Autonomous Equity Trading using Deep Reinforcement Learning

Jad Soucar (jadsoucar@g.ucla.edu)
Department of Mathematics, 520 Portola Plaza, University of California Los Angeles
Los Angeles, CA 90095 USA
Ninaad Surana (ninaadsurana@g.ucla.edu)
Department of Mathematics, 520 Portola Plaza, University of California Los Angeles
Los Angeles, CA 90095 USA

March 17, 2024

Abstract

In this paper, we explore the application of Deep Reinforcement Learning (DRL) to the domain of autonomous equity trading, with a particular focus on the use of Deep Q Networks (DQNs) to develop trading agents capable of navigating the complex and dynamic landscape of the financial markets. We introduce three variants of the DQN model; Vanilla DQN (V-DQN), Target DQN (T-DQN), and Double DQN (D-DQN). Our models incorporate a comprehensive set market indicators into the state space and several risk metrics into the reward function to guide the trading decisions towards not only profitability but also risk-adjusted returns. The risk metrics include the Sharpe Ratio, Sortino Ratio, and Treynor Ratio. We test each Q learning scheme outlined above with each risk metrics to determine the best trading agent. We find that most trading agents earn a percentage increase of around 6%-13%. After training, we find that incorporating the Sharpe ratio into the reward function produces the best return, and that the Double DQN algorithm is optimal across all risk metrics.

Keywords: Reinforcement Learning, Q Learning, Finance, Agent Based Modeling, Equity Trading

1 Introduction

In recent years, advancements in the fields of Deep and Reinforcement Learning have transformed algorithmic trading. With the surge in computational power and increasing amounts of financial data, traders and researchers have been exploring novel methods to create autonomous trading agents that can operate successfully in financial markets. One such strategy is Deep Q Learning, which has dominated various domains such as game playing, robotics, and now, financial trading.

With this in mind, we form the questions, how do we create such an agent, and guide it to make valid decisions? How do we ensure the stability of the agent? How do we improve upon previous attempts? We observed that in existing literature, many researchers had attempted to create such a Deep Q Learning system using ordinary Profit-Loss based Reward systems and state spaces that contain only the asset's price. In order to improve performance, we decided to incorporate a wider range of performance indicators and risk metrics into our agent's decision making process. We hope to create an agent that can mimic a human investor with strong insight into market trends and behaviour. The primary concerns surrounding autonomous trading are rooted in the belief that technical analysis does not account for underlying factors that drive growth. For example if an autonomous agent sees unexpected behaviour in the markets, or is unable to access long horizons of meaningful data, it may be unable to act as an informed human would by analyzing the key indicators and risk metrics. However, trading algorithms introduce their own advantages, such as a lack of emotion or bias, and the ability to place buy and sell orders instantly without need for time-intensive deliberation. Moreover, algorithms have the ability to identify trends and patters that may be difficult for a human trader to observe. Overall, we seek to combine all the advantages of both human and autonomous trading, and build an agent that can be continuously improved. In our implementation, we aim to contribute to the growing field of Reinforcement Learning (RL), and expand its applications into financial markets.

In this paper, we focus on two problems native to the use of RL in finance. The first issue is the noise contained in the underlying data, which can make training a Deep RL network difficult. To mitigate this problem we implement Fixed Target Distribution Deep Q Learning and Double Deep Q Learning and compare their efficacy to a traditional Deep Q Learning framework. Next we acknowledge that many RL trading agents only use their profit or loss to inform investment decision. However, in reality traders are often concerned with hedging risk. In an effort to replicate this type of trading behavior we use risk metrics as a way of assessing an agent's investment decisions. To do this, we implement different combinations of metrics such as the Sharpe Ratio, Sortino Ratio, and Treynor Ratio. We then present results of the agent's performance using different training schemes. We see that the agent always delivers promising rates of return, between 6% and 13%. Specifics on the implementation of our trading environment, algorithms used, and experimental results will be detailed in further sections.

1.1 Paper Organization

We begin with a brief introduction of necessary mathematical prerequisites of Q Learning. Next, we discuss the necessary financial prerequisites, before proposing several deep Q learning frameworks to solve the problem of optimal equity trading. We end with an evaluation of our results and consider possible future work on this problem.

2 Mathematical Background

In this section we aim to provide the relevant mathematical and programmatic background for implementing a Deep Q Learning algorithm. We discuss what Q learning is and how the principles from classical Q learning can be used to construct deep Q learning algorithms in Python. We will be borrowing heavily from Brunton and Kutz's book titled "Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control".

2.1 General Reinforcement Learning

Before we begin our discussion of Q learning, we will begin with an exposition of the two primary categories of RL: model-based and model-free. Model-based RL systems create a model of the environment, which includes the dynamics of how actions lead to subsequent states and rewards. Essentially, it predicts the future states and rewards for actions taken from a given state. Take for example the gambler problem, which provides explicit probability measure $P(s', s, a)$ that provides explicit probabilities as to whether the actor will win or lose a gamble. In other words, the gambler queries the environment for its transition dynamics. The second approach is model-free, which is where a system learns a policy or value function directly from interactions with the environment without constructing an explicit model of the environment's dynamics. The system learns what to do by trial and error, adjusting its actions based on the rewards received.

The second distinction that we'd like to note is two subcategories of model-free RL; off and on-policy learning. On-Policy learning algorithms learn the value of the policy that is currently being used to make decisions. Put simply, the policy used to generate behavior (exploration) is the same policy that is being evaluated and improved. Essentially, it learns on the job, using the data generated by its current strategy. By contrast off-policy learning algorithms can learn about one policy (the target policy) while using a different policy (the behavior policy) to generate behavior. This means it can learn from data that was generated from a previous policy or even from data generated by other agents. This separation allows for greater flexibility and efficiency in some contexts.

2.2 Q Learning

With the general background of RL out of the way, we note that Q learning is a model-free and off-policy training scheme. That means that Q learning does not rely on an environment that can produce exact transition probabilities and must instead use trial and error of optimal or sub-optimal actions in order to learn the optimal policy. In order to formalize this type of training scheme we first define the quality function $Q(s,a)$, which tells you the joint value/quality of taking an action a given a current state s.

