ChatGPT

Retriever-Augmented Memory for Multi-Episode Reinforcement Learning in Cryptocurrency Trading Agents

Abstract

Cryptocurrency markets are highly volatile and nonstationary, posing challenges for reinforcement learning (RL) trading agents. In this paper, we propose a novel framework to enhance RL-based cryptocurrency trading agents using Retriever-Augmented Memory (ReMem). ReMem – originally introduced for large language model agents – allows an agent to retrieve and refine past experiences during decision-making, rather than relying solely on static model parameters or short-term memory. We conceptually describe the ReMem approach and detail how integrating an explicit memory retrieval module into a trading agent can improve performance. The proposed architecture augments a traditional RL agent with an external memory of prior trading episodes and a retrieval mechanism to recall relevant experiences in new market conditions. This memory-augmented design is expected to boost profitability by reusing successful strategies, stabilize trading policies through continual learning across episodes, and increase risk sensitivity by recalling past adverse outcomes. We outline the ReMem-enhanced agent's decision loop, compare it to conventional memory-based or transformer-based RL agents, and discuss implementation considerations. The result is a conceptual blueprint for **ReMem-Trader** – a reinforcement learning trading agent that continually learns from its own evolving experience to adapt in dynamic cryptocurrency markets.

Introduction

Cryptocurrency trading presents a demanding testbed for sequential decision-making algorithms due to extreme price volatility, regime shifts, and unstable risk–return dynamics [1]. Traditional model-based approaches often fail under such nonstationary conditions, motivating the use of deep reinforcement learning (DRL) for trading strategy development [2] [3]. DRL agents can in principle learn adaptive policies directly from market experience, and indeed recent work has shown RL-based strategies outperforming static benchmarks in risk-adjusted returns [4] [5]. However, standard RL agents face limitations in how they utilize past experiences. After training, an agent's historical experiences are distilled into network weights and no longer directly accessible, which means valuable situational knowledge may be lost due to finite model capacity [6]. This is problematic in volatile markets: an agent might *learn* a profitable strategy in one market regime, only to forget or misapply it when conditions change and later recur.

Conventional solutions to maintain memory in RL include recurrent neural networks (e.g. LSTM-based policies) to capture temporal dependencies [4], or training with experience replay to reinforce past insights. While effective to a degree, these approaches treat memory implicitly. The agent cannot actively query specific past events when deciding on actions – it is limited by what its network weights have encoded and by a fixed-size hidden state. Transformer-based agents have also been explored, feeding entire past trajectories into the model context to achieve in-context learning of new tasks [7]. Yet this approach becomes infeasible for long trading horizons with high-frequency data, as lengthy episodes would exhaust the context window and computational budget [7] [8]. There is a need for a mechanism that enables *targeted reuse* of prior experiences without incurring huge model complexity.

In this paper, we propose to equip cryptocurrency trading agents with **Retriever-Augmented Memory (ReMem)**, a framework that explicitly stores and retrieves past experiences to inform current decision-making. Originally formulated for multi-step reasoning in language agents [9], the ReMem approach introduces a dedicated memory module and retrieval process within the agent's decision loop. Instead of treating the memory as a static buffer of recent history, the agent can actively **recall, evaluate, and update** its memory during trading. We hypothesize that such a memory-augmented RL agent – which we term **ReMem-Trader** – will exhibit improved performance through continual adaptation:

- **Higher Profitability:** by recalling successful strategies from similar past market conditions and reapplying them, the agent can achieve greater cumulative returns.
- **Policy Stability:** by retaining learned behaviors across episodes, the agent's policy updates become less volatile, reducing the risk of catastrophic forgetting and unstable swings in strategy.
- **Risk Sensitivity:** by remembering prior episodes with large losses or drawdowns, the agent can recognize warning patterns and adjust its actions to avoid or mitigate risk in analogous situations.

To realize these benefits, we present a conceptual architecture integrating ReMem into an RL trading agent. We describe how the agent's state, policy network, and learning process are modified to incorporate an external memory and a retriever for multi-episode experience reuse. A comparison with traditional memory-based RL (e.g. recurrent or transformer agents) is provided to highlight the novelty of our approach. Finally, we discuss implementation considerations such as memory indexing, retrieval algorithms, and maintaining performance in live trading. The remainder of this paper is organized as follows: **Section 2** reviews background on memory in reinforcement learning and the ReMem framework. **Section 3** details the proposed ReMem-Trader architecture and its decision-making process. **Section 4** offers a conceptual comparison to existing methods and examines practical implementation aspects. **Section 5** concludes with a summary and outlook for future work.

