QLLM: Do We Really Need a Mixing Network for Credit Assignment in Multi-Agent Reinforcement Learning?

Zhouyang Jiang[1], **Bin Zhang**[2], **Yuanjun Li**[1], **Zhiwei Xu**[1*]
[1]Shandong University
[2]The Key Laboratory of Cognition and Decision Intelligence for Complex Systems,
Institute of Automation, Chinese Academy of Sciences

Abstract

Credit assignment has remained a fundamental challenge in multi-agent reinforcement learning (MARL). Previous studies have primarily addressed this issue through value decomposition methods under the centralized training with decentralized execution paradigm, where neural networks are utilized to approximate the nonlinear relationship between individual Q-values and the global Q-value. Although these approaches have achieved considerable success in various benchmark tasks, they still suffer from several limitations, including imprecise attribution of contributions, limited interpretability, and poor scalability in high-dimensional state spaces. To address these challenges, we propose a novel algorithm, **QLLM**, which facilitates the automatic construction of credit assignment functions using large language models (LLMs). Specifically, the concept of **TFCAF** is introduced, wherein the credit allocation process is represented as a direct and expressive nonlinear functional formulation. A custom-designed *coder-evaluator* framework is further employed to guide the generation and verification of executable code by LLMs, significantly mitigating issues such as hallucination and shallow reasoning during inference. Furthermore, an **IGM-Gating Mechanism** enables QLLM to flexibly enforce or relax the monotonicity constraint depending on task demands, covering both IGM-compliant and non-monotonic scenarios. Extensive experiments conducted on several standard MARL benchmarks demonstrate that the proposed method consistently outperforms existing state-of-the-art baselines. Moreover, QLLM exhibits strong generalization capability and maintains compatibility with a wide range of MARL algorithms that utilize mixing networks, positioning it as a promising and versatile solution for complex multi-agent scenarios. The code is available at https://github.com/zhouyangjiang71-sys/QLLM.

Introduction

Recent advances in cooperative multi-agent reinforcement learning (MARL)(Oroojlooy and Hajinezhad 2023) have led to widespread applications in domains such as assembly lines (Kaven et al. 2024), logistics distribution (Yan et al. 2022), autonomous driving (Zhang et al. 2024b), and swarm robotics (Orr and Dutta 2023). In such scenarios, multiple agents learn to interact with the environment collaboratively to accomplish designated cooperative tasks. Given that agents receive a shared team reward, it is essential to accurately assess the individual contribution of each agent to facilitate effective training. A key challenge in this context is the credit assignment problem, where inaccurate attribution of individual contributions may result in the emergence of so-called ”lazy agents” (Liu et al. 2023).

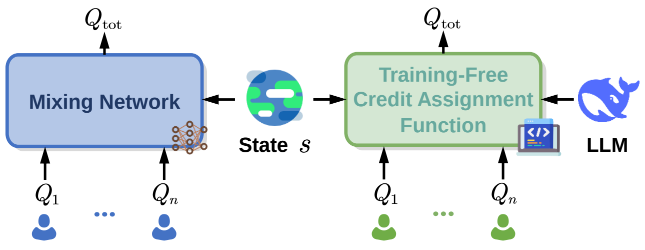

Figure 1: A comparison between the traditional value decomposition method using a mixing network (**Left**) and our proposed novel paradigm QLLM leveraging LLMs (**Right**). The traditional approach employs a neural network to model the nonlinear relationship between local Q-values and the global Q-value, whereas QLLM capitalizes on the extensive knowledge encoded within LLMs to directly generate a training-free credit assignment function.

To address the credit assignment problem in multi-agent systems, a variety of solutions have been proposed in the academic community. Attention-based credit assignment methods (Soydaner 2022) incorporate a learnable mechanism that adaptively allocates credit by computing attention weights over agents or their interactions. By explicitly modeling inter-agent dependencies, these methods enable the network to selectively attend to the most influential agents or environmental factors when estimating individual contributions. This dynamic weighting scheme enhances the expressiveness of the credit assignment process, particularly in heterogeneous or partially observable environments.

