Project Explanation: Evaluating Risk-Sensitive Reward Adaptation in a Deep Reinforcement Learning Framework for Portfolio Management

Richard Pogoson

September 2025

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my family and friends for their support throughout the progress of this project. I would also like to thank my supervisor, Callum, for his guidance.

Abstract

This project investigates the application of Deep Reinforcement Learning to portfolio management with a focus on evaluating risk-sensitive reward functions. While most implementations have primarily optimised for returns alone, few recent studies have incorporated Conditional Value-at-Risk (CVaR) into the reward function. This work builds on those efforts by proposing a novel CVaR-informed reward structure that penalises increases in downside risk, computed via historical simulation. The problem is modelled as an Approximate Markov Decision Process, and the agents are trained either using Proximal Policy Optimisation, or Twin-Delayed-Deep-Deterministic Policy Gradient with a custom feature extractor built either using Long Short-Term Memory or Convolutional neural networks. Three reward functions: logarithmic return, CVaR-informed return, and the Differential Sharpe Ratio, are evaluated across consistent experimental conditions. Random noise is injected into training data to improve market stochasticity and improve policy robustness. **Experimentation shows that...**

1 Methodology

1.1 Approximate Markov Decision Process Formulation

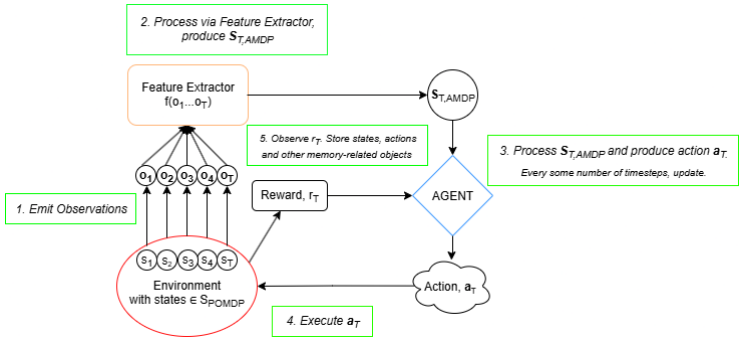

Figure 1.1: Conceptual Sketch of Environment and Agent Interaction

The above image details a sketch of the learning process. For ease of illustration, it is simplified, and in practice there are more than 5 observations processed by the feature extractor per time-step.

In line with the acknowledgement of Hu and Lin (2019), this problem will be framed with the agent trying to solve *something* generated by a partially observable environment. This is discussed in more detail below.

1.1.1 States

The true state-space $\mathcal{S}_{\mathcal{POMDP}}$ of the financial environment includes a wide range of latent and unobservable factors such as market sentiment and macroeconomic influence, most of which cannot be fully observed due to noise and limited information.

Instead, the feature extractor (LSTM or CNN) receives an observation vector $\mathbf{o}_t$ at each time step t, emmited by an unknown, underlying state $S_{t,POMDP}$. To limit dimensionality, I employ a subset of those indicators chosen by Soleymani and Paquet (2020) and Zou et al. (2024). These are among the most popular in the financial industry. Similarly to Zou et al. (2024), the observation vector is defined as:

$$\mathbf{o}_t = [\, P_t/P_0, \mathbf{c}_t, \mathbf{a}_t, \mathbf{M}_t, \mathbf{Y}_t, \mathbf{C}_t, \mathbf{G}_t, \mathbf{E}_t \,],$$

where:

- $P_t/P_0 \in \mathbb{R}^+$: portfolio value at time t, normalised by dividing by initial portfolio value;
- $\mathbf{c}_t \in \mathbb{R}^n_+$: adjusted close prices for each of the n stocks;

- $\mathbf{a}_t \in \mathbb{R}^{n+1}$: portfolio allocation weights (including cash);
- $\mathbf{M}_t \in \mathbb{R}^n$: MACD - captures trend momentum by comparing two moving averages;
- $\mathbf{Y}_t \in \mathbb{R}^n$: Momentum - determines the speed at which the price of an asset is changing;
- $\mathbf{C}_t \in \mathbb{R}^n$: CCI - determines cyclical trend of an asset price as well as strength and direction of that trend;
- $\mathbf{G}_t \in \mathbb{R}_+^n$: ATR - assesses the volatility of an asset in the market through a certain period;
- $\mathbf{E}_t \in \mathbb{R}_+^n$: EMA - a moving average that gives more weight to recent prices.

The full observation vector $\mathbf{o}_t$ has size $2 + 7n$.

To capture temporal dynamics, I construct a sequence of observation vectors over a lookback window of $T = 30$, based on the findings of Zou et al. (2024), where this value was shown to be effective:

$$\mathbf{X}_t = [\mathbf{o}_{t-T+1}, \ldots, \mathbf{o}_t] .$$

This sequence $\mathbf{X}_t$ is input to a feature extractor, which produces a final feature vector f_t. Thus, the feature extractor can be considered to provide the *approximate* state space (S_{AMDP}) within which the agent is acting. This vector aims to capture the temporal structure in the observation history and serves as the agent's internal representation of state at time t.

Although the underlying environment is partially observable, the feature extractor learns to approximate the true, underlying state space ($\mathcal{S}_{\mathcal{POMDP}}$) utilising this history of observations. Nonstationarity introduced through updates to the feature extractor prevents formalisation as a strict MDP, given that updates to the feature extractor parameters result in updates to the "state" space that the agent would interact with.

1.1.2 Actions

$\mathcal{A}$: Each action is a nonnegative continuous vector representing the agent's desired portfolio allocation across $n + 1$ assets (including cash), constrained to sum to 1. In PPO, this vector is generated by sampling from a Dirichlet distribution, whose concentration parameters are produced by the final layer of the actor network. In TD3, this vector is simply produced via a feed-forward network with a final softmax layer. For PPO, the use of a Dirichlet distribution has the following advantages:

1. In problems with continuous action spaces, Gaussian distributions are normally used to generate actions when Policy Optimisation methods such as PPO are used (Schulman, Levine, et al. 2015). This normally requires

using a network to produce as many means and standard deviations as the size of the action space, and use these to sample. In the continuous action environment used in this paper, the action vector is required to sum to 1 to ensure wealth is divided appropriately. Post-processing with a softmax would be required, which, if not handled carefully, would change the relative scaling of the action vector. This could make it harder for the agent to learn. A Dirichlet distribution handles this normalisation itself.

2. Implementation is simpler, given that the vector now returns a scalar when calculating its log-pdf value, making the PPO probability ratio calculation much easier to do (as opposed to summing supposedly independent log-probability action components for each normal distribution).

3. A continuous action space simplifies the discrete, though approximately continuous state space of Zou et al. (2024) with size $(2k+1)^{30}$, where k represents the number of shares that can be bought or sold at once. It bypasses the combinatorial explosion of large discrete action spaces and holds the advantage of an in-built non-deterministic exploration feature of stochastic action selection.

1.1.3 Reward

r: Three reward functions will be tested:

1. **Standard Logarithmic Returns (SLR)**: Referring to formula (1.7):
   (a) Let $x_t = P_t/P_{t-1}$
   (b) Return:
   $$f(x_t) = \ln(x) \tag{1.1}$$
   This function is appropriate, as it returns negative values for inputs where $P_t < P_{t-1}$, positive values when $P_t > P_{t-1}$ and 0 when $P_t = P_{t-1}$. It is scale-invariant, given that it considers the ratio of portfolio values over time, which allows it to dampen the effect of large return values, which is beneficial in RL, where unnormalised rewards can destabilise training.

2. **CVaR-Informed Reward**: Utilising the formula defined in (1.5), we define another component to supplement (1.1). This is denoted as $y_t = CVaR_t - CVaR_{t-1}$ at a given time, t. Formally, this reward is:
   $$g(x_t, y_t) = f(x_t) - \zeta \times y_t \tag{1.2}$$
   Where ζ denotes the risk aversion of the trader (or agent, in this case). This intends to make the agent select actions which do not result in large increases in Conditional Value-at-Risk - i.e., maximise returns whilst minimising large losses. It distinguishes itself from that used by Cui et al. (2023) in that it combines CVaR with portfolio returns, along with its temporal nature by penalising increases in CVaR over time - as opposed to penalising CVaR magnitude at each policy update.

