An application of reinforcement learning to the trading of options strategies.

Adam Keys
Department of Computer Science
University College London

Abstract

Through application of asynchronous, model-free, reinforcement learning techniques to options data, this work seeks to understand the viability of deep reinforcement learning in order scheduling, portfolio management, and trading efficacy. Namely, both deep Sarsa(λ) and deep Q-learning methods are applied to market data for Apple stock spanning March 2022 to March 2024. With an ability to trade at-the-money puts and calls, alongside 2 other puts and calls at lower and higher prices respectively, the learning agent will choose from a novel action space of linear combinations of different options (e.g. bull spread, straddle, butterfly, etc). Following the training of the models within a bespoke trading environment, the results, although mixed, do exhibit some preliminary evidence supporting the use of reinforcement learning in this context.

1 Relevant Background

1.1 Introductory Reinforcement Learning

Reinforcement learning (RL), at its very core, revolves around the *interplay* of an autonomous agent and an environment. Originating from the medium through which young children learn, the general premise involves some agent observing their current state s_t, a sufficient statistic for the environment, and then taking an action a_t from a predefined action space based on a policy π. This action is then enacted within the environment, which returns the quality of the action, quantified by the reward r_{t+1}, along with the new state the agent ended up at (s_{t+1}) as a result of taking the action. Thereafter, the agent takes a following action according their policy and the cycle continues ad infinitum or until a terminal condition is met. RL uses this sequence of events and the resulting data to determine the optimal policy π_*: how the agent should act in a given state in order to maximise the expected future reward.

The policy $\pi(a|s)$ represents the mapping from state to action, telling the agent which action to take at every possible state in the state space. Representing the total amount amount of reward an agent will get in expectation from a state, the value function $v_\pi(s)$ is defined given a policy. The value function is used to determine the total reward an agent can earn when following a policy and can thus be used to assess the its overall quality. Given the transitive nature of the value function, one can define a partial ordering over policies ($\pi \geq \pi'$ iff $v_\pi(s) \geq v_{\pi'}(s), \forall s$). From here, given any Markov decision process, the existence of an optimal policy, one at least as good as any other, follows as per Sutton & Barto (2018). Thus the theoretical groundwork is laid for arguably the most important idea in RL: the Bellman equation (Bellman 1958). This equation recursively relates the value of being in one state $v_\pi(s)$ to the immediate reward gained from stepping out of that state r_{t+1}, plus the discounted value of being in the next state. It is a necessary condition for optimality.

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ...|S_t = s] && (1) \\ &= \mathbb{E}_\pi[r_{t+1} + \gamma(r_{t+2} + \gamma r_{t+3} + ...)|S_t = s] && (2) \\ &= \mathbb{E}_\pi[r_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s] && (3) \end{aligned}$$

1.1.1 Model-Based Control

Control methods within dynamic programming seek to enforce the Bellman equation through iterative techniques, the foremost of which are *value iteration* and *policy iteration*. The former operates entirely within value space and utilises the following iterative update:

$$v_k(s) = \max_{a \in \mathcal{A}} \left(\mathcal{R}_a^s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_{k-1}(s') \right),$$

where $\mathcal{R}_a^s$ is the expected reward from taking action a in state s, $\mathcal{P}_{ss'}^a$ the probability of reaching state s' from s whilst taking action a, $\gamma \in (0, 1]$ is a discount factor, and $\mathcal{A}$ and $\mathcal{S}$ are the action and state spaces respectively.

The intuition here is that states adjacent to those with high rewards will also take on higher values through the max operator. As iterations progress, these rewards are propagated back through the state space such that the value function implicitly defines a path for an agent to follow. The optimal policy, which follows once the value function has converged, derives from acting *greedily* with respect to the value function. This means that an agent will choose the actions which maximise their expected reward as per the value function.

Policy iteration is another popular model-based methodology borrowed from dynamic programming. Unlike value iteration, this method explicitly defines a (typically random) policy which it then seeks to optimise. Policy iteration can be decomposed into two processes: evaluating a policy using iteration, then acting greedily with respect to the value function to improve the policy. The former involves using the following iterative update to learn a value function for a specified policy.

$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right)$$

Once the value function has converged to within a sufficient threshold, an RL agent will thereafter redefine their policy. This redefinition involves finding the actions which maximise the expected reward according to the newly evaluated value function. However, given the policy has now evolved, it also now corresponds to a completely different value function. As such it can be reevaluated using iterative policy evaluation and then once again acted greedily towards to define another policy. This process repeats until a convergence criterion is met on the policy, at which point the optimal policy has been found. Convergence is guaranteed for discrete environments where the model is known (Santos & Rust 2004).

1.1.2 Model-Free Control

A glaring issue with the aforementioned methodologies is that in the vast majority of use cases, there is no available model of the environment. This is especially pertinent in the case of financial problems, where dynamical systems are highly complex and often only partially observed. *Model-free RL* methodologies address this shortcoming through leveraging the environment itself to garner information about its inner workings. Specifically, an agent will *sample* the environment through interaction, taking action a from state s, receiving reward r and ending up at state s' . The resulting experience, often denoted with the tuple $\langle s, a, r, s' \rangle$, can be used to find the optimal policy.

Without knowledge of the state transition probabilities, model-free methods are unable to calculate expected rewards for given actions. As such, they must operate directly in action space, taking actions in an environment and estimating the corresponding rewards through sampling. The Sarsa control algorithm (Rummery & Niranjan 1994), deriving from the experience tuple $\langle s, a, r, s', a' \rangle$, adapts the policy iteration framework to the model-free setting using *temporal-difference (TD) learning*. Specifically, the algorithm updates state-action $Q(s, a)$ values, representing the expected reward from taking an action a from state s, with the following iterative update.

$$Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma Q(s', a') - Q(s, a)),$$

where α is a learning rate parameter, and a' is the action determined by the policy at state s'. This equation can be thought of as an incremental mean update, whereby a Q-value is updated in the direction of the difference between the immediate reward earned, plus the amount of future reward expected from state s' , encapsulated by the Q-value at that state, less the prior estimate for the value of this state-action pairing (see derivation in Appendix A.1).

Once again, a random policy π is initially defined and then evaluated with this iterative equation. Once the Q-values converge, a marginally more effective policy evolves from acting greedily with respect to the Q-values. This newly defined policy corresponds to a new set of Q-values and thus the process can be repeated until an optimal policy is converged upon. Notably, however, the method of policy improvement in the SARSA algorithm, known as ε-greedy policy improvement, differs slightly. This scheme attempts to balance an important paradigm in RL: the exploitation/exploration tradeoff (see Appendix A.3).

Q-Learning (Watkins & Dayan 1992) is another model-free control algorithm very similar to SARSA, differing in that it is an off-policy method. As exhibited in the iterative update below, Q-learning alters the TD target through the introduction of the max operator, thus leveraging the Bellman optimality equation. In particular, Q-learning updates Q-values in the direction of the immediate reward, plus the discounted maximum over the Q-values at the next state. Therefore, two policies are in operation simultaneously: the ε-greedy *exploration policy*, which determines the actions taken by the agent, and the *target policy* found in the TD target used in bootstrapping towards the true Q-values. This dichotomy enables Q-learning to learn from data generated by a policy which differs from the one which it is seeking to optimise, enabling more efficient data usage whilst learning.