Background

Memory in Reinforcement Learning Agents

In reinforcement learning, agents traditionally improve by compressing experience into model parameters via gradient updates. After training, however, specific episodic details are not directly accessible – the agent's **memory** of past situations is implicit. This paradigm has clear drawbacks: once learning is done, past experiences no longer play a direct role in decision-making, and new experiences take many updates to influence behavior [6]. Researchers have long sought ways to endow RL agents with more explicit memory mechanisms. Recurrent neural networks (RNNs) and Long Short-Term Memory (LSTM) networks are common extensions to RL policies that provide a form of internal state to remember recent observations. For instance, in a crypto trading context, an LSTM-enhanced agent can maintain information on recent price trends to inform its next action [4]. Such memory-augmented policies have demonstrated improved adaptability in volatile markets [4]. Nonetheless, RNN-based memory has limitations: the memory capacity is bounded by hidden state size, and long-term dependencies may still fade over time.

Another line of work uses **experience replay buffers** during training to ensure the agent repeatedly learns from past data. While this improves sample efficiency and stability, it still relies on the weight updates to indirectly retain knowledge. Once the training/update is done at a given time, the agent cannot dynamically refer back to a particular past experience when making a decision; the influence of that experience is baked (to an unknown degree) into the network weights [6]. This lack of direct, situational recall means the agent may fail to exploit potentially relevant past cases, especially in multi-episode settings where conditions recur after long gaps.

Recently, researchers have begun exploring **retrieval-based memory** in RL to address these issues. In retrieval-augmented RL, an agent is supplemented with an external memory store (a database of past experiences or trajectories) and a learned retrieval mechanism. Goyal *et al.* (2022) train an RL agent to query a database of past experiences and retrieve information useful for the current state [10]. By conditioning its policy on the content of retrieved experiences, the agent can utilize knowledge from an entire dataset of past trajectories, not just the current episode, leading to faster learning and higher performance [11]. This approach is akin to **case-based reasoning**, where the agent recalls a similar past situation and applies the lessons from that case to the current decision [12]. The success of retrieval-augmented RL in tasks like Atari games and multi-task problems suggests that explicitly leveraging episodic memory can significantly enhance learning efficiency and effectiveness.

Transformer-Based Decision Agents

The surge of interest in transformers and in-context learning has also influenced RL agent design. Approaches such as the **Decision Transformer** treat RL as a sequence modeling problem, feeding past states, actions, and returns into a transformer model to predict the next action (essentially leveraging the transformer's capacity to encode long trajectories) [13]. These transformer-based agents can exhibit *in-context adaptation*, learning new tasks from a few on-the-fly examples similarly to large language models. However, a major challenge arises: complex trading environments or long horizons produce very long episodes, which are impractical to fit entirely into the model's context window [7]. Keeping thousands of time-steps of price data and trades in context would be computationally expensive and may exceed memory limits, especially as transformer complexity grows quadratically with sequence length. Furthermore, not all details from a long history are relevant for the current decision – some may be noisy or irrelevant.

To mitigate this, researchers have proposed combining transformers with retrieval. **Retrieval-Augmented Decision Transformer (RA-DT)** is one such example, where an external memory is used to store past trajectories and only the most relevant fragments (sub-trajectories) are retrieved and fed into the transformer's context [8] [14]. By using a vector similarity search, RA-DT finds past experience snippets similar to the current situation and incorporates them via cross-attention, thereby reducing the effective context length and focusing the model on pertinent experiences [14]. This approach is analogous to retrieval-augmented generation in NLP, but applied to decision making in RL. It addresses the scalability issue: rather than relying on an unwieldy fixed-size context, the agent can tap into an *external* memory on demand. Still, RA-DT and similar methods mainly treat retrieval as a one-shot context augmentation; the agent itself does not *modify* its memory content, and the control flow of decision-making remains unchanged (the retrieval happens automatically per time-step).

