The use of Reinforcement Learning in Algorithmic Trading

The Impact of Function Approximation Methods on Model Performance

Robert Mertens[1]

Supervisor(s): Neil Yorke-Smith[1], Amin Sharifi Kolarijani[2], Antonis Papapantoleon[1]

[1]**EEMCS, Delft University of Technology, The Netherlands**
[2]**Mechanical Engineering, Delft University of Technology, The Netherlands**

Name of the student: Robert Mertens
Final project course: CSE3000 Research Project
Thesis committee: Neil Yorke-Smith, Amin Sharifi Kolarijani, Antonis Papapantoleon, Julia Olkhovskaya

Abstract

The application of Reinforcement Learning (RL) to algorithmic trading offers the potential for developing adaptive, profitable strategies, yet its success is highly dependent on the methods used for function approximation. This thesis systematically investigates how different function approximation methods impact the performance and generalization of an RL-based agent trading in the volatile EUR/USD Forex market. The study evaluates a range of architectural choices, including network size and shape, the division of parameters between actor and critic networks, various activation functions, and different feature extraction techniques. While no tested configuration achieved consistent profitability on unseen data, the findings highlight a critical trade-off between a model's learning capacity and its ability to generalize. Notably, the bounded Sigmoid activation function showed superior performance on evaluation data compared to the unbounded ReLU, suggesting it provides a form of implicit regularization that promotes robustness. In contrast, variations in network shape and the actor-critic parameter division did not significantly affect on out-of-sample performance, indicating that other experimental factors may have been a bottleneck.

1 Introduction

Artificial Intelligence (AI) has become increasingly prevalent in financial markets, with its ability to analyze vast datasets and identify complex patterns exceeding traditional models [1, 2]. Reinforcement Learning (RL), a subfield of AI, is particularly well-suited for trading, as it allows an agent to learn optimal decision-making strategies through direct interaction with the market environment [3, 4]. Despite this potential, the widespread adoption of RL in the volatile Forex market remains limited. Key challenges include the noisy nature of financial data, the risk of overfitting, and the difficulty of generalizing learned strategies to new market conditions. The performance of RL agents is critically dependent on the function approximators used to represent value and policy functions, yet there is a gap in understanding how specific choices in their design impact trading effectiveness.

1.1 Motivation

While existing research demonstrates that RL models can outperform traditional methods [5, 3], many studies do not isolate the comparative impact of different function approximation techniques. The effects of model complexity on convergence and stability have been studied in broader RL contexts, but their specific implications within the dynamic Forex environment remain underexplored [3, 6]. This paper addresses this gap by investigating the central research question: "**How do different function approximation methods impact the performance of a Reinforcement Learning model trading in the Forex Market?**" We explore this through the following sub-questions:

- How do network architecture choices, specifically the size, shape, and division of parameters between actor and critic, affect model performance and learning stability?
- What is the impact of different activation functions on training speed and the ability to generalize to unseen data?
- How does the method of feature extraction—handcrafted technical indicators versus automated extraction via a Convolutional Neural Network (CNN)—influence trading performance?
- How does L2 regularization (weight decay) affect the model's ability to learn and avoid overfitting?

1.2 Structure of the paper

We begin by providing background on Reinforcement Learning and the Forex market. We then detail our methodology, custom trading environment, and experimental setup. Finally, we present and discuss the results, concluding with a summary of our findings and directions for future research.

1.3 Existing work

The exploration of Reinforcement Learning (RL) in financial markets, particularly in the Forex domain, has garnered significant attention in recent literature. Several studies have contributed to our understanding of RL applications, yet specific unanswered questions remain, particularly concerning the choice of function approximators and their effects on trading performance.

Regarding network architecture, Liu et al. highlight the challenges that RL models face in adapting to the volatile nature of financial markets. They note that the architecture of neural networks involved in RL critically influences the model's ability to stabilize and converge during learning processes [3]. They emphasize that while tailored architectures could yield better performance, the intricate balance between flexibility and overfitting remains a contentious point, suggesting that more systematic approaches are needed to explore architectural variations in depth. Furthermore, Ashish's work illustrates this complexity by comparing algorithms like Proximal Policy Optimization (PPO) and Advantage Actor-Critic (A2C), yet it stops short of offering definitive conclusions on the ideal architectural configurations tailored for Forex trading [5].

Secondly, in relation to the impact of activation functions on training speed and generalization, Meng and Khushi provide a comprehensive review of the varying activation functions employed within RL frameworks. They acknowledge their critical role in convergence behavior and training efficiency but note that empirical comparisons in financial contexts remain sparse [2]. The existing literature often fails to align theoretical insights with practical applications in trading environments, indicating a pressing need for targeted studies that explicitly link activation function choices with performance metrics in Forex trading.

When examining the method of feature extraction—handcrafted technical indicators versus automated approaches like Convolutional Neural Networks (CNNs)—the contrasting efficacy of these methods is underscored in the literature. The work by Sun et al. emphasizes the ongoing discussion regarding the predictive capabilities of classical technical indicators versus those learned through deep learning techniques [4]. Despite the enthusiasm for CNNs' promise in capturing complex market signals, comprehensive evaluations directly juxtaposing these methods' effectiveness in Forex-specific contexts remain limited, calling for research that bridges this gap.

Lastly, the effect of L2 regularization on learning and overfitting is critically assessed across studies. Both Ashish and Liu et al. acknowledge the risk of overfitting in RL models applied to the noisy financial data characteristic of Forex markets [3, 5]. However, while these studies advocate for regularization techniques as solutions, they often lack robust empirical backing that illustrates how these methods alter the learning trajectories and performance metrics in real trading scenarios. The debate on the optimal application of L2 regularization in preventing overfitting without compromising learning opportunity persists, indicating a need for deeper investigation focused on Forex modeling.

2 Background

2.1 Reinforcement Learning

Reinforcement learning (RL) is a machine learning paradigm where an agent learns to make optimal decisions through trial-and-error, guided by reward signals from an interactive environment. The underlying mathematical framework for RL is often based on Markov decision processes (MDPs), which provide a structured way to formalize decision-making scenarios where outcomes are partly random and partly under the control of a decision-maker [7, 8]. In an MDP, the process is defined by states, actions, and transition probabilities, enabling the description of the environment's dynamics and the modeling of decision-making problems [9]. These components work together to facilitate the agent's learning process where, through numerous interactions, the agent aims to discover an optimal policy that maximizes a discounted sum of rewards.

Algorithmic trading can be effectively formulated as a Reinforcement Learning problem. In trading, for example, an RL agent can learn to adjust its strategies based on the performance in previous trades while considering the temporal aspect of market movements, which aligns well with the learn-explore-decision cycle of RL [8, 10]. Moreover, the capability of RL to incorporate and handle delayed rewards makes it well-suited for such applications. [11].

2.2 Forex

The foreign exchange (Forex) market is a decentralized global marketplace where currencies are traded. It operates 24 hours a day, 5 days a week, facilitating the exchange of currencies among participants, including banks, financial institutions, corporations, and individual traders. This market is essential for international trade, enabling businesses to convert currencies for transactions and investments abroad [12, 13, 6]. The Forex market is considered the largest and most liquid financial market globally, with an average daily trading volume exceeding \$6 trillion, which demonstrates its significance in facilitating global economic activities [14].

Trades in the Forex market occur directly between parties through electronic trading platforms and over-the-counter (OTC) markets [14]. This decentralized nature allows for continuous operations and flexibility in trading. Trading involves the exchange of one currency for another, denoted as currency pairs (e.g., EUR/USD, GBP/JPY). Every transaction involves buying one currency while simultaneously selling another, reflecting the relative value of the currencies involved [12].

Trading volume in Forex refers to the total quantity of a currency being traded over a specific period, impacting price stability and indicating market activity levels. Higher volume can signal strength in market movements, while lower volume could indicate potential for volatility and slippage [14]. However, due to the decentralized nature of Forex, volume is notoriously difficult to measure [15, 16].

The ask price is the price at which a trader can buy a currency, while the bid price is the price at which a trader can sell. The difference between these prices is known as the spread [17]. These prices are typically grouped into time frames, which can be visualized as candlesticks that show the open, high, low, and close values for both ask and bid prices. Time frames may range from minutes to days, allowing for a variety of trading strategies [18].

Traders can adopt buy (long) strategies, expecting currency appreciation, or sell (short) strategies to take advantage of expected depreciation. Holding a currency involves maintaining positions for potential future gains or to mitigate risks [17].

2.3 Function Approximation

Function approximation refers to the process of estimating complex input-output relationships, especially in high-dimensional spaces where exact computation is infeasible. In RL, it is mainly used to approximate value functions, which guide decision-making by estimating expected rewards for different states and actions [19, 20]. Several methods have emerged for function approximation, each with distinct theoretical strengths and weaknesses.

Tabular Methods directly associate each state or state-action pair with a single value. They are particularly effective in small state spaces where the number of possible states allows for a complete enumeration. However, their scalability is severely limited, rendering them inefficient and impractical as the dimensionality increases [19].

Linear Function Approximation solves the scalability problem with tabular methods by predicting values using a linear combination of features extracted from the state space. However, it struggles with capturing complex relationships in non-linear datasets, which limits its applicability in many real-world scenarios [19].

Non-linear Function Approximation can model more intricate relationships in the data. However, their increased flexibility can lead to issues such as overfitting, especially in contexts where the amount of training data is limited relative to the complexity of the model [19].

Deep Reinforcement Learning Subset of non-linear function approximation methods, using neural networks to approximate value functions or policies. Deep reinforcement learning excels in handling high-dimensional inputs, and has achieved remarkable success in various applications [20]. However, the complexity of training deep networks poses significant challenges, including the need for substantial computational resources and the risk of instability and poor generalization [20, 21].

3 Methodology

3.1 Data Sourcing

Historical tick data for the EUR/USD pair was sourced from Dukascopy, a provider known for high-resolution data in academic research [22]. Initial downloads of minute-interval data showed significant gaps (20% missing rows) [23].

This prompted us to consider other data sources including Yahoo Finance, Capital.com, and the European Central Bank (ECB). However, these sources were either lacking the option for finer granularity data, only providing time frames at the resolution of hours or days, or were not deemed as reliable [24]. Since the performance of predictive models in Forex trading is strongly tied to data resolution and accuracy [22], we decided to reconsider Dukascopy.

Finally, we developed a custom pipeline to request raw tick data via the Dukascopy API and downsample it into one-hour OHLCV candles. This process reduced missing data to approximately 2%, which we left unfilled to avoid introducing imputation-related artifacts [25, 26].

3.2 Feature Engineering

The raw forex OHLCV data is difficult for the reinforcement learning agent to interpret. To enhance the agent's ability to perceive market conditions and the state of its portfolio, both static, pre-calculated market indicators and dynamic, step-dependent agent features are generated. Two distinct classes, `FeatureEngineer` and `StepwiseFeatureEngineer`, are employed for this purpose [27]. The overall data flow and the roles of these components are illustrated in fig. 1.

Static Market Features (`FeatureEngineer`)

The `FeatureEngineer` class batch-processes the entire historical dataset before the simulation begins. It applies a series of transformation functions sequentially to the raw OHLCV data to generate a static `market_features` table. This table includes pre-calculated technical indicators, such as the Relative Strength Index (RSI), which do not change during the agent's interaction with the environment. This entire process is completed once during initialization.

Dynamic State Features (`StepwiseFeatureEngineer`)

In contrast, the `StepwiseFeatureEngineer` generates features dynamically at each step of the simulation. This is necessary for features that depend on the agent's actions, which cannot be pre-calculated. For instance, it calculates the percentage of the agent's portfolio held in cash at the current moment. This function is executed at every time step within the environment, providing the agent with up-to-date information reflecting its most recent decisions.

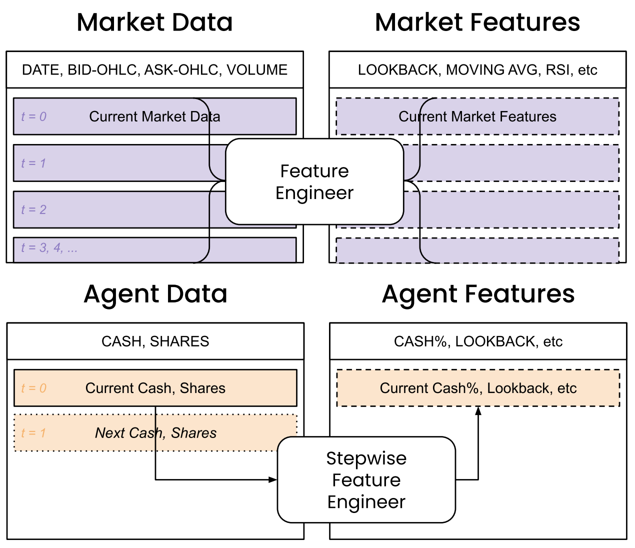


Figure 1: Graph showing the the 4 main tables used by the simulated Forex Environment. The `FeatureEngineer` generates the market features all at once during environment initialization. The `StepwiseFeatureEngineer` generates each row of agent features separately, during the trading simulation.

3.3 Forex Environment

To train and evaluate the reinforcement learning agent, a custom Gymnasium environment was developed [27]. It is designed to model the core mechanics of trading in a leveraged foreign exchange market, including transaction costs, bid-ask spreads, and portfolio dynamics.

State and Observation Space

The complete state of the environment includes the entire history of market data (OHLCV prices) and the agent's corresponding data (cash and shares held). However, current market and agent state is insufficient for the agent to make a proper decision. In an attempt to adhere to the Markov property, the observation provided to the agent may include multiple features that contain information about past timesteps. The feature vector fed to the agent at each step t is a concatenation of two components:

1. **Market Features**: The static, pre-calculated market features for time t (e.g., RSI, historical lookbacks), generated by the `FeatureEngineer`.
2. **Agent Features**: The dynamic, state-dependent features for time t (e.g., cash-to-equity ratio), generated by the `StepwiseFeatureEngineer`.