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_{a' \in \mathcal{A}} Q(s',a') - Q(s,a)),$$

1.1.3 Function Approximation

All of the aforementioned techniques can be extended into continuous action and state spaces through the use of function approximators which break Bellman's curse of dimensionality (Bellman 1966). In essence, these are functions which enable the generalisation from seen states to those which are unseen; one can imagine, for instance, a robot in a maze with a position 1mm away from another being likely to have the same expected reward in each. Function approximators utilise sampled state, action, and return data $(\{s_1, G_1\}, ..., \{s_n, G_n\})$ to fit a value or action-value function to data. This data, in effect, represents conventional training data for the approximator, upon which a loss criterion can be defined and an optimal parameter vector obtained using a gradient descent algorithm. Alternatively, linear models with analytic solutions can also be used.

1.2 Literature

There exists a relatively large amount of literature exploring the application of RL to finance. Liu et al. (2022), for instance, have developed a library named FinRL which enables the easy deployment of RL models to a vast array of different data types, sources, and frequencies. Their open-sourced contribution builds upon the convention set by OpenAI's Gym environments (Brockman et al. 2016), extending this framework to allow for training on large sets of financial data. FinRL utilises a layered structure consisting of the data layer, environment layer, and agent layer, enabling users to extend the system with their own functions and update the preexisting algorithms.

Although there are myriad other examples of research applying RL to finance, there are relatively few papers assessing the suitability of RL in the context of options trading. Most of the few examples that exist in the public domain relate RL to the hedging of directional risk through trading derivative securities, otherwise known as delta-hedging. The literature on this topic include Du et al. (2020), Ritter & Kolm (2018), Halperin (2019), and Bühler et al. (2018). Both Du et al. (2020) and Ritter & Kolm (2018) focus their efforts on RL with an objective function equal to the expected cost of hedging plus the variance of the said cost, ultimately using a single Q-function underpinned by a series of reasonable assumptions. Halperin (2019) constructs a risk-adjusted MDP for a discretised Black-Scholes Model (Black & Scholes 1973), with the option price and optimal hedge as arguments to an optimal Q-function. Pricing in the paper derives from Markowitz portfolio theory (see Markowitz 1952) wherein a learner attempts to dynamically optimise risk-adjusted returns for a replicating portfolio. The paper shows that their model converges to the true continuous-time Black-Scholes price and hedge ratio.

Du et al. (2020) explore the viability of deep reinforcement learning in replicating options across an array of strike prices and subject to discrete lotting and non-linear transaction costs. The authors explore the use of RL in *automatic hedging* (Ritter & Kolm 2018), whereby deep RL agents learn to hedge arbitrary derivative portfolios. Specifically, Du et al. (2020) define an action space consisting of trading integer units of the underlying and a reward function which incentivises reductions in the variance of the portfolio's P&L. Compared with traditional delta hedging, the authors' RL agent is shown to be at least as effective and oftentimes better. This is a significant result given the importance of delta-hedging in risk management and the pricing of options in the global economy.

Cao et al. (2020) similarly investigate the usefullness of RL in the hedging of portfolios. They illustrate the difference between traditional delta hedging and *optimal hedging*, wherein an agent attempts to minimise an objective function containing the mean cost of hedging and its standard deviation. Considering the cases of an asset following a geometric Brownian motion and a stochastic volatility process, the paper introduces a number of novel approaches. Firstly, different Q-functions are utilised for monitoring both the expected cost and the expected value of the cost squared (the variance) across different state-action combinations. Through separating the functions in this manner, a broader range of objective functions can be employed. The authors additionally compare the efficacy of two different approaches for reward creation: the P&L approach where the hedged position is revalued at every step, and the accounting approach where the cash inflows/outflows are utilised.

Research relating RL to options trading in the context of seeking alpha is understandably sparse; any paper evidencing a profitable trading system is better kept in the private domain for obvious reasons. That said, Wen et al. (2021) do indeed explore the application of RL to options trading with the purpose of generating P&L. The paper utilises open-high-close-low-volume (OHCLV) market data for the underlying of an option as the state space, with actions including doing nothing, purchasing a call, and selling a call. Applied to this "OTRL" framework are a series of RL algorithms including deep Q-learning, soft actor critic, and proximal policy optimisation. These algorithms were trained using Shanghai 50 ETF, and Shanghai & Shenzhen 300 ETF trading data across a multitude time-frames. The authors first train their RL models with the purpose of determining optimal stop-loss thresholds, which are then used in the final options trading scheme. Their results indicate that RL is indeed a viable method for effective options trading, outperforming the buy and hold null model by a significant margin. Interestingly, proximal policy optimisation, a policy gradient based method which seeks to parameterise and then optimise the policy directly, was the most effective model.

The following work delves much deeper than this treatment, and appears to be unique in the realm of published literature. Nonetheless, RL's suitability for this domain is clear, stemming from several key factors. Firstly, an abundance of financial data allows for RL agents to learn and adapt strategies based on historical trends. The natural quantification of reward in terms of portfolio P&L, alongside the episodic nature of options trading through expiry dates, also lend themselves to RL's approach. Finally, the complexity in manually developing deterministic trading algorithms underscores RL's value in learning strategies autonomously.

2 Data

In this application the selected asset is Apple (AAPL), the well-known American technology firm. The rationale behind this selection is two-fold: firstly, the stock is well traded and liquid, meaning there is likely to be less sparsity in options data over 30 minute intervals. The second reason relates to the scale of Apple's stock price, namely that it trades at c$180 across the period under consideration. Most exchanges allow options trading at strike prices in intervals of 1 point for assets trading under $200 (e.g. CBOE 2024). As such, although there is significant volume at each dollar interval of strike, the with more of the liquidity typically sitting at intervals of $5. Hence utilising a lower-valued asset such as Apple ensures that significant transaction data will be available at intervals of $5 about the spot price. This is opposed to, for instance, the much more expensive NASDAQ Index which has a far greater number of significant levels to trade at, leading to greater sparsity at each level.

OHLCV market data for Apple's stock was downloaded from Polygon (Polygon 2024), between March 2022 and March 2024. Given the intensity of the data-demands for (deep) neural networks and RL models in general, a lower time frame was chosen in order to maximise the number of data points per day. That said, options pricing data c15% about the current spot can be sparsely traded at any given time, leading to more frequent gaps in activity at the lower time frames. In order to balance these two demands, the 30 minute time frame was chosen, leading to c16,000 rows of data and c80,000 data points.

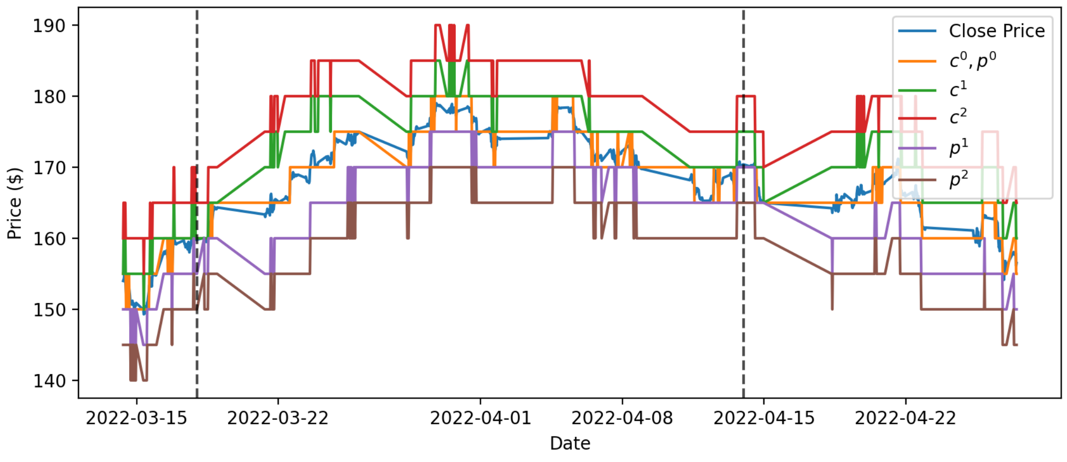

Figure 1: A visualisation of different strike prices for a trading period in 2022. The vertical dashed lines represent expiry dates.

