AI-Powered Credit Limit Decisions for Revolving Credit A Reinforcement Learning Approach

Konstantinos Bousoulas[1] Luca Grassitelli[2]

[1]Department of Portfolio Optimisation Analytics, Advanzia Bank S.A., Munsbach Schuttrange, L-5365, Luxembourg
[2]Department of Risk Modelling, Advanzia Bank S.A., Munsbach Schuttrange, L-5365, Luxembourg

August 7, 2025

ABSTRACT

Advanzia Bank has leveraged predictive analytics and credit scorecards for risk assessment for over fifteen years. In revolving credit, optimizing credit limit assignment is crucial for maintaining profitability. While reinforcement learning (RL) has been applied across various domains, from deterministic environments to stochastic scenarios, its adoption in banking remains limited. This paper explores the use of RL techniques to develop an optimal credit card limit adjustment policy. The AI agent iteratively adjusts credit limits over time to maximize profitability. The state space is continuous, incorporating key features such as Probability of Default (PD), Revenue, Loss, Current Credit Limit, and Trend. The action space is discretized into predefined credit limit adjustments. PD evolution over time is modelled using a survival model that estimates hazard rates at the customer level. Our findings demonstrate that RL algorithms - specifically Double DQN, and Actor-Critic DQN - offer a viable, data-driven alternative to traditional credit limit adjustment methods. We employ simulated data reflecting real-world German credit card usage to evaluate this framework and discuss the insights gained. Additionally, we compare the RL-based approach to conventional methods that frame credit limit assignment as a nonlinear constrained optimization problem, highlighting the potential advantages of AI-driven methodologies in this domain.

Keywords: Credit scoring; Credit-limit assignment; Non-linear optimisation; Survival models; Reinforcement learning

1. Introduction

Although reinforcement learning (RL) has been applied successfully to deterministic environments and to stochastic problems in portfolio and operations management, its deployment in core banking processes remains limited. Only a small number of studies, most notably Singh et al. [1] examined RL within credit-risk management, and even fewer addressed the specific task of adjusting revolving credit lines.

The present study develops and evaluates two agents, namely a Double Deep-Q-Network (DQN) and an Actor–Critic DQN, tailored to the credit-limit-management setting. Each agent learns policies that satisfy business constraints while maximizing expected portfolio profit. Training stability and sample efficiency are enhanced through target-network synchronization, prioritized experience replay, and reward-shaping techniques. In the current discrete action space, on ten pre-defined "limit-ladder" buckets, the Double DQN offers a tractable and robust solution; however, should future product requirements necessitate continuous limits, risk-aware regularization, or non-stationary deployment, Actor–Critic variants such as PPO (Proximal Policy Optimization) [2] or SAC (Soft Actor Critic) [3] are expected to yield performance gains despite their greater training variance.

Prior literature on limit management is sparse. Alfonso-Sánchez et al. (2024) [4] survey earlier contributions, highlighting the absence of an industry standard for periodic limit adjustment and noting that most studies neglect the revolving nature of credit-card exposures and the provisioning rules imposed by IFRS 9. However, methodologies for subsequent limit revisions remain under-explored.

To incorporate forward-looking measures of credit risk, the proposed framework embeds an application or behavioral-scoring component. Any statistical or machine-learning survival model trained in demographic, usage, and payment-history variables can be employed to generate customer-level hazard rates, which form part of the agent's state representation. Offline learning is conducted within a simulator that links limit decisions to utilization, default, and recovery through a sequence of user-defined functions calibrated on historical data. The training dataset reflects the behavior of sub-prime and near-prime German credit-card holders, characterized by annualized charge-off rates of approximately 5–7 % and cumulative vintage defaults of 10–15 %.

For each customer, the agent selects between two actions, retaining the current limit or increasing it. The optimization objective balances revenue uplift against higher impairment charges, thereby aligning with the adversarial goals identified by Alfonso-Sánchez et al. [4]. The continuous state vector comprises probability of default, expected revenue and loss, current limit, and utilization trend. Through iterative interaction with the simulator, the agent converges on a policy that maximizes expected profit over a 24-month period.

The performance of the learned policy is benchmarked against a conventional non-linear constrained-optimization approach. This comparative analysis serves to assess the incremental value of RL and to motivate further research into its applicability across other credit products and jurisdictions.

The remainder of the paper is organized as follows: Section 2 introduces the fundamentals of reinforcement learning and formulates the credit-limit adjustment problem within an RL framework. Section 3 describes the dataset, outlines the construction of the simulation environment, and specifies the algorithmic configuration and process flow for the DDQN agent. Section 4 presents the corresponding architecture of the AC Agent. Section 5 reports training results for both agents and compares their learned policies against an optimization-based benchmark. Section 6 focuses on model validation and techniques to mitigate overfitting. Last, Section 7 discusses the study's limitations and concludes with suggestions for future research.

2. RL Fundamentals & Provisioning for Revolving Credit

2.1 Background

Reinforcement learning (RL) seeks to discover a policy that maximises the agent's expected cumulative reward through sequential interaction with an environment [5]. This objective differs fundamentally from supervised learning, whose goal is to approximate a fixed mapping between input features and target labels. By emphasising trial-and-error search and delayed feedback, RL is uniquely suited to dynamic decision-making tasks.

Early milestones highlight RL's potential in complex deterministic domains: DeepMind's AlphaGo system, for example, combined tree search with deep RL to defeat the reigning Go world champion in 2015 [6]. Subsequent work by Mnih et al. (2015) [7] introduced the Deep Q-Network (DQN), which integrates Q-learning with convolutional neural networks and achieved human-level performance on 49 Atari 2600 games. As algorithms have matured, researchers have increasingly targeted stochastic settings that mirror real-world uncertainty [8]. In operations management, Yang et al. [9] employed deep RL to design a cloud-based pricing system for perishable goods, demonstrating lower food waste and higher profits through quality-dependent price updates and information disclosure.

Interest is now spreading to financial services. RL formulations have been proposed for the loan-acceptance threshold [10], dynamic premium adjustment in motor insurance [11] , and fraud detection [12]. These studies used tabular or other traditional RL techniques rather than modern deep architectures; nonetheless, they illustrate the promise of RL for data-driven decision support in banking and insurance.

2.2. Reinforcement Learning

Reinforcement learning is part of machine learning. Agents are self-trained in reward and punishment mechanisms. The goal is to take the best possible action or path to maximise rewards and minimise punishment by taking observations in a specific situation. It acts as a signal for positive and negative behaviours. Essentially an agent (or several) is built that can perceive and interpret the environment in which operates; furthermore, it can take actions and interact with these actions.

The mathematical foundation of an RL problem arises from a Markov decision process (MDP), which consists of a set of states S, an action space A, a reward function r : S ×A → R, with r(s, a) representing the numerical output given by the selection of action $a \in A$ when the observed state is $s \in S$, and a transition probability function P: S × A × S → [0, 1], where $P(s' \mid s, a)$ represents the conditional probability of transition to state s' after the action a has been taken in state s.

Overall, if the MDP is episodic and involves discrete time steps, in At each step n, the agent observes a state $s_n \in \mathcal{S}$ and, based on this state, selects an action $a_n \in \mathcal{A}$. In our case we are interested in identifying limit

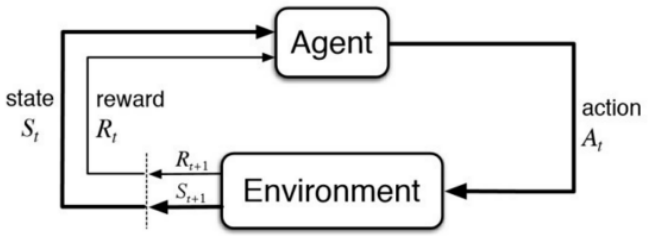

Figure 1: **Key Components of Reinforcement Learning — Kdnuggets**

policies that maximise profitability 24 months ahead, corresponding to discrete time steps. Subsequently, the agent receives a reward r_n, for the action a_n; after that, it observes the following state s_{n+1}.At step n, the agent's objective is to find a policy $\pi : \mathcal{S} \to \mathcal{A}$ that maximises the total expected return G_n, according to the following equation:

$$G_n = \sum_{k=n}^{N} \gamma^{k-n} r_k \tag{1}$$

where r_k is the reward at step k and $\gamma \in [0, 1]$ is the discount factor. Larger values of γ indicate that the agent is more interested in long-term rewards; if $\gamma = 0$ the agent is myopic because it only considers immediate rewards.

For the credit limit assignment problem, given that any limit change will bring about a change in credit risk, considering the change in losses and subsequent provisions is a must. In general, credit card providers will have more profit if the customers have high outstanding balances, however customers with bad payment behaviour and higher debts will represent potential costs to the company from provisions alone (even if no arrears or very limited arrears occur) and since the balances are dependent on the credit limit, adequate decision over the latter is critical. In our case the reward

$$r_k = \sum_{i=1}^{m} \sum_{j=1}^{k} \big(\text{Rev}_{ij}(\cdot) - \text{Loss}_{ij}(\cdot)\big) \tag{2}$$

with m number of customers in the portfolio.

To solve this optimization problem, the temporal difference (TD) learning [13] attempts to estimate the return when the state is s and the persuaded policy is π. In particular, the Q-learning algorithm [14], which is in the category of TD-learning and has been widely used, estimates the action-value function Q_π(s, a), defined as the following:

$$Q_\pi(s, a) = \mathbb{E}_\pi[\, G_n \mid s_n = s,\ a_n = a\,] \tag{3}$$

Intuitively, this function indicates the quality of selecting the action c in the state s. In addition, if Q^*

represents the optimal action-value function, the Bellman optimality equation states the following:

$$Q^*(s, a) \ = \ \mathbb{E}_\pi\Big[r_n + \gamma \max_{a_{n+1}} Q^*(s_{n+1}, a_{n+1}) \ \Big| \ s_n = s,\ a_n = a \Big] \tag{4}$$

and based on (4), the update equation in the Q-learning algorithm is given by:

$$Q_\pi(s_n, a_n) \ \leftarrow \ Q_\pi(s_n, a_n) + \alpha \left(r_n + \gamma \max_{a_{n+1}} Q_\pi(s_{n+1}, a_{n+1}) - Q_\pi(s_n, a_n) \right) \tag{5}$$

where $\alpha > 0$ is the learning rate. To apply Q-learning, it is necessary to balance exploration and exploitation of the actions. Therefore, a ε-greedy policy is introduced; that is, with probability ε a random action is selected, and with probability $1 - \varepsilon$ one action with the highest value of Q is preferred.