Retriever-Augmented Memory (ReMem) Framework

The **ReMem** framework takes memory augmentation a step further by deeply integrating it into the agent's decision loop. Proposed in the context of LLM-based agents, ReMem stands for a *"reasoning–acting–memory-*

refining" loop [9]. It introduces memory as an *active* component of decision-making rather than passive storage. Concretely, at each decision step the agent can choose to: **Think**, **Act**, or **Refine Memory** [9]. The **Think** operation produces internal reasoning (e.g. intermediate calculations or plan), **Act** triggers an external action (such as executing a trade or outputting a decision), and **Refine** allows the agent to retrieve information from its memory, prune irrelevant entries, or reorganize stored knowledge [15]. This triad of operations is executed in a loop: the agent may perform several Think or Refine steps (e.g. retrieving multiple past experiences and deliberating on them) before finally choosing an Act to conclude the time-step [16].

By formalizing memory operations as part of the agent's action space, ReMem essentially defines a **Memory-augmented Markov Decision Process**. In this MDP extension, the state includes not only the environment observation but also the agent's current memory state and any ongoing chain-of-thought [16]. The agent's policy must decide not just *what external action to take*, but also *when and how to query or update memory*. Notably, this makes the memory a first-class entity: *"in contrast to standard [agents], memory is no longer a fixed buffer. It becomes an explicit object that the agent reasons about and edits during inference."* [16]. In other words, the agent is aware of its memory content and can intentionally manage it, rather than simply consuming whatever context is given. Such a framework enables **continual adaptation** – the agent can reflect on what it has learned so far, refine its knowledge base, and then proceed, all during the live decision process.

The original ReMem was demonstrated with large language models, but the core idea is general: an intelligent agent equipped with a retriever-augmented memory can improve over time *within its deployment*, not just during training. It can integrate new experiences and refine its strategy on the fly. This capability is particularly attractive for multi-episode decision-making problems, where an agent faces a sequence of related tasks or environments. Cryptocurrency trading naturally fits this description: each trading session or market regime can be seen as an episode from which lessons should be carried over. By leveraging ReMem, a trading agent could retrieve experiences from previous episodes (e.g. past bull or bear markets, previous crashes or rallies) and apply those insights to current decisions. The next section proposes an architecture to realize this vision.

Proposed Method: ReMem-Enhanced Trading Agent

We propose **ReMem-Trader**, a reinforcement learning agent for cryptocurrency trading augmented with the Retriever-Augmented Memory framework. The design extends a standard RL trading agent by incorporating an explicit memory module and integrating memory operations into the agent's decision-making process. Figure 1 (conceptual) outlines the architecture: the agent consists of four main components analogous to those identified in Evo-Memory [17] – a base policy model (F), a retrieval module (R), a context constructor (C), and a memory update mechanism (U). We describe each component and how they interact in the trading context:

- **Base Model (F)** – This is the underlying RL agent (e.g. a policy network or Q-network) that maps the current state and retrieved memory context to an action decision. It can be implemented using any suitable DRL algorithm (DQN, DDPG, PPO, etc.) along with function approximation (neural network). In our design, the base model receives an *augmented state* as input: not only the usual market features (prices, indicators, positions, etc.), but also information returned from the memory retrieval. The base model's output is a trading action, such as buy, sell, hold, or portfolio allocation weights for assets, which is then executed in the market environment.

- **Memory Store** – An external database that holds the agent's past experiences across episodes. Each **memory entry** in this store could represent a salient segment of a trading episode. For example, an entry might encode a particular market context (a pattern of price movements, volatility regime, etc.), the action the agent took, and the outcome (profit/loss or reward received, possibly with risk metrics like drawdown). Entries could be stored at different granularities (full episodes, or key event snippets within episodes). The memory is dynamic: after every trading episode (or even after every time-step or significant event), new experiences are added via the update module, and outdated or less useful experiences might be pruned or summarized during memory refinement.

- **Retrieval Module (R)** – A mechanism to fetch relevant entries from the memory store given the current situation. This module can be implemented as a **similarity-based retriever**, for instance using a vector embedding model: the current state (or recent sequence of states) is encoded into a query vector, which is then compared against embeddings of stored memory entries to find the closest matches [14]. Domain-specific design is crucial here – for trading, the retriever could use price trend features or latent representations of market conditions to identify past periods that "look like" the present. The retrieved results (e.g. top-k similar experiences) are then passed to the context constructor.