To formally define $Q(s,a)$ we first introduce some notation. We let $R(s',s,a)$ be the reward of transitioning from state s to s' through action a, γ be the discount factor of future reward, and $V(s')$ be the expected value over all possible future rewards given a current state s'. With that we define $Q(s,a)$ as

$$\begin{aligned} Q(s,a) &= \mathbb{E}\left(R(s',s,a)+\gamma V(s')\right) \\ &= \sum_{s'} P(s'|s,a)(R(s',s,a)+\gamma V(s')) \end{aligned} \tag{1}$$

In other words, Q is the expected sum of the instantaneous reward of the state-action pair (s,a) along with the discounted future rewards of being at a new state s' brought on by (s,a). Using the quality function $Q(s,a)$ we can define an optimal policy π and value function $V(s)$, that considers which action a is optimal and what the expected reward from taking that action is.

$$V(s) = \max_a Q(s,a) \tag{2}$$

$$\pi(s) = \arg\max_a Q(s,a) \tag{3}$$

We can even rewrite the equations above in terms of Bellman's equation, which tells us that the value of being in state s is the expected current and future return of applying the optimal action a.

$$V(s) = \max_\pi \mathbb{E}\left(R(s',s,a)+\gamma V(s')\right) \tag{4}$$

We've defined what a q value is and how to construct the quality function $Q(s,a)$, but now we define a recursive equation to update $Q(s,a)$ as the agent learns through trial and error.

$$Q^{\text{new}}(s_k,a_k) = Q^{\text{old}}(s_k,a_k) + \alpha\left(\underbrace{r_k + \gamma\max_a Q(s_{k+1},a) - Q^{\text{old}}(s_k,a_k)}_{\text{TD Error}}\right) \tag{5}$$

Let's dissect this update equation. As we engage in trial and error learning, we update our $Q(s,a)$ by slowly nudging our q values up or down by a factor of the difference between the actual current and future reward $r_k + \gamma\max_a Q(s_{k+1},a)$ and our predicted reward $Q^{\text{old}}(s_k,a_k))$. This difference is sometimes referred to as the "Temporal Difference (TD) error". We also note that α in this case is the learning rate.

We draw the readers attention to two interesting features of the update equation. First to the term $\gamma\max_a Q(s_{k+1},a)$. The reason why we maximize Q over a is to guarantee that the quality $Q(s_k,a_k)$ is a function of the optimal actions that can be taken given a new state s' brought on by (s_k,a_k). r_k on the hand, which is the reward derived from the pair (s_{k-1},a_{k-1}) is not required to be the optimal reward, and will most likely be sub-optimal during the training process. This is precisely the reason why Q Learning is an off-policy learning scheme.

Second we note that when calculating the TD error, the agent calculates the future reward by looking one step into the future. This idea of looking forward in time by one time step to calculate error is why the error is called the TD error. Naturally, the degree to which the agent looks into the future can be modified.

2.3 Deep Q Learning Variations

Now we discuss the process of recasting Q Learning into training schemes that incorporate deep learning methods like neural networks. This is especially useful in settings where the state space is too large to reasonably store all q values in a Q table. Here, a deep learning approach seeks to parameterize $Q(s,a)$ as dependant on some weights θ such that

$$Q(s,a) \approx Q(s,a,\theta) \tag{6}$$

In order to optimize for the parameters θ within a Q Learning framework we use a loss function that is eerily similar to the TD error defined in equation (5). This framework is known as a Vanilla Deep Q Network (V-DQN).

$$\mathcal{L} = \mathbb{E}[(r_k + \gamma \max_a Q(s_{k+1}, a, \theta) - Q(s_k, a_k, \theta))^2] \tag{7}$$

With the loss function properly defined, we can move on to the neural network architecture, and how it allows us to approximate the Q function. In practice, we find that the architecture of the network varies based on the use case. For example a team of DeepMind engineers in 2013 coupled the loss function described above with several convolutions layers and fully connected layers. The inputs were several consecutive frames for the game, which represented the agent's state. And the output was one of several possible action that the agent could take. Regardless of the architecture used the value contained in each output node approximates the value $Q(s,a)$ for the associated (s,a) which corresponds to the network's $(\text{Input}, \text{output})$ pair.

The deep Q networks described above can be difficult to train due large variance in the estimated Q function. To mitigate this problem we can introduce two separate schemes to deal with this problem.

2.3.1 Stabilization Schemes

First we introduce the fixed target distribution Q learning framework, known as T-DQN. This framework involves introducing a second neural network to approximate the temporal difference target, instead of using the same main network to predict the quality of the next state. Under this framework TD Error and TD Target are defined as follows

$$\underbrace{\overbrace{r_k + \gamma \max_a Q(s_{k+1}, a; \theta^-)}^{\text{TD Target}} - Q^{\text{old}}(s_k, a_k; \theta)}_{\text{TD Error}} \tag{8}$$

where $Q(s', a'; \theta^-)$ is the Q value predicted by the target network for the next state-action pair, and $Q(s, a; \theta)$ is the Q value predicted by the main Q network for the current state-action pair.

$$TD^{\text{T-DQN}}_{\text{target}} = r + \gamma \max_{a'} Q(s', a'; \theta^-), \tag{9}$$

However, unlike the main network, instead of updating the target network at every training step, which can lead to high variance in the target Q values and general instability in training, the target network's weights are only updated after a fixed number of steps or episodes. This creates a stable target distribution for the Q values against which the policy network is updated. However, given the training scheme of the target network it is functionally just a delayed copy of the main network. Because the target and actual Q values are being estimated by similar networks, we run the risk of overestimating the target when our data has high variance. To mitigate the problem of overestimation we can propose another framework called the Double Deep Q Network (D-DQN).

Like fixed target distribution Q learning, a double deep Q learning framework relies on a target network and a main network. However in this case, the main Q network is responsible for choosing the best action, but the target network is used to evaluate the quality of that action. Its a fine difference that can be formulated as follows

$$TD^{\text{DDQN}}_{\text{target}} = r + \gamma Q(s', \underset{a'}{argmax} Q(s', a'; \theta); \theta^-), \tag{10}$$

3 Financial Background

In this section we transition to some necessary financial background for creating our trading agent. Specifically we introduce several widely used market features and risk metrics. For the sake of this section and the rest of the paper, assume that

$$S_t = \text{Price at time } t \tag{11}$$

3.1 Market Features

We introduce 4 commonly used market features: Moving Average Convergence Divergence (MACD) Aguirre et al. (2020), Relative Strength Index (RSI) Adrian (2011), Commodity Channel Index (CCI) Maitah et al. (2016), Average Directional Index (ADX) Gurrib (2018).

3.1.1 MACD

In order to define MACD, we must first define the Exponential Moving Average (*EMA*)

$$\text{EMA}_t = \left(S_t \times \frac{2}{N+1}\right) + \text{EMA}_{t-1} \times \left(1 - \frac{2}{N+1}\right) \tag{12}$$

where N is the number of periods that the EMA is calculated over. For example $N = 12$ means that *EMA* is calculated over 12 days.