For CVaR calculations, a confidence level of 95% was selected. Lower percentiles, such as 75%, were not chosen as they would cause the risk boundary to shift too frequently under volatile market conditions, potentially increasing instability during training.

3. **Differential Sharpe Ratio (DSR):** The Sharpe Ratio, SR, is defined by:

$$SR = \frac{R_p - R_f}{\sigma_p} \tag{1.3}$$

where:

- R_p refers to the expected returns of the portfolio for a given time scale
- R_f refers to the risk-free rate - the rate of return on a benchmark, presumably risk-free investment.
- σ_p refers to the standard deviation of the portfolio's returns (over the same time scale used for R_p)

The *differential* Sharpe ratio, D_t, is defined in Moody and Saffell (1998) by:

$$D_t \equiv \frac{dS}{d\eta} = \frac{B_{t-1}\Delta A_t - \frac{1}{2}A_{t-1}\Delta B_t}{\left(B_{t-1} - A_{t-1}^2\right)^{3/2}} \tag{1.4}$$

where:

$$A_t = A_{t-1} + \eta\Delta A_t = A_{t-1} + \eta(R_t - A_{t-1})$$
$$B_t = B_{t-1} + \eta\Delta B_t = B_{t-1} + \eta(R_t^2 - B_{t-1})$$

- A_t operates as an exponential moving average of returns until time t
- B_t operates as the exponential moving average of squared returns - operating much like a variance, since it tells us how spread out the returns are.
- η is a hyperparameter specifying a "decay" or "learning" rate, which represents how quickly the reward function adapts to new returns. A higher number specifies a faster adaptation speed.

The numerator optimises for the weighted change in return averages, while the denominator adjusts for volatility. The objective of this reward function is to ensure high returns with low variance in an online manner.

1.1.4 Transition Probabilities

The transition function $p(f' \mid f, a)$ defines the probability of transitioning to a new abstract state representation f' given the current abstract state representation f and action a. Here, f and f' refer to the latent states produced by the LSTM Feature Extractor, which processes windows of market observations. In a given period where the feature extractor's weights remain static (and so the "state

space" viewed by the agent is fixed), the approximate-MDP being solved by the agent can be viewed as stochastic. This would be because the underlying S_{POMDP} that generates a given observation can be modelled as not generating observations deterministically, but rather according to an unknown probability distribution . Therefore, the resultant state space of the approximate MDP (S_{AMDP}) would be stochastic, as the feature extractor processes a history of non-deterministic inputs.

1.1.5 Discount Factor

γ: A value between 0 and 1 that determines the present value of future rewards.

1.1.6 Termination Condition

This occurs when either or both of the following conditions are true:

1. The portfolio value decreases by 30% or more.
2. The trading period ends.

This formulation is nonstandard, but it defines an approximate MDP, represented by the tuple $(\mathcal{S}_{\mathcal{POMDP}}, \mathcal{S}_{\mathcal{AMDP}}, \mathcal{A}, r, p(f' \mid f, a), \gamma)$. Through training, the feature extractor intends to provide a better approximation to the underlying POMDP using a fully observable approximation.

1.2 Calculating CVaR

The Historical Method is the simplest method. It simply reorganises actual historical returns, putting them in order from worst to best. It then assumes that history will repeat itself, from a risk perspective. Following the calculation method used by Hull (2021), consider a portfolio with 1-day returns over 100 days, denoted by:

$$R_1,\, R_2,\, \ldots,\, R_{100}.$$

To calculate the 95% one-day CVaR we:

1. **Sort them**
   Sort these returns in ascending order (i.e., from the worst to the best):

   $$R_{(1)} \leq R_{(2)} \leq \cdots \leq R_{(100)},$$

   where $R_{(1)}$ is the worst return and $R_{(100)}$ is the best.

2. **Determine the 95% VaR**
   Since the 95% VaR corresponds to the threshold below which the worst 5% of outcomes lie, and 5% of 100 days is 5 days, we identify:

   $$\text{VaR}_{0.95} = R_{(5)}.$$

3. **Compute the 95% CVaR**
   The 95% Conditional Value-at-Risk (CVaR) is the average of the losses on these worst 5 days:

   $$\text{CVaR}_{0.95} = \frac{1}{5} \sum_{i=1}^{5} R_{(i)}. \tag{1.5}$$

The two other dominant methods - the *Monte Carlo* and *Variance-Covariance/Parametric* approaches - are outlined briefly for context:

Both methods use statistical metrics such as volatilities and correlations of the risk factors and the sensitivities of the portfolio values with respect to these risk factors in order to approximate a VaR/CVaR value.

Monte-Carlo Method: This uses market volatilities and correlations and, through a multivariate normal probability distribution, performs rollouts to predict changes in asset prices over time, and therefore estimate what the VaR (and therefore CVaR) is.

Variance-Covariance/Parametric Method: This assumes a closed form for the return distribution of the portfolio, and using a parametric model simplifies the problem of VaR calculation to that of estimating the parameters of the distribution (MathWorks 2024). Once the distribution is estimated, CVaR can be computed as the expected loss beyond the VaR threshold.

This paper employs the Historical method with the intention of evaluating the agent based on its actual, realised returns rather than on theoretically derived

approximations from market variables, ensuring that penalties directly relate to the outcomes of its actions.

1.3 Computing a Change in Portfolio Value

Let a_t represent an action for timestep t. This is a vector of size n, where n - 1 corresponds to the number of assets being managed, with the first entry being kept for cash. This represents the desired wealth allocation for the next time step. A change in portfolio value is computed as follows:

1. Let a_t be the agent's desired allocation for time $t+1$, and a_{t-1} the current allocation at timestep t.
2. Update the allocation:
   - If a previous allocation exists, apply a liquidity constraint via convex combination:
     $$a_t \leftarrow (1-m)\, a_{t-1} + m\, a_t \tag{1.6}$$
     where $0 \leq m \leq 1$ is the maximum allocation change parameter (liquidity control - set to 1 in this paper).
   - Otherwise, if no prior allocation exists (i.e., at $t = 0$), initialise:
     $$a_{t-1} \leftarrow [1, 0, 0, \ldots, 0],$$
     placing 100% of the portfolio in cash, and set:
     $$a_t \leftarrow a_t.$$
3. If the current portfolio value P_{t-1} is undefined, initialize:
   $$P_{t-1} \leftarrow P_0,$$
   where P_0 is the starting cash.
4. Compute the wealth distribution:
   $$W = P_{t-1} \cdot a_t.$$
5. Adjust for asset price changes:
   - Let $\mathbf{p}_t = [0, p_1, p_2, \ldots, p_{n-1}]$ denote the vector of percentage price changes between timestep t and timestep $t+1$. Cash (0) is presumed to have no return.
   - Update each asset's wealth by applying the return:
     $$W' = (1 + \mathbf{p}_t) \circ W$$

- *Note*: P_t elsewhere denotes the total portfolio value at time t; here, lowercase bold $\mathbf{p}_t$ refers specifically to the vector of per-asset price returns.

6. Compute the transaction cost (if a previous allocation exists):

$$TC = \tau \cdot P_{t-1} \cdot \sum_{i=1}^{n} \left| (a_t)_i - (a_{t-1})_i \right|,$$

   where τ is the transaction cost rate (0.02%).

7. Compute the new portfolio value:

$$P_t = \sum_{i=1}^{n} W_i' - TC. \tag{1.7}$$

2 Reinforcement Learning Algorithms

2.1 PPO: Theory

This algorithm choice primarily follows the precedent set by Zou et al. (2024), while also reflecting PPO's status as one of the most advanced policy-based RL techniques (Schulman, Wolski, et al. 2017). It requires a stochastic policy and possesses the following characteristics:

1. **On-policy**: PPO evaluates and improves the same policy that it is actively using during training to collect experiences, rather than relying on external policies.
2. **Online**: PPO is designed to update incrementally, processing data sequentially as it is received, rather than requiring an entire dataset upfront for training.