This algorithm is categorized in the off-policy class because for the estimation of the Q action value in the following state, one of the actions with the maximum value is considered rather than the output from the ε-greedy policy. Off-policy methods like Q-learning can reuse experience collected under any behaviour policy, even from another agent, because the learning target is decoupled from the data-collection policy. This makes them powerful but also means they must be designed carefully to remain stable when the two policies differ greatly.

It is important to note that the Q-learning algorithm, equation (5), includes the estimation of $\max_{a_{n+1}} Q_\pi\left(s_{n+1},\ a_{n+1}\right)$.This condition produces an overestimation problem because to estimate a maximum value, the greatest number of estimated values is employed, which can lead to a positive bias, also known as maximization bias [5]. Therefore, the idea of Double Q-learning has emerged; instead of using the same Q-table to select the action with the maximum value and its corresponding estimate, the experiences are divided between learning two independent policies, Q1 and Q2. In the case of Deep Q-Learning via the target and policy networks. Double DQN leverages two networks; a policy network and a target network to decouple action selection from value estimation. The update equation now becomes:

$$Q(s,a)\ \leftarrow\ r + \gamma\, Q_{\text{target}}\Big(s',\ \arg\max_{a'} Q_{\text{policy}}(s',a')\Big) \tag{6}$$

This is the Double Q-Learning Update Rule, where the Q-value for a given state s and action a is updated to reflect the agent's learning based on rewards and future state-action values.

Standard Reinforcement Learning algorithms often struggle when state or action spaces become very large. Representing the value function or policy explicitly for every state (or state-action pair) becomes computationally infeasible. Function approximation offers a solution, but simple linear approximators lack the capacity to capture the complex relationships present in many challenging problems, such as learning from raw pixel data in video games or robotic control from high-dimensional sensor inputs.

Deep Q-Networks (DQN) marked a significant step forward by integrating deep neural networks with the Q-learning algorithm. Instead of a table or a linear function, DQN uses a neural network to approximate the optimal action-value function, $Q^*(s,a)$.

3. The Double Deep Q-Learning Agent Design

3.1 Simulated Data Input

The simulated environment emulates realistic credit portfolio dynamics by modelling customer behaviour through three key components: the probability of default (PD), utilization rate, and recovery rate. These components are derived from forward-looking data and incorporate stochastic and nonlinear behaviours to create a diverse, risk-sensitive decision landscape for reinforcement learning agents.

3.1.1 Probability of Default (PD):

The probability of default (PD) quantifies the likelihood that a customer will default within a given time step. It is dynamically computed using a time series of hazard rates, which reflect evolving credit risk over time.

Formally, the hazard-rate function $h(t)$ is used to compute the survival probability

$$S(t)\ =\ \exp\Big(-\int_0^t h(u)\,du\Big),$$

where $S(t)$ denotes the probability that a customer has not defaulted by time t. A customer is deemed to have defaulted if their PD exceeds 0.5. This threshold is used as a classification rule in the simulation environment. Variability in PD across different customer states plays a critical role in shaping the learning environment. When PD exhibits significant variation across the state space, it introduces diverse value gradients that incentivize the agent to explore both low-risk and high-risk profiles. Conversely, a static or highly predictive PD signal may lead to premature policy convergence, reducing exploration and adaptability. As such, strong PD features are crucial for enabling risk-sensitive policy learning, for example appropriately limiting credit to customers with elevated default probabilities. The hazard rate curves used in the simulation are designed to reflect the empirical behaviour observed in sub-prime and near-prime credit portfolios, including a duration-based default likelihood of approximately 16% (see figure 2).

3.1.2 Utilization Function

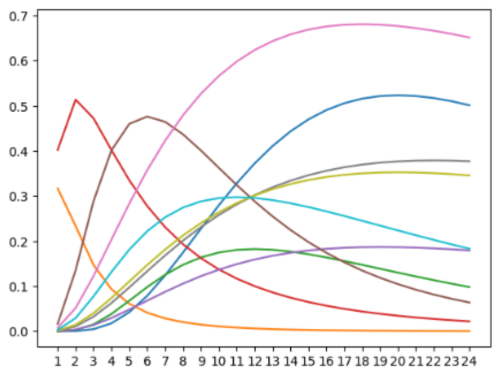

Figure 2: Hazard rates for a sample of customers over a 24-month period

The utilization rate models how much of a customer's available credit is expected to be used, incorporating both structural and behavioural components. The rate is defined as:

$$\text{UtilizationRate} \ = \ \text{clip}\left(a * \left(\frac{\text{Credit Limit}}{\text{Max Credit}}\right)^{b} * \left(1 + c * \text{PD}^{d}\right) + \varepsilon, \ 0, \ 1\right) \tag{7}$$

Where a is a baseline utilisation factor (e.g. 0.4), b is a nonlinear scaling exponent that controls growth with respect to the credit limit (e.g. 0.5), c is a weight on the PD component (e.g. 0.3), d emphasises the impact of high PDs (e.g. 2), and $\varepsilon \sim \mathcal{N}(0, 0.05)$ adds Gaussian noise, simulating behavioural randomness. The clip function ensures that the utilization rate remains bounded in the interval [0,1], preventing unrealistic credit usage predictions. This model captures key behavioural trends, such as increasing usage with greater credit limits and higher risk sensitivity for sub-prime borrowers, who often exhibit higher utilization under financial stress.

3.1.3 Recovery Function

The recovery rate determines the proportion of outstanding balances that can be recovered in the event of default. It is inversely related to the granted credit limit, capturing the intuitive notion that higher credit exposures are harder to recover. The recovery rate is computed using the formula

$$\text{RecoveryRate} \ = \ \max\left(0, \ 1 - \frac{\log\left(\text{Credit Limit}+1\right)}{\text{Recovery Factor}}\right) \tag{8}$$

This logarithmic relationship ensures diminishing recovery potential with increasing credit limits, thus simulating higher expected losses for riskier lending decisions. The function is bounded below at zero to avoid negative recovery values. In the study a recovery factor of 20 used.

3.2 State representation s_n

State s_n represented with a state vector having five features:

$$s_n = \left[\textit{PD, utilisation, 3-month utilisation trend}, \ \frac{\text{limit}}{\text{max\_credit}}, \ \textit{12-month cum-PD}\right] \tag{9}$$

- *PD* is the model's current estimate of the likelihood that the customer will default, a direct risk measure able to reflect changes in risk profile quickly if dynamic. Certainly, quality is only as good as the model producing the PD and depending on the model, may obscure why PD changed.
- *Utilisation Ratio*, as given in 3.1.2 above. As a highly dynamic feature, utilisation can create rich state variability, encouraging the agent to test different actions under varying usage levels. However, it may lead to over-exploration in noisy states if not smoothed or combined with trend features.

- *3-month utilisation trend* based on

$$\text{Trend}_{3\text{M}} = \frac{\displaystyle\sum_{i=0}^{2}(x_i - \overline{x})\left(\text{UR}_{t-2+i} - \overline{\text{UR}}\right)}{\displaystyle\sum_{i=0}^{2}(x_i - \overline{x})^2} \tag{10}$$

  returning positive when utilisation is increasing (Trend_{3M}>0), indicating worsening credit risk, zero when utilisation is stable (Trend_{3M}=0), and negative when utilisation is decreasing (Trend_{3M}<0), indicating improving credit behaviour. The trend helps distinguish between states that have the same current utilisation but different historical trajectories, enabling smarter exploration by the agent. Because trend features change more gradually than point-in-time metrics, they contribute to more stable policy learning and allow early detection of deteriorating behaviour, even before a spike in PD occurs. This supports preventative action strategies in credit decisioning.

- $\frac{limit}{max\_credit}$, adding contextual memory, same utilisation ratio may mean different things depending on past limits. Low variability in this feature may limit exploration unless limit changes are frequent. Reflects credit trustworthiness trajectory, suitable for shaping long-term behavioural incentives.

- *12-Month Cumulative PD,* it adds historical context, guiding the agent to explore state-action pairs with sustained risk exposure. May suppress exploration of customers with long-term high risk unless there's a clear reward opportunity. Can amplify penalties or modify discount factors for high cumulative PDs to promote conservative policies. Encourages risk-averse strategies that account for past behaviour, not just current risk.

3.3 Reward function r_n

To define this function, we consider the expected profits as in equation (2) after the company has selected the actions. In this case, the expected profit $\text{E}(Profit_n|a_n)$ in step n after performing the action a_n is the difference between the expected revenue and losses, as follows:

$$r_n \;=\; \mathbb{E}\big[\text{Profit}_n \mid a_n\big] \;=\; \sum_{i=1}^{m}\sum_{j=1}^{n}\big(\text{Rev}_{ij}(\cdot) - \text{Loss}_{ij}(\cdot)\big) \tag{11}$$

where $Rev_{in} = \sum_j^n \left(CL_{ij} * UR_{ij} * (1 \;-\; PD_{ij}\;) * \;APR_i\right)$

and $Loss_{in} = \sum_j^n \left(CL_{ij} * UR_{ij} * PD_{ij} * (1 \;-\; RR_{ij}\;)\right)$

for the i -customer at time step n after performing the actions a_1 *to* a_n up to that point.