Action Space

A significant design decision was to define the agent's action not as a discrete buy or sell order, but as a desired target exposure. The action space can be configured as either:

- **Continuous**: A single float value in the range $[-1.0, 1.0]$, where -1.0 represents a 100% short position, 1.0 a 100% long position, and 0.0 a neutral (all cash) position.

- **Discrete**: A set of integers that are mapped to evenly spaced exposure levels (e.g., $[-1.0, -0.5, 0.0, 0.5, 1.0]$, for n_actions $= 5$)

The range of possible actions is set to be $[-1.0, 1.0]$ by default, but can be any subset thereof. For example one could disallow going short by setting an action range of $[0.0, 1.0]$. For this paper the action range is kept as the default, as any modifications would be out of scope.

Episode Life-cycle
An episode is defined as a single simulation run. An episode begins when the environment is initialized or reset and proceeds step-by-step through the historical dataset. The episode concludes under one of two conditions:

1. **Termination**: The agent's total equity falls to or below zero. This represents bankruptcy and ends the episode prematurely.
2. **Truncation**: The agent successfully navigates the entire dataset, reaching the final time step.

Upon conclusion, the environment logs the complete history of the episode—including all market data, features, agent actions, and resulting portfolio values—into a pandas DataFrame.

Market Assumptions
At each step, the environment receives the agent's target exposure. It then calculates the required trade volume to shift the current portfolio allocation to this new target. The simulation executes this trade based on the following rules to model a realistic market:

1. **Transaction Costs**: A percentage-based fee is applied to every trade, reducing the cash proceeds from a sale or increasing the cost of a purchase.
2. **Leverage Constraint**: The environment enforces a hard leverage limit of 100%. The agent cannot buy assets worth more than its available cash, nor can it take on a short position that it cannot back with its current equity, preventing it from incurring debt or facing a margin call within the simulation's rules.
3. **Bid-Ask Spread**: Buy orders are executed at the higher `ask` price, and sell orders at the lower `bid` price, accurately reflecting the cost of crossing the spread.
4. **Zero Market Impact**: Market prices evolve independently of agent actions. This is a common simplification in algorithmic trading simulations and reflects a "price-taking" agent [28].

3.4 Trading Logic

This section describes the trading logic used in the simulation. We use commonly accepted symbols for the following formulae, but in case there is any doubt a list of symbols and exact descriptions are provided in appendix B.1.

Setup
The entire trading process starts upon environment reset. Starting capital is cash-only, so $E_0 = C_0$, and $S_0 = 0$. Since we assume Zero Market Impact, this initial capital does not have any impact on the simulation. Everything onward is just relative. The agent makes a trade at the start of each time period. For simplicity we assume that no slippage occurs and trades are executed "instantly". Each trade updates the cash and shares, which will then remain constant until the next trade. The components of trade execution logic are depicted in fig. 2.

For the determination of position value we uphold Mark-to-Market (MTM), this means valuing open positions based on current market prices rather than historical purchase costs. If the number of shares S_t is positive, the market price is the bid price (what others are willing to pay). If S_t is negative, the ask price is used (the price to buy back the asset).

For each timeframe there are 5 key points: Open, High, Low, Close, and Pre-action (in that order). Pre-action occurs at the open of the next timeframe, but before the trade is executed, using the cash and shares from the previous timestep. For calculating the exposure, equity and position value, we focus on their value right before a trade is executed. This way, the impact of the trade on their respective values can be most accurately observed.

Action $a_t \in \mathcal{A} \rightarrow$ Target Exposure $x_t \in \mathcal{X}$

First, each raw action a_t provided by the agent is mapped to a target exposure x_t. We abbreviate n_actions as N_a.

$$\begin{aligned} x_t &= a_t & a_t &\in [-1, 1] & N_a &= 0 \\ x_t &= -1 + \frac{2 \cdot a_t}{N_a - 1} & a_t &\in \{0, 1, \ldots, N_a - 1\} & N_a &> 0 \end{aligned}$$

Determining Trade Size

Based on the target exposure x_t a number of shares to trade ΔS_t is determined.

$$\begin{aligned} V_t^\star &= x_t \cdot E_t && \text{Target Position Value} \\ S_t^\star &= \begin{cases} \dfrac{V_t^\star}{P_t^{\text{ask}}} & x_t > 0 \quad \text{(long)} \\ \dfrac{V_t^\star}{P_t^{\text{bid}}} & x_t \le 0 \quad \text{(short)} \end{cases} && \text{Target Shares} \\ \Delta S_t &= S_t^\star - S_{t-1} && \text{Shares to Trade} \end{aligned}$$

Executing buy order

If $\Delta S_t > 0$ we buy shares. We limit the amount of shares bought to a maximum, to avoid going more than 100% long.

$$\begin{aligned} S_t^{\max} &= \frac{C_{t-1}}{(1+\kappa) \cdot P_t^{\text{ask,open}}} \qquad \text{(B.2)} \\ S_b &= \min(\Delta S_t, S_t^{\max}) \\ C_t &= C_{t-1} - S_b \cdot (1+\kappa) \cdot P_t^{\text{ask,open}} \\ S_t &= S_{t-1} + S_b \end{aligned}$$

Executing sell order

If $\Delta S_t < 0$ we sell shares. We limit the amount of shares sold to a maximum, to avoid going more than 100% short.

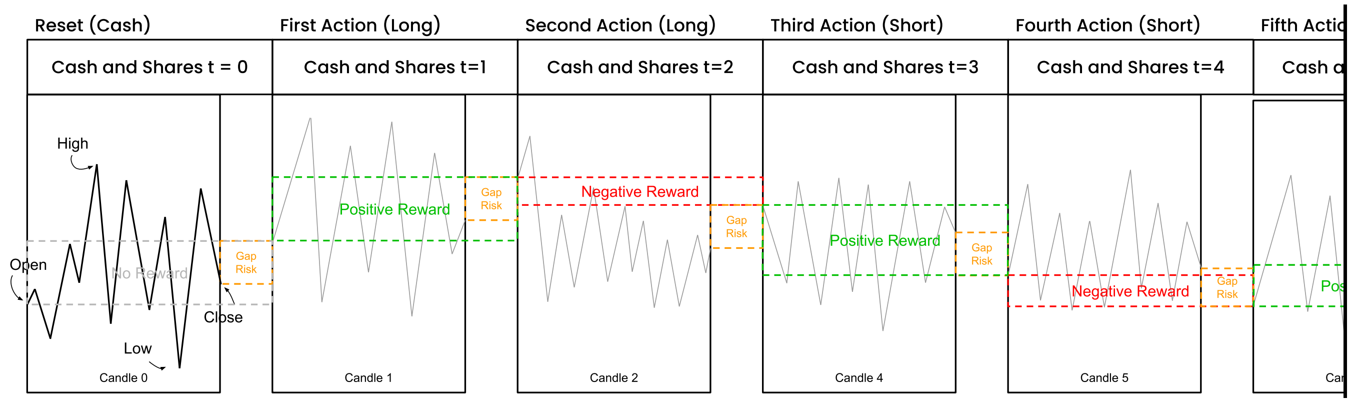


Figure 2: Graph showing the trade execution process. Trades happen at the start of each timestep, starting from the second (first is reset). Each trade updates the cash and shares. The change in equity of a trade is calculated as the difference between the equity right before the trade, and the equity right before the next.

$$\begin{aligned} S_t^{\max,-} &= \frac{C_{t-1} + 2 \cdot S_{t-1} \cdot P_t^{\text{ask,open}}}{2P_t^{\text{ask,open}} - (1-\kappa) \cdot P_t^{\text{bid,open}}} \qquad \text{(B.3)} \\ S_s &= \min(-\Delta S_t, S_t^{\max,-}) \\ C_t &= C_{t-1} + S_s \cdot (1-\kappa) \cdot P_t^{\text{bid}} \\ S_t &= S_{t-1} - S_s. \end{aligned}$$

3.5 Rewards

Standard Rewards

The primary objective of a trading agent for this paper is to maximize its capital. Following the zero market impact assumption, the size of the agent's starting equity has no impact on the simulation of the environment. This implies the agent's objective is to maximize the annualized profit ratio.

To align with this goal, the default reward function is defined as the change in the agent's total equity from the previous time step to the current one.

$$r_t = E_t - E_{t-1}$$

This simple, dense reward structure directly incentivizes profitable actions at every step. Common variants are the difference in log equities, the ratio in equity between timesteps, or adding some heuristic for risk to incentivize the agent to be more cautious in their trade execution (risk-adjusted rewards). A significant downside to most of these simple reward functions however, is their short-sightedness. An action that yields a large short-term reward might lead to worse long-term outcomes.

Dynamic Programming Rewards

To overcome this short-sightedness, a future-aware reward signal was developed. Using dynamic programming, we pre-compute an approximation of the optimal value function $V_t^*(\epsilon)$ over a discretized state space of time and portfolio exposure ϵ. The reward given to the agent is then based on the optimality of its action relative to the computed optimal policy. Specifically, the reward is the difference between the agent's actual one-step return and the return predicted by the optimal value function, $r_t = \log(E_{t+1}/E_t) + V_{t+1}^* - V_t^*(\epsilon_t)$. This incentivizes the agent not just to seek immediate profit, but to take actions that lead to states with higher future potential. A full derivation is laid out in appendix C.

4 Experimental Setup

4.1 Baseline

For each of the following experiments. We start from a baseline, and modify parts of function approximation to determine their impact on model performance. This baseline was developed through combining aspects from different papers, systematic testing with hyperparameters, discussion with peers, and original thought.

Environment setup

Following section 3, we made the following decisions for the baseline with regard to the environment setup:

- Currency Pair: EUR/USD only.
- Data source: tick data sourced from Dukascopy downsampled to 1 hour candles, as described in section 3.1.
- Data split: 5 years of data (2020-2025), split 70% training, 30% evaluation.
- Continuous action space spanning $[-1, 1]$.
- A 0.005% commission (called transaction cost in formulae) on top of the cost of crossing the spread [29].
- Dynamic Programming optimal informed reward function as described in section 3.5.
- Observation space consisting of 16 carefully selected features to capture price momentum, trend, reversals, volatility, agent exposure, and time of day/week [30].
- Trading logic as described in section 3.4.

Model Algorithm

We primarily considered actor-critic methods due to their potential to combine the strengths of both policy-based and value-based reinforcement learning. Two prominent algorithms in this category are Advantage Actor-Critic (A2C) and Soft Actor-Critic (SAC). A2C has been shown to provide more stable and efficient training with lower variance, and has been successfully applied in financial tasks such as stock and portfolio trading [31, 32, 33, 34].

SAC, on the other hand, is an off-policy method that optimizes both expected return and policy entropy, allowing for more exploratory and robust behavior. It has demonstrated state-of-the-art performance in various continuous control tasks, though it has seen less application in trading contexts so far [35, 36, 37]. Ultimately, we selected SAC for our experiments due to its enhanced stability, sample efficiency, strong performance in continuous action spaces, and entropy-regularized objective [36, 37]. The specific hyperparameters used are laid out in appendix A.1.

4.2 Experiments

During training, models are saved every episode. Then we evaluate each of the models deterministically, repeatedly taking the highest probability actions for a single episode on both the training data and the evaluation data. This gives a general idea of model performance across training episodes. For each experiment, models were trained for 50 episodes across 5 seeds in an attempt to provide robust results.

Before evaluating the reinforcement learning agents, we first establish performance benchmarks using several simple, non-learning models. These baselines implement basic, heuristic-based trading strategies, to provide a crucial reference point. We have six experiments.

(1) Network Size This experiment aims to determine the effect of model capacity on performance. We systematically vary both the depth (number of hidden layers) and the width (number of neurons per layer) of the networks. We test depths of 1, 2, 3, and 4 layers, which we term ”shallow,” ”moderate,” ”deep,” and ”very deep,” respectively. For each depth, we test widths of 8, 16, 32, and 64 neurons, termed ”narrow,” ”moderate,” ”wide,” and ”very wide.” This creates a grid of 16 different network sizes, allowing us to observe the trade-offs between underfitting with small networks and potential overfitting or increased training difficulty with larger ones.

(2) Network Shape While network size determines the total number of parameters, the shape—how those parameters are arranged across layers—can also influence the learning dynamics. In this experiment, we test five different network shapes while keeping the total number of learnable parameters N_p approximately constant. The exact shapes are laid out in table 1, Reasoning for these parameters is described in appendix A.2.

Name	**Architecture**	N_p	N_n	$N_{\textbf{flop}}$
shallow	36, 36	2557	73	5041
inv_funnel	19, 29, 44	2572	93	5051
funnel	37, 24, 16	2550	78	5022
flat	28, 28, 28	2577	85	5069
diamond	22, 40, 22	2571	85	5057

Table 1: Model comparison with architecture, parameter count N_p, neurons N_n, and Floating Point Operations N_{flop}

(3) Actor-Critic Parameter Division In the SAC algorithm, the actor and critic have separate networks. This experiment investigates whether performance is sensitive to how the total computational budget (number of parameters) is divided between them. We test five configurations laid out in table 2. All networks have a depth of 2. Calculation of these network widths is described in appendix A.3.

Experiment Name	$W_{\textbf{actor}}$	$W_{\textbf{critic}}$	N_p	$N_{\textbf{flop}}$
No Bias	32	32	4290	8450
Moderate Actor Bias	36	27	4232	8336
Moderate Critic Bias	27	36	4232	8336
Large Actor Bias	41	19	4144	8166
Large Critic Bias	19	41	4144	8166

Table 2: Network division configurations. Each experiment varies the layer widths of the actor W_{actor} and critic W_{critic} to bias parameter allocation, while keeping the total parameter count (N_p) and FLOPs (N_{flop}) roughly constant.

(4) Activation Functions The choice of activation function introduces non-linearity into the network, enabling it to learn complex relationships. While ReLU is a common default, other functions may offer advantages. This experiment compares the performance of the baseline network architecture when using six different activation functions: ReLU, Leaky ReLU, Sigmoid, SiLU (Swish), Tanh, and ELU.