Table 1: Strike Price Methodology

Strike	Formula
c_t^2	$c_t^1 + 5$
c_t^1	$\text{ceil}(\text{close}_t + 2\sigma_{W_t}\left(1 - \frac{t}{T}\right), 5)$
$c_t^0 = p_t^0$	$\text{round}(\text{close}_t, 5)$
p_t^1	$\text{floor}(\text{close}_t - 2\sigma_{W_t}\left(1 - \frac{t}{T}\right), 5)$
p_t^2	$p_t^1 - 5$

The RL agent will have access to an at-the-money call and put, and options at two further levels either below or above the spot rate. To calculate these levels, outlined in Table 1, rounding to the nearest $5 was used alongside a linearly decreasing multiple of the weekly volatility as the option tends to expiration. Naturally as the third Friday of the month nears, when monthly options typically expire, the transition density of the stock's expiration price reduces in variance. Thus, the strikes at which the agent should have available should correspondingly close around the spot price to reflect this.

Once all unique combinations of strike price, option type, and expiry are calculated, the corresponding OHLCV data was downloaded from Polygon from the first instance until expiry. Crucially, if an option could be traded at any given point in time, the pricing data must be available in full until expiry so the environment can always derive accurate portfolio values. Once the data was backfilled to close any gaps, and its availability checked upon the strike space, it was split into "episodes". Noting the distinction to *training* episodes, this segmentation of data involved splitting the stock and strike data into sets which encapsulated the length of time between one option expiring, and the next expiring four weeks later. Hence there is only one option expiry that can be traded at any given time.

3 Methodology

3.1 Trading Environment

RL models learn from feedback, trail and error; an agent takes an action and observes the resultant effect both in terms of the state space and, more importantly, the reward. As such it is necessary for an agent to operate within an *environment*, a system whereby an agent can efficiently take actions and observe their impact. In the context of options trading, such an environment is necessarily quite complex: there are 12 strategies traded long and short, each comprised of combinations of call and put options, often at differing strike prices. For the sake of reduced complexity, calendar spreads, wherein options with different expiries are traded, will not be utilised.

Much of RL-based research centres around prebuilt environments in OpenAI's Gym package (Brockman et al. 2016). Given the complexity of the input data and action space in this case, however, a custom-environment tailored for options trading was build from the ground up. The environment is instantiated as a class with a randomly selected training period linked to an expiry date. Then, the corresponding data for the underlying and options is cached, in addition to a number of state and tracking variables. An important attribute defined is the *type map*, which specifies the number and types of options in a strategy, along with the direction (short/long). For instance, the entry for the "butterfly" strategy consists of four calls with direction [1, -1, -1, 1]. This data is critical for the pricing computation and logic. Arguably the most important attribute defined is *time*, a counter which keeps track of where the system is in the underlying stock data as the agent steps through the environment.

The *step* method enables the agent to take actions and progress through the environment. Taking as input an action ("risk_rev", "straddle", "call", etc), a direction (1 or -1), and the strike prices at which to trade (e.g. "c_1", "p_2"). Thus to go long a straddle in this system, the agent would step with an action equal to "straddle", a direction equal to 1, and (predefined) strike prices "p_0", "c_0" . Within the class, this data is then passed to a function which creates the position and adds it to the portfolio. Notably, the system will check for preexisting instances of strategies with the opposite direction to the one to be traded. If found, the strategy will not be added to the portfolio, and instead the preexisting instance will be traded off the books in the current time step. Thus it is impossible to be both long and short a given strategy simultaneously. Although, of course, there are certain merits to trading a strategy both long and short, and initially only matches with exactly the same strikes were removed; there were too many trades taking place and thus the system was altered.

Every new position added or removed from the portfolio inventory is passed to the *trade* function. This function alters the stock of cash in the portfolio, enabling strategies to be traded both long *and* short. As with a real portfolio, the environment has an initial stock of cash ($1000) which is incremented when an option is sold short, and decremented when an asset is bought. Hence, when an asset is traded, the portfolio's value, the total amount of cash held plus the value of its assets, does not change within the current time step; the change in cash is exactly the reverse of the change in the book value of the assets. Thereafter, when the assets appreciate/depreciate, the value of the portfolio can increase and decrease accordingly. When assets are sold, the gain/loss is realised in the cash[1].

Along with managing the cash stock, the portfolio itself must be valued in full through asset valuation. The system does not calculate the value of specific strategies ex-ante, with this pricing logic handled as and when the agent steps through the environment. Specifically, after a position has been added to the inventory and the cash altered accordingly, the portfolio itself is valued through the *portfolio_eval* method. This method iterates over each strategy, valuing it as the sum of the value of its constituent call and put options. The different P&L variables are then calculated and the total value at that time point is returned.

Finally, the environment calls the *calc_analytics* method, responsible for calculating the state variables and reward. The method retrieves a series of precalculated, stock price dependent measures and calculates the remaining state variables dependent upon the agent's prior actions. Once these measures are calculated, they are returned to the agent along with the immediate reward. From this point, the agent observes the new set of variables and makes their next decision, once again stepping through the environment.

3.2 Duelling Deep Q-Learning

Deep Q-learning is arguably the most popular form of RL at the moment, and for good reason. This off-policy, asynchronous, methodology combines greater data efficiency with the impressive ability of deep neural networks to fit to data. Deep Q-learning extends the discrete Q-learning framework outlined earlier into a continuous state space using function approximation. Rather than using a tabular format to relate state-action pairs to Q-values, a (deep) neural network is used in its place, taking as input the states, passing this data through the network, and outputting Q-values for each action within the action space. As the network fits to the data over time, the agent's behavioural policy evolves according to the typical ε-greedy framework, optimising the exploitation/exploration tradeoff across episodes. Therefore, as the agent learns more about the state space and which actions typically

[1]Note that the P&L will already reflect this change

yield the most reward, a sophisticated and oftentimes successful policy can be learned. There are a number of different schemes which can be used to decrement epsilon as episodes progress. In this instance, a final value for epsilon is set to 0.1, and the geometric decay rate which will achieve this target in the total number of episodes is applied to epsilon recursively.

RL agents that learn online update their parameters incrementally after observing a stream of experience. In the most simple case, this experience is thereafter discarded. The issue with this system in the deep Q case are two-fold. Firstly, the adjacent states produce strongly correlated state/reward experience which violates the assumption of independent and identically distributed input data for stochastic-gradient descent algorithms frequently used in training. Moreover, the instant forgetting of potentially valuable and rare experiences is highly data-inefficient; these experiences can and should be used in multiple updates. Lin (1992) introduces *experience replay* to remedy these issues. Experience replay operates as a cache $\mathcal{D}_T = \{e_1, ..., e_T\}$ of experience $e_t = (s_t, a_t, r_t, s_{t+1})$ which is updated at every step, and then uniformly sampled in order to generate training data. Using this structure decorrelates the data and, through restricting its maximum size, ensures that *recent* data can be used multiple times, improving data efficiency (see Mnih et al. 2013). The use of experience replay usually reduces the total experience required for learning and, in doing so, exchanges cheap computation and some memory for reduced environment experience, something far more costly (Schaul et al. 2016). To implement experience replay in this case, a custom data structure was created built upon the standard deque class, with a maximum size of 2000 and a batch size of 100.