PPO was introduced to address instability in earlier policy gradient methods. Vanilla policy gradient algorithms can make excessively large updates when advantage estimates are high, leading to collapsed or divergent policies. Trust Region Policy Optimisation (TRPO) mitigated this by imposing a constraint on the KL divergence between the old and new policy distributions. However, TRPO's approximation of this KL constraint requires second-order derivatives to be calculated, making TRPO complex and computationally expensive. PPO simplifies this by using a *clipped* surrogate objective that implicitly controls the policy update step size. This prevents large deviations from the previous policy while still allowing for efficient learning. The idea is to optimise a first-order approximation of the TRPO objective while avoiding the need for explicit trust region constraints.

These attributes may allow PPO to adapt its policy to changing market dynamics, which is critical for handling volatile and non-stationary financial environments. Furthermore, PPO's online nature allows a natural integration of recurrent neural networks, such as LSTMs. This makes it well-suited for capturing temporal dynamics in financial data.

2.1.1 Rollout and Experience Collection:

For the LSTM-based feature extractor, at the start of each episode, the Feature Extractor, Value Network (Critic), and Policy Network (Actor) hidden states are initialised. At each time step, the environment produces the observation matrix, $\mathbf{X}_t$, which is then transformed by a Feature Extractor into the feature vector f_t. This vector is then passed to the agent, which samples an action, a_t, using the actor network and returns the associated log-probabilities, ℓ_t. Simultaneously, the critic produces a value estimate for the same feature vector. It is then executed in the environment, and the reward and done flag are observed. The done-flag specifies whether the ensuing state is terminal (i.e., a state from which

no other states can be reached). The tuple $(\mathbf{X}_t, a_t, \ell_t, V_t, r_t, d_t, hiddenstates)$ is stored in the memory buffer.

Note: Throughout this paper "hidden state" will be used to refer to both the hidden and cell states of an LSTM. LSTMs differ from standard recurrent neural networks in their use of an additional "cell" state alongside the standard hidden state, which helps to act as a long-term memory holder.

2.1.2 Advantage Estimation using GAE:

Every T timesteps (where T represents the update frequency), the agent is updated. First, an advantage estimate, $\hat{A}_t$ is computed for each time step, returning an advantage vector:

$$\hat{A}_t = \sum_{k=t}^{T} \gamma^{\,k-t} \lambda^{\,k-t} \Big(r_k + \gamma\, V_{k+1}(1 - d_k) - V_k \Big)$$

It requires three components:

1. The Temporal Difference (TD) error, $r_k + \gamma\, V_{k+1}(1-d_k) - V_k$: this measures the discrepancy between the expected return (V_t) and the actual return ($r_k + \gamma\, V_{k+1}(1 - d_k)$) after taking an action. It should be noted that in Reinforcement Learning, the value of a state intends to encode the value of expected rewards that can be gained from that state - as such, while V_t is an estimate, comparing it to the actual reward recieved from that state with the estimated value of the following state intends to provide a difference between the estimated value of that state (by the critic's value function) and the actual value of that state.
2. The Discount Factor at time $k - t$, $\gamma^{\,k-t}$: this takes into account how far into the "future" state k is from the beginning of the rollout.
3. GAE Lambda, λ: this controls the trade-off between bias and variance in the advantage estimate. A value of $\lambda = 0$ reduces variance with more bias (akin to one-step TD, given that experiences beyond one time-step in the future are ignored), while $\lambda = 1$ reduces bias but increases variance (similar to Monte Carlo return estimation)

In sum, A_t measures whether or not the action is better or worse than the policy's default behaviour (Schulman, Moritz, et al. 2015).

2.1.3 Batch Processing and Importance Sampling:

We then process the full batch of recent experience, denoted $B = \{t = 0, \ldots, T - 1\}$, since no mini-batching is used in this paper. From this batch, we extract the observation matrices $\mathbf{X}_B$, the corresponding actions a_B, and the old log-probabilities ℓ_B^{old}. We compute the feature vectors $f_B = \mathcal{F}(\mathbf{X}_B)$ and pass

them through the current policy π_θ to obtain the updated action distribution $\pi_B = \pi_\theta(f_B)$.

For each timestep $t \in B$, we compute the log-probabilities of the previously taken actions under the current policy and form the probability ratios:

$$\rho_t = \frac{\pi_\theta(a_t \mid f_t)}{\pi_{\theta_{\text{old}}}(a_t \mid f_t)}$$

The ratio ρ_t arises from importance sampling, which allows PPO to reweight experiences collected under the old policy $\pi_{\theta_{\text{old}}}$ while evaluating updates under the current policy π_θ. This is essential for stable on-policy learning, ensuring that updates account for distribution shift while taking into account the likelihood of the vector-action pairs with respect to the current policy compared to the old policy.

2.1.4 Policy Update and Loss Functions:

The actor loss is then calculated using the PPO-Clip objective:

$$L^{\text{actor}} = -\mathbb{E}_b\big[\min\big(\rho_b \tilde{A}_b,\ \text{clip}(\rho_b, 1-\epsilon, 1+\epsilon)\tilde{A}_b\big)\big]$$

The negativity in the formula arises from the fact that policy gradient methods operate via gradient *ascent* as opposed to the more common gradient *descent*. Flipping the sign of this objective informs the optimiser to "minimise the negative" of the policy objective, which is mathematically equivalent to "maximising the positive" of the policy objective. The policy gradient objective is to maximise the advantage. With respect to the policy function, this can be colloquially referred to as the probability of taking advantageous (or higher-reward-yielding actions).

When reading the below, consider the idea of "minimising the negative" advantage. More "positive" losses are considered unbeneficial, and more "negative" losses are considered desirable.

Presume the advantage is *positive*:

- **If** $\rho > 1 + \epsilon$, the $\min(\text{clip}(\dots)\dots)$ returns $1 + \epsilon$, so the loss is clipped.
- **If** $\rho < 1 - \epsilon$, the $\min(\text{clip}(\dots)\dots)$ returns ρ, meaning that the loss gets more negative, but not as much as the favoured case where $\rho > 1 - \epsilon$.

The ratio here is bounded to $1 + \epsilon$.

Presume the advantage is *negative*:

- **If** $\rho > 1 + \epsilon$, the $\min(\text{clip}(\dots)\dots)$ returns ρ, meaning that the loss gets more positive. The gradient descent will push the policy to reduce the probabilty of poor actions as the loss therefore becomes larger conceptually.

A probabily distribution with a higher likelihood of selecting a poor actions is strongly optimised, since ρ can be very large (in theory). Therefore, note that the objective is not symmetric.

- **If** $\rho < 1 - \epsilon$, the $\min(\text{clip}(\dots)\dots)$ returns 1 - ϵ, meaning that the loss gets more positive, but not as much since it is limited by 1-ϵ. A probability distribution with a lower likelihood of poor actions is more weakly optimised.

The critic loss measures the deviation between the estimated value $V_\phi(f_t)$ and the target return $\hat{A}_t + V_t^{\text{old}}$, using the huber loss metric:

$$L^{\text{critic}} = \text{Huber}\Big(V_b^{\text{new}},\ \hat{R}_b\Big)$$

After these batch updates are concluded, the memory buffer is cleared. The pseudocode for this is detailed on the next page:

Algorithm 1 PPOLSTM, GAE, and Shared Feature Extractor

Require: Learning rate α, discount γ, GAE λ, clip ϵ, batch size B, PPO Epochs E
Require: Actor π_θ (LSTM), critic V_ϕ (LSTM), feature extractor $\mathcal{F}_\psi$
Require: Entropy coefficient $c_{\text{ent}} \in \{0, \text{small}\}$
Require: Memory buffer $\mathcal{M}$ that stores $(\mathbf{X}_t, a_t, \log \pi_\theta(a_t \mid f_t), V_t, r_t, d_t, \text{hidden})$