(5) Feature Extraction: Technical Indicators vs. CNN The baseline model using handcrafted technical analysis features is compared against another using a Convolutional Neural Network (CNN) for feature extraction. The CNN alternative uses two convolutional layers to extract cross-feature and temporal patterns from a sliding window of 48 time steps of minimally processed OHLC and volume data. Details on the structure of the CNN are laid out in appendix A.4.

(6) Regularization: Weight Decay To mitigate overfitting, a common regularization technique is weight decay (L2 regularization), which adds a penalty to the loss function proportional to the squared magnitude of the network weights. This experiment investigates the impact of the strength of this regularization on model performance. Five different values for the weight decay coefficient are tested: $10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}$. Note that the baseline does not have weight decay.

5 Results

For each of the models, we focus on two aspects: overall performance as the profit ratio, and trade performance as the total number of trades. A trade is defined as the period during which the agent maintains a position, either long or short, beginning from the moment the position is assumed and ending when the agent transitions to the opposite position. All of the graphs that will be discussed in the following sections are shown in their entirety in appendix E.

(1) Network Size We observe the following aspects: wider networks get the highest profit ratios on the training data, but lose their edge when trading on evaluation data, see fig. 3.

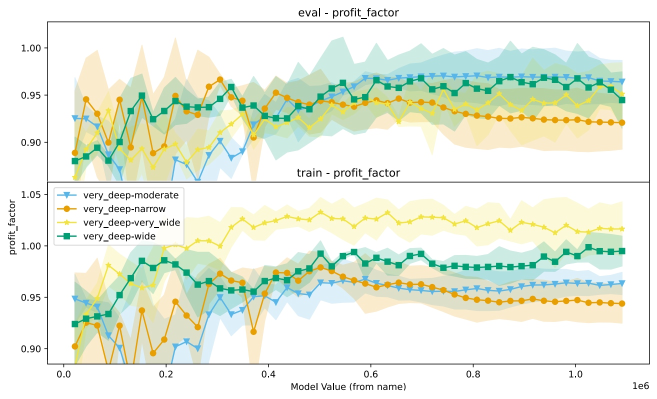


Figure 3: Profit factor across network widths for very deep networks on the evaluation and training environment.

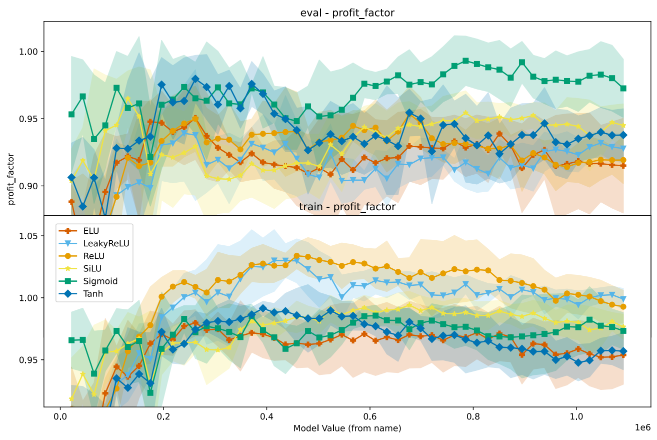


Figure 4: Profit factor across activation functions on the evaluation and training environment.

None of the network sizes was able to get profit on the unseen data. The networks with the highest profit ratio are those that collapse into a single action, which happens most frequently to smaller (shallow or narrow) networks.

(2) Network Shape Network shape did not seem to have a significantly impact either the profit ratio or the number of trades executed, given a fixed number of parameters. All networks performed approximately the same.

(3) Actor-Critic Parameter Division A moderate or large critic bias improves performance on training data, this effect is lost on unseen data. We notice dividing parameters over networks has no significant effect on out of sample performance. We also notice that the stronger the bias the less actions agents take, with some collapsing into single actions.

(4) Activation Functions As shown in fig. 4: (Leaky)ReLU learned the fastest and had the highest profit ratio on training data, whereas models using Sigmoid activation functions had the best performance on evaluation (unseen) data across activation functions, and one of the highest profit factors across all models evaluated.

(5) Feature Extraction: Technical Indicators vs. CNN The agent informed using CNN extracted features collapsed into a single action. This suggests the features extracted were not informative enough for the agent to overcome the hurdle of transaction costs. The model using technical indicators had a lower profit ratio, but was consistently trading.

(6) Regularization: Weight Decay Increasing the decay factor has no significant effect on the baseline model up until about 0.001, from which onward the agent becomes unable to learn and collapses into a single action.

6 Discussion

6.1 Interpretation

In this section we discuss the factors contributing to some of the observed results.

(3) Parameter division The higher profit ratio on training data of critic-biased networks can be attributed to the critic's role as the Q-value function in the SAC algorithm. A network with more parameters has a higher capacity to model the complex, non-linear relationships between the market state and its potential value. This allows the critic to provide a more precise and detailed learning signal to the actor, enabling it to better exploit the specific patterns and nuances present in the training dataset, resulting in a higher profit ratio. This advantage is likely lost on the unseen data due to overfitting on noise.

(4) Activation Functions The differing performance between (Leaky)ReLU and Sigmoid activation functions highlights a classic trade-off between model flexibility and regularization. ReLU is an unbounded function, which grants the network higher representational power to fit the training data very closely, including its noise, leading to better performance on that data but risking overfitting. In contrast, Sigmoid is a bounded function that squashes activations into a fixed range, acting as a form of implicit regularization. This constraint prevents the network from reacting excessively to outliers or noise, forcing it to learn more general and robust patterns that are more likely to persist in the unseen evaluation data.

(5) Feature Extraction The failure of the agent using CNN-extracted features to learn the information necessary to trade likely stems from the specific network architecture. While handcrafted technical indicators represent strong, expert-driven priors designed to isolate market phenomena like trend and momentum, the CNN must learn useful representations from scratch. This, combined with the highly specific design and noisy input, likely produced uninformative features.

(6) Weight decay The observation that weight decay provided no benefit and was detrimental at higher values suggests that the baseline model was not significantly overfitting. Regularization techniques are effective only when there is excess model capacity to constrain. The baseline network, with its relatively small architecture ([32, 32]) and curated set of 16 features, is likely already capacity-limited and does not have enough parameters to excessively memorize the training data. Larger decay factors consequently induced underfitting by excessively penalizing all network weights, causing the policy to collapse.

6.2 Generalizability

There are a few reasons why the generalizability of the results can be called into question. The first reason is the high variance and low seed count; the second is the lack of variety in evaluation environments; and the third is the possibility of a performance bottleneck. These reasons may contribute to an inability to derive clear conclusions from the data.

First observation is the high variance in all of the results. This, combined with the low seed count of 5 suggests that a significant portion of the difference in mean performance across methods may be attributed to variance. The low seed count could also be a contributing factor for the high variance in the variance itself, rather than one method being truly more ”stable” than another.

Furthermore, there is a lack of variety in the evaluation environment. The evaluation was conducted on a single, fixed out-of-sample dataset, meaning the results may be specific to that particular market period. While techniques such as k-fold cross-validation are common for establishing more robust results, they were not used for this paper. This was primarily due to the temporal dependency of the Reinforcement Learning problem; using a ”middle fold” for validation would disrupt the chronological order of data, violating the Markov property essential for the agent's learning process. Additionally, the computational cost of training multiple models for each experiment was infeasible given the available resources.

In cases where no significant performance difference was observed, such as the network shape experiments (Experiment 2), the outcome may be explained by the bottleneck principle. This principle suggests that an unvaried, limiting factor in the experimental setup may be constraining overall performance, thereby masking the potential impact of the variable being tested. For the network shape experiments, it is plausible that the baseline's relatively small network size and limited 16-feature space acted as this bottleneck, causing the network shape to have no meaningful influence on the final trading performance.

6.3 Problem complexity

The main difficulties of developing a reinforcement learning agent that reliably gets positive returns on unseen data can be reduced to the following.

Changing position (Trading) comes with the penalty of commission, and spread costs. Therefore agents are incentivized to just hold a single position. To overcome this local optimum, the agent must develop enough of a predicting capability to justify actual consistent trading. However, since the signal to noise ratio is quite low in Forex Trading, the moment the agent ”catches on” and has enough insight to trade, it is already close to overfitting, causing poor performance on unseen data. This balance is one of the main difficulties in getting generalized model performance.

One method of decreasing noise is to decrease the granularity of the data. This aggregates values across timeframes, decreasing fluctuations in price and volume. However, this also eliminates the possibility to exploit these fluctuations in price, exponentially reducing the maximum possible return.

6.4 Usage of Future-Aware rewards

Finally, the usage of future-aware rewards can be taken into doubt. After all, a reinforcement learning agent is already trying to optimize the discounted sum of rewards (also called the return). Wouldn't future-aware rewards cause the agent to optimize the ”return of the return”? Wouldn't pulling some estimation of future reward into the present reward signal, lead the agent to focus on optimizing the wrong thing?

Given that the goal is to maximize the equity ratio, the following property should hold: ”If the return for one policy is higher than that for another, then the equity ratio should also be higher.”. Following the mathematical reasoning described in appendix C.3, this property holds for both standard (short-sighted) and the optimal value function based (future-aware) rewards. This means optimizing the future aware reward is equivalent to optimizing the equity ratio, alleviating the concern of misaligned goals.

7 Conclusion

This research sought to determine how different function approximation methods impact a Reinforcement Learning agent's trading performance in the EUR/USD Forex market. Our investigation, which systematically evaluated the effects of network architecture, activation functions, and feature extraction, found that while no configuration achieved reliable profitability on unseen data, the choice of specific components has a significant and discernible impact on model behavior.

The primary contribution of this work is the empirical demonstration of a critical trade-off between a model's capacity to learn and its ability to generalize. Key findings indicate that the unbounded ReLU activation function led to faster learning on the training set, whereas the bounded Sigmoid function demonstrated superior performance on evaluation data, suggesting it provides a form of implicit regularization that promotes robustness. Conversely, other architectural choices, such as network shape and parameter division between the actor and critic, had no significant impact on out-of-sample performance.

Future work should address the reliability concerns noted in this study by repeating experiments with more random seeds and across different out-of-sample time periods. A promising direction involves combining the most successful elements identified, such as a more sophisticated Convolutional Neural Network (CNN) for feature extraction paired with a larger, appropriately regularized network. An alternative research avenue is to reframe the problem as a supervised learning task, using the optimal policy derived from dynamic programming as ground truth. This could offer a more stable and computationally efficient path to developing a profitable trading agent by leveraging techniques such as data shuffling unavailable in a strict RL context.

8 Responsible Research

This research was conducted in adherence to the principles of responsible research, emphasizing ethical considerations and the reproducibility of our findings. Our methodology is designed to be transparent and verifiable.

8.1 Ethical Considerations

The primary goal of this research is to contribute to the scientific understanding of Reinforcement Learning models in financial markets. All data used in this study is publicly available historical Forex data sourced from Dukascopy, which mitigates concerns regarding privacy and sensitive information. We acknowledge that research in algorithmic trading carries societal implications; our focus remains on the academic exploration of model performance rather than the development of a commercial trading system.

We are acutely aware of the potential for scientific misconduct, such as data fabrication or manipulation. To prevent such issues, our data processing pipeline, from raw tick data aggregation to feature engineering, is explicitly documented in Section 3.1 and 3.2. Furthermore, the study's conclusions are based solely on the results obtained through the described methodologies.

8.2 Reproducibility

A central tenet of credible computational science is reproducibility, which requires that sufficient details of the research are available for independent verification. We have taken several steps to ensure our work is reproducible.

- **Data and Code Sharing:** The complete source code for the trading environment, the RL agent, and the analysis scripts are publicly available in a GitHub repository [27]. We provide links to the data source, Dukascopy, allowing for independent acquisition of the raw data [23].
- **Methodological Transparency:** Our paper provides a detailed description of the experimental setup, including the environment, the baseline model, specific hyperparameter configurations, and the exact algorithms used.

9 Acknowledgements

The overarching project "The use of Reinforcement Learning in Algorithmic Trading" was conducted as part of the CSE3000 Research Project course in collaboration between a group of five members. I would like to acknowledge and thank my fellow group members—Finn van Oosterhout, Mihai Radu Serban, Justas Bertašius, and Yavuz Hançer—for their valuable contributions, teamwork, and dedication throughout the project.

References

[1] Ganesh Marimuthu. Algorithmic trading in forex markets: The impact of ai-driven strategies on liquidity and market efficiency. *International Journal of Scientific Research in Computer Science Engineering and Information Technology*, 2025.

[2] Terry Lingze Meng and Matloob Khushi. Reinforcement learning in financial markets. *Data*, 2019.

[3] Yang Liu, Qi Liu, Hongke Zhao, Pan Zhen, and Chuanren Liu. Adaptive quantitative trading: An imitative deep reinforcement learning approach. *Proceedings of the AAAI Conference on Artificial Intelligence*, 2020.

[4] Shuo Sun, Rundong Wang, and Bo An. Reinforcement learning for quantitative trading. *arXiv preprint arXiv:2109.13851*, 2021.

[5] Malla Venkata Sai Ashish. Algorithmic trading using machine learning. *International Journal of Scientific Research in Engineering and Management*, 2025.

[6] Hana Jamali, Younes Chihab, Ivan Garcia-Magarino, and Omar Bencharef. Hybrid forex prediction model using multiple regression, simulated annealing, reinforcement learning and technical analysis. *IAES International Journal of Artificial Intelligence (IJ-AI)*, 2023.

[7] F. Garçia and E. Rachelson. Markov decision processes. *Markov Decision Processes in Artificial Intelligence*, pages 1–38, 2013.

[8] B. Gasperov, S. Begusic, P. Posedel, and Z. Kostanjcar. Reinforcement learning approaches to optimal market making. *Mathematics*, 9:2689, 2021.