Naturally, as experience builds up, the neural network function approximator must be trained and fit to the data (Q_*) through minimising some objective function $J(\mathbf{w})$ which takes parameter vector $\mathbf{w}$ as input. As such an objective function and an algorithm which updates the parameters of the network must be specified. For the former, as below, the mean squared error between the sampled estimate for the return Q_* and the neural network's prediction $\hat{Q}$ is chosen.

$$\begin{aligned}
J(\mathbf{w}) &= \mathbb{E}_\pi[(Q_* - \hat{Q}(s, a; \mathbf{w}))^2], \\
Q_* &:= r + \gamma \max_{a'} \hat{Q}(s', a'; \mathbf{w}^-) \\
\Delta\mathbf{w} &= -\frac{1}{2}\alpha\nabla J(\mathbf{w}) \\
&= \alpha\mathbb{E}_\pi[(Q_* - \hat{Q}(s, a; \mathbf{w}))\nabla_{\mathbf{w}}\hat{Q}(s, a; \mathbf{w})]
\end{aligned}$$

For the stochastic gradient descent algorithm, which samples the expectation shown above, the choice is *Adam* (Adaptive Moment Estimation; Kingma & Ba 2017). The innovation this algorithm brings is to have learning rates for each parameter evolve dynamically dependent upon the data encountered, allowing for more efficient step sizes. This process is akin to a car moving along different terrains, choosing to slow down when the path twists and turns, and accelerating when there is a straight road (representing the dynamics of the gradient). This adaptable nature is at the epicentre of Adam's efficiency, allowing the algorithm to traverse the terrain, or the loss function, more efficiently than algorithms with fixed step sizes.

van Hasselt et al. (2015) show that, in certain Atari environments, using deep Q-learning with a single neural network leads to the overestimation of action values under specific conditions. This phenomenon arises due to the positive bias introduced by the maximisation function within the TD-target in the Q-learning update. Moreover, using only a single network can lead to parameter instability through the nature of estimating a value in the direction of its own estimate. van Hasselt et al. (2015) propose a simple solution to these problems: use two deep Q-networks. In this manner, Q-learning targets Q_* can be estimated with respect to a slower, lagging network known as the *target* network. These targets are used in updating the *policy* network, which the agent uses to determine which actions to take. In separating the two networks in this manner, and syncing them periodically, van Hasselt et al. (2015) show that much more stable learning can be achieved. As such, this modification is employed in this work, with the sync rate hyperparameter set to 2000 steps.

Training RL agents can be very computationally expensive, and any such technique reducing this burden can be highly advantageous. Noting that Q-values were typically quite tightly distributed about a mean value, Wang et al. (2016) proposed their *duelling* methodology. Under this scheme, the final layer of each neural network splits off into two: the advantage stream, and the value stream. Each stream is a fully connected layer to the network, with each then combining to produce a single Q-value for each action. The specification used here, shown below, calculates a Q-value as the

Table 2: Duelling Deep Q-Learning Summary

Step	Description
1. Generate Experience	Sample the environment with action a_t using an ε-greedy behaviour policy.
2. Store transition	Append the new data $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$ to the experience replay $\mathcal{D}_t$, $\max\lvert\mathcal{D}\rvert = 3,000$.
3. Sample mini-batch	Retrieve 100 items from $\mathcal{D}_t$ with uniform probability.
4. Compute Q-targets	Compute Q-targets using the target network's parameters, $\mathbf{w}^-$, using both advantage and value streams.
4. Optimise policy network	Apply Adam to the MSE to update the policy network: $\mathbb{E}_{s,a,r,s'}[(r + \gamma \max_{a'} Q(s', a'; \mathbf{w}_i^-) - Q(s, a; \mathbf{w}_i))^2]$
5. Sync networks	Given 2,000 steps have elapsed, copy the policy network's weights to the target network.
6. Repeat	Continue this process until average rewards stop increasing or for a desired number of training episodes.

sum of the value stream V, for a given state s, plus the relative advantage A for the action a under consideration, less the mean advantage for the state, given additional parameter vectors $\boldsymbol{\theta}$ and $\boldsymbol{\eta}$.

$$\hat{Q}(s, a; \mathbf{w}, \boldsymbol{\theta}, \boldsymbol{\eta}) = V(s; \mathbf{w}^-, \boldsymbol{\theta}) + A(s, a; \mathbf{w}^-, \boldsymbol{\eta}) - \frac{1}{|A|} \sum_a A(s, a; \mathbf{w}^-, \boldsymbol{\eta})$$

The benefit in using this architecture is in part the greater ability of the network to learn the value of each state. This arises due to the continual update of the value function as the network learns across all actions, rather than learning the function for a *single* action at a time as with the single-stream system. This greater allocation of resources to the value function enables better approximation of the state values, and thus more effective deployment of temporal difference-based methods (Wang et al. 2016).

3.3 Deep Sarsa(λ)

A modern variation of the seminal contribution from Sutton (1988), deep Sarsa(λ) is a control algorithm for problems with continuous state spaces. Building on the algorithm described in Section 1.1.2, deep Sarsa(λ) utilises a neural network as a function approximator in place of the traditional Q-table. Moreover, the λ part of the algorithm denotes the difference in reward structure. Instead of simply taking the immediate reward plus the Q-value of the next action, Sarsa(λ) uses G_t^λ which incorporates a discounted stream of rewards:

$$G_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)},$$
$$G_t^{(n)} = r_{t+1} + \gamma r_{t+2} + ... + \gamma^{n-1} r_{t+n} + \gamma^n Q(s_{t+n})$$

In this framework, λ is a hyperparameter representing the extent to which rewards are to be discounted, with smaller values indicating a preference towards more immediate rewards. Notably, Sarsa(λ) becomes traditional Sarsa when $\lambda = 0$, and Monte Carlo control when $\lambda = 1$. In the context of the application here, a relatively high discount rate is applicable. This is because options trading has lagging effects, in that when an option is purchased, the immediate reward is close to 0. Over time, as the value of the option increasingly changes, the rewards will correspondingly change too. The true impact of the short-sale of an option, for instance, is not observable after a single time period. Apart from the difference in reward calculation, deep Sarsa(λ) shares some similarities with deep Q-learning. In this algorithm there is only a single deep Q-network, with the second made redundant due to the removal of recursive bootstrapping whereby a network's estimates are updated in their own direction[2].

As with the previous methodology, an experience replay is utilised with the same size, and the Adam optimiser is applied to the data to update parameters in the direction of the objective function's

[2]This also reduces the bias inherent to estimating Q-values at the cost of increased variance; there is no longer an estimate of the reward expectation yet more stochasticity is introduced over n-steps than one.

Table 3: Deep Sarsa(λ) Summary

Step	Description
1. Generate Experience	Sample the environment with action a_t using an ε-greedy behavioural policy.
2. Compute λ-returns	Once the episode $\mathcal{E}_i$ has concluded, the λ-return for each action is calculated.
3. Store transition	Append the new data $e_t = (s_t, a_t, G_t^\lambda), \forall t \in \mathcal{E}_i$ to the experience replay $\mathcal{D}_t$, $\max |\mathcal{D}| = 3,000$.
3. Sample mini-batch	Retrieve 300 items with uniform probability from $\mathcal{D}_t$.
4. Optimise network	Apply Adam to the MSE to update the Q-network: $\mathbb{E}_{s,a,G^\lambda}[(G_t^\lambda - Q(s, a; \mathbf{w}_i))^2]$
6. Repeat	Continue this process until average rewards stop increasing or for a desired number of training episodes.

minima. Moreover, in order to maintain training efficiency, the duelling architecture is once again employed, allowing for the simultaneous learning of the value and action-value functions in each update. An unfortunate consequence of updating in this manner is that the algorithm is quasi-offline. Given the agent must gather all the rewards and experience in an episode in order to calculate the λ-returns, optimisation of the network can only occur in-between episodes, rather than at each step within an episode. In order to compensate for this reduced frequency of parameter update, the mini-batch size was increased two-fold to 200.