Initialise Adam on $\{\theta, \phi, \psi\}$ with step size α
for each training epoch **do**
  for each episode **do**
    Reset env; reset LSTM hidden states for actor, critic, and (if applicable) $\mathcal{F}_\psi$
    for $t = 0, 1, \ldots$ **do**
      $f_t \leftarrow \mathcal{F}_\psi(\mathbf{X}_t)$;
      Sample $a_t \sim \pi_\theta(\cdot \mid f_t)$ (train) or use mean action (eval); record $\ell_t = \log \pi_\theta(a_t \mid f_t)$
      $V_t \leftarrow V_\phi(f_t)$; step env with a_t to get r_t, $\mathbf{X}_{t+1}$, done d_t
      Store $(\mathbf{X}_t, a_t, \ell_t, V_t, r_t, d_t,$ hidden states$) \in \mathcal{M}$
      if update boundary reached **then**
        for each E **do**
          Bootstrap: if non-terminal, set $V_T \leftarrow V_\phi(\mathcal{F}_\psi(\mathbf{X_T})$; last hidden); else $V_T \leftarrow 0$
          GAE: set $g \leftarrow 0$; for $t = T-1, \ldots, 0$:
            $\delta_t \leftarrow r_t + \gamma(1 - d_t)V_{t+1} - V_t$
            $g \leftarrow \delta_t + \gamma\lambda(1 - d_t)\, g; \quad A_t \leftarrow g$
          Returns: $\hat{R}_t \leftarrow A_t + V_t$
          Form mini-batches $\mathcal{B}$ of size B from stored indices
          for each batch $b \in \mathcal{B}$ **do**
            Recompute $f_b \leftarrow \mathcal{F}_\psi(\mathbf{X_b})$
            $\ell_b^{\text{new}} \leftarrow \log \pi_\theta(a_b \mid f_b); \quad V_b^{\text{new}} \leftarrow V_\phi(f_b)$
            $\tilde{A}_b \leftarrow \dfrac{A_b - \mu(A_b)}{\sigma(A_b) + 10^{-8}}$
            $\rho_b \leftarrow \exp(\ell_b^{\text{new}} - \ell_b^{\text{old}})$
            $L_{\text{actor}} \leftarrow -\mathbb{E}_b\big[\min\big(\rho_b \tilde{A}_b,\ \text{clip}(\rho_b, 1-\epsilon, 1+\epsilon)\tilde{A}_b\big)\big]$
            $L_{\text{critic}} \leftarrow \text{Huber}\big(V_b^{\text{new}},\ \hat{R}_b\big)$
            $H_b \leftarrow \mathbb{E}[\mathcal{H}(\pi_\theta(\cdot \mid f_b))]$ ($c_{\text{ent}} = 0$ in implementation)
            $L \leftarrow L_{\text{actor}} + \frac{1}{2} L_{\text{critic}} - c_{\text{ent}} \frac{H_b}{\dim(\mathbf{a})}$
            Backprop on $\{\theta, \phi, \psi\}$, clip global grad-norm at 1.0, Adam step
          end for
          Clear $\mathcal{M}$ and continue interaction
        end for
      end if
    end for
  end for
end for

2.1.5 Specific Implementation Details (PPO)

It should be noted that the PPO-variant opted for here is nonstandard. When implementing PPO with a recurrent policy, multiple implementation details should be taken into account to remain faithful to RL theory and on-policy assumptions.

1. The hidden states for the actor, critic and (when LSTM) feature extractor must be reset every episode to maintain the first-order Markov assumption - i.e., a state is strictly dependent on the state that came before it, and none other. Not resetting the hidden state would mean that a new episode would begin with the first state being implicitly dependent on the last state of the previous episode (given that that first state is passed the hidden state from the previous episode, which encodes temporal dynamics from that previous episode).

2. An entropy bonus is sometimes added to the loss funtion in PPO to encourage exploration, given that distributions with higher degrees of entropy are more "random". This is employed in this paper, and the entropy bonus is further scaled by the dimension of the action vector (or the number of concentration parameters of the dirichlet). The Dirichlet distribution (being a continuous distribution), has its entropy measured via "differential entropy". The differential entropy maximum decreases linearly with respect to the number of concentration parameters, and thus scaling it ensured it did not dominate the loss function. Further, adding this penalty helps to stabilse the network. The differential entropy is maximised where all α are equal to 1, and thus adding this penalty not only enforces exploration but pushes concentration parameters to this more stable value around 1.

3. Hidden-State handling becomes extremely complicated. In recurrent RL algorithms, the action is a function of both the state input and the prior hidden state. Therefore, to maintain the on-policy nature of PPO, one must store both the states and the hidden states computed during the rollout to ensure the importance sampling function of PPO is maintained. Given that the feature extractor is also trained in this process, it was required to store the observation matrix $\mathbf{X}_t$ also in order to recompute feature representations.

4. Some PPO implementations normalise advantages via a z-scoring method - i.e., subtracting the mean and dividing by the standard deviation to stabilise the advantage signal. This is something that is implemented in this paper.

5. It is very common to instead utilise a mean-squared error to optimise the critic as opposed to the Huber loss. Due to early instability from large critic-losses, we employ the Huber loss which is quadratic up until 1, and then linear, aiming to promote stability by reducing (with respect to the

mean-squared loss), the error beyond a value of 1.

2.2 PPO: Architectures

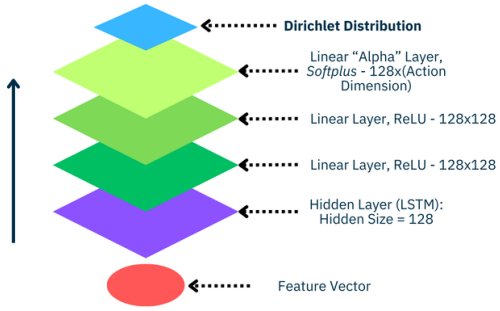

Figure 2.1: Actor Network Structure (PPO)

The Actor utilises an LSTM layer for temporal encoding of state inputs, having received f_t as input from the feature extractor. The LSTM's last hidden state flows through two ReLU-activated dense layers. The third layer is also a linear layer, but produces an output the size of the action dimension. This is supposed to represent the *alpha*, or concentration parameters, of the Dirichlet distribution. The concentration parameters control the "weight" of belief about the allocations. This is appropriate as PPO requires a distribution to be produced by its actor (which Softmax is not).

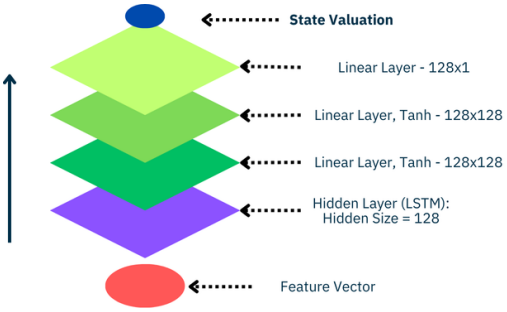

Figure 2.2: Critic Network Structure (PPO)

The Critic also processes state inputs via an LSTM layer. The LSTM's last hidden state is passed through a series of three fully connected layers. The first two utilise Tanh activation functions, but the third does not, as normalisation is not required here.

2.3 TD3: Theory

TD3 (Twin-Delayed Deep Deterministic Policy Gradient) differs significantly from PPO. TD3 is both:

1. **Off-policy**: TD3 trains "target networks" - copies of the networks being used to acquire experiences from the environment.
2. **Experience-reusing**: TD3 can update incrementally, but requires the entire history of encountered experiences at training time to do so. Unlike PPO, it does not clear the replay buffer after training.

TD3 was intended to be an improvement on its predecessor, Deep Deterministic Policy Gradient (DDPG) - another actor-critic method for continuous action spaces. DDPG required a pair of neural networks: an actor network, learning a deterministic (as opposed to stochastic, like PPO) mapping from states to actions, and a critic which aimed at estimating the Q-Value of state-action pairs. This is analogous to the value funtion, $V_\phi(f_t)$, found in PPO, but instead computes the expected long-term reward for a state-action pair, as opposed to just a state. This combination aimed at combining the strengths of value-based and policy optimisation approaches, however, the determinism of the action selection could lead to critic value-function overestimation, due to the actor being trained to select the action that maximises the critic's q-value for a give state. If this overestimation accumlates over training, the network can become unstable. (Lillicrap et al. 2016) also note that the algorithm is sensitive to hyperparameter settings, requiring careful tuning to balance the updates of the actor and critic networks.

TD3 mitigates thes problems with several improvements. As outlined by (Fujimoto et al. 2018), it:

- Introduces double Q-learning. TD3 has two standard critic networks, as opposed to DDPG's one. It selects the smaller of two Q-values computed by the critics for a given state-action pair, intending to reduce overestimation bias.
- Employs delayed policy updates. The critics are updated more frequently than the actor, ideally meaning that the critic has more time to stabilise before influencing the actor's updates.
- Adds noise to the target actor during training, encouraging generalisation and ideally reduces the risk of overfitting specific actions.