The MACD is a trend-following momentum indicator that shows the relationship between two moving averages of a stock's price, whereas the signal line is the EMA of that same difference. The MACD can help identify the overall trend. When the MACD Line is above the Signal Line, it's considered bullish. When the MACD Line is below the Signal Line, it's considered bearish.

$$\text{MACD Line} = \text{EMA}_{12}(S_t) - \text{EMA}_{26}(S_t) \tag{13}$$

$$\text{Signal Line} = \text{EMA}_9(\text{MACD Line}) \tag{14}$$

3.1.2 Relative Strength Index (RSI)

The RSI is a momentum oscillator that can be defined as follows. The RSI is a momentum oscillator that measures the speed and change of price movements. RSI oscillates between zero and 100. Traditionally, RSI is considered overbought when above 70 and oversold when below 30. Before defining RSI, we must first define a few terms. G_t represent the gain at time t, where $G_t = \max(S_t - S_{t-1}, 0)$. L_t represent the loss at time t, where $L_t = \max(S_{t-1} - S_t, 0)$.

The average gain and average loss over a specified period N (traditionally 14) are calculated as follows:

$$\text{Average Gain} = \frac{1}{N}\sum_{i=1}^{N} G_{t-i} \tag{15}$$

$$\text{Average Loss} = \frac{1}{N}\sum_{i=1}^{N} L_{t-i} \tag{16}$$

The relative strength (RS) is the ratio of average gain to average loss:

$$RS = \frac{\text{Average Gain}}{\text{Average Loss}} \tag{17}$$

The Relative Strength Index (RSI) is then calculated using the RS value:

$$RSI = 100 - \left(\frac{100}{1+RS}\right) \tag{18}$$

3.1.3 Commodity Channel Index (CCI)

The CCI compares a securitie's current price change with its average price change over a given period. It's used to identify cyclical trends in commodities but can be applied to other types of assets. Though it can vary by market and the time frame you're looking at, CCI readings above +100 typically indicate overbought conditions, suggesting a price reversal might be near. Readings below -100 indicate oversold conditions, possibly heralding a bullish reversal. In order to formalize CCI we first define Typical Price (TP), and Simple Moving Average of TP (SMATP).

$$\mathrm{TP} = \frac{S_{t,\mathrm{high}} + S_{t,\mathrm{low}} + S_t}{3} \tag{19}$$

$$\mathrm{SMATP} = \frac{\sum \mathrm{TP}_t}{N} \tag{20}$$

$$\text{Mean Deviation} = \frac{1}{N}\sum_{t=0}^{N} |TP - SMATP| \tag{21}$$

$$\mathrm{CCI} = \frac{\mathrm{TP} - \mathrm{SMATP}}{0.015 \times \text{Mean Deviation}} \tag{22}$$

3.1.4 Average Directional Index (ADX)

The Average Directional Index (ADX) is a technical analysis indicator used to quantify the strength of a trend. The ADX itself doesn't indicate trend direction, but it measures the strength of the current trend, whether up or down. Values above 25 usually indicate a strong trend, while values below 20 indicate a weak trend or trading range. Values in between suggest a developing trend. However, in order to define ADX, we must once again define several new terms.

H_t represents the high price at time t, L_t represents the low price at time t, C_t represents the close price at time t. DM^+ and DM^- denote the positive and negative directional movements, respectively. TR denotes the true range. $+DI$ and $-DI$ represent the positive and negative directional indicators, respectively. We can define each of these terms bellow.

$$DM^+ = \max(H_t - H_{t-1}, 0) - \min(L_t - L_{t-1}, 0) \tag{23}$$

$$DM^- = \max(L_{t-1} - L_t, 0) - \min(H_{t-1} - H_t, 0) \tag{24}$$

$$TR = \max(H_t - L_t, |H_t - C_{t-1}|, |L_t - C_{t-1}|) \tag{25}$$

$$+DI = 100 \times \frac{DM^+}{TR} \tag{26}$$

$$-DI = 100 \times \frac{DM^-}{TR} \tag{27}$$

$$ADX = 100 \times \frac{\text{EMA of } |(+DI) - (-DI)|}{TR} \tag{28}$$

3.2 Risk Metrics

We now introduce 4 risk metrics, that allow a market agent to evaluate the risk of an investment: Sharpe Ratio, Sortino Ratio, Treynor Ratio, Beta.

3.2.1 Beta (β)

Beta (β) measures a security's sensitivity to market movements. It represents the tendency of a security's returns to respond to swings in the market and is a key component in the Capital Asset Pricing Model (CAPM).

$$\beta = \frac{\mathrm{Cov}(R_s, R_m)}{\mathrm{Var}(R_m)} \tag{29}$$

3.2.2 Sharpe Ratio

The Sharpe Ratio evaluates the performance of an investment compared to a risk-free asset, after adjusting for its risk. It is used to understand how much excess return is being received for the extra volatility that one bears for holding a riskier asset.

$$\text{Sharpe Ratio} = \frac{R_p - R_f}{\sigma_p} \tag{30}$$

3.2.3 Sortino Ratio

The Sortino Ratio improves upon the Sharpe Ratio by focusing only on the downside deviation instead of the total standard deviation. This makes it a better measure of the risk-adjusted return when the investor is concerned about downside risk.

$$\text{Sortino Ratio} = \frac{R_p - R_f}{\sigma_d} \tag{31}$$

3.2.4 Treynor Ratio

The Treynor Ratio is similar to the Sharpe Ratio but uses beta (β) instead of standard deviation to measure volatility. It assesses the returns earned in excess of the risk-free rate per unit of market risk and is particularly useful for diversified portfolios.

$$\text{Treynor Ratio} = \frac{R_p - R_f}{\beta_p} \tag{32}$$

4 Model Development

In this section we develop our model by first defining the environment,state,action, and reward space. Then we choose a specific deep Q network architecture, and define the control flow of our model.

4.1 Environment Setup

The stock market is a multi-agent complex system where various agents interact with each other through the purchase or sale of various financial assets. For the sake of our model we restrict our RL agent to trading stock in publicly traded companies. The limit order book (LOB) plays a crucial role in how stocks are priced.

The LOB is a real-time database that lists all open buy orders (bids) and sell orders (asks) for a stock at various price levels. Bids are listed in descending order with the highest price at the top, while asks are listed in ascending order with the lowest price at the top. The LOB facilitates price discovery as the matching of buy and sell orders based on price and time priority determines the current price of the stock. This processes of matching prices based on the difference between the bid and ask price (spread), can be directly observed through the dynamic pricing of the asset. In other words, despite the motivation's of the agents remaining hidden, the state of the environment can be readily observed at the current price S_t. The agent can attempt to recognize patterns in the price S_t, and construct market features similar to those defined in section 3.1.