[9] N. Bäuerle and U. Rieder. Markov decision processes. *Jahresbericht der Deutschen Mathematiker-Vereinigung*, 112:217–243, 2010.

[10] H. Wang and X. Zhou. Continuous-time mean-variance portfolio selection: a reinforcement learning framework. *SSRN Electronic Journal*, 2019.

[11] H. Wang, T. Zariphopoulou, and X. Zhou. Exploration versus exploitation in reinforcement learning: a stochastic control approach. *SSRN Electronic Journal*, 2019.

[12] C. Peng and Z. Zhao. Exploring new trends in the global foreign exchange derivatives market based on the european and american financial markets. *Advances in Economics, Management and Political Sciences*, 53:188–194, 2023.

[13] S. Ogbeide. Empirical assessment of foreign exchange market effect on the nigerian emerging economy. *Management and Economics Review*, 3:102–109, 2018.

[14] Y. Yong, D. Ngo, and Y. Lee. Technical indicators for forex forecasting: a preliminary study. *Computational Intelligence in Information Systems*, pages 87–97, 2015.

[15] H. Wang, Y. Yuan, Y. Li, and X. Wang. Financial contagion and contagion channels in the forex market: a new approach via the dynamic mixture copula-extreme value theory. *Economic Modelling*, 94:401–414, 2021.

[16] A. Geromichalos and K. Jung. An over-the-counter approach to the forex market. *International Economic Review*, 59:859–905, 2018.

[17] K. Pakhrudin, K. Kamaruddin, and F. Ahmad. Trader hub system development using van k tharp expectancy theory to analyse retail forex trading system performance. *Malaysian Journal of Computing*, 5:523, 2020.

[18] M. Öztürk, İ. Toroslu, and G. Fidan. Heuristic based trading system on forex data using technical indicator rules. *Applied Soft Computing*, 43:170–186, 2016.

[19] Z. Yang, C. Jin, Z. Wang, M. Wang, and M. Jordan. On function approximation in reinforcement learning:

optimism in the face of large state spaces. *arXiv preprint arXiv:2011.04622*, 2020.

[20] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, H. Fan, L. Sifre, G. Driessche, T. Graepel, and D. Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–359, 2017.

[21] A. Fickinger, H. Hu, B. Amos, S. Russell, and N. Brown. Scalable online planning via reinforcement learning fine-tuning. *arXiv preprint arXiv:2109.15316*, 2021.

[22] L. Abednego and C. Nugraheni. Forex data analysis using weka. *Computer Science & Information Technology*, 2020.

[23] Dukascopy Bank SA. Forex historical data feed, 2025. Accessed: 2025-06-01.

[24] K. Bednarz. Portfel markowitza w transakcjach na rynku forex. *Przeglad Organizacji*, Vol. 55:603–622, 2024.

[25] Fabiola Santore, Eduardo Cunha de Almeida, Wagner Hugo Bonat, Eduardo H. M. Pena, and Luiz S. Oliveira. A framework for analyzing the impact of missing data in predictive models. *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, 2020.

[26] T. Kim, W. Ko, and J. Kim. Analysis and impact evaluation of missing data imputation in day-ahead pv generation forecasting. *Applied Sciences*, 9:204, 2019.

[27] Robert Mertens. Tud-cse-rp-rlinfinance. https://github.com/Fo3nix/TUD-CSE-RP-RLinFinance, 2025. Accessed: 2025-06-22.

[28] John Moody and Matthew Saffell. Learning to trade via direct reinforcement. *IEEE Transactions on Neural Networks*, 12:875–89, 07 2001.

[29] FOREX.com. When does forex.com charge commissions? https://www.forex.com/en/help-and-support/raw-spread-account/, 2025. Accessed: 2025-06-19.

[30] Zihao Zhang, Stefan Zohren, and Stephen Roberts. Deep reinforcement learning for trading. *arXiv preprint arXiv:1911.10107*, November 2019.

[31] H. Trinh, S. Bae, and Q. Tran. Improving traffic efficiency in a road network by adopting decentralised multi-agent reinforcement learning and smart navigation. *Promet - Traffic&Transportation*, 35:755–771, 2023.

[32] Y. Han, J. Liu, D. Tian, and Q. Zhang. A novel anti-risk method for portfolio trading using deep reinforcement learning. *Electronics*, 11:1506, 2022.

[33] S. Singh, V. Goyal, S. Goel, and H. Taneja. Deep reinforcement learning models for automated stock trading. *Applied Technology and Data Science*, 2022.

[34] N. Yousefi. Deep reinforcement learning for tehran stock trading. *Journal of Novel Engineering Science and Technology*, 1:37–42, 2022.

[35] C. Wang and K. Ross. Boosting soft actor-critic: emphasizing recent experience without forgetting the past. *arXiv preprint arXiv:1906.04009*, 2019.

[36] Z. Shi and S. Singh. Soft actor-critic with cross-entropy policy optimization. *arXiv preprint arXiv:2112.11115*, 2021.

[37] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*, 2018.

A Experiment details

A.1 Baseline Hyperparameters

The baseline SAC hyperparameters are described in table 3

Hyperparameter	Value
`policy`	MlpPolicy
`activation_fn`	ReLU
`net_arch`	vf,qf=[32,32]
`learning_rate`	3×10^{-4}
`buffer_size`	200,000
`learning_starts`	1,000
`batch_size`	256
`tau`	0.005
`gamma`	1.0
`ent_coef`	auto
`gradient_steps`	2
`train_freq`	48

Table 3: SAC hyperparameters used in training

A.2 Shape parameter calculation

The network shapes used in the experiments (e.g., flat, diamond, funnel) were determined by solving a constraint satisfaction problem with a precise objective function. The goal was to find a set of network architectures, one for each shape type, where the total number of trainable parameters is as close as possible across all types. This ensures that any observed performance difference is due to the network's architectural shape, not its capacity.

The process begins by defining constraints to generate a large pool of valid candidate architectures for each shape type. The primary constraints are:

1. **Layer sizes are not extreme:** The number of neurons in any hidden layer must be within a reasonable range, set between 16 and 64 neurons.
2. **The shapes are distinct:** For non-uniform shapes like 'funnel' or 'diamond', a scaling factor determines how the layer width changes. This factor is constrained to be between 1.5 and 2.0 to ensure the shape is distinct and meaningful.

From the pools of valid candidates, a search algorithm is employed to find the optimal combination. For each shape type, its list of candidate architectures is sorted by the total parameter count. The algorithm then uses a multi-pointer search strategy, iterating through the sorted lists to find a set of architectures (one for each shape type) that minimizes the absolute difference between the maximum and minimum parameter counts in the set. This optimization yields the specific hidden layer dimensions for each shape that are subsequently used in the experiments, attempting to ensure a fair comparison.

A.3 Division sizes

The experiment on actor-critic parameter division required calculating network widths that would result in a specific parameter count. The goal was to allocate a fixed total parameter budget between the actor and critic networks according to a predetermined ratio (e.g., 60%/40%), while keeping the network depth constant.

This was achieved by first establishing the relationship between a network's width and its total number of parameters. For a feed-forward network with an input layer of size d_{in}, an output layer of size d_{out}, and N hidden layers of a uniform width w, the number of parameters P can be expressed as a function of w. Based on the implementation used to solve for the width, this relationship is formulated as:

$$P = (N-1)w^2 + (d_{in} + d_{out} + N)w + (1 - \text{bias\_term})$$

Rearranging this gives a quadratic equation in terms of the width w:

$$(N-1)w^2 + (d_{in} + d_{out} + N)w + (C - P) = 0$$

where C is a constant related to the bias terms. This equation can be solved for w using the standard quadratic formula, yielding the required network width to achieve a specific parameter count P.

For the experiment, the process is as follows:

1. A total parameter budget, P_{total}, is defined based on the baseline model's architecture.
2. For a given division ratio δ (e.g., 0.75 for a large bias), the individual parameter budgets for the actor (P_{actor}) and critic (P_{critic}) are calculated:
$$P_{actor} = \delta \cdot P_{total}$$
$$P_{critic} = (1-\delta) \cdot P_{total}$$
3. The quadratic equation is then solved independently for the actor and critic, using their respective parameter budgets (P_{actor} and P_{critic}), to find the necessary uniform layer widths, w_{actor} and w_{critic}.

This method ensures that the total model capacity remains constant while precisely controlling the distribution of parameters between the two key components of the SAC algorithm.

A.4 CNN Architecture

The goal of the CNN is to automate the process of feature extraction from raw, windowed time-series data, serving as an alternative to handcrafted technical indicators. The network is designed to process a sliding window of market data and produce a compact feature vector for the policy and value networks.

Input Data Preprocessing

The CNN does not operate on raw price data directly. Instead, it takes a window of 48 time steps, where each step contains a set of minimally processed features derived from the original OHLC (Open, High, Low, Close) and volume data. The transformations are:

- Stationarity Transformation: Absolute price levels are non-stationary. To address this, all OHLC columns are converted into percentage returns relative to the previous time step.
- Volume Normalization: Trading volume can vary drastically. To prevent outliers from disproportionately influencing the network, the volume is normalized using a robust scaler (based on interquartile range) over a 100-step lookback window.
- Basic Shape Features: Simple, unscaled ratios such as the high-low range (hl_ratio), open-close range (oc_ratio), and bid-ask spread (spread), all relative to the closing price, are included.

These provide the network with raw, fundamental information about the shape and liquidity of each candlestick.

Architecture Design and Rationale

The CNN architecture, is specifically tailored for financial time-series data. It consists of two convolutional layers followed by a pooling layer and a final linear projection.

Layer 1 The first layer is the most critical. Unlike typical image CNNs that use small, square kernels, this layer uses a kernel with a height equal to the total number of input features and a temporal width of 3. This design forces the convolution to capture patterns by looking at all input features simultaneously across a very short time window (the current step and its immediate neighbors). It effectively collapses the feature dimension, learning 32 unique filters that represent complex cross-feature relationships (e.g., "what is the relationship between high returns, low volume, and a widening spread at this moment?").

Layer 2 After identifying instantaneous cross-feature patterns, the second convolutional layer is designed to find temporal relationships within those learned patterns. It operates on each of the 32 feature maps from the previous layer independently, looking for patterns across a 5-step time window. This allows the model to learn sequential information, such as trends or momentum, from the abstract features generated by the first layer.

Layer 3 A max-pooling layer is applied to the temporal dimension. This serves two purposes: first, it reduces the sequence length by half, which decreases the number of parameters in the subsequent fully-connected layer and helps control overfitting. Second, it provides a degree of local time-invariance, making the learned features more robust to slight shifts in the timing of patterns.

Layer 4 After the convolutional and pooling blocks, the resulting feature maps are flattened into a 1D vector. A final linear layer projects this vector onto a fixed-size embedding of dimension 27. This acts as a final aggregator, producing the compact feature representation that is ultimately passed to the actor and critic networks. The output dimension of 27 was chosen to be comparable to the number of features in the handcrafted "technical_analysis" experiment, ensuring a fair comparison of feature extraction methods.

Finally, the 27-dimensional feature vector from the CNN is concatenated with a separate vector of non-windowed data (e.g., current_exposure, time-of-day features). Adding up to 32 features in total. This hybrid approach allows the model to leverage both automatically learned patterns from the price and volume history via the CNN and explicit, "necessary" engineered features representing the agent's current state and temporal context.

B Trading Logic Clarifications

B.1 Notation

- $t \in \{0, 1, \ldots, T-1\}$ the current timestep.
- $P_t^{\text{bid,open}}, P_t^{\text{bid,high}}, P_t^{\text{bid,low}}, P_t^{\text{bid,close}}$: the bid prices measured at open, high, low, close of period t.
- $P_t^{\text{ask,open}}, P_t^{\text{ask,high}}, P_t^{\text{ask,low}}, P_t^{\text{ask,close}}$: the ask prices measured at open, high, low, close of period t.
- κ: transaction cost percentage (e.g. $\kappa = 0.001$ for 0.1%)
- $a_t \in \mathcal{A}$: action taken by agent at start of period t.
- $x_t \in \mathcal{X}$: target exposure corresponding to action a_t.
- C_t, S_t: cash and number of shares held during period t. From the moment action a_t has been executed until just before action a_{t+1} is executed.
- $\epsilon_t \in \mathcal{E}$: exposure of the portfolio at the start of timestep t before a_t is executed. $\epsilon_t = p_t / E_t$
- p_t: position value at the start of timestep t before a_t is executed.
$$p_t = C_{t-1} \cdot P_t^* \quad P_t^* = \begin{cases} P_t^{\text{bid,open}}, & S_{t-1} \geq 0, \\ P_t^{\text{ask,open}}, & S_{t-1} < 0. \end{cases}$$
- E_t: equity at the start of timestep t before a_t is executed. $E_t = C_{t-1} + p_t$
- N_a, N_x, N_ϵ number of elements in $\mathcal{A}, \mathcal{X}, \mathcal{E}$ respectively. Only defined when the space is discrete.

B.2 Derivation for `max_shares_to_buy` in `buy_shares`

Our goal is to find the maximum number of shares we can buy S_{buy} such that
$$\frac{\text{Position Value}}{\text{Equity}} \leq 1,$$
is always satisfied. We solve for the limit case where exposure is exactly 1.