3.3.1 State Space

A state space must posses all the potential information which an agent could need in order to make an informed decision. In the case of options trading, there are a vast number of important variables which an agent could use to make decisions, from patterns in weather data to supply chain disruptions, commodity prices, implied volatility surfaces, and naturally the price data itself. For the purposes of this work, a total of 19 state variables were included, with some being calculated dynamically as the agent traded, and others calculated ex-ante to save on computation at run-time.

Most of the pre-calculated features which the model takes as input are analytic in nature. The distances to the 7, 30, and 100 day moving averages, for instance, are common indicators borrowed from the realm of technical analysis. Of particular importance are the distances to the strike levels at which the agent can trade. For example, the distance to the closest call, which is approximately at-the-money, is of critical importance to the agent when trading strategies such as the straddle. Without knowledge of the whereabouts of the option strikes, the agent is essentially trading blind. A forward-looking, although admittedly weak, alpha signal comes from an LSTM model trained on OHLCV data for the underlying. This three-layer model was trained on a rolling window of 600 bars, with an equal number predicted into the future before retraining. The model was cross-validated using a modest 2-fold validation, with a final architecture of [64, 32, 16] nodes chosen. The first 600 time points, which lacked prior data, were predicted using a model trained on the final 600 points. In this manner, data leakage was avoided, and some predictive analytics were included in the state space. A full breakdown of the state variables can be seen in Table 4.

3.3.2 Action Space

The action space from which the agent can select from is comprised of 12 different strategies, along with the option to do nothing. Options "strategies" simply refer to baskets of linear combinations of calls and puts, often with different strike prices, which enable the user to realise unique payoffs dependent on the underlying (see Wilmott 2007). One such example previously mentioned is the "straddle", involving purchasing a put and a call at the same strike price. The resulting payoff structure is a "V" shape about the strike price, and is bought when the user expects significant volatility in the underlying. To better reflect true market conditions, the trading environment also enables the agent to sell each of the 12 strategies short, bringing the total size of the action space to 25. Returning to the straddle example from the short perspective, an agent would write both a call and a put option at the same price, incrementing their cash position in the process. Were the options to expire worthless, this increase in cash would remain as profit for the agent. The full list of strategies can be found in Table 5, and a visualisation of all the payoffs can be found in Appendix A.4.

Table 4: State Variables

State	Description
Distance above/below	Current distance to the nearest significant price level (multiple of 10) above and below.
MA distance (7,30,100)	The signed distance to the nearest 7, 30, and 100 period moving averages. When not enough periods have elapsed, the highest possible current moving average is used ($MA_{100}(t = 70) = MA_{70}(t = 70)$).
$\%\Delta$ in stock price (1,10,50)	Representing the current momentum of the price of the underlying across different time horizons.
$\%\Delta$ to strike prices [C,P]$\times$[0,1,2]	The most important feature: measuring the % distance to the puts and calls available to trade at different intervals above and below the spot price.
P&L (1,7,30)	Change in the value of the portfolio over different time periods.
LSTM Estimate	Predictive analytic representing an LSTM's estimate for the next period's stock price, trained using OHLCV for the underlying.

Table 5: Action Space

Strategy	Description
Call (1,2,3)	Vanilla call option to be bought at one of c^0, c^1 or c^2
Put (1,2,3)	Vanilla put option to be bought at one of p^0, p^1 or p^2
Butterfly	Upside down "V" through purchasing a call (c^0), short two calls at c^1 and long a final call at c^2.
Straddle	Purchase of a put and call at p^0 and c^0 respectively.
Strangle	Slightly wider than a straddle: purchase a put and call at p^1 and c^1 respectively.
Risk reversal	Write a put below the current spot (p^1) and buy a call above (c^1).
Bull & bear spreads	Buy a call at c^1, and sell one at c^2. The inverse in the bear case.

3.3.3 Reward

For the purposes of this trading agent, the reward will simply be the P&L over the period from which the agent is stepping out from. Hence, any changes in the value of assets held on the books will be immediately reflected in the reward communicated back to the agent.

4 Results

With the training of any RL agent, the most important thing to observe is a steadily increasing average reward as the agent learns and begins to act more greedily with respect to their Q-values. Unfortunately, as exhibited in Figure 2, the duelling deep Q architecture did not converge on any meaningful policy which was able to successfully exploit the environment. It appears as if the agent begins to take more and more risk, leading to wider fluctuations in the portfolio value as the episodes progress. Note that the rewards here, although technically denominated in dollars, are to be interpreted with caution. This is because there is no specified trading lot size which is a function of the total portfolio value; the agent only purchases/sells singular options. This, along with the size of the initial cash balance, could be scaled arbitrarily to induce virtually any final return of the same sign. Of more importance is the shape of the final rewards schedule, which can be fit to match a specific returns profile through the use of leverage.

The deep Sarsa(λ) architecture performs with relative consistency, as can be observed in Figure 2, with the agent able to fit to the in-sample training data. The continual increase in average reward indicates that the agent, which begins by taking purely random actions, is able to make informed decisions which increase rewards as the number of episodes increase. Given the average reward is simply an inter-episodic average of the P&L, this increasing reward indicates that the agent is becoming more proficient at trading in the environment, utilising the different strategies to find profitable trades. The relatively higher performance of Sarsa(λ) indeed makes intuitive sense. The

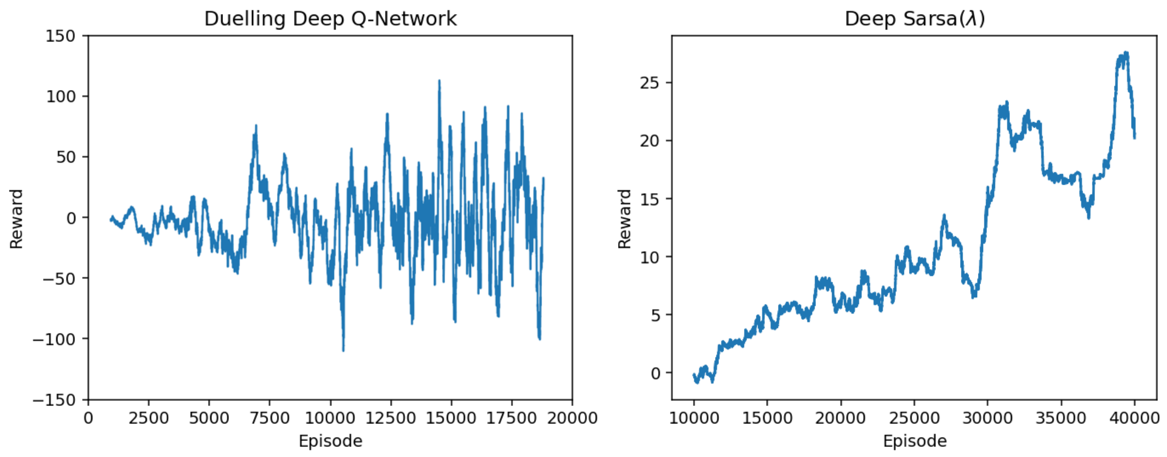

Figure 2: Rolling average of in-sample rewards for each model during training.

effects of an asset's purchase or sale on a portfolio occurs over a number of time steps as that asset's value changes more and more. Thus, utilising a TD target corresponding only to the immediate reward, plus the reward estimated by the target network, is a structurally insufficient method. Far too much importance is placed in the ability of the network's predicted Q-value to accurately represent the future stream of rewards from any given state. Naturally, the use of TD-targets which look far further ahead, and thus eliminate the need for bootstrapping, makes the algorithm more effective in mapping actions to rewards.

