 23rd International Conference on Machine Learning, ICML '06, New York, NY, USA, 2006, ACM, pp. 673–680, <https://doi.org/10.1145/1143844.1143929>, <http://doi.acm.org/10.1145/1143844.1143929>.
- [11] J. A. SIRIGNANO, *Deep learning for limit order books*, Quantitative Finance, (2018), pp. 1–22.
- [12] R. S. SUTTON AND A. G. BARTO, *Introduction to Reinforcement Learning*, MIT Press, Cambridge, MA, USA, 1st ed., 1998.
- [13] H. VAN HASSELT, A. GUEZ, AND D. SILVER, *Deep reinforcement learning with double q-learning*, CoRR, abs/1509.06461 (2015), <http://arxiv.org/abs/1509.06461>, <https://arxiv.org/abs/1509.06461>.
- [14] C. J. C. H. WATKINS AND P. DAYAN, *Q-learning*, Machine Learning, 8 (1992), pp. 279–292, <https://doi.org/10.1007/BF00992698>, <https://doi.org/10.1007/BF00992698>.

## Appendix A. Appendices.

**A.1. Transforming Inventory-Action Domain.** For all inventory-action pairs  $\{(q, x) : (q, x) \in (0, q_0]^2, x \leq q\}$ , we perform the following non-linear transformation into the do-

main  $[-1, 1]^2$ . First, shift and normalize

$$(A.1) \quad \hat{q} = \frac{q}{q_0} - 1, \quad \hat{x} = \frac{x}{q_0}.$$

Next, define the radial distance, angle, and ratio as

$$(A.2) \quad \begin{aligned} r &= \sqrt{q^2 + x^2}, \\ \theta &= \tan^{-1} \left( -\frac{x}{q} \right), \quad \text{and} \\ \zeta &= -\frac{x}{q}, \end{aligned}$$

respectively. Transform the radial distance via

$$(A.3) \quad \tilde{r} = \begin{cases} r \sqrt{(\zeta^2 + 1) (2 \cos^2(\frac{\pi}{4} - \theta))}, & \theta \leq \frac{\pi}{4} \\ r \sqrt{(\zeta^{-2} + 1) (2 \cos^2(\theta - \frac{\pi}{4}))}, & \theta > \frac{\pi}{4}. \end{cases}$$

Finally convert the polar coordinates into the domain  $[-1, 1]^2$  to produce the features

$$(A.4) \quad \tilde{q} = -\tilde{r} \cos(\theta), \quad \text{and} \quad \tilde{x} = \tilde{r} \sin(\theta).$$

This produces the transformation of the inventory/action pairs as shown in Figure 3.