The Limit Condition:
$$\frac{\text{Position Value}}{\text{Equity}} = 1$$
$$\text{Position Value} = \text{Equity}$$
$$\text{Equity} = \text{Updated Cash} + \text{Position Value}$$
$$\text{Position Value} = \text{Updated Cash} + \text{Position Value}$$
$$\text{Updated Cash} = 0.$$

Define Final State Variables in terms of S_{buy}:
Let:

- $S_{\text{curr}} = \text{Current Shares}$
- $C_{\text{curr}} = \text{Current Cash}$
- $P_{\text{ask}} = \text{Ask Price}$
- $P_{\text{ask\_eff}} = P_{\text{ask}} \cdot \big(1 + \text{transaction cost pct}\big)$

Then:

- $\text{Updated Shares} = S_{\text{curr}} + S_{\text{buy}}$
- $\text{Updated Cash} = C_{\text{curr}} - S_{\text{buy}} \cdot P_{\text{ask\_eff}}$

Substitute into the Limit Condition and Solve for S_{buy}:
$$C_{\text{curr}} - S_{\text{buy}} \cdot P_{\text{ask\_eff}} = 0$$
$$S_{\text{buy}} \cdot P_{\text{ask\_eff}} = C_{\text{curr}}$$
$$S_{\text{buy}} = \frac{C_{\text{curr}}}{P_{\text{ask}} \cdot \big(1 + \text{transaction cost pct}\big)}.$$

This formula gives us the precise number of shares to buy to hit 100% leverage.

B.3 Derivation for `max_shares_to_sell` in `sell_shares`

Our goal is to find the maximum number of shares we can sell S_{sell} such that
$$\frac{\text{Position Value}}{\text{Equity}} \geq -1,$$
is always satisfied. We solve for the limit case where exposure is exactly -1.

1. The Limit Condition:
$$\frac{\text{Position Value}}{\text{Equity}} = -1$$
$$\text{Position Value} = -\,\text{Equity}$$
$$\text{Equity} = \text{Updated Cash} + \text{Position Value}$$
$$\text{Position Value} = -\big(\text{Updated Cash} + \text{Position Value}\big)$$
$$2 \cdot \text{Position Value} = -\,\text{Updated Cash}.$$

2. Define Final State Variables in terms of S_{sell}:
Let:

- $S_{\text{curr}} = \text{Current Shares}$
- $C_{\text{curr}} = \text{Current Cash}$
- $P_{\text{ask}} = \text{Ask Price}$
- $P_{\text{bid\_eff}} = P_{\text{bid}} \cdot \big(1 - \text{transaction cost pct}\big)$

Then:

- $\text{Updated Shares} = S_{\text{curr}} - S_{\text{sell}}$
- $\text{Updated Cash} = C_{\text{curr}} + S_{\text{sell}} \cdot P_{\text{bid\_eff}}$
- $\text{Position Value} = \big(S_{\text{curr}} - S_{\text{sell}}\big) \cdot P_{\text{ask}}$

3. Substitute into the Limit Condition and Solve for S_{sell}:
$$2 \cdot \big(S_{\text{curr}} - S_{\text{sell}}\big) \cdot P_{\text{ask}} = -\big(C_{\text{curr}} + S_{\text{sell}} \cdot P_{\text{bid\_eff}}\big)$$
$$2\, S_{\text{curr}}\, P_{\text{ask}} - 2\, S_{\text{sell}}\, P_{\text{ask}} = -\, C_{\text{curr}} - \, S_{\text{sell}}\, P_{\text{bid\_eff}}$$
$$S_{\text{sell}}\, \big(P_{\text{bid\_eff}} - 2\, P_{\text{ask}}\big) = -\big(C_{\text{curr}} + 2\, S_{\text{curr}}\, P_{\text{ask}}\big)$$
$$S_{\text{sell}} = \frac{-(C_{\text{curr}} + 2\, S_{\text{curr}}\, P_{\text{ask}})}{P_{\text{bid\_eff}} - 2\, P_{\text{ask}}} = \frac{C_{\text{curr}} + 2\, S_{\text{curr}}\, P_{\text{ask}}}{2\, P_{\text{ask}} - P_{\text{bid\_eff}}}.$$

This formula gives us the precise number of shares to sell to hit -100% leverage.

C Dynamic Programming Rewards

C.1 Derivation of Optimality

State Space Discretization The continuous exposure space $[-1, 1]$ is discretized into N_ϵ equally spaced exposure levels for state representation, and the target exposure space is discretized into N_x equally spaced levels for action selection:

$$\mathcal{E} = \left\{-1, -1 + \frac{2}{N_\epsilon - 1}, \ldots, 1 - \frac{2}{N_\epsilon - 1}, 1\right\} \quad (1)$$

$$\mathcal{X} = \left\{-1, -1 + \frac{2}{N_x - 1}, \ldots, 1 - \frac{2}{N_x - 1}, 1\right\} \quad (2)$$

For a given exposure value $\epsilon \in [-1, 1]$, the corresponding discrete index is computed as: $\text{idx}(\epsilon) = \lfloor(\epsilon + 1) \cdot (N_\epsilon - 1) \cdot 0.5 + 0.5\rfloor$

Dynamic Programming Formulation The optimal trading problem is formulated as a finite-horizon Markov Decision Process. At each timestep t and current exposure level ϵ, we define:

- **Value function** $V_t(\epsilon)$: the maximum expected cumulative log-equity from a state with exposure ϵ at time t to the terminal time T.
- **Policy function** $\pi_t(\epsilon)$: the optimal target exposure x_t^* for a state with exposure ϵ at time t.
- **Worst-case function** $Q_t^{\min}(\epsilon)$: the minimum one-step continuation value among all possible target exposures for a state with exposure ϵ at time t.
- **Immediate equity ratio** $r_t(\epsilon, x)$: the ratio between next equity and current equity (E_{t+1}/E_t) for state-action pair (ϵ, x) at time t

Backward Induction Algorithm The optimal value and policy tables are computed using backward induction, starting from the terminal timestep $T - 1$ and working backwards to $t = 0$:

1. **Terminal condition:** $V_{T-1}(\epsilon) = 0 \quad \forall \epsilon \in \mathcal{E}$
2. **Recursive relation:** For $t = T - 2, T - 3, \ldots, 0$ and each exposure level $\epsilon \in \mathcal{E}$:

$$V_t(\epsilon) = \max_{x \in \mathcal{X}} \{\log(r_t(\epsilon, x)) + V_{t+1}(\epsilon')\}$$

where:

- ϵ' is the exposure after taking action x from exposure ϵ at timestep t and following market movements until just before action x_{t+1} is executed.
- Transaction cost are taken accounted of.

Associated Formulae Following the dynamic programming formulation and the backward induction algorithm, we can directly define the optimal policy π_t and worst-case one-step continuation value function $Q_t^{\min}$.

$$\pi_t(\epsilon) = \arg\max_{x \in \mathcal{X}} \{\log(r_t(\epsilon, x)) + V_{t+1}(\epsilon')\}$$

$$Q_t^{\min}(\epsilon) = \min_{x \in \mathcal{X}} \{\log(r_t(\epsilon, x)) + V_{t+1}(\epsilon')\}$$

Equity Transition Model For each state-action pair (ϵ, x) at timestep t, the next exposure ϵ' is computed as follows:

1. **Reverse equity calculation:** First, we decompose the current equity and exposure into cash and shares. We use current equity $E_t = 1$ such that the resulting equity is directly the equity ratio $E_{t+1}/E_t = E_{t+1}$.

$$(C_{t-1}, S_{t-1}) = \text{reverse\_equity}(P_t^{\text{bid,open}}, P_t^{\text{ask,open}}, 1, \epsilon)$$

2. **Trade execution:** Then we use the cash and shares to execute the trade to achieve target exposure x.

$$(C_t, S_t) = \text{trade}(x, P_t^{\text{bid,open}}, P_t^{\text{ask,open}}, C_{t-1}, S_{t-1}, \kappa)$$

3. **Market evolution:** Using the resulting cash and shares we compute the equity after market movements.

$$E_{t+1} = \text{calculate\_equity}(P_{t+1}^{\text{bid,open}}, P_{t+1}^{\text{ask,open}}, C_t, S_t)$$

4. **Next exposure:** Finally, we compute the resulting exposure:

$$\epsilon' = \frac{E_{t+1} - C_t}{E_{t+1}}$$

Interpolation for Continuous States Since the agent operates in continuous exposure space while the DP tables are discrete, bilinear interpolation is used to estimate values for arbitrary current exposure levels:

$$V_t(\epsilon) = (1 - \alpha) \cdot V_t(\epsilon_{\text{low}}) + \alpha \cdot V_t(\epsilon_{\text{high}})$$

where ϵ_{low} and ϵ_{high} are the nearest discrete exposure levels, and α is the interpolation weight determined using relative closeness to each discrete exposure level.

C.2 Future Aware Reward formulations

The DP-based reward is not a single function but a family of related functions designed to provide robust, future-aware feedback by comparing the agent's performance against the optimal and worst-case scenarios pre-computed by dynamic programming. We distinguish between three primary formulations.

A. Bottom-up Reward This reward function quantifies how much better the agent's action was compared to the worst possible action at the current state. It provides a reward based on the advantage over the minimum performance baseline.

$$r_t^A = \left(\log r_t^{\text{true}} + V_{t+1}(\epsilon_{t+1}) - Q_t^{\min}(\epsilon_t)\right) \cdot \text{norm}$$

where $r_t^{\text{true}} = E_{t+1}/E_t$ is the true equity ratio, $V_{t+1}(\epsilon_{t+1})$ is the interpolated optimal future value, $Q_t^{\min}(\epsilon_t)$ is the interpolated worst-case value from the current state, and "norm" is a global normalization factor.

B. Top-down Reward This reward function measures how much worse the agent's action was compared to the optimal action, effectively quantifying the sub-optimality loss.

$$r_t^B = \left(\log(r_t^{\text{true}}) + V_{t+1}(\epsilon_{t+1}) - V_t(\epsilon_t)\right) \cdot \text{norm}$$

The value is always less than or equal to zero (before normalization and clipping), with a reward of zero indicating that the optimal action was taken. This "top-down" approach penalizes any deviation from the optimal policy π^*.

Motivation for a Third Reward Function The raw values of these future-aware rewards are often highly skewed, with a distribution resembling an exponential decay where most rewards are very close to zero and a few are exceptionally large. Normalizing these values using a single global factor can cause the vast majority of reward signals to become numerically insignificant. This may lead to slow or ineffective learning, as the agent receives meaningful feedback only in rare, high-stakes situations. To address this, a third formulation is introduced that intends to fix this issue.

C. Percentile-normalized Reward This reward function decouples the quality of an action from the inherent importance of the state. First, we define the "state importance" $I_t(\epsilon_t)$ as the range between the best and worst possible one-step outcomes:

$$I_t(\epsilon_t) = V_t(\epsilon_t) - Q_t^{\min}(\epsilon_t)$$

This value is then used to construct a two-part reward. A "goodness" score is calculated by normalizing the agent's performance within this range, and it is then scaled by the pre-computed percentile rank of the state's importance.

$$\begin{aligned} \text{Goodness}_t &= 2 \cdot \frac{Q_t^{\pi}(\epsilon_t, x_t) - Q_t^{\min}(\epsilon_t)}{I_t(\epsilon_t)} - 1 \\ r_t^C &= \text{Goodness}_t \cdot \text{Percentile}(I_t(\epsilon_t)) \end{aligned}$$

This formulation ensures a consistent reward range for decision quality at every state, while still emphasizing the importance of making good decisions in critical situations.

C.3 Proof of Order-Equivalence

Objective We aim to prove that for two policies, π_A and π_B, the total return under the DP-based reward functions for π_A is greater than for π_B if and only if the final equity for π_A is greater than for π_B, given the same starting equity. We assume an undiscounted setting ($\gamma = 1$).

Preliminaries An episode runs for $T-1$ timesteps from $t = 0, \ldots, T-2$. The value function at the terminal timestep $T-1$ is defined to be zero: $V_{T-1}(\epsilon) = 0$ for all $\epsilon \in \mathcal{E}$.

Baseline: Simple Log-Equity Rewards

First, we establish the property for a simple reward function equal to the one-step log-equity ratio.

1. **Simple Reward Definition:** The reward at timestep t is defined as:
$$r_t = \log\left(\frac{E_{t+1}}{E_t}\right)$$
2. **Total Return Calculation:** The total return for a policy π is the sum of these rewards:
$$G(\pi) = \sum_{t=0}^{T-2} r_t = \sum_{t=0}^{T-2} \log\left(\frac{E_{t+1}}{E_t}\right)$$
3. **Telescoping Sum:** This sum forms a telescoping series which simplifies to the total log-equity ratio over the episode:
$$G(\pi) = \log\left(\frac{E_{T-1}}{E_0}\right)$$
4. **Equivalence Proof:** Comparing two policies, π_A and π_B, starting from the same initial equity E_0:
$$G(\pi_A) > G(\pi_B) \iff \log\frac{E_{T-1}^A}{E_0} > \log\frac{E_{T-1}^B}{E_0}$$
As the logarithm function is strictly monotonic, this is equivalent to:
$$G(\pi_A) > G(\pi_B) \iff \frac{E_{T-1}^A}{E_0} > \frac{E_{T-1}^B}{E_0}$$
Thus, maximizing the simple log-equity return is perfectly equivalent to maximizing the final equity.

Bellman Decomposition and Sub-Optimality Loss

To analyze the DP rewards, we introduce a key identity derived from the Bellman equations.

Sub-Optimality Loss Definition: For any state ϵ_t and an action x_t from policy π, the Bellman inequality is $V_t(\epsilon_t) \geq Q_t^{\pi}(\epsilon_t, x_t)$, where $Q_t^{\pi}(\epsilon_t, x_t) = \log(E_{t+1}/E_t) + V_{t+1}(\epsilon_{t+1})$. We define the single-step sub-optimality loss $\delta_t \geq 0$ as:

$$\begin{aligned} \delta_t(\pi) &\equiv V_t(\epsilon_t) - Q_t^{\pi}(\epsilon_t, x_t) \\ \delta_t(\pi) &= V_t(\epsilon_t) - \left(\log\left(\tfrac{E_{t+1}}{E_t}\right) + V_{t+1}(\epsilon_{t+1})\right) \end{aligned}$$

Log-Equity Identity Derivation: Rearranging the definition of δ_t and summing over the trajectory:

$$\sum_{t=0}^{T-2} \log\left(\frac{E_{t+1}}{E_t}\right) = \sum_{t=0}^{T-2} \left(V_t(\epsilon_t) - V_{t+1}(\epsilon_{t+1})\right) - \sum_{t=0}^{T-2} \delta_t(\pi)$$

The left side is $\log(E_{T-1}/E_0)$. The first term on the right is a telescoping sum that simplifies to $V_0(\epsilon_0) - V_{T-1}(\epsilon_{T-1})$. Given the terminal condition $V_{T-1} = 0$, we arrive at the identity:

$$\log\left(\frac{E_{T-1}}{E_0}\right) = V_0(\epsilon_0) - \sum_{t=0}^{T-2} \delta_t(\pi)$$

Since ϵ_0 is the same regardless of policy, the factor $V_0(\epsilon_0)$ is constant. This shows that maximizing final equity is equivalent to minimizing the cumulative sub-optimality loss.