Although successfully fitting an RL agent to training data is challenging in of itself, this cannot be conflated with out-of-sample success. Unfortunately, with the data limits provided by the data source, there are only 4 of 25 (16%) periods (months) to be used for testing. Figure 3 nonetheless reports a more holistic breakdown of the in and out-of sample efficacy of each model architecture. This data is collected by running the training and test data through a special testing environment, involving passing the state space at each step through the fully trained network, having removed the stochastic component of the ε-greedy policy ($\varepsilon = 0$). As to be expected based on the lack of convergence for the duelling deep Q-network, both in and out-of-sample performance is poor. There are dramatic swings in the episode returns, and by episode 3 the entire stock of cash was depleted. On the other hand, the deep Sarsa(λ) results are marginally more promising. With a mean P&L of $26 across the entirety of the train and test set, the results are however still not statistically significant (H_0: $r = 0$; H_1: $r \neq 0$; $p = 0.51$). Out-of-sample, the deep Q architecture has a lacklustre performance, with a mean return of approximately 0. In contrast, the deep Sarsa(λ) model is able to generate a large return in the 21st period (Dec 2023 - Jan 2024), which increases the mean return to 34.2 ($p = 0.61$).

Figure 4 shows a more traditional "backtest" wherein the monthly returns for each options period are summed together to see as to how the portfolio would have evolved were the model trading across the trading horizon. As can be inferred from the prior set of results, the performance of the agent trained under the duelling deep Q architecture was poor; the model traded away all of its cash, dipping into negative equity, before returning to a final balance of circa $40. In contrast, the Sarsa(λ)-trained agent was able to trade much more effectively on the whole, returning 62% across the entire domain and 13% out of sample (Table 6). Compared to the null model involving a simple buy and hold of the underlying, Apple stock, this performance is on the surface commendable, beating the return by 46% across the entire domain. The same trend holds true out-of-sample, wherein Sarsa(λ) returns 13% to Apple's -5% and the S&P 500's 11%.

5 Discussion

For the sake of robustness, it is important to clarify as to what the presented results actually signify, and as to what one should ideally take away from them. Firstly, there should be no illusion that either RL model could or should be traded in its current form. The returns, both in- and out-of-sample,

Table 6: Deep Sarsa(λ) Summary

Return (%)	RL Model		Null Model	
	DDQN	Sarsa(λ)	B&H Apple	B&H S&P 500
Entire domain	-97	62	16	21
In-sample	-83	48	23	13
Out-of-sample	-13	13	-5	11

suffer from an extraordinary amount of variance. This is why the results are statistically insignificant and as to why there have been no Sharpe ratios included – the standard deviation in returns (a common proxy for risk) is astronomically high for financial standards. Creating an agent able to trade with a relatively high Sharpe ratio was never a likely outcome; that would be much further down the model's development pipeline. What the results do signify is that there is significant *potential* in the application of RL to options trading: models can fit to options data, and with a number of alterations, could improve such that they are viable in a production environment.

The hypothesis as to the predominant issue plaguing the efficient operation of the models in this work relates to reward assignment, or the lack thereof. A very sensible tweak to the reward structure used in this work would be to have the reward per action completely isolated. Presently, the reward is taken as a function of the P&L of the *entire* portfolio, whereas it would likely be more efficient to have this as action specific, such that the agent knows the exact change in P&L resulting from each action taken, rather than an indistinct portfolio average. Moreover, it would be advantageous for the reward to scale non-linearly with drawdown, thus allowing for a greater penalisation of periods involving significant loss. In general, a greater integration of risk-management practices would be highly beneficial to the model's stability, and would help increase the Sharpe ratio through mediating the fat tails of the returns distribution.

The list of extensions which could be included in the framework created thus far is virtually endless. A foundational improvement which could be made, for instance, is to incorporate the use of the Cython library (Behnel et al. 2011) in the trading logic. This library facilitates the integration of Python and C++, a compiled and thus much faster language. The initial plan for the development of the trading environment was for numerous functions to be written in the Cython language, allowing for faster training through compiling frequently used functions. Unfortunately, this was abandoned for the sake of time management. An alternative to Cython is Numba (Lam et al. 2015), a similar package which automatically compiles selected code at run-time. Once initially compiled, each additional function call is to the compiled instance, allowing for enormous computational saving and enhancements. Ultimately, a faster environment allows for quicker training, which in turn allows for a more comprehensive validation of hyperparameters and exploration of novel extensions.

As the scale of the trade sizes increases, or the available liquidity in targeted markets decreases, positional slippage and averse selection become more prominent. Therefore, were this framework to be applied to, for instance, market making or trading with far larger volumes, it would be prudent to utilise a limit order book (LOB) simulator, such as ABIDES (Byrd et al. 2019). ABIDES, and other equivalent simulators, enhance the reliability of backtesting results by simulating a dynamic LOB which is dependent upon the liquidity added or withdrawn from the book. In doing so, the market orders which are placed in this framework would have a corresponding adverse effect in the LOB, thus reducing profitability. More accurately mirroring true market dynamics in this instance is of critical importance when implementing models into production. A promising application of ABIDES, alongside a deep duelling Q-network architecture, to market making in equities can be found in Nagy et al. (2023).

Another natural progression would be to use a wider variety of RL models, such as those from the actor-critic class. Interestingly, in their research of a similar application, Wen et al. (2021) find that *proximal policy optimisation*, an actor-critic method, outperforms the traditional deep Q architecture. Finally, with greater data availability, it would have been interesting to observe the results given the greeks were included in the state space. It could be the case that the model and network are able to learn these non-linear dependencies without their explicit definition, but nonetheless it would be good to test that assertion.