Thus, TD3 becomes more stable and effective algorithm for solving RL problems in continuous action domains.

2.3.1 Experience Collection

TD3 differs slightly from PPO in this respect. For the LSTM-based feature extractor, all standard and target network hidden states are initialised. At each

time step, the environment produces the observation matrix, $\mathbf{X}_t$, which is then transformed by a Feature Extractor into the feature vector f_t. This vector is then passed to the agent, which selects an action, a_t, using the actor network, along with hidden states for standard and target actor and critic networks. This action is 'perturbed' with some noise according to some normal distribution with mean 0 and standard deviation ϵ. The action is then executed in the environment, and the reward and done flag are observed and stored, however, the environment also returns the next observation matrix, $\mathbf{X}_{t+1}$ in this case. The tuple $(\mathbf{X}_t, a_t, r_t, \mathbf{X}_{t+1}, d_t, hiddenstates)$ is stored in the memory buffer.

2.3.2 Training Period

The algorithm initialises parameters for its policy network (actor) and two q-function networks (critics). If not yet initialized, the replay buffer is initialised as empty. A target policy network (the target actor) is also initialised alongside two target q-function (target critic) networks and these are assigned the same weights.

If there is sufficient memory (at least *batchSize* experiences) within the replay buffer to learn from, then the agent samples from the buffer *batchSize* experiences. The target actions are computed by the target actor with a perturbation of ϵ. The target q-values are also computed by the target critics as a function of:

$$y \leftarrow r + \gamma(1-d)\ \min\big(Q_{\phi_1^{\text{tgt}}}(f', a'),\ Q_{\phi_2^{\text{tgt}}}(f', a')\big)$$

Where:

- r denotes the rewards from each approximate state, f
- $(1-d)$ evaluates to a boolean reflecting whether f', the following state, is a terminal state (0 if terminal, 1 otherwise)
- γ denotes the discount factor which denotes how short-sighted the agent is (a lower value ensuring that the agent assigns low values to temporally distant states)
- $\min\big(Q_{\phi_1^{\text{tgt}}}(f', a'),\ Q_{\phi_2^{\text{tgt}}}(f', a')\big)$ denotes the minimum state-action value estimate of the two target critic networks to prevent overestimations

The above formula is simply a standard q-valuation function, where the value of a state-action pair is denoted by the reward received from the state-action pair leading to the state plus the discounted value of the state, policy-chosen action pair of the ensuing state. This "state, policy-chosen action pair" is valued at zero if the ensuing state in question is a terminal state.

The non-target critics also value the state-action pairs and the Mean-Squared Error (MSE) loss is calculated for each critic. Using stochastic gradient descent, each non-target critic is updated using this loss and, less frequently, (as the

target network is updated every $actor\_network\_frequency$ timesteps) back-propagation is run on the actor network and all target networks are updated via a convex combination of its current weights and the corresponding non-target network weights, where the new target weights are weighted by the following sum:

$$\tau * non\_target\_network\_weight + (1 - \tau) * target\_network\_weight$$

To explain, a value of $\tau = 0.005$ (the value used in our code) would mean that the updated target network weight becomes 0.5% less 'like' the number it was before, and 0.5% more 'like' the corresponding non-target network weight.

This batch learning is run a pre-specified number of times, and when this pre-specified number is reached, this process begins again with the agent taking another action. If at any state f, the ensuing state, f', is a terminal state, the environment resets and a new episode begins, restarting the loop. The pseudocode for this is found on the next page.

Algorithm 2 TD3 with LSTM, Shared Feature Extractor (critic-owned), and Target Smoothing

Require: Actor step size α, critic step size β, discount γ, soft update τ, policy delay d_π, batch size B, actor noise σ, target noise $\tilde{\sigma}$
Require: Feature extractor $\mathcal{F}_\psi$, target feature extractor $\mathcal{F}_\psi^{\text{tgt}}$; actor μ_θ (LSTM); critics Q_{ϕ_1}, Q_{ϕ_2} (LSTM)
Require: Replay buffer $\mathcal{D}$ storing $(\mathbf{X}_t, a_t, r_t, \mathbf{X}_{t+1}, d_t,$ hidden states$)$

Hard copy targets: $\theta^{\text{tgt}} \leftarrow \theta$, $\phi_i^{\text{tgt}} \leftarrow \phi_i$ for $i=1,2$, $\psi^{\text{tgt}} \leftarrow \psi$
for each episode **do**
  Reset env and all LSTM hidden states (actor, critics, feature extractors)
  for $t = 0, 1, \ldots$ **do**
    $f_t \leftarrow \mathcal{F}_\psi(x_t;$ hidden$)$
    Action (train): $a_t \leftarrow \Pi\big(\mu_\theta(f_t) + \epsilon,\ \epsilon \sim \mathcal{N}(0, \sigma^2)\big)$ ▷ Π projects to simplex
    Step env with a_t to get $(r_t, \mathbf{X}_{t+1}, d_t)$
    Store $(\mathbf{X}_t, a_t, r_t, \mathbf{X}_{t+1}, d_t,$ hidden states$) \in \mathcal{D}$
    if update condition holds **then**
      for $j = 1$ to $\min\big(\lfloor |\mathcal{D}|/B \rfloor,$ max updates$\big)$ **do**
        Sample mini-batch B from $\mathcal{D}$ with corresponding hidden snapshots
        $f \leftarrow \mathcal{F}_\psi(\mathbf{X}), \quad f' \leftarrow \mathcal{F}_\psi^{\text{tgt}}(\mathbf{X})$
        Target actions: $a' \leftarrow \Pi\big(\mu_{\theta^{\text{tgt}}}(f') + \tilde{\epsilon},\ \tilde{\epsilon} \sim \mathcal{N}(0, \sigma^2)\big)$ ▷ Π projects to simplex
        TD target: $y \leftarrow r + \gamma(1-d)\ \min\big(Q_{\phi_1^{\text{tgt}}}(f', a'),\ Q_{\phi_2^{\text{tgt}}}(f', a')\big)$
        Critic loss: $L_c \leftarrow \frac{1}{|B|} \sum_{(x,a)} \Big[\big(Q_{\phi_1}(f,a) - y\big)^2 + \big(Q_{\phi_2}(f,a) - y\big)^2\Big]$
        Update critics & feature extractor: $\phi_1, \phi_2, \psi \leftarrow \text{Adam}_\beta\big(\nabla L_c\big)$
        if $j \bmod d_\pi = 0$ **then**
          Freeze critics; stop-grad through $\mathcal{F}_\psi$**:** $\tilde{f} \leftarrow \text{sg}\big(\mathcal{F}_\psi(X)\big)$
          Policy loss: $L_\pi \leftarrow -\frac{1}{|B|} \sum_{\tilde{f}}\ Q_{\phi_1}\big(\tilde{f},\ \mu_\theta(\tilde{f})\big)$
          Update actor: $\theta \leftarrow \text{Adam}_\alpha\big(\nabla L_\pi\big)$
          Soft updates: $\phi_i^{\text{tgt}} \leftarrow (1-\tau)\phi_i^{\text{tgt}} + \tau\phi_i$ (for $i = 1, 2$); $\quad \theta^{\text{tgt}} \leftarrow (1-\tau)\theta^{\text{tgt}} + \tau\theta$; $\quad \psi^{\text{tgt}} \leftarrow (1-\tau)\psi^{\text{tgt}} + \tau\psi$
        end if
      end for
    end if
    if $d_t = 1$ **then**
      break
    end if
  end for
end for

2.3.3 Specific Implementation Details (TD3)

Similarly to PPO, the TD3 architeture used is recurrent, and nonstandard. Similar architectural decisions were made in order to make this work. And some other notable improvements were made to the standard TD3 algorithm.

1. Similar hidden-state considerations to those taken when implementing PPO were also taken into account with TD3.
2. TD3 generally employs "online stochastic batching" to sample experiences from its replay buffer. We extend this by making it such that experiences are sampled linearly probabilistically according to how recently they occur. This intends to ensure that learning is accelerated, given that more recent experiences, intuitively, ought to be more informative than older ones. The downside to this is that it can also accelerate un-learning, or catastrophic forgetting.
3. The replay buffer size employed in this algorithm is rather small, with a size of 5×10^4. Generally, replay buffers for TD3 tend to be much larger, but due to the size of the observation matrix (with just one being 30 timesteps * 200+ features), memory constraints forced a smaller number to be used.