Proof for Top-Down Reward (DPRewardFunctionB)

This reward function measures the sub-optimality of an action relative to the best possible action.

Reward Definition and Total Return: The core of the "Top-down" reward is the advantage relative to the optimal value function, which is precisely the negative sub-optimality loss:

$$r_t^B = \log\left(\frac{E_{t+1}}{E_t}\right) + V_{t+1}(\epsilon_{t+1}) - V_t(\epsilon_t) = -\delta_t(\pi)$$

The total return is therefore:

$$G^B(\pi) = \sum_{t=0}^{T-2} r_t^B = -\sum_{t=0}^{T-2} \delta_t(\pi)$$

Equivalence Proof: To maximize the total return $G^B(\pi)$ is to maximize $-\sum \delta_t(\pi)$, which is equivalent to minimizing the total sub-optimality loss $\sum \delta_t(\pi)$. From our identity in the previous section, this is equivalent to maximizing the final log-equity $\log(E_{T-1}/E_0)$. The order-equivalence property therefore holds exactly.

Extension to Other Rewards

The order-equivalence property can also be shown to hold for the other reward formulations under reasonable assumptions. The proofs follow a similar structure but are less direct than the one presented for the "Top-down" reward. For the sake of brevity, these detailed derivations are not included here.

D Relevant Terminology

D.1 Reinforcement Learning

- **Reinforcement Learning**: machine learning paradigm where an agent learns to make sequential decisions by interacting with an environment.
- **Agent**: decision making entity
- **Environment**: everything the agent interacts with.
- The interaction proceeds in discrete time steps $t = 0, 1, 2, \ldots$. At each timestep:
  1. The Agent observes the current state.
  2. The Agent selects an action according to a policy.
  3. The Agent executes the action.
  4. The environment transitions to a new state, and provides a reward to the agent.
  5. The Agent uses the reward and new state to update internal knowledge.
- The goal for the agent is to determine a policy, that maximizes the total reward.

D.2 Reinforcement Learning (Formally)

- **State** s: representation of the environment at a given step.
- **States** S: the set of all possible states.
- **Actions** $A(s)$: the set of all possible actions a in state s
- **Reward function** $R(s, a, s')$: (expected) immediate reward received after taking action a in state s, and transitioning to state s'.
- **Transition function** $P(s'|s, a)$: probability of transitioning to state s' when taking action a in state s.
- **Policy function** $\pi(a|s)$: (probabilistic) mapping of states $s \in S$ to actions $a \in A(s)$
- **Return** G_t: The cumulative discounted reward starting from timestep t:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

- **Discount factor** $\gamma \in [0, 1]$: Determines how much the agent values future rewards.
- **State-Value Function** $V^\pi(s)$: Expected return from state s under policy π:

$$V^\pi(s) = \mathbb{E}_\pi [G_t \mid s_t = s]$$

- **Action-Value Function** $Q^\pi(s, a)$: Expected return from state s, taking action a, and then following policy π:

$$Q^\pi(s, a) = \mathbb{E}_\pi [G_t \mid s_t = s, a_t = a]$$

- **Optimal State-Value Function** $V^*(s)$: Maximum expected return achievable from state s:

$$V^*(s) = \max_\pi V^\pi(s) = \max_{a \in A(s)} Q^*(s, a)$$

- **Optimal Action-Value Function** $Q^*(s, a)$: Maximum expected return achievable from state s and action a:

$$Q^*(s, a) = \max_\pi Q^\pi(s, a)$$

- **Optimal Policy** π^*: Policy that maximizes expected return from all states:

$$\pi^*(s) = \arg\max_\pi V^\pi(s) = \arg \max_{a \in A(s)} Q^*(s, a)$$

- **Advantage Function** $A(s, a)$: Measures how much better or worse an action is compared to the average:

$$A(s, a) = Q(s, a) - V(s)$$

D.3 Function approximation

- **Function approximation method**: refers to how the agent represents and estimates complex functions like the state-value function $V^\pi(s)$, action-value function $Q^\pi(s, a)$, policy function $\pi(a|s)$, transition function $P(s'|s, a)$, or reward function $R(s, a, s')$.
- **Tabular methods**: Each state, or state-action pair is mapped to a value. This mapping can be represented using a table. Only works for discrete, and small state-action spaces.
- **Linear function approximation**: the function is estimated as a weighted sum of features. Allows for continuous, and larger state-action spaces, but are limited in their ability to capture complex patterns or interactions.
- **Non-linear function approximation**: the function is estimated using non-linear models (e.g., neural networks, decision trees, kernel methods) . Captures more complex relationships but can be harder to train.

D.4 Deep Reinforcement Learning

- **Deep learning**: subfield of machine learning using multilayered neural networks.
- **Deep Reinforcement Learning** (**Deep RL**): Applies deep learning to RL algorithms. Can be utilized in function approximation, feature extraction, and joint-learning.

D.5 Reinforcement Learning Model Categories

- **Policy vs Value-based** methods: Value-based methods try to learn a value function, and then derive the optimal policy using it. Policy-based methods try to learn the optimal policy directly.
- **Policy-gradient methods**: subset of policy-based methods that use gradient descent on the expected return to approximate the optimal policy.
- **Actor-critic methods**: combines value and policy based methods. It uses two separate components: the actor, that is responsible for selecting actions (representing the policy), and the critic, evaluating the actions taken by the actor (using a value function).
- **Model-free vs. model-based**: Model-free methods learn directly from interactions without understanding the environment's dynamics, while model-based methods build a model of the environment in order to plan ahead.
- **On-policy vs. off-policy**: On-policy methods learn from the actions taken by the current policy, while off-policy methods learn from actions taken by a different policy (e.g., a past or exploratory one).
- **Algorithm**: the overall procedure used to learn how to behave optimally. It defines how the agent updates its policy or value estimates based on experience.

D.6 Trading

- **Foreign Exchange Market** (Forex): also called currency market, is a globalized decentralized market for the trading of currencies.
- **Forex currency pair**: a combination of currencies that are traded against each other. First currency being the base currency, second being the quote currency (BASE/QUOTE). The pair shows how much quote currency you need to buy one unit of the base currency. Ex. EUR/USD, GBP/USD, USD/JPY, etc.
- **Candlestick**: a visual representation of price movement over a specific timeframe. It provides four key pieces of information. Open: Price at the beginning of the timeframe. High: Highest price during the timeframe. Low: Lowest price during the timeframe. Close: Price at the end of the timeframe. Volume: Total transaction amount during the timeframe (sometimes included in the data, imperfect accuracy for Forex due to decentralization).
- **Timeframe**: the duration that each data point (or candlestick) represents on a chart. It determines the granularity of the data the RL agent learns from.
- **Ask, Bid, Spread**: Ask is the price for which you can buy shares, bid price is the price for which you can sell shares, and the spread is the difference between ask and bid.
- **Buy, Sell, Hold**: Buy is the trading quote currency for the base currency. Selling is trading the base currency for the quote currency. Hold is keeping the same amount of each.
- **Long, Short, Cash**: Long is holding the base currency, having a positive positional value, expecting it to appreciate relative to the quote currency. Short is borrowing the base currency and immediately trading it for the quote currency, having a negative positional value (debt), expecting the base to depreciate. Cash refers to a neutral position, holding only the quote currency and having no exposure to the base currency.
- **Equity**: difference between total assets and liabilities.
- **Mark-to-market**: method of valuing assets and liabilities at their current market price rather than their original purchase price.

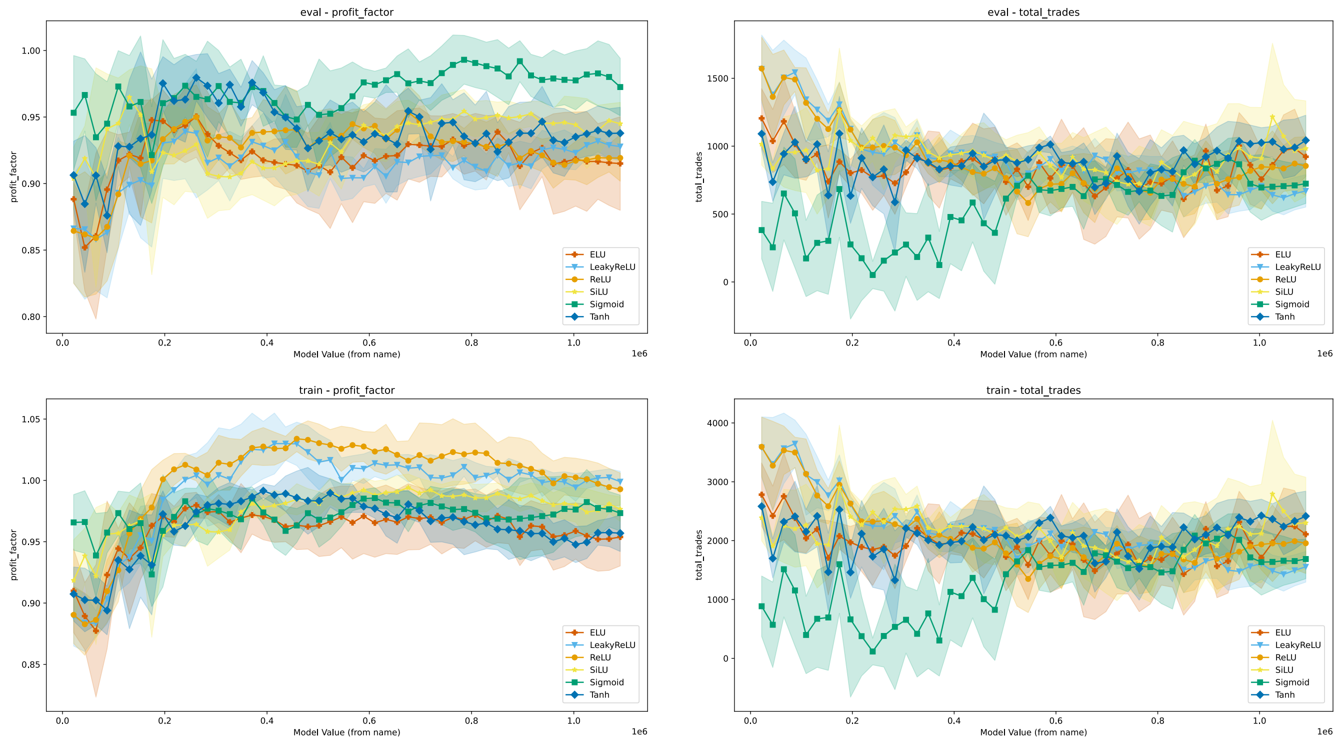
E Graphs

Table 4: Baseline Train Results

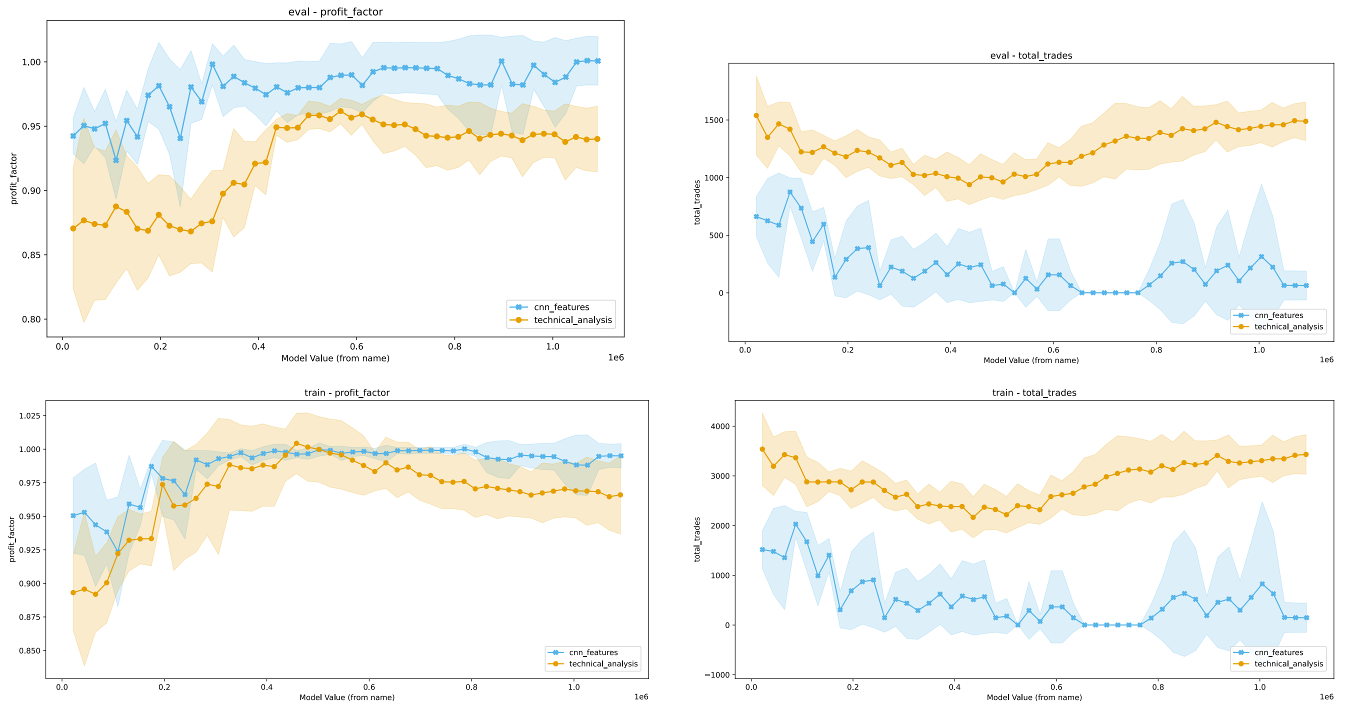
Model	Profit Factor	Total Trades
long	0.9983245	1
short	1.0013813	1
cash	1.0	0
random	0.76632106	10893
ideal	45.37837	8090