Above we describe the observable features of the environment, however, for our model we also assume that the RL agent has a set of personal rules that they must follow. The first rule is that the agent can only place a trade of one share per transaction. The second rule we impose, for the sake of simplicity, is that our agent can only place one trade per day. Third we restrict the agent to buying whole shares, and exclude any option of purchasing or selling fractional shares. Fourth, we only allow the RL agent to purchase a single stock. While there are models that allow RL agents to construct portfolios by analyzing several stocks we restrict our RL agent to a single stock for simplicity. Fifth, we allow the agent to borrow money infinitely, such that it can also place a buy order at any point. However, the agent can only place a sell order if it has shares in its inventory. We consider this to be analogous to a trader depositing more money in their trading account. Although we note the presence of transaction fees. However, due to the relatively small volume of trades made by the RL agent, we consider these fees to be negligible, and as a result omit them.

4.1.1 State Space

We initially define our state space to model the task of trading our chosen stock. As such our state s at time t is:

$$\hat{s_t} = p_t \tag{33}$$

where p_t is the current price of a single share of stock. However, as stated above, a market participant will often times calculate additional performance indicators. So we allow the agent to incorporate the market indicators listed in section 3.1 into its state. This is analogous to a trader who observes and allows those indicators to influence their trading decisions. By including market indicators into agent's state, we are formally allowing the deep Q network to use those metrics when calculating $Q(s,a)\ \forall s \in S, \forall a \in A$. With the inclusion of market indicators are state resembles the following.

$$\tilde{s_t} = [p_t,\ MACD,\ RSI,\ CCI,\ ADX] \tag{34}$$

Additionally, we make two key observations about the deficiencies of the state as defined in equation 34. First the trader is more concerned about the difference in price Δ_t (As defined in 4.1.3) then the magnitude of the price itself, since the Δ is more closely related to the agent's profit. Second the trader not only has a memory but ought to use its memory of previous market movements when making a trading decision. Using these two observation's we define an improved state that gives the RL agent access to a tuple of the last N state delta's.

$$\begin{aligned} s_t &= [\Delta_{t-N+1}, \Delta_{t-N+2}, \ldots, \Delta_N] \\ &\in \mathbb{R}^{|\tilde{s_t}| \times N} \end{aligned} \tag{35}$$

where Δ is defined as follows

$$\Delta_t = p_t - p'_t \tag{36}$$

and $p_{t'}$ and p_t come from states $s_{t'}$ and s_t respectively. Specifically, t' is the time of purchase of the cheapest share in the agent's inventory. The intuition behind this construction decision is that an agent will always prioritize locking in profits through selling the stock in its inventory that will generate the largest Δ.

4.1.2 Action Space

The action space of the trading agent is whether to buy, hold, or sell the stock in each time step at the given state. As stated in the environment setup, our agent can only place one trade per day, and the transaction can only be of one share. As such, we define our action space to be:

$$A = \{-1, 0, 1\} \tag{37}$$

where -1 and 1 represent sales and purchases respectively, and 0 means holding.

4.1.3 Reward Function

The reward we set depends on the action our agent decides. If the agent places a sell order, we reward it according to the returns it realises, if the agent places a buy order, we reward it according to the risk of buying that share (i.e a small risk will generate a larger reward), and if the agent chooses to hold for that time step, we give it zero reward. Specifically, the reward R given the states at time t and t' with action a_t is defined as

$$R(s_t, a_t, s_{t'}) = \begin{cases} RiskMetric(s_t) & \text{if } a_t > 0, \\ \Delta_t & \text{if } a_t < 0, \\ 0 & \text{if } a_t = 0. \end{cases} \tag{38}$$

where our Risk Metric is a function to be chosen from those defined in Section 3.2, and Δ_t is defined as in equation 36. We expect that incorporating risk into our reward function will guide the trading agent to make risk sensitive trades similar to a human trader.

4.2 Deep Q Network Setup

4.2.1 Architecture

We implement a Deep Q Network System to compute optimal Q functions and choose appropriate actions. Here we choose three frameworks outline in section 2.1 to implement and test; V-DQN, T-DQN, and D-DQN.

Regardless of our choice of framework, we define all neural networks to have identical architecture. We choose an Artificial Neural Network with six fully connected layers including: our input layer of size $|\tilde{s}_t|N$, a hidden layers of size 128, two hidden layers of size 256, a hidden layer of size 128, and our output layer of size $|A_t|$. Additionally, we define the consistent learning rate to be $\alpha = 0.001$. For clarity, we graphically illustrate the architectures of each DQN we've chosen to implement bellow.

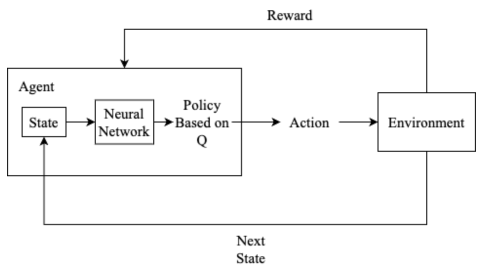


Figure 1: V-DQN Framework

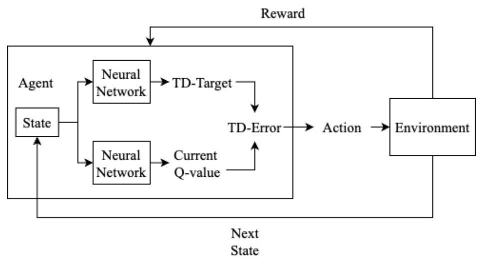


Figure 2: T-DQN Framework

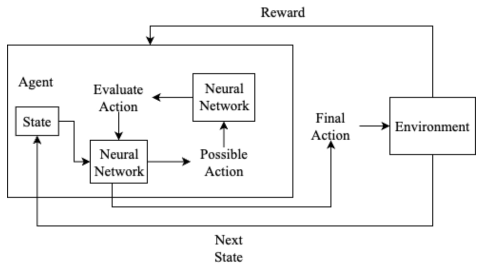


Figure 3: D-DQN Framework

4.2.2 Policy

In all choices above, we follow the ε-greedy policy to catalyze off-policy actions by introducing randomness. Formally ε-greedy is implemented by replacing the $\max_a Q(s_{k+1}, a)$ term in equation 5 with $Q(s_{k+1}, \tilde{a}_{k+1})$ where

$$\tilde{a}_{k+1} = \begin{cases} \text{random action from } A(s_t), & \text{with probability } \varepsilon \\ \arg\max_a Q(s_t, a), & \text{with probability } 1 - \varepsilon \end{cases} \tag{39}$$

As training continues we slowly take $\varepsilon \to 0$ using a simulated annealing strategy. The reason why the ε-greedy strategy can be beneficial, is because it allows the agent to explore the space of possible actions freely at the beginning of the training process while also emphasizing exploitation of the agents accumulated knowledge toward the end of training. Please note that for the purpose of feeding the state into the deep Q network we must first normalize by applying a soft max function to each column of the matrix s_t, then we flatten the matrix.