- **Context Constructor (C)** – This component synthesizes the information from retrieval into a form that the base model can consume. In NLP-based ReMem, this would involve constructing a prompt or context window [17]; in our RL setting, it could involve concatenating features or computing summary statistics from retrieved experiences. For example, if the memory returns two past episodes where a rapid price drop occurred with certain indicators preceding it, the context constructor might distill common features of those episodes (e.g. "similar volatility spike and order-book imbalance observed") and feed this as additional input features. Alternatively, we could employ an attention mechanism in the base model: the current state and retrieved memory entries are combined through an attention layer (akin to how RA-DT fuses sub-trajectories [14]). This allows the model to focus on the most pertinent parts of retrieved experiences when computing the action decision.

- **Memory Update (U)** – After each decision (or episode), the agent uses this function to update its memory store. This involves writing a new experience entry composed of the current state, action, outcome, and any useful metadata (reward achieved, success/failure, risk metrics). Importantly, the update may also **evolve** the memory: if the memory is at capacity or if some entries are deemed obsolete, the update function can remove or compress them [17]. In a long-running trading deployment, the agent might accumulate an enormous amount of data, so U could implement policies for memory management (for instance, keeping the memory to the most recent N episodes, or using a sliding window of past months, or prioritizing experiences with high information value).

Decision Loop with Memory Integration: The ReMem-Trader operates in discrete time-steps (e.g. each time-step could be a trading interval like one hour, or each episode could be one day of trading). At each decision point, the following loop is executed:

1. **Observe Current State:** The agent observes the current market state (prices, technical indicators, inventory holdings, etc.). Let us denote this observation as s_t.

2. **Retrieve Relevant Experiences:** The retrieval module encodes s_t (and possibly recent history h_t) into a query and finds the top relevant past experiences from memory. Suppose it retrieves a set ${e_{t1}, e_{t2}, \dots, e_{tk}}$ of experience entries that occurred under market conditions similar to s_t. Each retrieved entry might include context like "when indicator X and Y had values (x, y) and volatility was high, the agent's action was Sell, which resulted in a large loss" or conversely a profitable outcome. The retrieval is essentially bringing potentially useful *lessons* to the agent's fingertips.

3. **Memory Reasoning (Refine):** Before committing to an external action, the agent can internally reason about the retrieved information. This corresponds to the **Think** and **Refine** operations from ReMem. In practice, this might mean the agent's policy network processes the current state *together with* the memory context and evaluates possible outcomes. The agent could also decide to ignore or discount certain retrieved items if they seem irrelevant (this could be learned, or implemented via a gating mechanism in the network). A key part of refinement is that the agent may update the *importance weighting* of retrieved memories: for instance, if two past experiences were retrieved but one closely matches the current context while the other is a looser match, the agent can weight them accordingly in its decision process (or even choose to do an additional query/refinement round for more similar cases).

4. **Act:** Finally, the agent selects and executes a trading action a_t (e.g. buy 10 BTC, or shift portfolio allocation) based on the enhanced decision input. This action is output by the base model which has now considered both the raw state s_t and the retrieved memory context. The trade is carried out in the environment (market simulator or live market), and the agent observes the immediate reward r_t (which could be profit or loss from that action, etc.).

5. **Memory Update:** The new experience $(s_t, a_t, r_t, \text{any outcome info})$ is encoded into an entry and added to the memory store via the update function U. If this step concludes an episode (for example, end of day or end of a trading session), the summary of that episode is stored. The update function may also prune older experiences if memory size is limited or compress similar experiences to maintain a tractable memory.

This loop repeats at each time-step. Notably, steps 2 and 3 (Retrieve and Refine) correspond to the **memory reasoning** aspect unique to ReMem. If the agent determines that memory is not needed for a trivial decision, it could bypass heavy retrieval and just act (this could be an optimization learned over time, effectively learning when to trust its own policy vs. when to consult memory). In critical or novel situations, the agent might even retrieve multiple rounds or do extra internal "thinking" before acting – analogous to how a human might pause trading to recall historical episodes during a sudden market crash.