2.4 TD3: Architectures

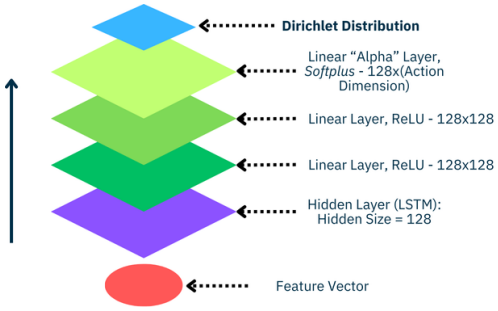

Figure 2.3: Actor Network Structure (TD3)

The TD3 actor networks are very similar to the those of the PPO agent. However, the final layer in TD3 is simply a fully connected layer - the outputs of which are processed by a softmax activation layer to ensure simplex constraints.

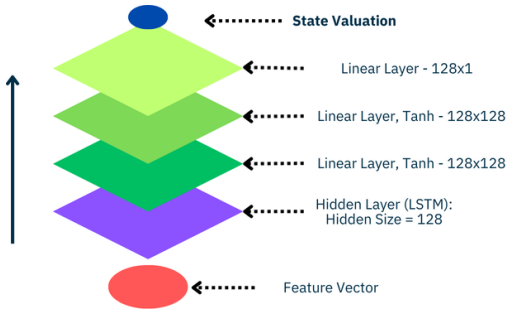

Figure 2.4: Critic Network Structure (TD3)

The TD3 critics are also very similar to that of PPO, the only thing thing differing being the number of neurons in each layer, wiht 512 in the first fully connected layer, and 384 in the the second. This is also the case for the TD3 actor.

3 Feature Extractors

Algorithm 3 Feature Extractor $\mathcal{F}_\psi$ (LSTM or CNN)

Require: Extractor type $\tau \in \{\text{LSTM}, \text{CNN}\}$, parameters ψ; optional projection head $g(\cdot)$
Require: For LSTM: initial states $h_0, c_0 \in \mathbb{R}^{\text{layers} \times \text{batch} \times \text{hidden}}$
for each timestep t **do**
  Receive observation matrix $\mathbf{X}_t$
  if $\tau = \text{LSTM}$ **then**
    $(Y_t, (h_t, c_t)) \leftarrow \text{LSTM}_\psi(\mathbf{X}_t, (h_{t-1}, c_{t-1}))$
  else ▷ $\tau = \text{CNN}$
    $Y_t \leftarrow \text{CNN}_\psi(\mathbf{X}_t)$
    $f_t \leftarrow \text{AAP}(Y_t)$ ▷ adaptive avg pool
  end if
  $f_t \leftarrow g(f_t)$ ▷ e.g., some number of linear and/or normalisation layers and activations
  Pass f_t to appropriate RL algorithm.
end for

3.1 LSTM

The LSTM Feature Extractor, mentioned in the prior section, functions similarly to that implemented by Zou et al. (2024). It iterates through the observation matrix $\mathbf{X}_t$ until it has fully passed through. The last hidden state after the final iteration is passed into linear layers and output as a final feature vector, f_t.

3.1.1 Argument for Statelessness

It should be noted that the LSTM feature extractor is stateless with respsect to each timestep of the environment. The extractor maintains the hidden state whilst iterating each observation $\mathbf{o}_i$, but each time the environment is stepped, the built-up hidden state is reset - emulating the one-day feature extractor implemented by (Zou et al. 2024). The reasoning for this is as follows - presume the feature extractor retains the hidden state at each timestep:

1. If there exists an observation matrix of shape [T, F], with T observation vectors, each of shape F. Recall that the most recent observation vector possesses data up until (and including) time T.
2. At timestep t, the feature extractor then processes this data and ends up with hidden state h_T.
3. If you pass h_T into the feature extractor to process the next [1:T+1, F] matrix, then you inadvertently include "future" market data for the first timestep, because h_T semantically encodes data from $\text{F}_{0:30}$. Thus, F_1, the first vector from 1:T+1, would be processed with a hidden state containing future meaning at the next environment timestep.

Arguably, this forces the construction to learn shorter-time patterns, or limits it to learning time patterns of length T, however, for exceedingly volatile market data, this could be beneficial.

3.1.2 Structure

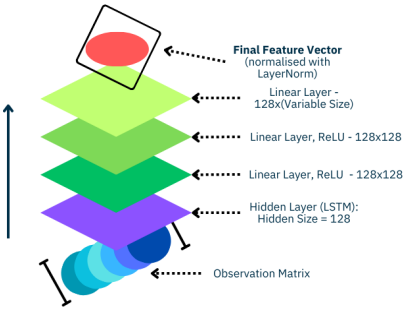

Figure 3.1: LSTM Feature Extractor Network Structure

The LSTM feature extractor is shown above. It operates with a single-layer LSTM module to capture temporal dependencies, followed by three fully connected layers for further refinement. The first two fully connected layers incorporate a ReLU activation function for nonlinearity purposes (as opposed to Tanh which is known to lead to vanishing gradient problems). The final output is a feature vector, the size of which was varied in experimentation. A final layernorm is applied to the feature vector - this is insipred by the normalisation layer included in transformers, which intends to prevent internal covariate shift - i.e., a change in the distribution of activations in layers during training. Preventing this can speed up convergence and stabilise learning. LSTMs intend to mitigate RNNs struggle with long-term memories by providing two types of memory - *long* and *short*. It utilises 'gates' to manage the long-term memory:

1. **A Forget Gate**: dictating how much of the long-term memory should be forgotten.
2. **An Input Gate**: dictating what the input should contribute to the long-term memory.
3. **An Output Gate**: dictating what the new input should contribute to the short-term memory.

These gates intend to prevent the vanishing/exploding gradient issue by preserving the error signals back through time, given that there are multiple routes for the error to propagate.

3.1.3 Specifc Implementation Details

Some specific implementation details should be noted:

1. In this neural network, the forget gate bias is initialised to 2. Given that this gate operates using a sigmoid function, it means that the gate initially starts with an output close to 1 and therefore should remember more at each step (as values closer to 0 mean "forget more"). Backpropagation through time means that the gradients flowing backward in time would decay slower (as they are affected by the gate value) and thus longer-term dependences may become more learnable, at least early in training.
2. The learning rate of the extractor (as shown later in the parameter table) is a fixed multiple (2) of the actor and critic learning rates. During experimentation, the extractor was shown to have lower gradients. While this does not fix that issue, it increases the effective step size, making the extractor learn faster even if the gradient signal is comparably small.

3.2 CNN

Convolutional Neural Networks (CNNs) should provide a reasonable comparison to the LSTM given its superiority in (Jiang et al. 2017)'s use. Convolutional Neural networks are popular among visual applications, given that their convolutional layers are translationally-equivariant - i.e., if the input is shifted, the output feature map is also shifted - allowing the CNN to detect patters regardless of their position. This, combined with the fact that convolutional layers use what is called a *convolutional kernel* or *filter* gives them an advantage over more standard multi-layer perceptrons (MLPs). MLPs would be forced to flatten input data, meaning that they would lose 2D relationships between pixels. CNNs instead preserve multidimensional structure and are able to learn local patterns with these *kernels*. CNNs also reuse the same small kernel across the image, which helps to reduce parameters required, which would reduce overfitting given that there are more degrees of freedom in the CNN model.