4.2.3 Loss Function

We've defined the V-DQN loss function in equation 7 but in practice the loss is implemented using various approximations. First the expectation in the Q Learning loss function is approximated using experience replay by sampling a mini-batch of experiences from the replay buffer. Given a replay buffer D that contains experiences (s, a, r, s'), a mini-batch of N experiences $B = \{(s_i, a_i, r_i, s'_i)\}_{i=1}^N$ is sampled uniformly at random from D. Using this strategy we can approximate the loss function in equation 7 (V-DQN Loss) as follows. The same strategy can be applied to the loss functions for T-DQN and D-DQN.

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \left[\left(r_i + \gamma \max_{a'} Q(s'_i, a'; \theta) - Q(s_i, a_i; \theta) \right)^2 \right] \tag{40}$$

The next adjustment practitioners use is the integration of the the Huber loss. This is designed to make our loss function quadratic for small values of the error and linear for large values of the error. The parameter δ effectively determines the sensitivity of the loss function to outliers. Previously we defined TD Error in equation 5, which for the sake of integrating the Huber loss we'll equate to δ. This final improvement yields the following loss function.

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} L(\delta) \tag{41}$$

where

$$\delta = \left(r + \gamma \max_{a'} Q(s', a') \right) - Q(s, a) \tag{42}$$

$$L(\delta) = \begin{cases} \frac{1}{2}\delta^2 & \text{for } |\delta| \leq 1, \\ |\delta| - \frac{1}{2} & \text{otherwise.} \end{cases} \tag{43}$$

4.2.4 Optimizer

Lastly, we use the Adam optimizer in all frameworks. This is an adaptive optimization algorithm used in training our neural networks. It incorporates momentum to accelerate convergence in relevant directions and introduces bias correction to improve parameter update accuracy. By computing adaptive moments of gradients, Adam scales updates effectively and ensures numerical stability with a small constant in the denominator. These combined features make Adam a robust and efficient choice for optimizing our neural networks.

4.2.5 Hyper parameters

We use the following hyper parameters which we keep consistent across all implementations and frameworks: our discount rate $\gamma = 0.95$, the parameters for our epsilon-greedy policy $\varepsilon_0 = 1$, $\varepsilon_{\text{decay}} = 0.995$, $\varepsilon_{\text{min}} = 0.01$, the window size $N = 10$, and the number of training episodes per framework $E = 10$.

5 Experimental Design & Results

For the purpose of this paper we use daily Google stock data, including low,high,close,and adjusted close prices. Specifically we use Jan 1, 2018 - Jan 2019 data to train the model and Jan 1, 2019 - Jan 2020 data to validate the model we also allow the agent to retrieve the market indicators listed in section 3.1. to incorporate into its state space defined in section 4.1.1. All computations regarding the market indicators are defined in section 3.1.

We then test the following variations to find the best performing trading agent. Bellow are our results expressed in terms of percentage growth over the time. However, its important that we explain how we obtain the percent growth value given that we don't incorporate the idea of starting balance into our model. So in order to calculate percent growth we assume that the initial balance B_0 is the minimum balance across the episode. Intuitively if we assume $B_0 = |\min t B_t|$ we functionally enforce that the agent never goes bankrupt. Using this "implied initial balance" we calculate the percent change using $\frac{B_T - B_0}{B_0}$.

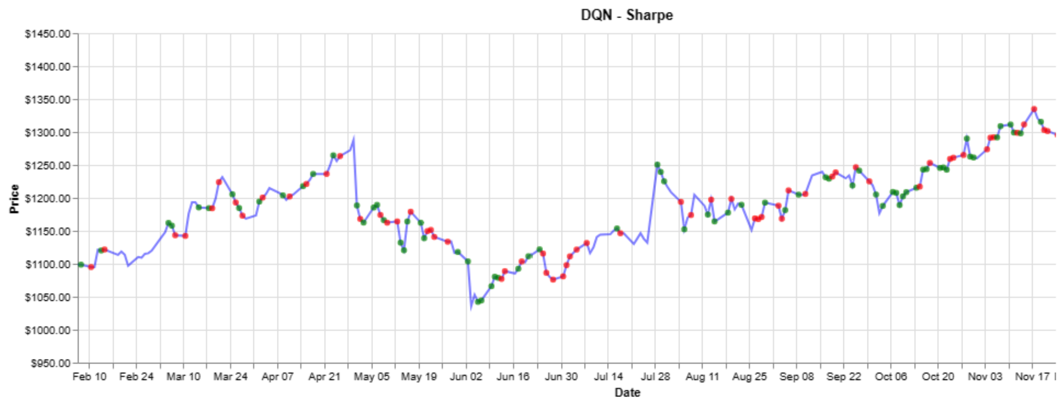
Metric	V-DQN	T-DQN	D-DQN
Sharpe	8.14%	13.05%	11.21%
Sortino	0.62%	-0.087%	2.61%
Treynor	6.68%	6.15%	10.26