Example Scenario: To illustrate, imagine the agent is trading Bitcoin and suddenly the price drops 5% in an hour on high volume. The current state s_t reflects a sharp downturn and fear in the market. The agent's retrieval module might pull from memory two past episodes: one from 6 months ago when a similar drop occurred due to regulatory news, and another from 2 years ago when a flash crash happened. In the first retrieved case, the agent's strategy to hold (no sell) proved wise as the market rebounded, yielding a positive reward; in the second case, failing to sell led to a large loss as the crash deepened. The context constructor feeds these scenarios into the policy network. The ReMem-Trader then *reasons* – perhaps via attention – that the current drop more closely resembles the flash crash in terms of order book dynamics, thus it should be cautious. It decides to reduce the position (an **Act** of selling some assets) to mitigate risk.

Afterward, the outcome (say the crash continued and the action saved loss) is recorded. The agent might also **Refine** memory by marking the flash-crash example and the current experience as connected, or by pruning less relevant memories to focus on important patterns like “crash scenarios.”

Expected Improvements with ReMem Integration

By integrating this ReMem framework into a trading agent, we anticipate several key improvements over conventional RL agents:

- **Improved Profitability:** The agent can **reuse successful strategies** from its memory. When a profitable pattern or arbitrage opportunity recurs, the agent recalls the prior instance and executes the proven action more quickly and confidently. This reduces the learning curve for each new pattern and allows the agent to capitalize on opportunities that it might otherwise miss or learn slowly. Empirical evidence from retrieval-augmented RL shows faster learning and higher scores on complex tasks [11], which in trading terms translates to better returns over time.

- **Stabilized Trading Policy:** With memory, the agent’s policy evolves more **smoothly** across episodes. Instead of relearning from scratch in each new market context, the agent builds on a growing repository of experiences. This continuity makes the trading behavior more consistent and robust. Notably, experience reuse can prevent dramatic policy shifts because the agent is reminded of past lessons that restrain it from taking ill-considered actions. This aligns with observations that even simple experience replay across tasks yields more efficient behavior [18] [19]. Overall, ReMem-Trader is less prone to catastrophic forgetting; it retains useful behaviors (e.g. risk management tactics learned in a crash) and carries them forward, leading to a steadier performance profile.

- **Enhanced Risk Sensitivity:** The memory module can store not just rewards but also **risk outcomes** (e.g. maximum drawdown in an episode, volatility experienced). By retrieving experiences with high losses or high volatility, the agent effectively gains a form of *risk memory*. When current conditions resemble those dangerous scenarios, the agent is alerted to potential risk. It can then act more cautiously (for instance, reducing leverage or avoiding trades) to prevent repeating past mistakes. In essence, the agent internalizes a risk management policy shaped by historical extremes. This should lead to improved risk-adjusted performance – for example, higher Sortino or Sharpe ratios – as the agent avoids large drawdowns. Prior studies have found RL agents with appropriate memory or regularization achieve better **risk-adjusted returns and portfolio stability** than those without [20] [5], and our approach aims to amplify that effect by explicitly learning from past risky episodes.

In summary, the ReMem-enhanced agent continuously learns not only *from* the environment but also *about its own learning*, leveraging past episodes to inform future decisions. This meta-level adaptation is expected to yield a trading agent that is more profitable, reliable, and prudent in the face of the cryptocurrency market’s uncertainties.

Discussion

Comparison with Traditional Approaches

Our proposed ReMem-Trader differs fundamentally from traditional memory mechanisms in RL. In a standard LSTM-based trading agent, the memory is limited to a fixed-length hidden state that gets updated each step; the agent cannot refer back to a specific past event beyond what the RNN dynamics preserve. In contrast, ReMem-Trader has access to an **explicit, addressable memory** of potentially unbounded size. This avoids the capacity bottleneck of RNN memory – detailed information from arbitrarily long ago can be retrieved if relevant, instead of being forgotten or overwritten [6]. Similarly, compared to **transformer-based agents** that pack recent trajectories into the context, ReMem allows the agent to **focus on a few key past episodes** rather than drag along an entire history. This targeted retrieval is computationally more efficient and scalable to long horizons [7]. It also means the agent's decision is informed by *selected experiences* rather than a raw timeline, potentially reducing noise and distraction from irrelevant past data.