Table 5: Baseline Eval Results

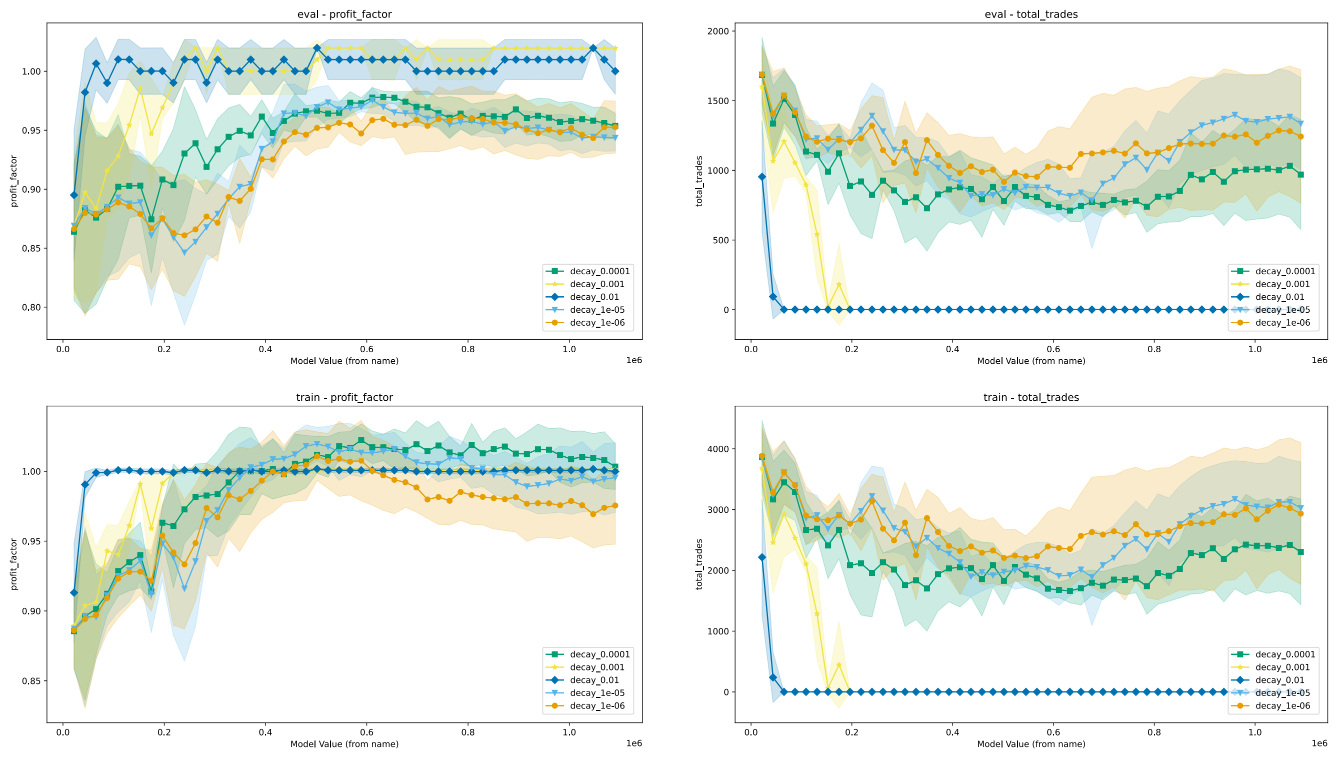
Model	Profit Factor	Total Trades
long	0.98054975	1
short	1.0195684	1
cash	1.0	0
random	0.7544354	4595
ideal	32.65881	3193



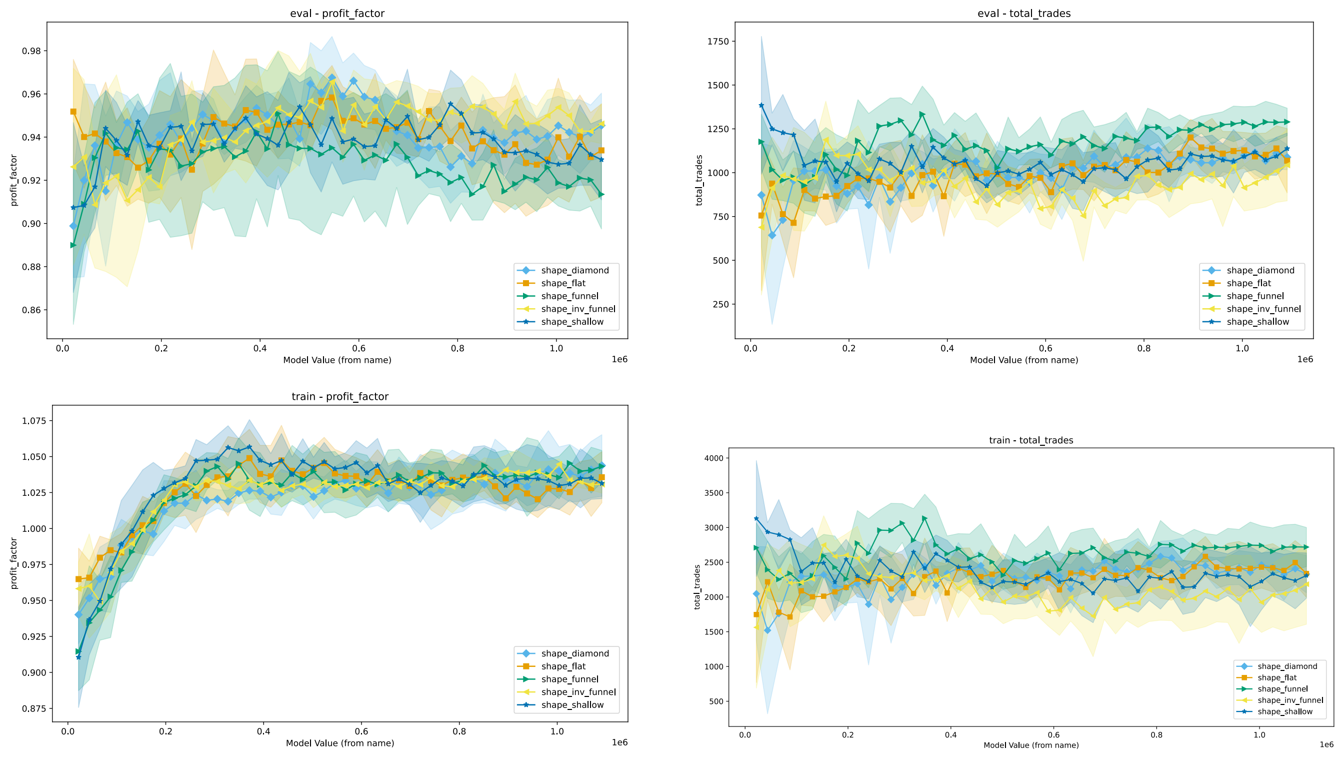
Profit factor (left) and total number of trades (right) for evaluation environments eval (top) and train (bottom) across **activation functions**.



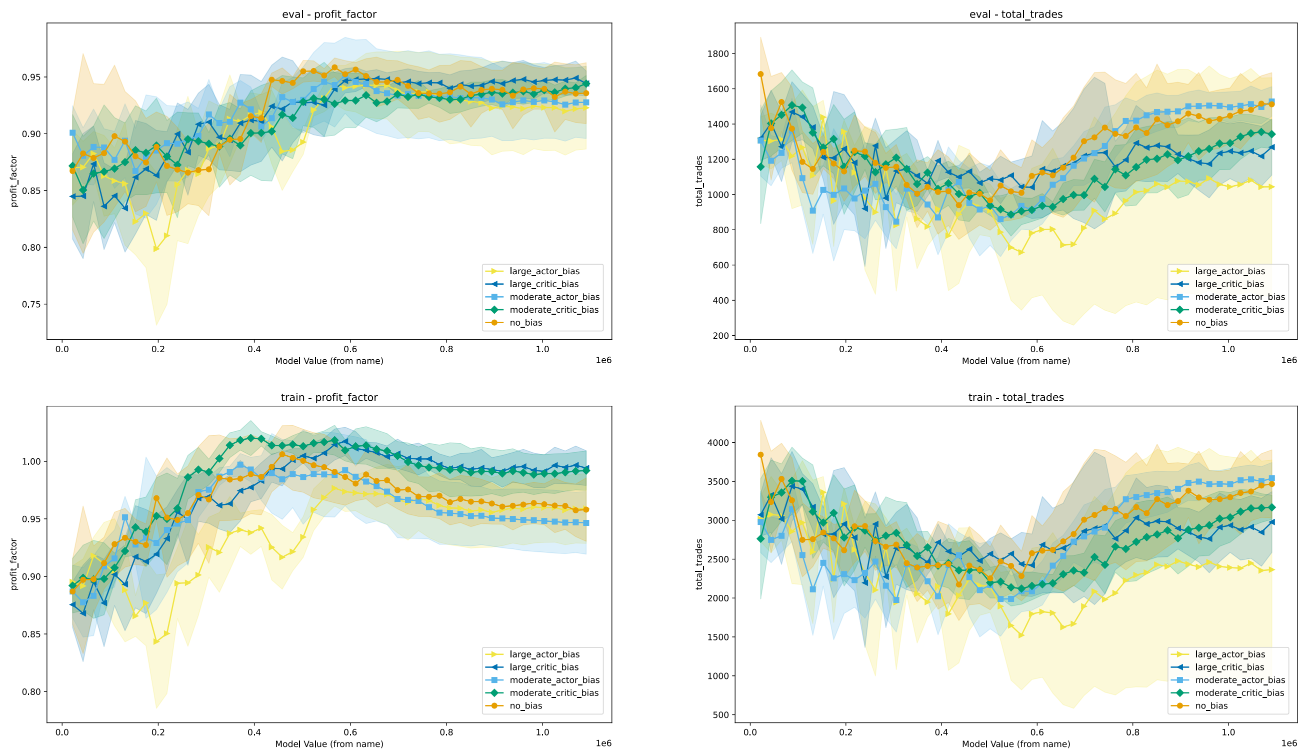
Profit factor (left) and total number of trades (right) for evaluation environments eval (top) and train (bottom) across **feature extraction methods**.



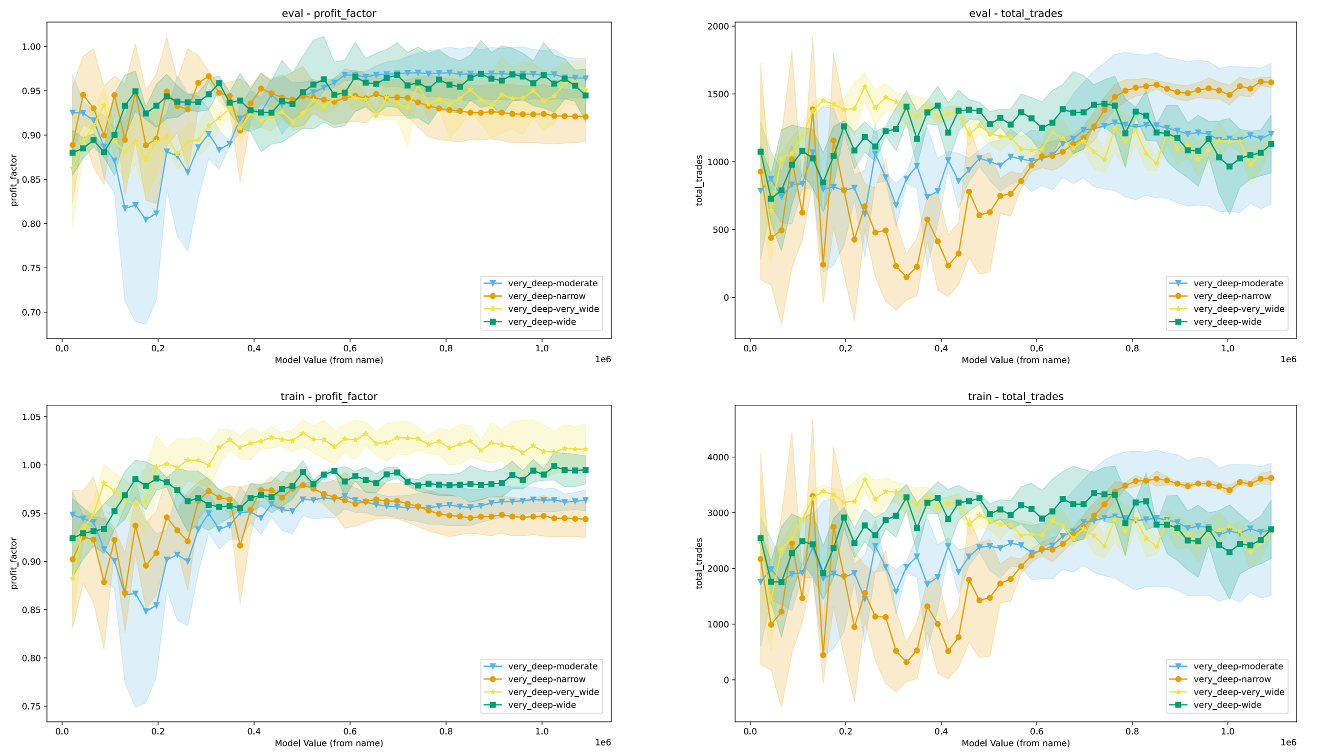
Profit factor (left) and total number of trades (right) for evaluation environments eval (top) and train (bottom) across **decay factors**.