Table 1: Performance Metrics of Different DQN Variants

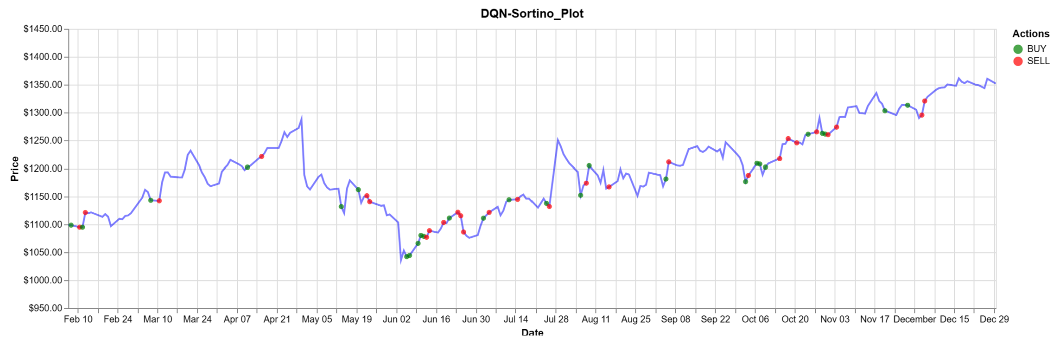
We note that the percentage growth in Google's stock from 2019-2020, was 27.75%, so each trading agent is still under performing the market. From table 1, we observe that the Sharpe ratio is better suited to guide the agent towards favorable trading decisions. We speculate that this is the case because the Sharpe ratio considers both systematic and unsystematic risk along with downside and upside risk through its use of the standard deviation of portfolio returns in the denominator. This makes it a comprehensive measure of risk-adjusted return, suitable for evaluating strategies that might be exposed to a wide range of market risks. On the other hand the Treynor ratio uses beta in the denominator and focuses on the systematic risk of the stock itself relative to the overall market. Sortino's ratio is also more limited in scope then the Sharpe ratio since it focuses on downside risk. We also invite the readers attention to the plots bellow, where one can observe an interesting feature in the D-DQN plots. We observe that the returns for the D-DQN algorithms outperform the other frameworks and that the buy and sell orders are more sparse then V-DQN. We hypothesize that this happens because Double DQN is able to effectively stabilize the TD target which Shields the agent's purchase behavior from idiosyncratic price shocks that may cause sporadic traeding behavior.