3.2.1 Structure

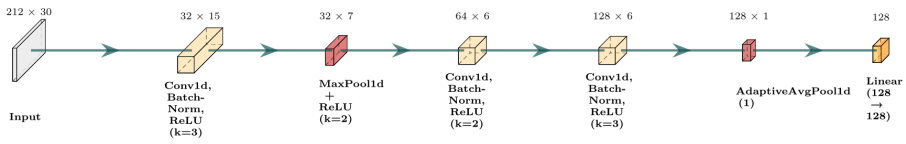

Figure 3.2: CNN Feature Extractor Network Structure

The CNN Feature extractor draws its inspiration from AlexNet - a revolutionary Convolutional Neural Network architecture that was trained to classify images in the ImageNet LSVRC-2010 contest (Krizhevsky et al. 2012). Due to the much lower dimensionality of the input vector compared to AlexNet images, only three convolutional layers were judged to be sufficient for the final layer to have a large receptive field. A Batch-Normalisation function is implemented after each

convolutional layer to stabilise learning and a ReLU to provide nonlinearity. The MaxPool following the first Convolutional Layer helps to shrink temporal size, given that the convolutions occur across the time dimension. This also means that the network can become invariant to small shifts and, in theory, more stable. The second convolutional layer doubles the number of channels, which might help the network to learn short-term patterns, and the third increases receptive field further whilst ideally being able to learn even longer term patterns. By this point, Given our architecture (Conv1: $K = 3, S = 2$; MaxPool: $K = 2, S = 2$; Conv2: $K = 2, S = 1$; Conv3: $K = 3, S = 1$ with $P = 1$), the receptive field for each neuron in the third layer is:

$$R_{\text{final}} = 17,$$

so each output unit of the last convolution covers a temporal window of 17 input steps.

This relies on the following formula, where the receptive field R at layer ℓ is computed recursively as

$$R_\ell = R_{\ell-1} + (K_\ell - 1)\, J_{\ell-1}, \qquad J_\ell = J_{\ell-1} \cdot S_\ell,$$

with $R_0 = 1$, $J_0 = 1$, where K_ℓ and S_ℓ denote the kernel size and stride of layer ℓ.

The adaptive average pool averages values for each channel - meaning that the outputs of each kernel used in the third convolutional layer are *summarized*. This is followed by a fully connected layer, which aim to serve a similar purpose to AlexNet's dense layers, which provides more flexibility in being able to mix outputs from different averaged channels from the adaptive average pool layer.

3.3 End-to-end Training

From a higher-level perspective, the feature extractor combined with an RL algorithm can be viewed as an encoder - i.e., one that processes some multi-dimensional input and outputs an embedding (in this case, a portfolio allocation). As such, it makes theoretical sense to train this model in an end-to-end fashion, with the feature extractors having errors backpropagated to them with the goal of providing feature representations that are beneficial to both the actor and critic(s).

3.3.1 PPO

In PPO, the extractor is encouraged to generate features that are beneficial to both the actor and critic. It aims to minimise the following loss:

$$L \leftarrow L_{\text{actor}} + \tfrac{1}{2} L_{\text{critic}} - c_{\text{ent}} H_b$$

3.3.2 TD3

In TD3, things are slightly different. The feature extractor is critic-owned. This is because:

1. The actor and critic in TD3 have different learning schedules - attempting to optimise the feature extractor with both might introduce instability and would be exceedingly complicated.

2. The actor loss is implicitly linked to the critic - the actor aims to produce actions that maximise the critic's Q-valuation, whereas the critic aims to minimise the mean temporal difference error for both critics - given this, it is clear to see that the critic holds a level of priority over the actor and therefore (for stability purposes) the feature extractor aims to produce representations that benefit the critic. Optimising using the actor's greedy maximisation goal instead may force the critic to produce high valuations, leading to the same problem of overestimation bias that TD3 sought to solve.

4 Experimental Design

4.1 Dataset

Three separate portfolios, from which 30 stocks are selected, are assessed. The model is trained and evaluated on historical stock data. The stocks used are denoted in the appendix, in Tables A.1, A.2, A.3.

The dates range from the period of 01/01/2009 to the most recent date available. At the time of writing, this is 10/09/2025. This is on a daily time scale as opposed to hourly, since longer time scales are generally less volatile. This therefore yields ~4000 days of data to train on, but when matching and dealing with missing data, some portfolios end up having at least close to ~3800 days of data.

Figure 4.1: Train/Validation/Test Data Splits

To simulate these interactions, a custom time-series environment was implemented. The environment is designed to expose market state observations, execute actions in the form of portfolio allocations, apply transaction costs, update the portfolio, and compute rewards based on selected reward functions. It also includes visual rendering capabilities.

4.1.1 Sliding Windows

In most RL projects, 4000 timesteps of data are not nearly enough to see meaningful progress. Agents are often required to be trained for hundreds of thousands of timesteps. Therefore, to provide the agent with enough data to train, I decided to use overlapping windows of the training data as episodes. This is illustrated below:

Figure 4.2: Sliding Window Methodology

To determine the number of windows, the following calculations were performed:

1. Set the training episode length:

$$L_e = \left\lfloor \frac{N}{2} \right\rfloor$$

   where N is the size of the training data.

2. Set the timestep shift:

$$\Delta_t = \left\lfloor \frac{L_e}{10} \right\rfloor$$

3. Calculate the number of windows W using:

$$W = \frac{N - L_e}{\Delta_t} + 1$$

It should be noted that the overlapping windows are not sequentially stacked. This was done to reduce temporal correlation between samples and to increase robustness - specifically in the case of TD3, which stores all previous experiences.

The choice of $L_e = \left\lfloor \frac{N}{2} \right\rfloor$ ensures that each training episode covers at least half of the training data. Setting $\Delta_t = \left\lfloor \frac{L_e}{10} \right\rfloor$ balances the overlap between episodes while maintaining a sufficient number of timesteps overall.

For consistency, the following naming conventions are used throughout Sections 4 and 5:

- **Hyperparameter Training:** Training PPO agents on the training set (perturbed data), varying the feature embedding size.
- **Validation Evaluation:** The learning curves of the agents being trained are assessed here, using a weighted area-under-the-curve metric.
- **Reward Training:** Re-training agents using the best hyperparameters on the combined training and validation sets. For each reward function type (e.g., standard logarithmic returns, CVaR, or differential Sharpe ratio), at least one agent is trained, each reward function can be analysed. To account for variability due to random initialisation, each reward function is evaluated across multiple base seeds (1, 3, 5, 7 and 9).
- **Reward Testing:** Final testing on a held-out, unperturbed test set using deterministic policy sampling for both algorithms.

For clarity, it should also be noted that "epoch" in this paper will refer to a single pass through all windows of the training data. For the hyperparameter tuning, there are 11 training episodes per epoch. For the reward tests, there are 16.

4.1.2 Noise and Seeds

An important component of this project - one that distinguishes it from a purely supervised learning approach - is the introduction of stochastic perturbations to the training data. Following the methodology of Liang et al. (2018), each episode's data is perturbed with Gaussian noise sampled from $N(0, 0.01)$ before the episode begins. This value was chosen according to early experimentation, where higher levels tended to unrealistically distort features, but lower levels proved to provide not much of a challenge.

I extend this by ensuring that the noise takes into account the standard deviations of each market data feature (closing prices, EMA, etc.). This ensures that features are proportionally perturbed according to their respective values. The noise is sampled once per episode and applied to the raw market data, but it does not produce a uniform shift across time. Instead, each timestep receives a distinct perturbation based on the structure of the sampled noise matrix (equivalent to the size of the training episode data). Furthermore, this noise is seeded to ensure that every episode within the training period receives a different level of noise, thereby ensuring that the market data in every episode is different.

This is structurally similar to *BipedalWalker* (OpenAI 2022), which (in its normal version) adds minor perturbations to the terrain each episode. This formula (defined in 4) also ensures reproducibility. The training and validation environments are initialised as follows:

On the test data, no noise is added. This allows testing agents on real, unperturbed market data and proper observation of agent behaviour.

Algorithm 4 Preprocess Data

Require: Base Seed B, epoch E

for every episode **do**
    Receive episode window i where $i \sim \text{Uniform}\{1, 2, \ldots, W\}$;
    Seed noise generation: seed $q = B + (100 \times i) + E$.
    Apply noise, $N(0, 0.01) \times \sigma_{feature}$, according to q to this episode i for all features
    Calculate new percentage price changes for the new closing prices.
end for
Run the training algorithm(1 or 2)

4.2 Evaluation Metrics

Given that it seemed correct to reward configurations that not only led to high returns on the evaluation set, but stable returns and learning, a normalised area-under-the-curve metric was employed to evaluate the agents' performance over the training set. The normalisation is used to take into account different time-lengths. Naturally, since the curves are not necessarily differentiable all throughout, the trapezoid rule is used to approximate the area.