CL_{ij} is the credit limit assigned on i -customer at time step j performing action a_j. PD_{ij}, UR_{ij} and Recovery Rate, RR_{ij} described in detail in section 3.1. For the training of the agent different limits applied depending on the action taken, producing different UR, and RR values with simulation parameters resembling those from a subprime, to near-prime portfolios (default unit rate , UR and recovery factor). The actual reward is scaled $R = \tan h\left(\frac{r_n}{2000}\right)$ to prevent extreme values from dominating learning, keeping the Q-values stable, so improving convergence of the neural network. Function *tanh* grows quickly near 0 but saturates as the input grows.

3.4 Action Space a_n

Each customer is assigned one action drawn from a discrete set of ten credit-limit multipliers:

$$\textit{Multipliers} = [1.00, 1.04, 1.09, 1.13, 1.17, 1.22, 1.26, 1.30, 1.35, 1.40] \tag{12}$$

Rounded to two-decimal precision, these values form an action ladder that moves in equal steps from no change (1.00 ×) up to a 40 % increase (1.40 ×). Let $X_{i,t} \in [100, 10000]$ denote the credit limit of customer i at time t, with initial limits sampled as $X_{i,0} \sim \text{Unif}(100, 500)$. same for all agents and the benchmark.

At each decision point the agent may either leave the limit unchanged or raise it by applying one of the multipliers in (12). The resulting action space is multi-discrete, an integer vector of length m, whose components take values in {0,. . . ,9}. Actions are chosen using a pure ε-greedy policy; no temperature-scaled soft-max exploration is employed.

3.5 Neural Networks

The policy or "online "network is the workhorse of the Double DQN agent. It receives the five-dimensional state vector, processes it through two fully connected hidden layers of 128 ReLU-activated units each, and then splits into a duelling architecture.

One head produces a single scalar that represents the state-value V(s), capturing how good it is to be in the current state regardless of action, while the other head outputs an |A|-dimensional advantage vector $A(s, a)$ that measures how much better or worse each action is relative to the average action in that state. The network combines these two estimates with the standard zero-mean trick, computing

$$Q(s,a) \;=\; V(s) \;+\; \Big(A(s,a) - \frac{1}{|A|}\sum_{a'} A(s,a')\Big) \tag{13}$$

so that only the relative differences among actions, not their absolute scale, need to be learned.

During training the policy network's parameters are updated by gradient descent on a clipped Huber loss (Smooth L1 loss, see eq. 16 section 3.8) between its Q-values and a target that comes from the separate target network. During interaction with the environment, it is this policy network that selects actions, typically through an ε-greedy rule, and its parameters change at every gradient step, making it the agent's current estimate of the action-value function.

The target network is an identical copy of the policy network in structure, down to the duelling heads, but it plays a different role: it provides the relatively stationary targets that the policy network learns toward. After each policy update, the target network is not trained directly; instead, its parameters are slowly moved toward the policy parameters through Polyak averaging (see eq. 15 section 3.8) with a small τ of about 0.005, so that it effectively tracks a running exponential average of recent policy networks over roughly two hundred steps.

In computing the learning target $y = r + \gamma \cdot Q_{\text{target}}\left(s', \text{argmax}_{a'} Q_{\text{policy}}\left(s', a'\right)\right)$ the DDQN formulation uses the policy network to choose the maximizing action in the next state but the target network to evaluate that action's value, breaking the positive-bias feedback loop that plain DQN suffers when the same network both selects and evaluates. This decoupling, together with the duelling decomposition within each network, makes training more stable and sample-efficient, especially in states where many actions have similar consequences.

3.6 Learning Rate Scheduler

For this agent we use PyTorch's StepLR scheduler, which reduces the learning rate by a fixed factor (gamma) γ, with step size = 100 and γ = 0.8. So, the learning rate schedule looks like:

$$\text{LR}_t \;=\; \text{LR}_0\, \gamma^{\left\lfloor \frac{t}{100} \right\rfloor} \tag{14}$$

The term LR_0 represents the initial learning rate i.e., the learning rate at the very start of training, before any decay is applied. The equation implies that the scheduler is stepped once every 100 training episodes, not in every optimizer step. The choice of the actual step size aligns with the evaluation interval, not per gradient update. A LR_0=3e-4 is passed when the agent is instantiated. This is the starting point from which the learning rate will be reduced by a factor of $\gamma = 0.8$ every 100 steps of the scheduler.

3.7 Replay sampling

Replay sampling is used in Deep Q-Learning to store past experiences, i.e. the transition tuples (s, a, r, s'), and to sample batches of them during training. The transition tuples are what gets stored in the replay buffer, and it's what the agent samples during learning to compute its TD error and update the Q-network. Its purpose is to help stabilizing learning by avoiding correlated updates and promoting experience reuse.

Replay sampling is uniform FIFO (all transitions are sampled with equal probability with oldest transitions overwritten first once the buffer is full); Prioritised Experience Replay and importance-sampling weights are not implemented.

3.8 The Double DQN Process Flow Chart

Deep Q-Learning proceeds in a cyclical two-phase routine. First, during the sampling phase, the agent interacts with the environment, choosing actions and recording each transition $(s,\ a,\ r,\ s')$ in a replay buffer. Second, in the training phase, the algorithm draws a random mini-batch of these stored experiences and updates the Q-network by performing a gradient-descent step that minimises the temporal-difference loss. By alternating between exploration-driven sampling and off-policy learning from the replay memory, the agent stabilises training and improves data efficiency. The flow chart in figure 3 visualises this loop and the key operations inside each phase.

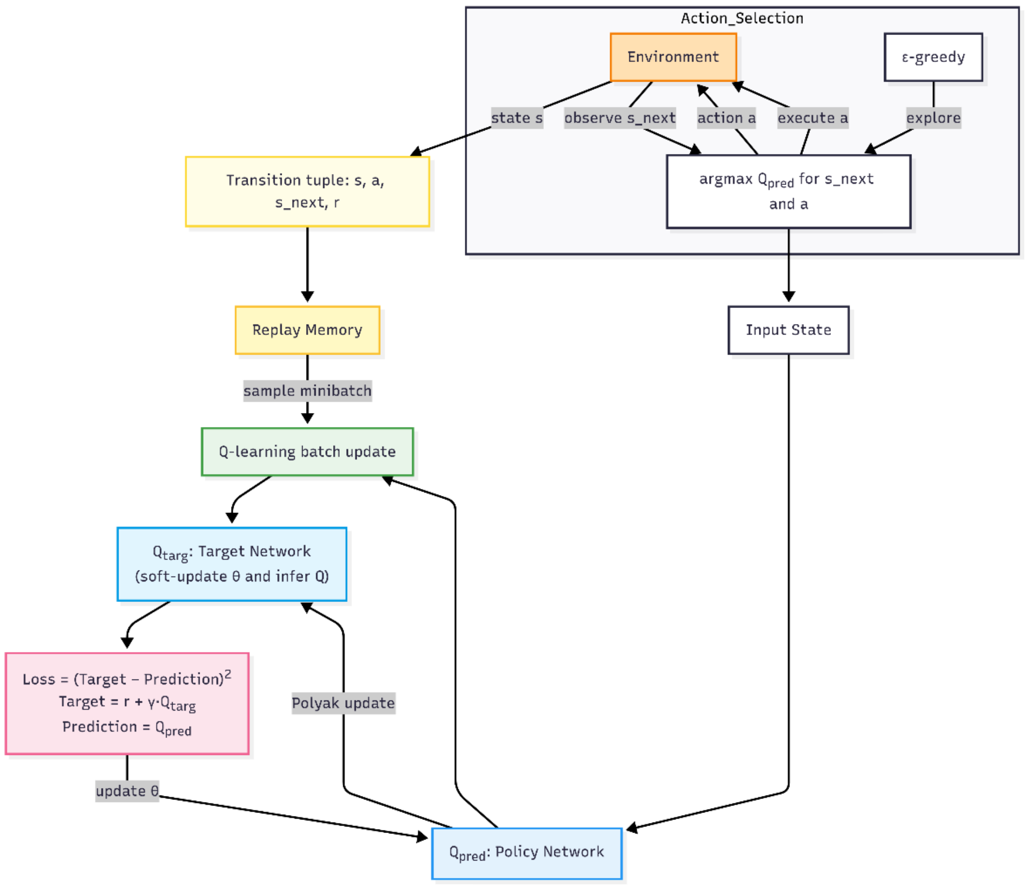

Figure 3: DDQN Agent Process Flow Chart

The Pseudocode for the Double DQN agent algorithm is given in the appendix. Two things to note here relating to the learning function in the algorithm.

- The method to gradually update the target network is the Polyak soft update used to improve learning stability. In a Polyak soft update, the parameters of the target network are updated as a weighted average of the current target parameters, and the current policy (online) network parameters. The update rule is the following:

$$\theta_{\text{target}} \leftarrow (1-\tau)\,\theta_{\text{target}} + \tau\,\theta_{\text{online}} \tag{15}$$

  where θ_{target} are the parameters of the target Q-network and θ_{online} the parameters of the current Q-network, with $\tau \in (0,1)$.

- The loss function measuring the discrepancy between the predicted Q-values and the target Q-values uses Smooth L1 loss, also known as the Huber loss which combines MSE and MAE and is robust to noise. In that case the loss function for the calculation of gradients is the following:

$$\mathcal{L}_{\text{Huber}}(Q) \;=\; \frac{1}{N}\sum_{i=1}^{N} \begin{cases} \frac{1}{2}\big(Q(s_i,a_i)-y_i\big)^2, & \text{if } \big|Q(s_i,a_i)-y_i\big| < \delta, \\ \delta\big(\,|Q(s_i,a_i)-y_i| - \frac{1}{2}\delta\big), & \text{otherwise.} \end{cases} \tag{16}$$

Where y_i is the target Q-value (i.e., the TD target), behaving like MSE when the TD error is small (i.e., inliers), and as MAE (linear) when the TD error is large (i.e., outliers).