5.1 V - DQN Plots

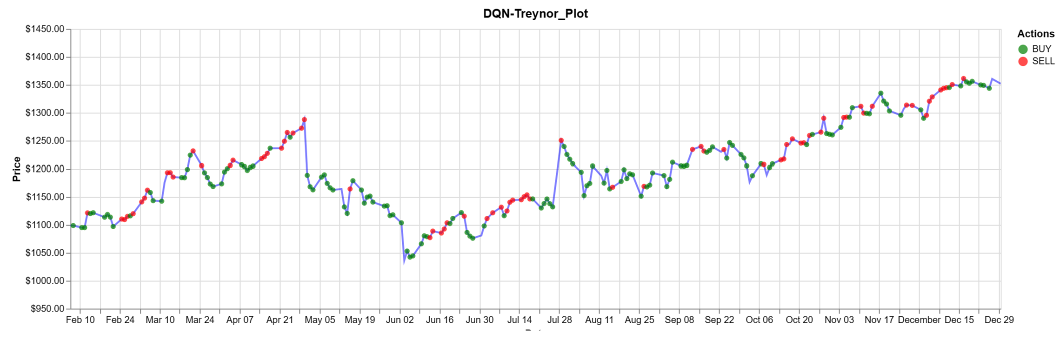
5.1.1 Sharpe



5.1.2 Sortino Plots

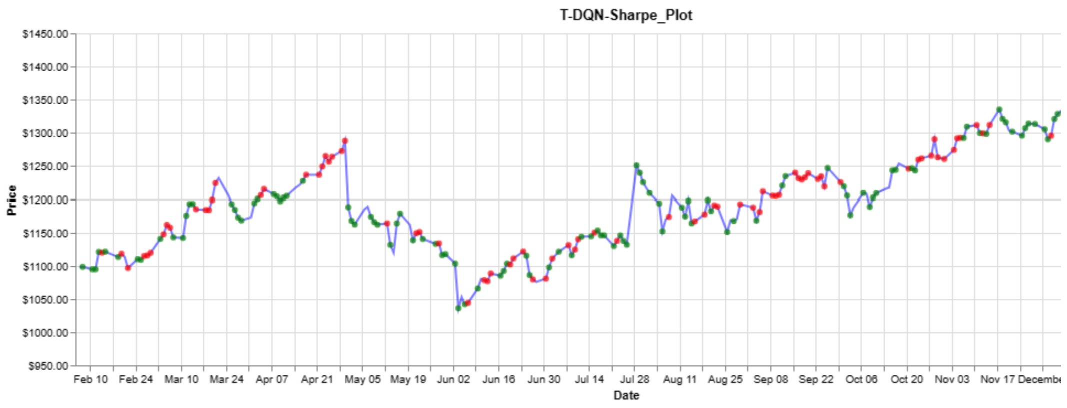


5.1.3 Treynor Plots

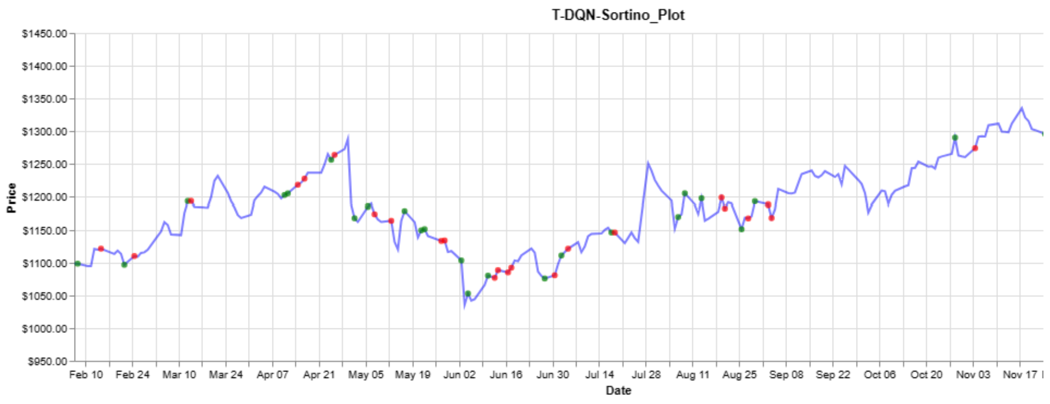


5.2 T - DQN Plots

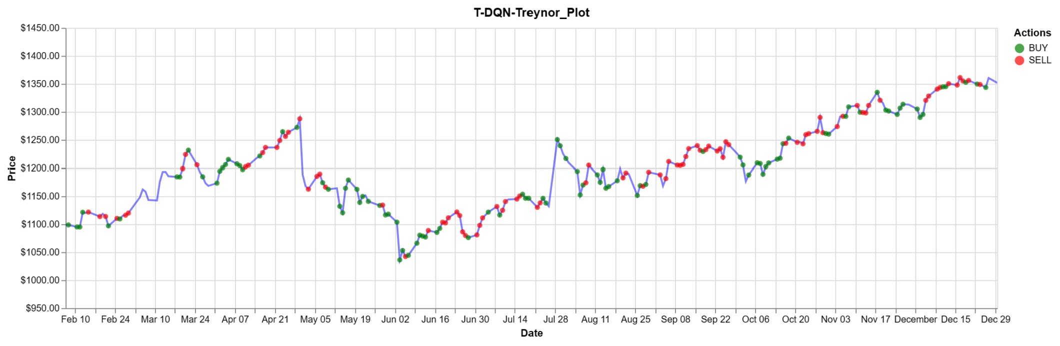
5.2.1 Sharpe



5.2.2 Sortino

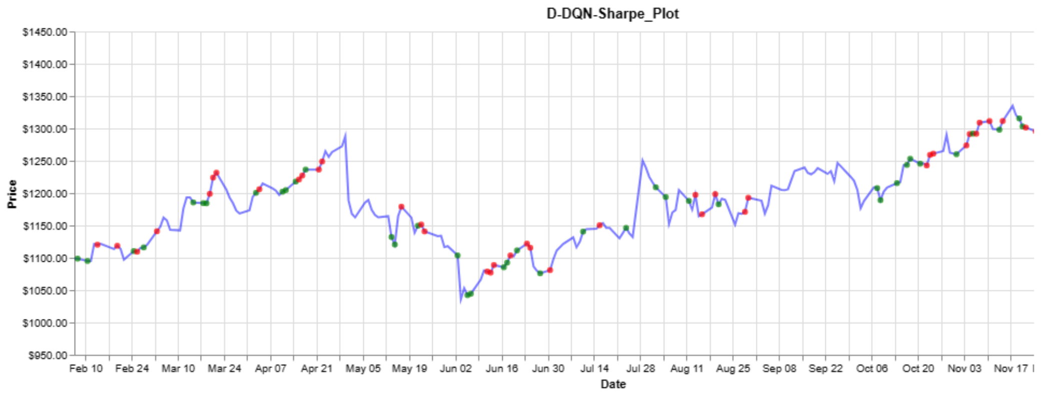


5.2.3 Treynor

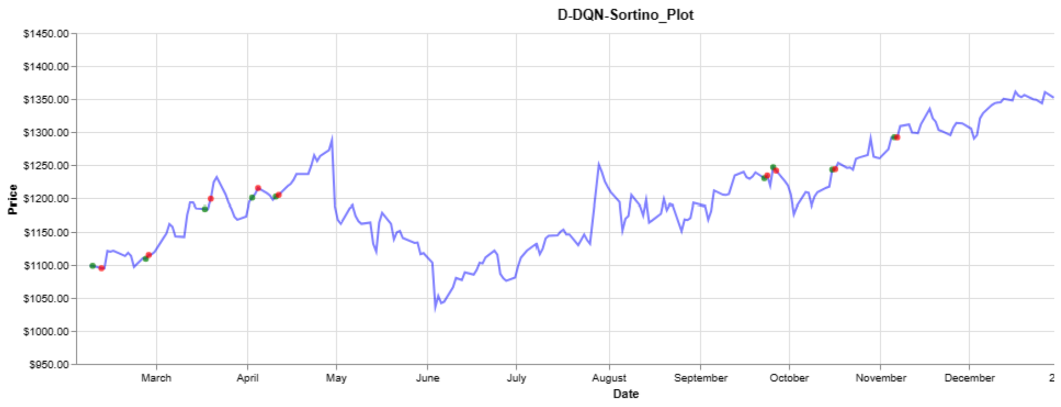


5.3 V - DQN

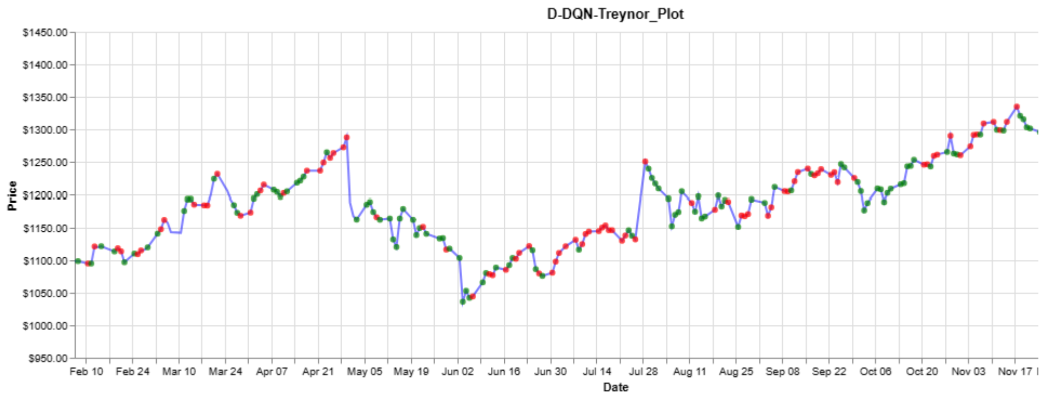
5.3.1 Sharpe



5.3.2 Sortino



5.3.3 Treynor



6 Future Improvements

While our agent yields positive returns, we identify numerous areas of possible further exploration to enable it to outperform that market. The first is the lack of training data. By presenting our agent with daily closing prices we limit the amount of actions it can take to one per day. This means that feedback is only as frequent as the agent's trade. Possible improvements to this could be training on a longer price history, or to provide more frequent pricing information. Next, we recognize that an upgraded agent could have the ability to place a trade more than one unit of stock per action. Specifically, we could allow the agent to trade up to k units of stock per day, where k would be another hyper-parameter to adjust. To accomplish this, the Deep Learning framework would need to be deeper in the number of layers and nodes, accounting for the higher dimensional action space. Further, we acknowledge that different neural network frameworks may present various advantages. For example, implementing a Long-Short Term Memory (LSTM) Recurrent Neural Network (RNN) could incorporate the concept of price memory without needing to manually maintain a window of states. Along the same lines, a Concurrent Neural Network (CNN) could be incorporated to understand our state space matrix and produce more informed evaluations of our action-state value functions. Additionally, incorporating more episodes in our training schemes could be greatly influential in yielding stronger results, but this would require more computing power and runtime. Lastly, limitations on when the agent can buy and sell units based on a notion of current balance vs. current holdings would make this project more seamlessly transferable to real world trading applications. In particular, introducing restrictions such as only being able to buy as many units as allowed by current balance, or having to sell units if balance is too low would more accurately simulate how humans trade.

References

1. Briola, Antonio, et al. Deep Reinforcement Learning for Active High Frequency Trading, 19 Aug. 2023, `arxiv.org/abs/2101.07107`.

2. Brunton, Steven Lee, and José Nathan Kutz. Data-Driven Science and Engineering: Machine Learning, Dynamical Systems and Control. Cambridge University Press, 2021.