Algorithm 5 Compute Weighted AUC

Require: Sequence of values $y_{0:T}$; optional abscissa $x_{0:T}$ (defaults to $0, 1, \ldots, T$)

If x not provided: $x \leftarrow [0, 1, \ldots, T]$

Normalise time: $\hat{x}_t = \dfrac{x_t - \min(x)}{\max(x) - \min(x)}$

Compute weights: $w_t \leftarrow \hat{x}_t$

Weighted values: $\tilde{y}_t \leftarrow y_t \cdot w_t$

Area under weighted curve:

$$A = \int \tilde{y}(x)\, dx \quad \approx \quad \text{trapz}(\tilde{y}, x)$$

Normalising constant:

$$Z = \int w(x)\, dx \quad \approx \quad \text{trapz}(w, x)$$

Normalised weighted AUC:

$$\text{AUC} = \begin{cases} A/Z, & Z > 0 \\ 0, & Z = 0 \end{cases}$$

return AUC

4.3 Test Metrics

The metrics used to evaluate the agent's performance on the test set are:

1. **Cumulative Return (CR)**: The total return of a portfolio at the end of

the trading stage and is calculated as:

$$CR = \frac{P_{end} - P_0}{P_0}$$

where P_{end} is the portfolio's final value, and P_0 is the initial value.

2. **Maximum Drawdown (MDD)**: The largest percentage decline from a historical peak in portfolio value over the trading period, reflecting worst-case loss:

$$\text{MDD} = \max_{t \in [0,T]} \left(1 - \frac{P_t}{\max_{\tau \in [0,t]} P_\tau}\right)$$

where P_t is the portfolio value at time t.

3. **Sharpe Ratio (SR)**: This is defined in section 1.1.3:

$$SR = \frac{R_p - R_f}{\sigma_p}$$

Since price movements can often be noisy for a given set of hyperparameters, making visual assessments challenging. To address this, a scoring mechanism was developed to evaluate the performance of each hyperparameter configuration. The score is computed as:

$$\text{score} = \frac{\bar{CR} - \bar{CR}_{\text{random}}}{\bar{MDD}} \times |\bar{SR}| \tag{4.1}$$

4.4 Network Parameters

The following parameters were used:

Parameter	Value
γ	0.99
GAE Lambda	0.98
Clip Parameter	0.2
Batch Size	48
Learning Rate (α)	0.0003 *(extractor uses 0.0006)*
FC1 Units	128
FC2 Units	128
Feature Output Size	*varied*
Epochs	4
Entropy Coefficient	0.01
Actor Noise	0
Normalize Advantages	True
Learning Frequency	48
Strategy	PPOLSTM
Hidden State Size	512

Table 4.1: Training parameters of PPOLSTM agent.

Parameter	Value
γ	0.99
Actor Learning Rate (α)	0.001
Critic Learning Rate (β)	0.001
Soft Update Coefficient (τ)	0.005
Feature Output Size	*varied*
Batch Size	100
FC1 Units	512
FC2 Units	384
Actor Noise	0.01
Target Noise	0.01
Actor Delay (d_π)	2
Hidden State Size	512

Table 4.2: Training parameters of TD3 agent.

5 Results

6 Conclusion

...

In future, more rigorous hyperparameter tuning should be investigated. Time constraints limited this, along with limited data. Furthermore, data normalisation methods should be explored more. A potential method to be used in the future may be a rolling z-score mechamism, however, tuning this would be complicated, given that one would have to figure out the horizon to normalize over - further, it is not clear what effect this might have on certain indicators' ability to convey information. Instead, a pre-norm layernorm could be applied to the feature extractor, which would also include learnable parameters.

References

Cui, Tianxiang, Shusheng Ding, Huan Jin, and Yongmin Zhang (2023). "Portfolio constructions in cryptocurrency market: A CVaR-based deep reinforcement learning approach". In: *Economic Modelling* 119, p. 106078.

Fujimoto, S., H. Hoof, and D. Meger (2018). "Addressing function approximation error in actor-critic methods". In: *International Conference on Machine Learning*, pp. 1587–1596.

Hu, Y. J. and S. J. Lin (2019). "Deep reinforcement learning for optimizing finance portfolio management". In: *2019 Amity International Conference on Artificial Intelligence (AICAI)*. IEEE, pp. 14–20.

Hull, J. C. (2021). *Options, Futures, and Other Derivatives*.

Jiang, Z., D. Xu, and J. Liang (2017). *A deep reinforcement learning framework for the financial portfolio management problem*. arXiv preprint arXiv:1706.10059.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton (2012). "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems* 25.

Liang, Z., H. Chen, J. Zhu, K. Jiang, and Y. Li (2018). *Adversarial deep reinforcement learning in portfolio management*. arXiv preprint arXiv:1808.09940.

Lillicrap, T.P., J.J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra (2016). "Continuous control with deep reinforcement learning". In: *arXiv.org*. Accessed: 29 December 2024. URL: `https://arxiv.org/abs/1509.02971`.

MathWorks (2024). *Estimate VaR for Equity Portfolio Using Parametric Methods*. `https://www.mathworks.com/help/risk/estimate-var-using-parametric-methods.html`. Accessed 21 October 2024.

Moody, J. and M. Saffell (1998). "Reinforcement learning for trading". In: *Advances in Neural Information Processing Systems*. Vol. 11.

OpenAI (2022). *BipedalWalker Environment: Terrain Generation Code*. `https://github.com/openai/gym/blob/master/gym/envs/box2d/bipedal_walker.py`. Function _generate_terrain(), Accessed: 1 May 2025.

Schulman, J., S. Levine, P. Abbeel, M. Jordan, and P. Moritz (June 2015). "Trust region policy optimization". In: *International conference on machine learning*. PMLR, pp. 1889–1897.

Schulman, J., P. Moritz, S. Levine, M. Jordan, and P. Abbeel (2015). *High-dimensional continuous control using generalized advantage estimation*. arXiv preprint arXiv:1506.02438.

Schulman, J., F. Wolski, P. Dhariwal, A. Radford, and O. Klimov (2017). *Proximal policy optimization algorithms*. arXiv preprint arXiv:1707.06347.

Soleymani, F. and E. Paquet (2020). "Financial portfolio optimization with online deep reinforcement learning and restricted stacked autoencoder—DeepBreath". In: *Expert Systems with Applications* 156, p. 113456.

Zou, J., J. Lou, B. Wang, and S. Liu (2024). "A novel deep reinforcement learning based automated stock trading system using cascaded LSTM networks". In: *Expert Systems with Applications* 242, p. 122801.

A Appendices

A.1 Appendix A: Extra Tables

Index	Stock Tickers
Dow Jones Industrial Average	MMM
	AXP
	AAPL
	BA
	CAT
	CVX
	CSCO
	KO
	NVDA
	XOM
	GS
	HD
	INTC
	IBM
	JNJ
	JPM
	MCD
	MRK
	MSFT
	NKE
	PFE
	PG
	RTX
	TRV
	UNH
	VZ
	V
	WBA
	WMT
	DIS

Table A.1: Dow Jones Industrial Average Constituents

Index	Stock Tickers
SSE 50	600028.SS
	600030.SS
	600031.SS
	600036.SS
	600048.SS
	600050.SS
	600150.SS
	600276.SS
	600309.SS
	600406.SS
	600519.SS
	600690.SS
	600760.SS
	600809.SS
	600887.SS
	600900.SS
	601088.SS
	601166.SS
	601318.SS
	601328.SS
	601398.SS
	601600.SS
	601601.SS
	601628.SS
	601668.SS
	601857.SS
	601888.SS
	601899.SS
	601919.SS
	601988.SS

Table A.2: SSE 50 Constituents

Index	Stock Tickers
FTSE 100	RMV.L
	IMI.L
	ICG.L
	PRU.L
	AAL.L
	KGF.L
	DGE.L
	PCT.L
	FRES.L
	UTG.L
	SDR.L
	WTB.L
	HSX.L
	LMP.L
	VOD.L
	HSBA.L
	SGRO.L
	IMB.L
	MRO.L
	BA.L
	CNA.L
	AV.L
	TATE.L
	BARC.L
	BEZ.L
	PHNX.L
	AHT.L
	BP.L
	ANTO.L
	RTO.L

Table A.3: FTSE 100 Constituents