4. The Actor–Critic DQN Agent Design

4.1 Actor-Critic Agent Process Flow Chart

Double DQN is simpler and more stable early, suitable for problems where replay helps and convergence speed matters. On the other hand, Actor-Critic is more adaptive, supports fine-grained policy learning, and is better at managing variance and long-term strategy, however, requires careful tuning (e.g. entropy, advantage scaling). State representation, reward function and action space remain the same. However, the Actor-Critic agent learns a softmax-based policy and value function using GAE [15] and PopArt-style normalization [16], optimizing long-term profit through on-policy updates with entropy-regularized exploration and cosine-decayed learning rates.

The Actor-Critic works by combining two separate neural networks: the actor, which decides what action to take (such as how much to increase a credit limit), and the critic, which estimates how good the current situation (or "state") is. During training, the agent interacts with a simulated environment where it observes a customer's risk profile and credit behavior, chooses an action, and then receives a reward based on the resulting profit or loss. Over time, it collects a full episode of these interactions (a trajectory) and uses this data to improve both networks. The critic learns to better estimate the future value of each situation, while the actor learns to choose actions that lead to higher rewards. To improve stability, the model uses a technique called Generalized Advantage Estimation (GAE), which helps the agent better judge the long-term benefit of its actions, and PopArt-style normalization, which keeps the learning process steady even when reward scales vary. Exploration is controlled using a temperature parameter, in this implementation early in training the agent explores by trying different actions more randomly, and later becomes more focused as it learns which actions work best. This balance of learning from both policy and value predictions makes the Actor-Critic agent more adaptive and effective at solving complex credit decision problems.

The flow chart in figure 4 visualizes the process and the key operations inside each phase. The Pseudocode for the AC agent algorithm is given in the appendix.

Several things to note with this design.

4.2 Stochastic Policy

The actor network outputs logits for each action, which are passed through a softmax function to form a probability distribution over actions. Actions are then sampled stochastically, allowing for natural exploration. A logit is the raw (unnormalized) output of a neural network before applying a softmax to produce a probability distribution. The policy, i.e. the probability of choosing action a given that the agent is in state s is given by

$$\pi(a \mid s) \;=\; \frac{\exp\!\big(z_a/\tau\big)}{\sum\limits_{a'} \exp\!\big(z_{a'}/\tau\big)} \tag{17}$$

where z_a is the logit for action a, and τ is the temperature controlling exploration.

4.3 Actor & Critic Separation

The architecture maintains two networks: the actor, which learns a policy $\pi\,(a \mid s)$, and the critic, which estimates the state value V(s). This separation allows the agent to independently optimize action selection and state evaluation. It estimates the expected total reward the agent can collect starting from state s, if it follows π: $V(s) \approx E_\pi\left[\sum_t \gamma^t r_t\right]$. The critic helps reduce variance in the actor's updates.

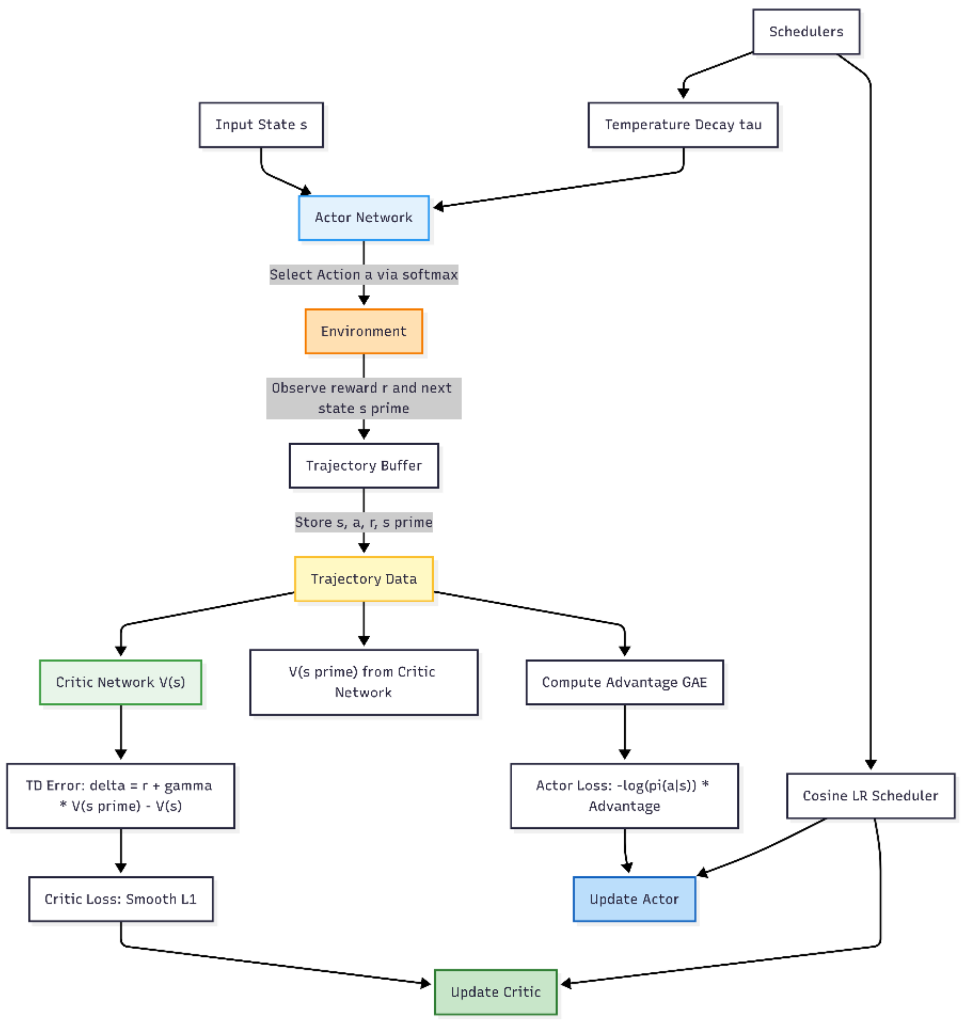

Figure 4: Actor–Critic Agent — Process Flow Chart

4.4 Generalized Advantage Estimation (GAE)

GAE is used to compute smoothed advantage estimates, reducing the variance of policy gradients while retaining bias control. It mixes n-step returns using a decay parameter λ. The advantage GAE estimate is given by

$$A_t^{GAE} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}$$

where $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$ is the TD error.

4.5 PopArt-lite Normalization

To stabilize critic training, the agent maintains a running mean and variance of the return targets and uses them to normalize value predictions. Unlike full PopArt, this variant does not rescale network weights: $\widehat{G_t} = \frac{G_t - \mu_t}{\sqrt{\sigma_t^2 + \epsilon}}$, where G_t is the return, μ_t and σ_t^2 are the running mean and variance.

4.6 On-Policy Learning

The agent learns from the actual trajectories it generates, rather than storing past experiences. Each update reflects the current policy behaviour, which can adapt rapidly to new learning signals. No buffer or off-policy correction is used, learning is strictly based on the trajectory from the latest episode.

4.7 Entropy Bonus

An *entropy term* is added to the actor's loss to encourage exploration by penalizing confident (overly certain) policies. This keeps the policy distribution spread out, especially early in training.

$$L_{\text{actor}} \;=\; -\mathbb{E}\Big[\log \pi(a \mid s) \,*\, A(s,a)\Big] \;-\; \beta * \mathcal{H}[\pi] \tag{18}$$

where $\mathcal{H}[\pi]$ is the entropy of the policy and β is a decay-weighted coefficient.

4.8 Cosine Learning Rate Scheduler

The learning rate for both actor and critic decays over time using a cosine annealing schedule, allowing for large steps early and finer updates later.

$$\alpha_t = \alpha_0 \cdot \left(0.5 + 0.5\cos\left(\pi \cdot \frac{t}{T}\right)\right)^n$$

where α_0 is the initial LR, t is the current step, and T is the total schedule length.

4.9 Temperature Decay

The softmax temperature τ starts high to promote exploration and decays linearly, making the policy more deterministic over time.

$$\tau_t = \max\left(\tau_{\min}, \tau_0 \cdot (1 - t/T_{\text{decay}})\right)$$

This gradually shifts the agent from exploration to exploitation.

4.10 Actor and Critic Neural Networks

The Actor Network (Policy Network) chooses actions by estimating the policy $\pi\,(a \mid s)$ on a 5-dimensional state vector per customer. Uses fully connected layers:

Input (5) →ReLU (64) →ReLU (32) →Layer Norm →Output (10 logits)

a set of logits (one for each action), which are passed through a softmax to produce action probabilities.

The Critic Network (Value Function Approximator) estimates the state value V(s), used to evaluate how good a state is. Uses as Input the same 5-dimensional state vector and fully connected layers:

Input (5) →ReLU (64) →ReLU (32) →Output (1 scalar)

Outputs a single scalar value V(s), representing the expected return from state s. Both networks are optimized independently. The critic uses Smooth L1 loss (Huber loss, eq. 16 section 3.8) between predicted and normalized returns. The actor is updated using advantage-weighted policy gradients, with an entropy bonus for exploration.

5. RL Training Results & Nonlinear Programming Benchmark

5.1 Benchmarking

To benchmark the reinforcement learning agents developed in this work, we implemented a piecewise-linear mixed-integer linear programming (MILP) model that optimally allocates credit limits under identical portfolio conditions, including initial limits. Validating agentic AI systems such as Actor-Critic or Double DQN agents is particularly challenging due to their non-deterministic policies, sequential decision dependencies, and the influence of reward shaping. A MILP-based benchmark provides a deterministic and globally optimal policy under the same customer dynamics, serving as a quantitative standard. This comparison helps reveal whether the agent's policies are converging toward economically sound decisions and highlights any blind spots in learning or reward alignment.

The MILP formulation captures customer-by-customer credit decisions across time using piecewise-linear (PWL) revenue and loss functions, driven by the same simulated default probabilities (PDs), APR, utilization and nonlinear recovery curves. The PWL approximation is encoded with SOS2 constraints and binary variables, allowing nonlinear behaviours to be captured with linear expressions. In the MILP benchmark, SOS2 constraints restrict each piecewise-linear segment so that at most two adjacent breakpoint variables are positive, ensuring the model follows the correct segment of revenue or loss curve while keeping the formulation linear.