Another important distinction is how memory is treated in the decision process. In conventional RL (and even in transformer context methods), memory is passive – it influences the agent only through the data it provides (or the context it forms) but the agent does not *change* the memory content during decision-making. By contrast, ReMem introduces **memory as a decision variable**. The agent can choose to retrieve or ignore, and even modify its memory (e.g. remove an experience that it deems misleading after some evaluation). This leads to a richer form of agency. The trade-off is increased complexity: the agent's policy space is larger (deciding when to Think, Act, or Refine). However, this self-reflective capability is precisely what can yield adaptive advantages in nonstationary settings. The agent effectively performs a form of continual learning at test time, something standard agents do not do. Notably, ReMem achieves this adaptation **without altering the base model weights** – improvements happen through memory updates and better use of past knowledge [21]. This is appealing for deployment, since it means a fixed model can improve over time, analogous to human traders who get better with experience without fundamentally changing their decision-making circuitry.

In summary, compared to traditional memory-based RL agents, ReMem-Trader provides **greater memory capacity, targeted recall, and active memory management**. Compared to transformer-based in-context learners, it offers **efficient scaling and dynamic memory use**. The memory becomes a source of strength rather than a limitation: instead of being constrained by what fits in a context window or hidden state, the agent can reach out to a vast repository of past knowledge. The design follows the recent trends in AI that emphasize *retrieval-augmented reasoning*, bridging the gap between purely learned models and explicit knowledge bases [22] [19].

Implementation Considerations

While conceptually promising, implementing the ReMem-enhanced trading agent raises several practical considerations:

Memory Representation and Indexing: Choosing the right representation for experiences is crucial. Each trading experience (episode or snippet) should be encoded in a way that similarity search is meaningful. This could involve training an embedding model on historical market data so that episodes with similar market dynamics end up close in embedding space. Unsupervised approaches (e.g. autoencoders or contrastive learning on time-series) might be used to derive such embeddings. The system might employ an approximate nearest neighbor index (like FAISS or HNSW) to enable fast retrieval of relevant experiences even as the memory grows.

Retriever Training: The retrieval module itself can be learned. During initial training of the agent, one could jointly optimize the retriever to fetch experiences that lead to good decisions. Alternatively, a simpler heuristic retriever (based on predefined features or distance metrics) could be used initially, and then refined with data. In an online trading setting, feedback signals (like whether the retrieved memory led to a correct decision) could be used to adjust the retrieval mechanism over time. This is analogous to tuning a search engine to return more useful results based on user clicks.

Memory Update & Pruning: Without limits, the memory will grow indefinitely as the agent continues trading. Some strategy for **memory management** must be in place. One approach is to retain a sliding window of recent experiences (ensuring the memory focuses on the current regime), possibly supplemented by a diverse subset of older experiences (to not forget rare but important events like crashes). Another approach is to periodically compress memory: for example, cluster similar experiences and merge them into a prototype experience. The ReMem framework's meta-memory operations (Refine) could be leveraged here – the agent itself could decide to remove certain entries that it finds unhelpful. Designing a robust criterion for pruning (e.g. low retrieval frequency, or outdated due to regime shift) will be important to prevent memory from cluttering with useless data.

Integration with RL Training: We propose ReMem primarily as a decision-time aid, but it can also influence the training phase. During training simulations, one could allow the agent to use retrieval to accelerate learning (similar to how human traders back-test or recall historical scenarios while training). Techniques from multi-task or meta-RL could be applicable, where the memory acts as a context for the policy. However, incorporating memory actions (Think/Refine) into training will complicate the RL algorithm, as the action space now includes cognitive operations. A possible simplification is to train the base policy in a normal way (e.g. using an off-policy algorithm with experience replay) and separately train the retriever/context constructor to feed helpful info to the policy. Then, at runtime, allow a limited form of the Think-Refine loop by retrieving once or a fixed number of times before action. This would avoid a blow-up of the action space while still providing the memory benefit.

Latency and Real-Time Constraints: In live trading, decisions often need to be made under tight time constraints (especially in high-frequency trading). Introducing retrieval and memory reasoning steps could add latency. It's essential to optimize the memory lookup (via efficient indexing) and possibly do some computations in parallel (e.g. encode state and query the memory asynchronously while the base model also does preliminary computations). If the agent decides to do multiple Think/Refine steps, one must ensure this loop terminates quickly enough. Setting a maximum number of retrieval iterations or a time-budget for decision can prevent the agent from over-thinking and missing the opportunity to act on time.