3. Dai, Zhenwen, et al. In-Context Exploration-Exploitation for Reinforcement Learning, 11 Mar. 2024, `arxiv.org/abs/2403.06826`.

4. De Asis, Kristopher, et al. Multi-Step Reinforcement Learning: A Unifying Algorithm, 11 June 2018, `arxiv.org/abs/1703.01327`.

5. Hill, R. Carter, et al. Principles of Econometrics. Wiley, 2018.

6. Luo, Yunfei, and Zhangqi Duan. Agent Performing Autonomous Stock Trading under Good and Bad Situations, 6 June 2023, `arxiv.org/abs/2306.03985`.

7. Mnih, Volodymyr, et al. Playing Atari with Deep Reinforcement Learning, 19 Dec. 2013, `arxiv.org/abs/1312.5602`.

8. Pricope, Tidor-Vlad. Deep Reinforcement Learning in Quantitative Algorithmic Trading: A Review, 31 May 2021,
`arxiv.org/abs/2106.00123`.

9. Ravichandiran, Sudharsan. Deep Reinforcement Learning with Python: Master Classic RL, Deep RL, Distributional RL, Inverse RL, and More with Openai Gym and Tensorflow. Packt, 2020.

10. Singh, Prabhsimran. "PSKRUNNER14/Trading-BOT: Stock Trading Bot Using Deep Q Learning." GitHub, 31 Dec. 2020, `github.com/pskrunner14/trading-bot/`.

11. Sutton, Richard S., and Andrew Barto. Reinforcement Learning: An Introduction. The MIT Press, 2020.

Appendices

A Additional Risk Metrics

A.1 Value at Risk (VaR)

Value at Risk (VaR) quantifies the maximum expected loss over a specified time period within a given confidence interval, assuming normal market conditions. It is widely used to assess the risk of a portfolio.

$$\begin{aligned}\mathrm{VaR}_{\alpha}(t) = &-\inf\{x \\ &: F(x) \\ &> 1-\alpha\}\end{aligned} \tag{44}$$

A.2 Expected Shortfall (ES) or Conditional VaR (CVaR)

Expected Shortfall (ES), or Conditional VaR (CVaR), measures the expected loss on days when there is a VaR breach. It provides a more comprehensive view of tail risk than VaR by averaging losses in the tail beyond the VaR threshold.

$$\mathrm{ES}_{\alpha} = \frac{1}{1-\alpha}\int_{\alpha}^{1} \mathrm{VaR}_{u}(L)du \tag{45}$$

B Additional Q Learning Schema

B.1 SARSA

SARSA is also a model-free RL scheme, but unlike classical Q Learning is on-policy. The advantage of SARSA is that it unlocks the full power of temporal difference learning by allowing the user to design an agent that can look n steps into the future when calculating the TD(n) error. In this setting we use all equations and definition listed above except for the update equation, which is now defined as follows.

$$Q^{\text{new}}(s_k, a_k) = Q^{\text{old}}(s_k, a) + \alpha(R_{\Sigma}^{(n)} - Q^{\text{old}}(s_k, a_k)) \tag{46}$$

$$R_{\Sigma}^{(n)} = (\sum_{j=1}^{n} \gamma^{j} r_{k+j}) + \gamma^{n+1} Q^{\text{old}}(s_{k+1}, a) \tag{47}$$

Its very important to note that in order to calculate $r_k, ..., r_{k+n}$ we must define which actions the agent will take at every future time step. Without that we can't know what the future rewards will be. So in order to solve this problem the SARSA model forces the agent to choose the optimal on-policy action at each projected time step.

There is a variation of TD learning that was developed by Sutton in 2015, where instead of looking at any n-step temporal difference, we take a exponential weighted average of all possible n-step temporal differences. This idea is known as TD-λ learning and can be formalized by defining the following update scheme.

$$R_{\Sigma}^{\lambda} = (1-\lambda)(\sum_{k=1}^{\infty} \lambda^{n-1} R_{\Sigma}^{(n)}) \tag{48}$$

$$Q^{\text{new}}(s_k, a_k) = Q^{\text{old}}(s_k, a) + \alpha(R_{\Sigma}^{\lambda} - Q^{\text{old}}(s_k, a_k)) \tag{49}$$

Recasting SARSA into a training scheme that incorporates neural networks would result in a loss function of the following form, where a user could implement n-step temporal difference learning or the TD(λ) framework described above.

$$\mathcal{L} = \mathbb{E}[(R_{\Sigma}^{(n)} - Q(s_k, a_k))^2] \tag{50}$$

B.2 $Q(\sigma)$ Learning

In 2018 Sutton and Barto first formulated the $Q(\sigma)$ method which was meant to unify the two fundamental approaches discussed above; Q Learning and SARSA. It introduces a parameter σ, which varies on a scale from 0 to 1, allowing the algorithm to interpolate between these two methods. This allows for $Q(\sigma)$ to adaptively balance between the exploration of new strategies (via Q Learning's off-policy nature) and the exploitation of current knowledge (via SARSA's on-policy nature), potentially leading to more efficient learning in complex environments.

$Q(\sigma)$ operates by dynamically adjusting the parameter σ for each time step of each episode, which determines the mix of on-policy (SARSA) and off-policy (Q Learning) learning. The updates to the Q values in $Q(\sigma)$ are based on a combination of the expected value (like in Q Learning) and the actual reward received plus the value of the next n state-action pair (like SARSA), with the balance between these two determined by σ. We can formalize $Q(\sigma)$'s update rule as

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma(\sigma_{t+1} Q(S_{t+1}, A_{t+1}) + (1 - \sigma_{t+1}) V_\pi(St + 1)) - Q(S_t, A_t)] \tag{51}$$

The natural question to ask is how σ_t is chosen at every time step. Asis, Hernandez-Garcia, Holland, and Sutton proposed in 2018 that in most settings σ should be gradually decreased over the course of the episode in a manner similar to simulated annealing.

Furthermore, just like SARSA, we can expand this formulation to include multiple "look-forward" steps. The variation of $Q(\sigma)$ that incorporate TD(n) learning.

When recasting this scheme to a deep learning problem, we can apply identical reasoning as for SARSA to generating a loss function of $Q(\sigma)$ Learning.

$$\mathcal{L} = \mathbb{E}[R_{t+1} + \gamma(\sigma_{t+1} Q(S_{t+1}, A_{t+1}) + (1 - \sigma_{t+1}) V_\pi(St + 1)) - Q(S_t, A_t] \tag{52}$$