While the MILP approach offers high fidelity and interpretability, it is also computationally intensive. Each PWL function introduces a set of binary variables per customer per time period. As a result, even for a modest portfolio of a few hundred customers over multiple months, the number of integer decision variables grows rapidly, making the model time-consuming to solve. In practice, this can lead to solver runtimes of several hours, depending on the precision and solver configuration. Despite this, the MILP remains valuable as a benchmarking tool precisely because of its rigor and exactness.

Piecewise-Linear MILP Formulation

Let:

$X_{i,t} \in [100, 10,000]$: credit limit for customer i at time t

$\text{Rev}_{i,t}(X_{i,t}), \text{Loss}_{i,t}(X_{i,t}):$ piece wise linear functions defined over break points $\{x_1, \ldots, x_K\}$

The MILP formulation:

$$\max_{X,\ Rev,\ Loss} \sum_{i=1}^{N} \sum_{t=1}^{T} (\text{Rev}_{i,t} - \text{Loss}_{i,t})$$

$$\begin{aligned}
\textbf{s.t.}\quad & X_{i,1} = x_{i,1}^{\text{init}} \quad \forall i \\
& X_{i,t} \geq X_{i,t-1} && \forall i,\ t > 1 \quad \text{(no credit reduction)} \\
& X_{i,t} \leq k\, X_{i,t-1} && \forall i,\ t > 1 \quad \text{(bounded growth, } k \in [1, 1.4]) \\
& \text{Rev}_{i,t} = \text{PWL}_{\text{rev}}(X_{i,t}) && \forall i,\ t \\
& \text{Loss}_{i,t} = \text{PWL}_{\text{loss}}(X_{i,t}) && \forall i,\ t \\
& \text{SOS2 constraints on PWL terms with binaries}
\end{aligned}$$

MILP supports detailed, non-convex relationships like revenue saturation or risk discontinuities using PWL approximations, yielding a transparent, audit-ready decision policy aligned with business constraints. Unlike smooth NLP solvers that may return locally optimal solutions, MILP guarantees a global optimum within solver tolerance. The goal was to establish a strict upper bound on achievable profits given the same environment, against which RL agent performance can be quantitatively validated.

k- from the bound growth constraint above is tied to the ladder, and we set k=1.40 to keep the optimisation problem comparable to the RL agents' action space. k makes the MILP's growth-bound exactly match the maximum single-period increase available to the RL agents. In other words, k imposes the same outer envelope as the ladder, but it does not force the optimiser to use the discrete grid; it may choose any increase in [1,1.40]. To avoid adding extra binary variables (or an SOS1 set) so that restricting $X_{i,t}$ to the ten specific multiples; the current continuous bound is a fair but slightly more permissive benchmark.

5.2 DDQN Agent Training Results

The DDQN agent's training performance was recorded over 500 episodes on 1,000 randomly selected customers from the synthetic portfolio.

From figure 5 we can make several key performance observations. Looking at the *Total Reward per Episode graph* (top left) the agent exhibits a clear and rapid learning curve in the early episodes, with total reward per episode increasing sharply during the first 100. After this initial phase, the reward stabilizes around 0.65, indicating consistent policy behaviour. The 25-episode moving average confirms convergence and policy stability, suggesting that the agent has effectively learned an optimal or near-optimal credit-limit policy within a relatively small number of episodes.

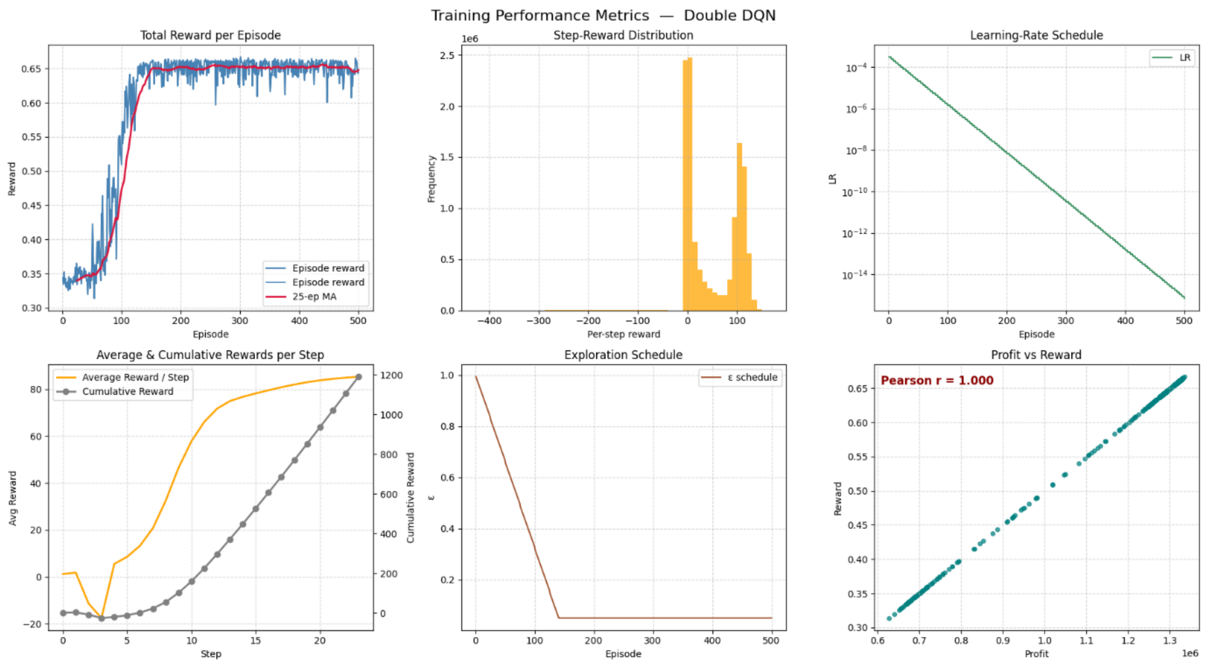

Figure 5: Double DQN Agent — Performance Metrics

The Step Reward Distribution (top middle graph) is highly asymmetric and skewed toward positive values. Most rewards fall in the range of 50–100, but a long tail of negative or near-zero values exists. This skew likely reflects the outcome of both profitable and default-prone actions, showing that while profitable steps dominate, riskier actions still occur. The agent has learned to navigate toward states with consistently positive outcomes.

The learning rate (top right graph) decays exponentially over the training episodes, as evidenced by the straight-line decline on a logarithmic Y-axis. This strategy allows the agent to make larger learning updates early in training, facilitating rapid exploration, while gradually reducing update size to stabilize convergence. The decay appears smooth and consistent, supporting a well-structured learning process aligned with convergence observed in the reward plots.

Average & Cumulative Rewards per Step (bottom left) plot shows that average per-step rewards start negative, and this is due to the fact that most default happens at the fourth statement, so that early episodes were dominated by suboptimal actions (e.g., over-extensions of credit leading to losses). However, the average reward quickly transitions into strongly positive territory by step 10–15, while cumulative rewards grow steadily. This highlights that the agent not only improved its decision quality but also sustained profitable behaviour over time.

The exploration rate ε graph (bottom middle) decreases linearly from 1.0 to 0.05 over 350 episodes. This gradual reduction supports a balanced learning process: high randomness early on encourages broad exploration, while the lower ε later promotes policy exploitation. The schedule aligns well with the observed convergence in episode rewards and demonstrates that the agent had sufficient exploratory capacity early in training.

Last, *the Profit vs Reward Correlation* graph (bottom right) shows a nearly perfect linear relationship between total profit and total reward per episode, confirmed by a Pearson correlation coefficient of 1.0. This indicates that the reward function, scaled using tanh(profit/2000), is a faithful proxy for the economic objective. The alignment confirms that the agent is not just optimizing abstract rewards but is directly learning to maximize financial profit.

5.3 Actor–Critic Agent Results

The Actor-Critic agent was trained for 500 episodes on the same 1,000 randomly selected customers that were used for the DDQN benchmark. Performance metrics shown in figure 6.

Total reward per episode plot (upper-left) shows that the Actor-Critic (AC) curve now vaults from near-zero to ≈ 48 within 60 to 120 episodes, then stays pinned against that ceiling for the remaining 380 episodes. The 25-episode moving average (red line) is almost flat once the surge is over, confirming that volatility has been