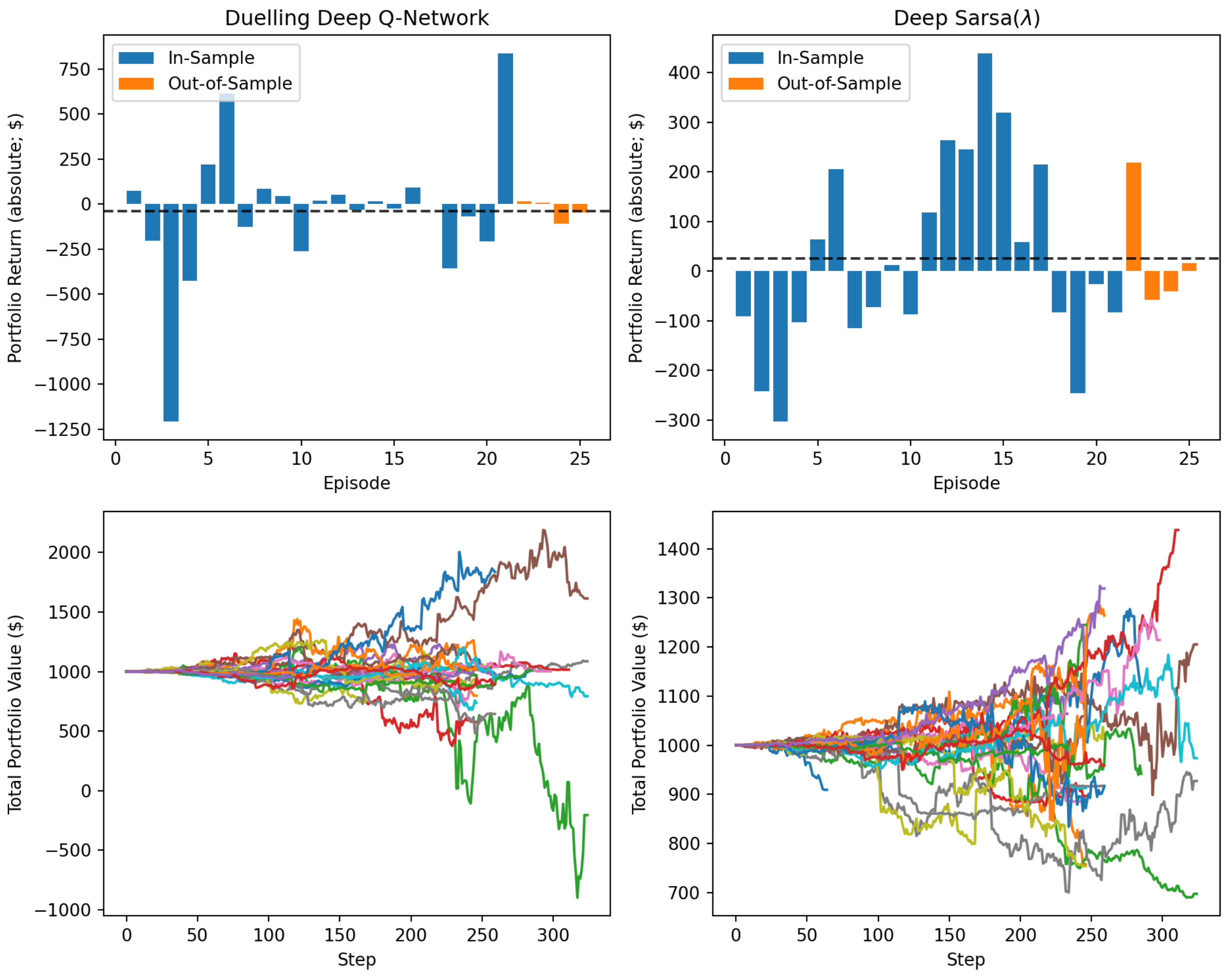

Figure 3: Summary statistics for the performance of the trained models, in- and out-of-sample.

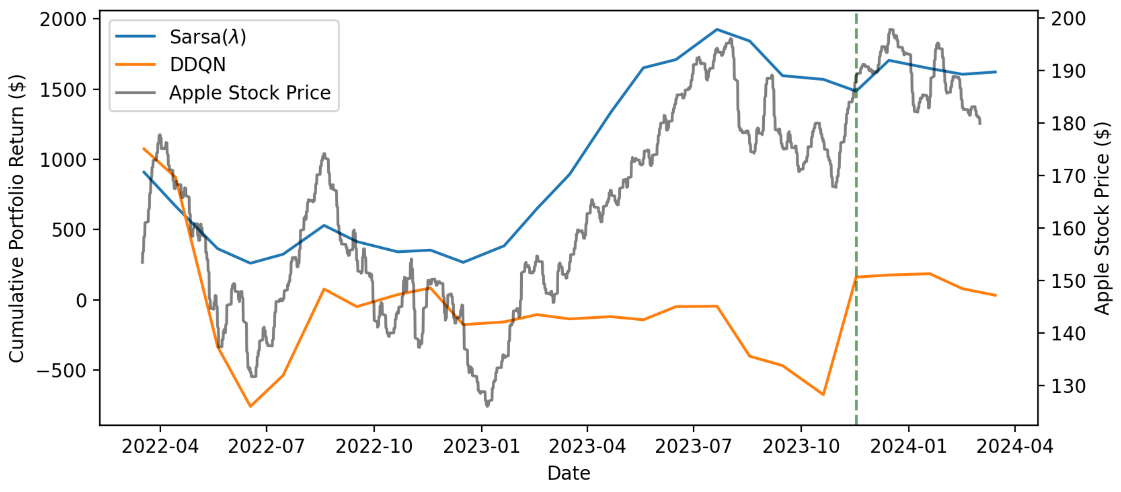

Figure 4: Backtest and comparison to null models.

A Appendix

A.1 Incremental Mean Update Derivation

An interesting lens through which to view incremental updates in reinforcement learning is outlined in Silver (2015), and is as follows:

$$\begin{aligned}\mu_k &= \frac{1}{k}\sum_{i=1}^{k} x_i \\ &= \frac{1}{k}(x_k + \sum_{i=1}^{k-1} x_i) \\ &= \frac{1}{k}(x_k + (k-1)\mu_{k-1}) \\ &= \mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1})\end{aligned}$$

Noting the similarities with, for example, the Sarsa update below, one can think of this as taking the difference between a new piece of information and the prior estimate for the mean, and updating the mean in this direction ($1/k = \alpha$).

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma Q(s',a') - Q(s,a)),$$

A.2 Differentiating model-free and model-based learning

An important taxonomy in RL is the distinction between model-based learning, whereby an agent has full knowledge of the information state through the reward function and state transition matrix $\mathbb{P}$, and model-free learning where such functions are not provided or practically impossible to observe. Optimisation techniques which are reliant on a model of the environment are typically borrowed from optimal control theory and dynamic programming. Whilst on the other hand model-free optimisation would be classified as more traditional "RL" (given an agent has to learn the environment through sampling, rather understanding its mechanics ex-ante). In order to understand the intuition behind more advanced models, it is useful to have a grasp of dynamic programming given this is the theoretical base from which many RL models derive.

A.3 Exploitation vs. Exploration

To illustrate this critical concept, consider the following pedagogical example from Silver (2015). There exists two doors which an agent can choose from, each providing a reward drawn from a discrete distribution. Initially, the agent chooses door A, earning 2 reward. The agent then chooses door B and earns 5 reward. At this point, were the agent to be acting greedily with respect to their Q-values, they would once again choose door B, earning say 7 reward. The agent from this point onwards will only choose door B, given it is providing a mean return of 6 which is greater than 3. However, through doing so, the agent is ignoring door A which in expectation offers a reward of 7, alternating between 2 and 12 with equal probability. Notably the reverse of this strategy is equally inefficient; choosing an action at random each step would lead to reduced rewards through the selection of sub-optimal actions with respect to the Q-values. The innovation which SARSA adopts, ε-greedy policy improvement, circumvents this fundamental tradeoff. With probability ε, an agent at a given step will take a random action, and with probability $(1 - \varepsilon)$, they will act greedily over their Q-values. Crucially, ε begins at zero, ensuring that the agent will explore the environment at random, and then decrements the value over time, allowing the agent to focus on maximising their reward once they have sufficiently explored the environment for optimal paths.