Risk Management Module: To explicitly cater to risk sensitivity, the implementation could tag experiences with risk outcomes (like "this episode had a max drawdown of X%"). The retrieval module could then be conditioned not just on similarity of state, but also triggered by risk-related signals. For instance, if the agent's current portfolio drawdown exceeds a threshold, it might explicitly retrieve experiences where past drawdowns occurred, even if the immediate state similarity is moderate. This ensures that in high-risk moments, the agent reviews what went wrong in the past, aligning with human risk management practices.

Evaluation Protocol: Demonstrating the effectiveness of ReMem-Trader would require careful evaluation. Traditional single-episode return metrics might not capture the value of memory. Instead, one would measure improvements over multiple episodes: e.g. does the agent adapt faster in a new market regime because of memory? We would compare against baselines like an LSTM-based agent and a transformer agent on scenarios with repeated patterns. Metrics could include cumulative profit, Sharpe ratio, and a measure of adaptation speed (how quickly performance improves when a previously seen scenario reappears). While full empirical analysis is beyond our scope here, defining such metrics is important for future work.

In conclusion, implementing ReMem in a trading agent will involve addressing these practical issues, but none appear insurmountable given modern techniques. Many components (fast similarity search, representation learning, off-policy RL) are well-studied. The novelty lies in their orchestration to create a self-improving, memory-rich trading agent.

Conclusion

We have presented a conceptual framework for enhancing cryptocurrency trading agents through **Retriever-Augmented Memory (ReMem)**, bringing ideas from advanced LLM-based agents into the realm of reinforcement learning for finance. The proposed ReMem-Trader architecture enables an RL agent to retrieve and reason with past experiences during decision-making, moving beyond the limitations of fixed-size memories and static policies. By treating memory as an interactive component – one that the agent can query and update in an ongoing manner – our approach aims to improve profitability, stability, and risk management in trading. The agent effectively learns *within* its deployment, continually refining its knowledge of market patterns and lessons from prior episodes.

We conceptually compared ReMem-Trader to traditional memory-based and transformer-based agents, highlighting that our approach offers targeted recall of relevant episodes and adaptive memory management that prior methods lack. These qualities make it well-suited for nonstationary environments like crypto markets, where an agent must recall distant events (e.g. bubbles, crashes) and adapt to evolving regimes. While our discussion has focused on the qualitative benefits, an important next step is to implement and empirically validate the ReMem-enhanced agent. Key challenges such as efficient memory indexing, determining similarity metrics for market states, and ensuring real-time operation will need to be addressed. We expect that insights from retrieval-augmented NLP and existing episodic control RL can guide these implementation aspects.

In a broader context, this work contributes to the vision of **agents that learn to learn from their own experience**. By endowing a trading agent with the capacity to remember and reason about the past, we move closer to AI systems that exhibit continual learning and self-improvement akin to human traders. Future research may explore extensions such as incorporating expert demonstrations into the memory (so the agent can retrieve not only its own experiences but also those of skilled traders), or applying the framework to other domains requiring long-term adaptation. We hope this proposal stimulates further investigation into retriever-augmented decision making in reinforcement learning, and ultimately, the development of more resilient and profitable autonomous trading agents for the financial markets.

---

[1] [2] [3] [4] [5] [20] Reinforcement Learning-Based Cryptocurrency Portfolio Management Using Soft Actor–Critic and Deep Deterministic Policy Gradient Algorithms
https://arxiv.org/html/2511.20678v1

[6] [10] [11] [12] proceedings.mlr.press
https://proceedings.mlr.press/v162/goyal22a/goyal22a.pdf

[7] [8] [13] [14] Retrieval-Augmented Decision Transformer: External Memory for In-context RL
https://arxiv.org/html/2410.07071v1

[9] [15] [16] [17] [18] [19] [21] [22] Google DeepMind Researchers Introduce Evo-Memory Benchmark and ReMem Framework for Experience Reuse in LLM Agents - MarkTechPost
https://www.marktechpost.com/2025/12/02/google-deepmind-researchers-introduce-evo-memory-benchmark-and-remem-framework-for-experience-reuse-in-llm-agents/