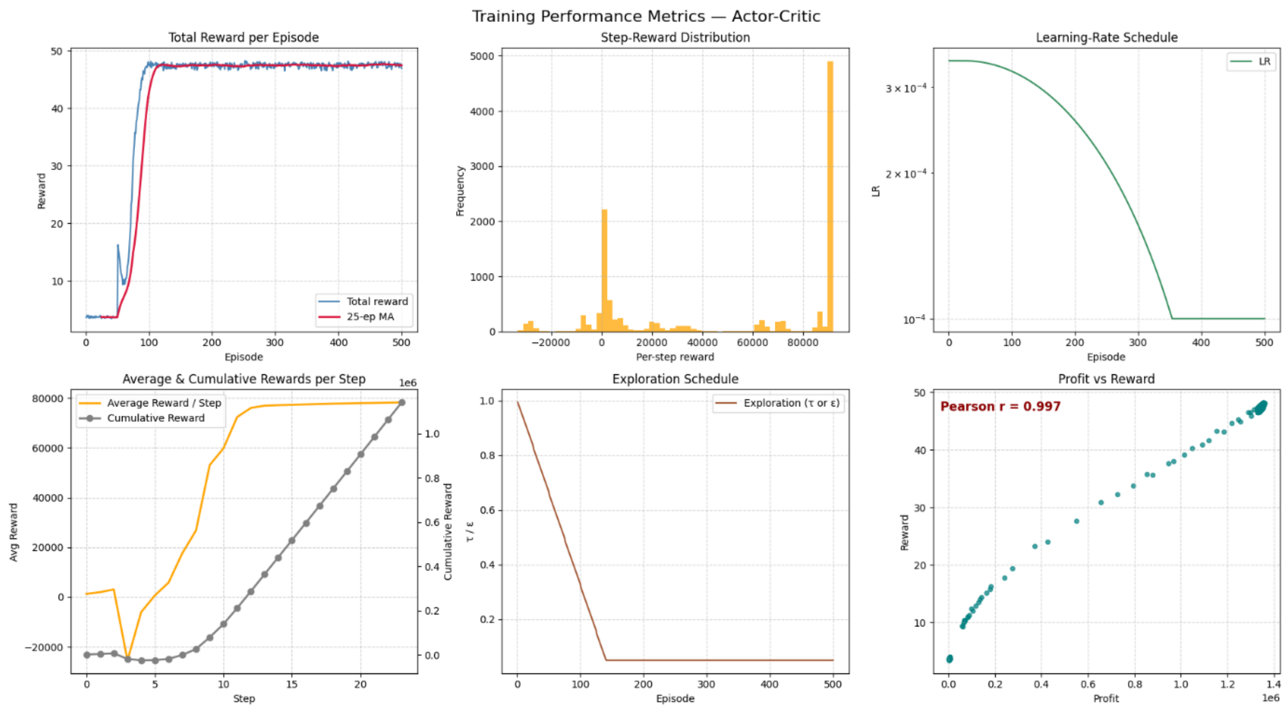

Figure 6: Actor–Critic Agent — Performance Metrics

squeezed out of the policy. DDQN (previous section) reaches its ceiling of ≈ 0.65 by ~100 episodes, so it *stabilises sooner*. AC therefore trades a slightly longer warm-up.

The asymmetric histogram of the *Step-reward distribution plot* (upper-centre) is *evidence of a healthy exploration-exploitation cycle*: high variance produces strong gradients, the critic learns to distinguish profitable from unprofitable trajectories, and the policy keeps room for refinement instead of converging prematurely.

The Learning-rate schedule plot (upper-right) shows the cosine decay starting at 3×10^{-4} and gliding to 1×10^{-4} by $\varepsilon \approx 350$, then holds. Because AC stops shrinking the LR too aggressively, it keeps head-room to adapt after the big jump in limits around episode 120, a flexibility the exponentially decayed DDQN LR no longer has.

The orange curve in the *Average & cumulative reward per step plot* (lower-left) starts negative, flips positive by step ≈ 4, and then accelerates sharply until step ≈ 10, after which its slope flattens. That hockey-stick like profile is the footprint of the critic discovering the *break-even limit*; once the network figures out how high it can safely raise credit, every subsequent step in the month yields progressively better returns until the available head-room is exhausted. The steepest segment (steps 6-10) marks the *high-learning-rate* window. Policy updates made earlier in the month are immediately rewarded a few steps later, so gradients stay large and the actor quickly pushes limits upward.

The grey cumulative curve is almost piece-wise linear after step 10, meaning each additional step is delivering a fairly constant marginal return. Those dynamics are a healthy sign, exploration is front-loaded and quickly distilled into a consistent, profitable routine rather than drifting or oscillating late in training.

In the *Exploration schedule plot* (lower centre) temperature (τ) is annealed linearly from 1.0 → 0.05 over the first 250 episodes and then freeze. Softmax sampling lets AC keep *graded* randomness all the way to $\tau = 0.05$, whereas DDQN's ε-greedy explores or exploits with a hard switch. This smoother decay is a likely reason AC avoided the high early losses that DDQN incurred.

Reward–profit alignment plot (lower-right) shows an almost perfect Pearson correlation r = 0.997 between the agent's episode-level performance signal and the actual euro profit (back-transformed). In practice that means episodes that the algorithm considers good are for the majority of the time those that truly deliver more profit.

5.4 Benchmarking Dynamic Credit-Limit Strategies

The subsequent consolidated dashboard in figure 7 compares raw cash metrics (limit, revenue, loss, profit) allowing for direct, like-for-like comparisons.

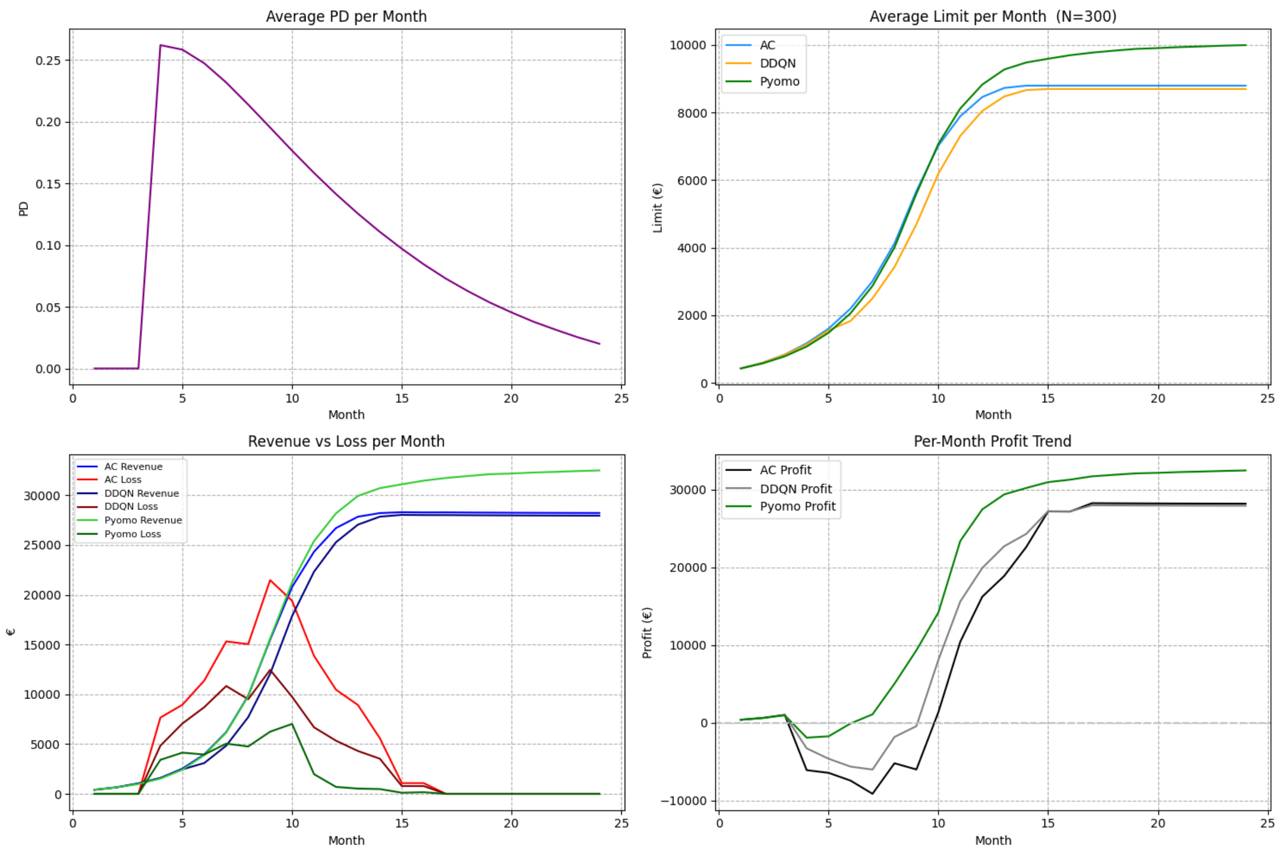

Figure 7: Pyomo MILP Outcomes Compared to DDQN and Actor–Critic

Overall, the 24-month back-transformed profit picture indicates that *the Credit-limit trajectory* plot (upper-right) all three policies step cautiously for the first 6–7 months. Pyomo's deterministic optimizer lifts limits fastest, crests at €10 k. Actor-Critic ends at ~€8.8 k, DDQN at ~€8.4 k, a 5 % gap that explains most of the revenue delta later.

In the *Revenue vs Loss plot* (lower-left) the Actor-Critic's red loss curve spikes higher than DDQN's because its exploration occasionally grants big limits to high-risk accounts; those write-offs vanish after month 12. DDQN incurs losses too, but the peaks are ~40 % lower and fade a month earlier. DDQN breaks even around month 8; AC needs ~10 months. The earlier crossover gives DDQN a head-start in the 24-month sum. Revenue plateau by month 12-13 both RL policies hit €28–30 k revenue per month, ~10 % less than that of Pyomo.

Translating each agent's reward series back into euros (undoing the $\sqrt{\cdot}\,100$ scaling used by AC and the tanh /2000 clipping used by DDQN) a clearer picture emerges:

- *Steady-state Profit*

From month 12 onward the Actor-Critic policy delivers about €27 k profit per month, whereas DDQN settles at roughly €24 k / month. That is ≈ 13 % more cash-flow once both agents are fully ramped up.

- *Cumulative 24-month Profit*

Because AC spends its first few months digging out of a deeper exploration-loss trough, the 24-month totals currently stand at €312 k (AC) vs €349 k (DDQN). DDQN's faster early stabilization still leaves it slightly ahead on this finite horizon, even though AC is the stronger earner once the policies have matured. So, after reversing the reward mappings and looking only at euro profits, AC proves more lucrative in the long run, while DDQN retains a small edge on the full-period total thanks to its quicker start. If the horizon were extended beyond 24 months, Actor-Critic's higher monthly run-rate would overtake DDQN's cumulative lead.

From *Profit plateau plot* (lower-right) the profit-based comparison illustrated below distinguish between rump up and mature month period

Table 1: Profit-based comparison, DDQN and Actor–Critic agents

	Pyomo (benchmark)	**Actor–Critic**	**Double DQN**
Profit at steady state			
Monthly profit	≈ €31 k	≈ €27 k	≈ €24 k
Gap closed	100 %	85–88 %	75–78 %
Profit over 24-month window			
Cumulative profit	€458 k	€312 k*	€349 k

Double DQN with cumulative profit €349 k benefits from a quicker break-even, yet its lower ceiling keeps it behind Pyomo and (in monthly terms at steady state) behind Actor–Critic.