A.4 Option Strategy Payoffs

The following is a generic example of the different payoffs available under the options strategies traded. Notably, only the long versions of these payoffs has been displayed, yet the short versions can

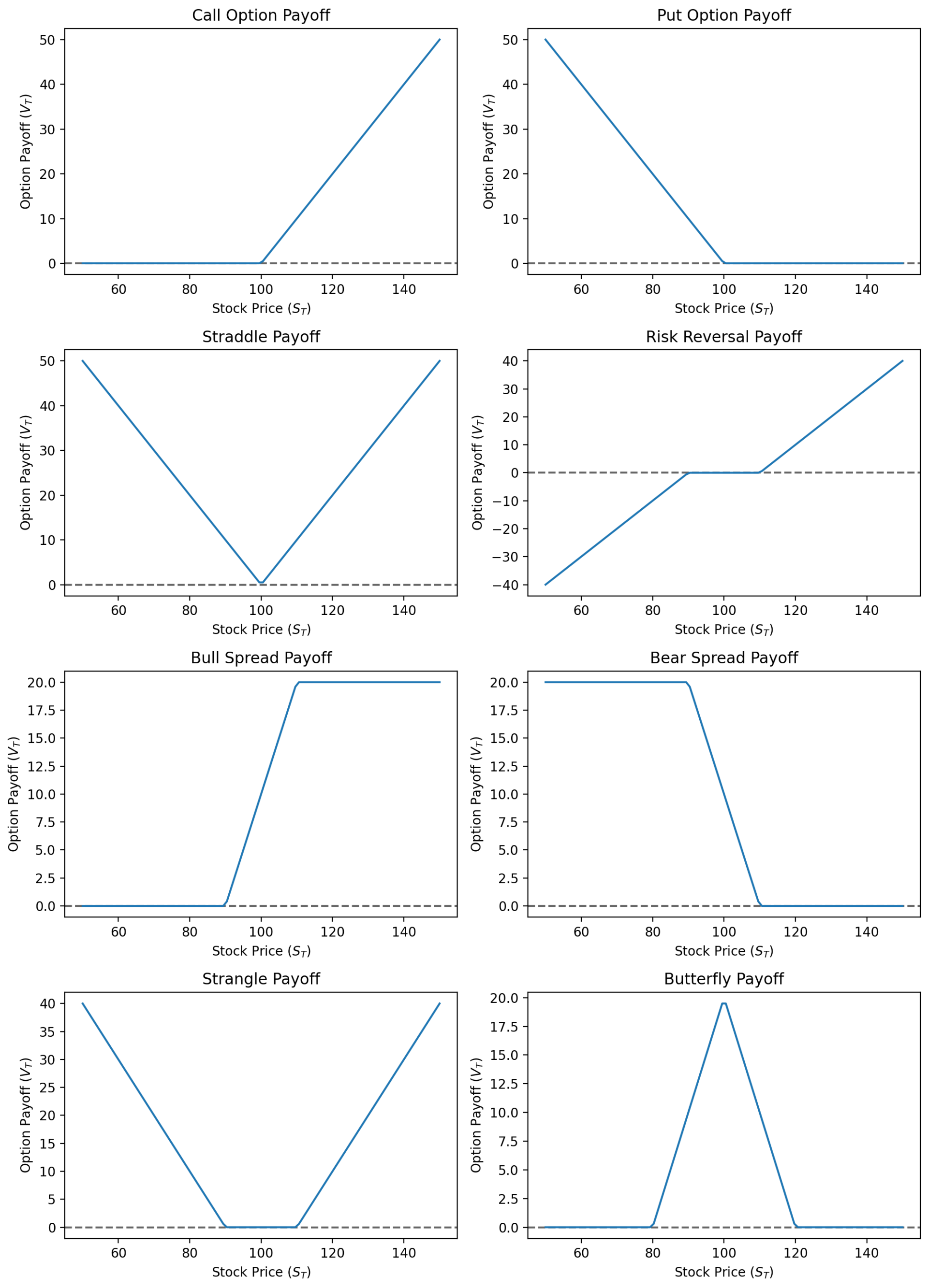

Figure 5: Long only option payoffs for all the different strategies traded.

be imagined as a mirror reflection of the corresponding long payoffs about the x-axis. The premium paid/received for an option is also not included in the payoff.

References

Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D. S. & Smith, K. (2011), 'Cython: The best of both worlds', *Computing in Science Engineering* **13**(2), 31–39.

Bellman, R. (1958), 'Dynamic programming and stochastic control processes', *Information and Control* **1**(3), 228–239.

Bellman, R. (1966), 'Dynamic programming', *science* **153**(3731), 34–37.

Black, F. & Scholes, M. (1973), 'The pricing of options and corporate liabilities', *Journal of Political Economy* **81**(3), 637–654.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J. & Zaremba, W. (2016), 'Openai gym'.

Byrd, D., Hybinette, I. & Balch, T. (2019), 'Abides: Towards high-fidelity market simulation for ai research'.

Bühler, H., Gonon, L., Teichmann, J. & Wood, B. (2018), 'Deep hedging'.

Cao, J., Chen, J., Hull, J. & Poulos, Z. (2020), 'Deep hedging of derivatives using reinforcement learning', *The Journal of Financial Data Science* **3**(1), 10–27.

CBOE (2024), 'Cboe options specification', `https://www.cboe.com/exchange_traded_stock/etp_options_spec/`.

Du, J., Jin, M., Kolm, P., Ritter, G., Wang, Y. & Zhang, B. (2020), 'Deep reinforcement learning for option replication and hedging', *The Journal of Financial Data Science* **2**, 44–57.

Halperin, I. (2019), 'Qlbs: Q-learner in the black-scholes(-merton) worlds'.

Kingma, D. P. & Ba, J. (2017), 'Adam: A method for stochastic optimization'.

Lam, S. K., Pitrou, A. & Seibert, S. (2015), Numba: a llvm-based python jit compiler, *in* 'Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC', LLVM '15, Association for Computing Machinery, New York, NY, USA.

Lin, L. (1992), 'Self-improving reactive agents based on reinforcement learning, planning and teaching', *Machine Learning* **8**, 293–321.

Liu, X.-Y., Xia, Z., Rui, J., Gao, J., Yang, H., Zhu, M., Wang, C. D., Wang, Z. & Guo, J. (2022), 'Finrl-meta: Market environments and benchmarks for data-driven financial reinforcement learning'.

Markowitz, H. (1952), 'Portfolio selection', *The Journal of Finance* **7**(1), 77–91.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. & Riedmiller, M. A. (2013), 'Playing atari with deep reinforcement learning', *CoRR* **abs/1312.5602**.

Nagy, P., Calliess, J.-P. & Zohren, S. (2023), 'Asynchronous deep double dueling q-learning for trading-signal execution in limit order book markets', *Frontiers in Artificial Intelligence* **6**.

Polygon (2024), 'Polygon website', `https://polygon.io/`.

Ritter, G. & Kolm, P. (2018), 'Dynamic replication and hedging: A reinforcement learning approach', *SSRN Electronic Journal* .

Rummery, G. & Niranjan, M. (1994), 'On-line q-learning using connectionist systems', *Technical Report CUED/F-INFENG/TR 166* .

Santos, M. S. & Rust, J. (2004), 'Convergence properties of policy iteration', *SIAM Journal on Control and Optimization* **42**(6), 2094–2115.

Schaul, T., Quan, J., Antonoglou, I. & Silver, D. (2016), 'Prioritized experience replay'.

Silver, D. (2015), 'Lecture series on reinforcement learning'.

Sutton, R. (1988), 'Learning to predict by the method of temporal differences', *Machine Learning* **3**, 9–44.

Sutton, R. S. & Barto, A. G. (2018), *Reinforcement Learning: An Introduction*, second edn, The MIT Press.

van Hasselt, H., Guez, A. & Silver, D. (2015), 'Deep reinforcement learning with double q-learning'.

Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M. & de Freitas, N. (2016), ‘Dueling network architectures for deep reinforcement learning’.

Watkins, C. & Dayan, P. (1992), ‘Technical note: Q-learning’, *Machine Learning* **8**, 279–292.

Wen, W., Yuan, Y. & Yang, J. (2021), ‘Reinforcement learning for options trading’, *Applied Sciences* **11**(23).

Wilmott, P. (2007), *Paul Wilmott Introduces Quantitative Finance*, 2 edn, Wiley-Interscience, USA.