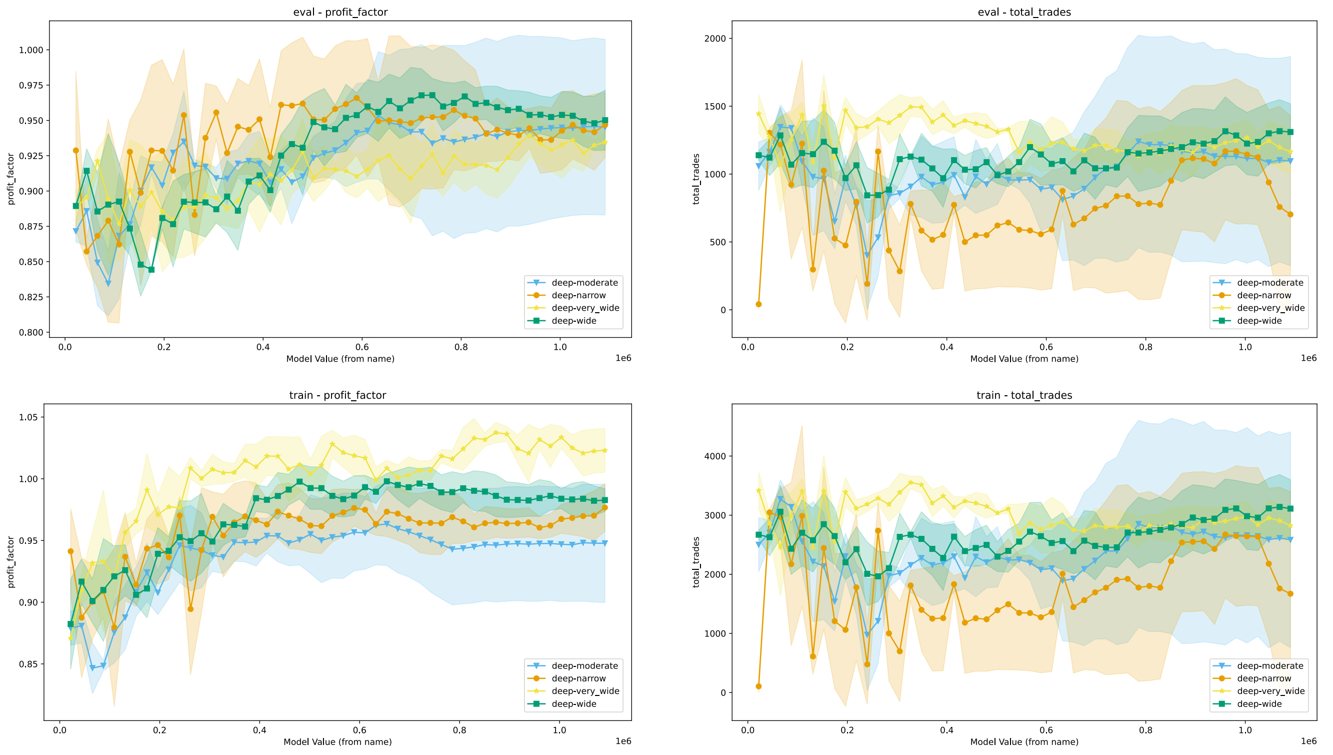
Profit factor (left) and total number of trades (right) for evaluation environments eval (top) and train (bottom) across **network shapes**.



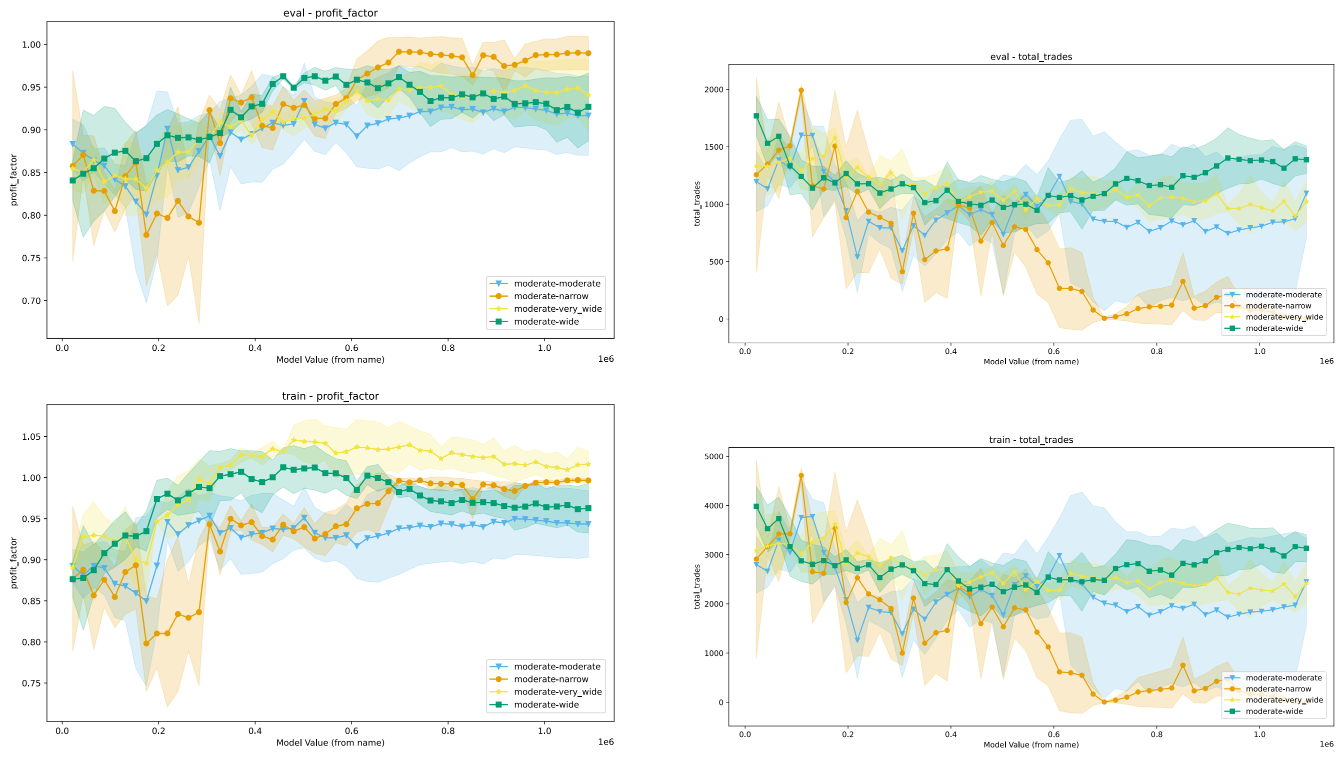
Profit factor (left) and total number of trades (right) for evaluation environments eval (top) and train (bottom) across **network divisions**.



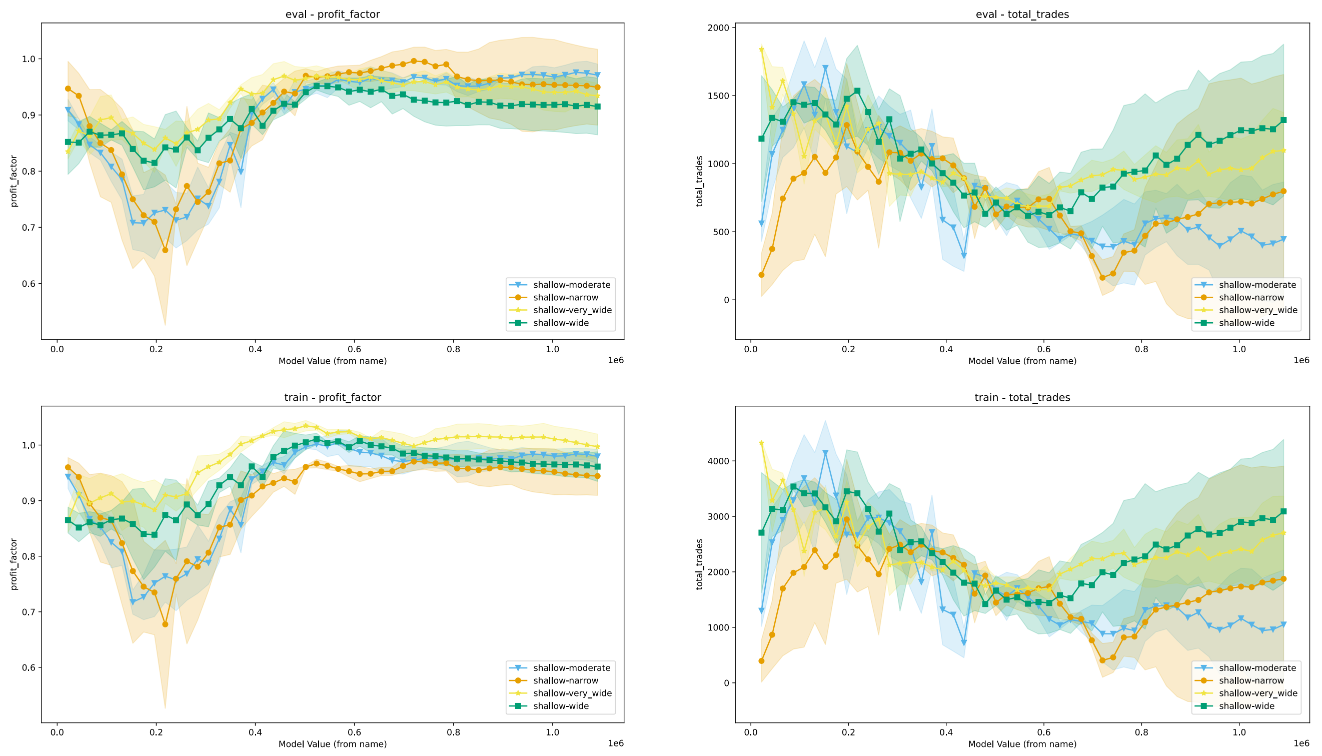
Profit factor (left) and total number of trades (right) for evaluation environments eval (top) and train (bottom) across **very deep networks**.



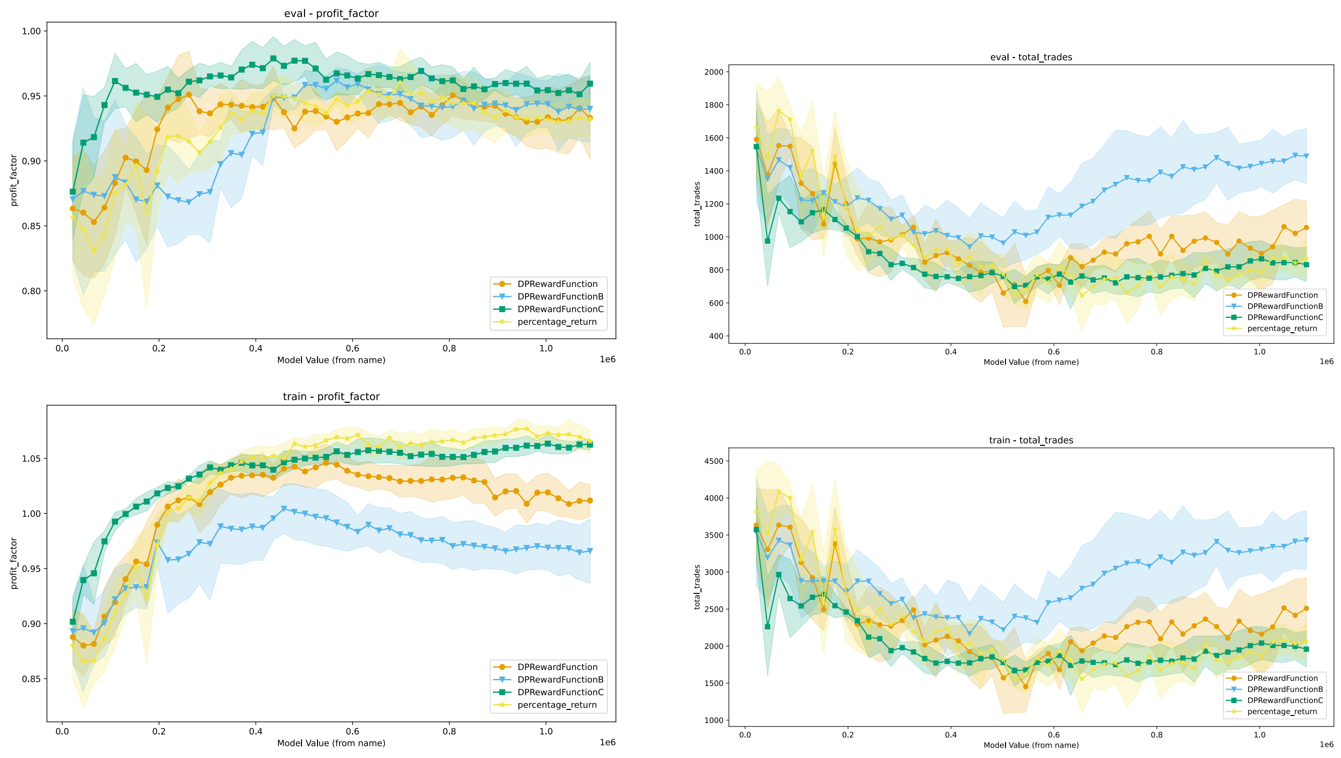
Profit factor (left) and total number of trades (right) for evaluation environments eval (top) and train (bottom) across **deep networks**.



Profit factor (left) and total number of trades (right) for evaluation environments eval (top) and train (bottom) across **moderate depth networks**.



Profit factor (left) and total number of trades (right) for evaluation environments eval (top) and train (bottom) across **shallow networks**.



Profit factor (left) and total number of trades (right) for evaluation environments eval (top) and train (bottom) across **reward functions**.