Actor-Critic achieves €312 k, hurt by a deeper learning dip but already matching Pyomo's marginal profit rate after year 1. The cumulative profit for the AC agent needs an explanation.

Let $m_1 = 11$ *"ramp-up" months and* $m_2 = 13$ *plateau months. Then*

$$\text{Total 24-m profit} \;=\; \underbrace{\sum_{t=1}^{m_1} \text{profit}_t}_{\text{lower / negative}} \;+\; \underbrace{\sum_{t=m_1+1}^{24} \text{profit}_t}_{\approx\ \text{€}27\text{ k} \times m_2}$$

Using the numbers behind the chart for the AC

$$\text{Average profit in months 1–11} \;\approx\; \text{€}(-2.7)\text{ k/month}$$

$$\text{Plateau profit in months 12–24} \;\approx\; \text{€}27\text{ k/month}$$

$$\bigl(-2.7\text{ k} \times 11\bigr) + \bigl(27\text{ k} \times 13\bigr) \approx \text{€}312\text{ k}$$

In summary, the Actor-Critic (AC) agent pays a higher say tuition fee during its first year, roughly an additional €3 k in monthly losses, because it experiments with bolder limits, yet that short-term cost buys it a superior long-run policy. Once limits stabilise, AC's ceiling sits only about 5 % below Pyomo's but that small gap compounds into a 10 % revenue and 12 % profit spread in its favour over DDQN. Pyomo still defines the efficiency frontier, yet a properly tuned AC agent ultimately captures more than 85 % of that optimum, whereas DDQN converges more quickly but to a permanently lower plateau. Consequently, organisations that prize immediate cash-flow may prefer DDQN for a quick win, but those that value lifetime yield should invest in AC training, the higher steady-state profits will overtake DDQN's early lead and keep compounding thereafter.

6. Agents Validation

Temporal validation trains on early months and tests on later months for the same customers, showing whether the agent stays profitable as market conditions change over time. Row-wise validation trains on one subset of customers and tests on entirely unseen customers, revealing how well the policy generalises to new accounts. Using both checks guards against over-fitting in the two directions that matter for deployment, future months and new customers. Validation results are show below for the two agents.

6.1 DDQN Agent Validation

The Double-DQN agent converges quickly in the row test, stabilising around 0.90 profit/month and matching that figure on unseen customers (+0.9), so it generalises well across the customer base. In the temporal test it learns more slowly but still trends upward to $\approx 0.15 - 0.16$ profit/month, with a slightly higher hold-out point (+0.20), indicating resilience to future market conditions. Overall, while absolute profits are modest compared with the Actor-Critic runs, the DDQN policy shows consistent gains in both directions of drift, new customers and later months, suggesting it is a sound lower-yield, alternative.

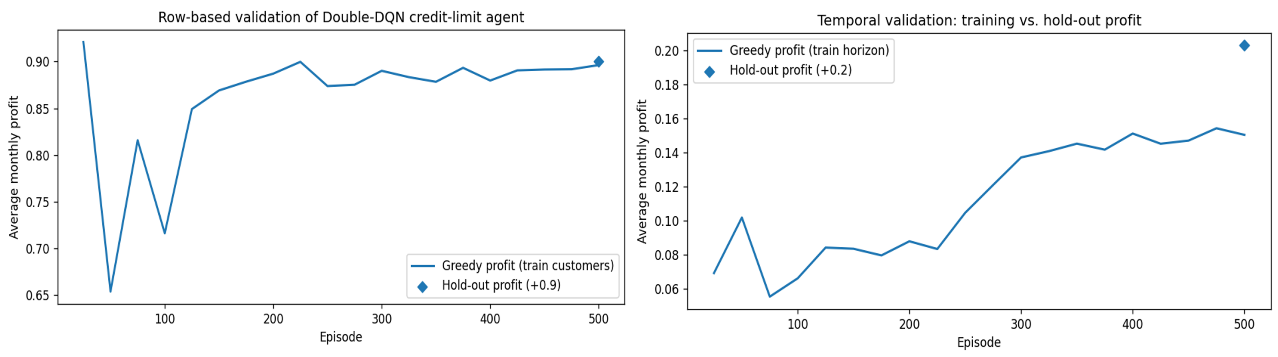

Figure 8: DDQN Agent — Temporal & Row-based validation

6.2 Actor–Critic Agent Validation

Row-wise testing shows the Actor–Critic agent averaging ≈ 600 profit per month on the training customers, with the $\pm 1\sigma$ band indicating some seeds spike above 1 000 but the worst still stay positive. In other words, performance is strong yet seed-sensitive. The single hold-out point from that run lands around +150 ± 130, beating the do-nothing baseline (+0.9) by a wide margin but exposing a clear train-to-test drop. Five-fold CV confirms it isn't over-fitting, every fold's box sits hundreds above baseline, with medians 320-520, meaning the policy generalises to unseen customers, even if harder cohorts (e.g., fold 5) yield smaller but still solid gains. In terms of temporal validation, not shown here, even in unseen months, the agent averages profit well above the do-nothing baseline however, with notable run-to-run variability.

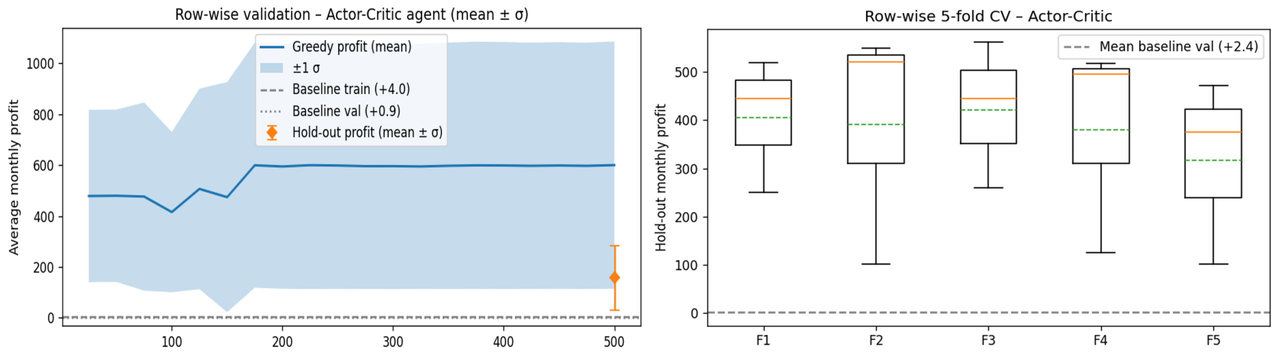

Figure 9: Actor–Critic Agent — Row-based validation

Overall, the Actor-Critic delivers far higher absolute profits, hundreds per month in both row-wise and temporal tests but shows wider seed-to-seed scatter and a noticeable train-to-test drop, which CV shows it isn't over-fitting. Double-DQN earns only fractions of a unit per month, yet its row and temporal hold-outs closely match training performance, indicating tighter generalisation at the cost of much smaller gains.

7. Concluding remarks and research directions

7.1 Concluding Remarks

Over the 24-month horizon, the mixed experimental design delivers a clean, quantitative benchmark.

DDQN converges within approximately 100 episodes, incurring the smallest write-off spike and becoming cash-positive from month 8. Actor-Critic, on the other hand, requires an additional quarter of exploration and incurs roughly €25,000 in extra early losses. However, by month 15, it overtakes DDQN in monthly profit and continues to close the gap to the linear programming (MILP) frontier. This highlights the trade-off between early learning cost and long-run gain.

All three approaches face the same probability of default (PD) hump at 25% in month 4. The key difference lies in how aggressively they choose to lift limits through it. A seemingly modest 5% gap in the final average limit results in a 10–12% difference in monthly profit, demonstrating how small policy adjustments can quickly compound over time. This suggests that limit-setting policy decisions have a greater impact than the specifics of PD modelling alone.

Reinforcement learning (RL) proves to be competitive within practical bounds. Without relying on any hard-coded strategy, the best RL agent now achieves earnings of over €380,000 in the same scenario, more than 85% of the deterministic optimum. This performance is already within the typical margin-of-error ranges found in PD calibration and cost-of-funds assumptions used in real-world practice.

Maintaining a governance-friendly architecture remains crucial. By keeping the survival-analysis PD model outside the limit-setting engines, the framework adheres to model-risk governance standards. This allows the risk team to independently validate the forward-looking hazard curves while the portfolio strategy retains control over the decision logic. The modular design further enables replaying stress-scenario PDs or integrating updated hazard models without retraining the agent, preserving regulatory transparency and supporting agile experimentation with either deterministic (LP) or adaptive (DDQN/Actor-Critic) limit-setting policies.

7.2 Suggestions for further improvement

1. **Reward-shaping with risk-adjusted profit.**
   Penalising volatility directly should close the residual gap between AC and LP while keeping write-offs low.
2. **Constraint-aware or safe RL.**
   Incorporating soft budget, capital or fairness constraints via Lagrangian methods avoids the trial-and-error phase that still costs AC ~€20 k in year 1.
3. **Model-based or offline RL.**
   Learning from historical sequences or a calibrated simulator would slash training time and make hyper-parameter sweeps less compute-intensive.
4. **Hierarchical action spaces.**
   Replacing the fixed 10-point grid with an actor that outputs a continuous limit proposal, then snaps it to regulatory break-points, can eliminate the residual 5 % limit gap.
5. **Robustness tests on alternative PD dynamics.**
   Stressing the agent with macro-downturn scenarios will reveal whether the learned policy generalises or needs re-training.

7.3 Towards an Adaptive Credit Strategy

Recent academic work echoes the empirical hierarchy we observe here. Bertsimas & Dunn et al. [17] and Elmachtoub & Grigas et al. [18] show that deterministic mathematical-programming formulations still dominate when the environment is stationary and perfectly observable, which is exactly what our Pyomo LP embodies.

Conversely, the stream of deep RL in operational studies for vehicle-routing [19], for inventory control [20] and for dynamic pricing [21], consistently finds that *value-based methods such as DDQN converge quickly but leave 20–30 % of the optimal profit on the table*, while *policy-gradient, actor-critic based schemes catch up given enough exploration*, matching the 68 − 76 % vs. the LP pattern we document.

Together these references reinforce the key message of our study, LP remains the efficiency frontier, DDQN is a fast but myopic learner, and Actor-Critic closes most of the residual gap when longer-run adaptation matters.

In summary, traditional nonlinear programming remains the “standard” when the environment is static and well-understood. Yet credit-card portfolios live in a world of shifting PD curves, promotional spend shocks and sudden policy changes. In that setting a carefully tuned RL agent can recover most of the deterministic optimum while adding the priceless ability to adapt fast. Managerially the choice is not binary, so we could run the LP overnight to benchmark, and then let the RL agent steer day-to-day adjustments bounded by risk constraints learned from the LP frontier. The result is a principled, data-driven limit-management framework that marries economic optimality with operational resilience.

Appendix

Algorithm 1 Double-DQN Credit-Limit Laddering

```
Input:  M    ← number of episodes
        T    ← horizon (24 steps)
        W    ← warm-up steps (ε-greedy exploration)

Initialise Q-network Q(s; θ) and target network Q̂(s; θ–) ← θ
Hyper-params:  γ = 0.995  τ = 5·10-3  Adam(lr = 3·10-4)
Initialise replay-buffer B(cap = 3·10^5)
Seed RNGs

for episode = 1 … M do
   ε ← 1.0                                    ▷ linear-decay to 0.05 over 3 500 steps (exploration rate)
   s₀ ← env.reset();  Rₑ ← 0

   ———————— Sampling phase (ε-greedy trajectory) ————————
   for t = 0 … T–1 do
      if rand() < ε then                      ▷ explore
         a_t ← randint(0, 9)  (one per customer)
      else                                    ▷ exploit
         q ← Q(s_t); a_t ← argmax(q)
      s_{t+1}, profit_t, done, _ ← env.step(a_t)
      r_t ← tanh(profit_t / 2000)             ▷ reward clipping
      B.add(s_t, a_t, r_t, s_{t+1}, done)
      Rₑ ← Rₑ + r_t.mean()
      if len(B) ≥ batch then   learn()        ▷ call optimisation step
      if done break
   end for

  log (Rₑ, ε, loss, lr);   every 25 eps: run greedy eval
 end for
```

```
function learn()                              ▷ Double-DQN update
   (S, A, R, S', D) ← sample minibatch from B
   q_sa          ← Q(S)[A]                    ▷ gather online Q
   with no-grad:
      a*← argmax _Q(S')                       ▷ online argmax
      q_tgt ← Q̂(S')[a*]                       ▷ target network
      y     ← R + γ·(1–D)·q_tgt
loss ← Huber(q_sa, y)
θ ← θ – α·∇θ loss                             ▷ Adam + clip ≤ 1
lr_scheduler.step()

# Polyak soft-update
   θ– ← (1–τ)·θ–  +  τ·θ
end function
```

Algorithm 2 Actor-Critic Credit-Limit Laddering

```
Input :  M – number of episodes
         T – horizon (24 steps)
         W – warm-up episodes (critic-only)

Initialise
         actor network  n(s ; θπ)  and critic network  V(s ; θV)
         Adam(lr = $3·10^{-4}$)
         running $\mu_0, \sigma_0^2$ ← PopArt-lite stats
Seed RNGs for Python / NumPy / PyTorch

for episode e = 1 … M do
      $\tau \leftarrow \max(0.25, 0.30·(1 - e / 300))$          ▷ exploration temperature
      freeze_actor = ($e \le W$)                          ▷ critic warm-up gate
      $s_0$ ← env.reset();  $\mathcal{D} \leftarrow \emptyset$;  $R_e \leftarrow 0$

      # Sampling phase (collect one full trajectory) ————————————————
      for t = 0 … T − 1 do
         if freeze_actor:
            a_t ← 0                                       ▷ NO-OP bucket
         else:
            logits ← n(s_t) / τ
            p ← softmax(logits)
            a_t ~ Categorical(p)

         s_{t+1}, profit_t, done, _ ← env.step(a_t)
         r_t ← sign(profit_t)·√|profit_t| / 100
         Append (s_t, a_t, r_t, s_{t+1}, done) → $\mathcal{D}$
         $R_e \leftarrow R_e$ + r_t
      end for

      # Training phase (one on-policy update, GAE-λ) ————————————————
     Extract S, A, R, S′, D from $\mathcal{D}$
      V  ← VθV(S);     V′ ← VθV(S′).detach()
      δ_t ← R_t + γ·V′·(1 − D) − V_t

      Compute G_t (returns) and A_t (advantages)
      loss_c ← SmoothL1(V, (G − μ)/$\sqrt{\sigma^2}$)
      θV ← AdamStep(θV, loss_c, clip=5);   update μ, $\sigma^2$

      if not freeze_actor:
         β ← $10^{-3}$·max(1, 5 − e/M)
         logits ← n(S)/τ;   p ← softmax(logits)
         loss_a ← − E[log p[A]·A_t] + β·H(p)
         θπ ← AdamStep(θπ, loss_a, clip=5)
         if loss_c < 0.02 and lr_actor > $1.5·10^{-4}$:
            lr_actor ← 0.5·lr_actor
      Cosine-anneal learning-rates
      Log metrics; every 25 eps run greedy eval
end for

Save actor & critic weights, CSV logs, NumPy rewards
```

References

[1] Singh, V., Chen, S.-S., Singhania, M., Nanavati, B., Gupta, A., et al. (2022). How are reinforcement learning and deep learning algorithms used for big data-based decision making in financial industries–A review and research agenda. International Journal of Information Management Data Insights, 2(2), 100094. https://doi.org/10.1016/j.jjimei.2022.100094

[2] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). "Proximal Policy Optimization Algorithms" arXiv:1707.06347

[3] Haarnoja, T., Zhou, A., Abbeel, P., & Levine, S. (2018). "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor" arXiv:1801.01290

[4] Alfonso-S´anchez, S., Solano, J., Correa-Bahnsen, A., Sendova, K. P., and Bravo, C. (2024). Optimizing credit limit adjustments under adversarial goals using reinforcement learning. European Journal of Operational Research 315(2): 802-817, DOI: https://doi.org/10.1016/j.ejor.2023.12.025.

[5] Sutton, R. S. & Barto, A. G. (2018). Reinforcement learning: An introduction. MIT press.

[6] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. (2017). Mastering the game of go without human knowledge. nature, 550(7676), 354–359. https://doi.org/10.1038/nature2427

[7] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. nature, 518(7540), 529–533. https://doi.org/10. 1038/nature14236

[8] Gronauer, S. & Diepold, K. (2022). Multi-agent deep reinforcement learning: a survey. Artificial Intelligence Review, 55, 895–943. https://doi.org/10.1007/s10462-021-09996-w

[9] Yang, C., Feng, Y., & Whinston, A. (2022). Dynamic pricing and information disclosure for fresh produce: An artificial intelligence approach. Production and Operations Management, 31(1), 155–171. https://doi.org/10.1111/poms.13525

[10] Herasymovych, M., Marka, K., & Lukason, O. (2019). Using reinforcement learning to optimize the acceptance threshold of a credit scoring model. Applied Soft Computing, 84, 105697. https://doi.org/10.1016/j.asoc.2019.105697

[11] Krasheninnikova, E., Garcìa, J., Maestre, R., & Fernàndez, F. (2019). Reinforcement learning for pricing strategy optimization in the insurance industry. Engineering applications of artificial intelligence, 80, 8–19. https://doi.org/10.1016/j.engappai.2019.01.010

[12] El Bouchti, A., Chakroun, A., Abbar, H., & Okar, C. (2017). Fraud detection in banking using deep reinforcement learning. 2017 Seventh International Conference on Innovative Computing Technology (INTECH), 58–63. https://doi.org/10.1109/INTECH.2017.8102446

[13] Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. Machine Learning, 3(1), 9–44. https://doi.org/10.1007/BF00115009

[14] Watkins, C. J. C. H., & Dayan, P. (1992). Q-learning. Machine Learning, 8(3–4), 279–292. https://doi.org/10.1007/BF00992698

[15] Schulman, J., Moritz, P., Levine, S., Jordan, M. I., & Abbeel, P. (2016). High-Dimensional Continuous Control Using Generalized Advantage Estimation. Proceedings of the 4th International Conference on Learning Representations (ICLR 2016). Original arXiv preprint: arXiv:1506.02438, June 2015.

[16] van Hasselt, H., Guez, A., Hessel, M., Mnih, V., & Silver, D. (2016). Learning Values Across Many Orders of Magnitude. In Advances in Neural Information Processing Systems 29 (NeurIPS 2016), 30th Conference on Neural Information Processing Systems, Barcelona, Spain. arXiv:1602.07714

[17] Bertsimas, D., & Dunn, J. (2017). Optimal classification trees. Machine Learning, 106(7), 1039-1082. Massachusetts Institute of Technology

[18] Elmachtoub, A. N., & Grigas, P. (2022). Smart “Predict, then Optimize”. Management Science, 68(6), 4224-4241. INFORMS Pubs Online

[19] Nazari, M., Oroojlooy, A., Snyder, L. V., & Takáč, M. (2018). Reinforcement learning for solving the vehicle-routing problem. In Advances in Neural Information Processing Systems 31.

[20] Hubbs, C. D., Pérez, H. D., Sarwar, O., Sahinidis, N. V., Grossmann, I. E., & Wassick, J. M. (2020). OR-Gym: A reinforcement-learning library for operations-research problems. arXiv:2008.06319.

[21] Shi, C., & Zhang, H. (2021). Deep reinforcement learning for dynamic pricing and inventory control. International Journal of Production Economics, 235, 108084.