Policy gradient approaches (Agarwal et al. 2021) tackle credit assignment from a differentiable optimization perspective. Each agent’s policy is updated based on gradients computed from shared rewards, typically combined with baselines or variance reduction techniques to improve learning stability. These methods can naturally handle continuous action spaces and stochastic policies, where credit assignment is implicitly achieved by estimating each

*Corresponding Author. Email: zhiwei_xu@sdu.edu.cn

agent's impact on the long-term return, either through joint trajectory sampling or individualized advantage estimation. In addition, value decomposition methods (Sunehag et al. 2017) have become increasingly popular. These methods address the credit assignment problem by constructing a joint value function that can be decomposed into individual utility functions associated with each agent. The decomposition is designed to ensure that the optimal joint policy can be recovered from the individual components, allowing for centralized training while preserving decentralized execution. These methods rely on specific structural assumptions that maintain the consistency between the global value and its local components, thereby enabling efficient coordination and scalable training.

However, most existing approaches predominantly rely on neural networks, either explicitly or implicitly, to perform credit assignment for each agents. A representative example of this is the *mixing network* employed in value decomposition methods. These approaches often suffer from high training costs, limited generalization ability, and a lack of sematic interpretability (Linardatos, Papastefanopoulos, and Kotsiantis 2020). With the advancement of large language models (LLMs) (Kasneci et al. 2023), new paradigms have emerged to address many long-standing challenges in MARL. By leveraging extensive pre-trained knowledge (Petroni et al. 2019), superior generalization capacity (Anil et al. 2022), and powerful code generation abilities (Xu et al. 2022a), LLMs have demonstrated potential as versatile auxiliary components in reinforcement learning pipelines. Recent studies (Qu et al. 2025) have shown the effectiveness of LLMs in generating dense, agent-specific rewards based on natural language descriptions of tasks and overall team goals. Empirical results reveal that such LLM-driven reward shaping leads to faster convergence and improved policy performance over existing baselines. Furthermore, the generalization and multitask learning capabilities of LLMs allow agents to quickly adapt to novel tasks with limited training samples, thus improving sample efficiency (Sanh et al. 2021). In tasks involving human interaction, LLMs can generate responses that more closely align with human expectations, enhancing interaction quality (Zhang et al. 2023b). Given the extensive applications of LLMs in the field of reinforcement learning, we propose to leverage them to address the credit assignment problem in multi-agent systems.

In this paper, we introduce **QLLM**, a novel value decomposition method that eliminates the need for a mixing network. To support its construction, we first propose a general *coder-evaluator* framework for LLM-based code generation. Within this framework, LLMs act as both a code generator and a code evaluator (Desmond et al. 2024). Upon receiving task and code generation instructions, the code generator produces a non-linear function called **T**raining-**F**ree **C**redit **A**ssignment **F**unction (**TFCAF**), which serves as a direct substitute for the mixing network in algorithms such as QMIX, VMIX, and RIIT (Rashid et al. 2020b; Su, Adams, and Beling 2021; Hu et al. 2021). Generated directly by the LLMs, this credit assignment function offers several advantages, including no need for training, immunity to high-dimensional state spaces, and enhanced semantic interpretability. Additionally, a code verification mechanism has been designed to ensure that the generated code runs correctly while mitigating hallucination issues (Martino, Iannelli, and Truong 2023) commonly encountered by LLMs. This framework significantly reduces the probability of errors when LLMs process complex textual prompts and enables the generation of a reliable **TFCAF** without direct interaction with the environment, thereby allowing it to replace the mixing network. Since LLMs modify only the mixing network in MARL algorithms, the TFCAF is applicable to almost any MARL algorithm that relies on mixing networks. An **IGM-Gating Mechanism** is further introduced to flexibly switch the monotonicity constraint on or off, adapting to tasks where the Individual-Global-Max principle holds or does not. The main contributions of this study are summarized as follows:

1. We propose the *coder-evaluator* framework for LLM-based code generation, which enables LLMs to generate a reliable credit assignment function under zero-shot prompting and without any human feedback, referred to as **TFCAF**. This framework is generalizable and can be extended to other MARL scenarios enhanced by LLMs.
2. **QLLM** is proposed by combining TFCAF and value function decomposition while remaining compatible with most MARL algorithms that employ mixing networks for credit assignment. Unlike existing methods, QLLM can accurately infer each agent's contribution without interacting with the environment.
3. Extensive experiments demonstrate that QLLM outperforms existing credit assignment algorithms, particularly in scenarios involving high-dimensional state spaces. Moreover, the integration of LLMs endows the contribution assignment among agents with semantic interpretability.

Related Work

Credit Assignment in MARL

In recent years, the credit assignment problem has emerged as a fundamental challenge in multi-agent systems. And policy-based approaches (Yu et al. 2022; Zhang et al. 2024a) were among the first to introduce credit assignment mechanisms. MADDPG (Lowe et al. 2017) extends the DDPG (Lillicrap et al. 2015) framework to multi-agent settings by incorporating centralized training with decentralized execution. This approach allows each agent to learn policies that account for the actions of other agents during training, thereby addressing the non-stationarity inherent in multi-agent environments. MADDPG utilizes an actor-critic architecture with centralized critics, where the centralized critic has access to global information, enabling more effective learning in mixed cooperative-competitive scenarios. Similarly, COMA (Foerster et al. 2018) leverages a counterfactual baseline to compute each agent's advantage function, following the same actor-critic framework as MADDPG. MAAC (Iqbal and Sha 2019) further enhances policy-based methods by introducing an attention mechanism into the critic network, leading to more precise credit assignment. However, most of these algorithms are not directly applicable to tasks with shared rewards.

Another widely adopted paradigm is the value decomposition approach (Rashid et al. 2020b; Xu et al. 2022b), which explicitly decomposes the globally shared return. As one of the earliest methods in this category, VDN (Sunehag et al. 2017) models the global value function as a linear summation of individual value functions. While simple and computationally efficient, VDN struggles to capture complex cooperative relationships due to its linear decomposition. QMIX (Rashid et al. 2020b) improves upon VDN by using a monotonic mixing network to non-linearly combine individual value functions, allowing for more expressive representations. However, the assumption of monotonicity between individual and global values may restrict its flexibility in highly complex environments. QTRAN (Son et al. 2019) and QPLEX (Wang et al. 2020) mitigate this limitation by relaxing the monotonicity constraint, enabling the model to learn richer representations, albeit at the cost of increased computational complexity. LICA (Zhou et al. 2020) adopts an alternative approach by introducing an independent credit assignment network that directly derives individual contributions from global information. Unlike methods focused solely on value decomposition, LICA explicitly models individual contributions, leading to more accurate credit assignment in cooperative tasks. RIIT (Hu et al. 2021) goes further by modeling implicit influence relationships between agents, enabling better credit assignment in tasks with long-term dependencies. Nevertheless, both RIIT and LICA continue to employ a mixing network structure analogous to that used in QMIX, in order to establish the mapping from individual agent Q-values to the global Q-value. Although the mixing network is a critical component for implementing credit assignment, it suffers from limited interpretability and slow convergence during training. Despite the advantages of value decomposition methods, achieving efficient and precise credit assignment in multi-agent systems remains a challenging open research problem.

LLM-enhanced Reinforcement Learning

Leveraging their large-scale pretraining and strong generalization capabilities, LLMs have been increasingly integrated into Reinforcement Learning to enhance performance in multitask learning, sample efficiency, and high-level task planning (Zhang et al. 2023a; Li et al. 2024). In the SMAC-R1 framework (Deng et al. 2024), agents utilize LLMs to generate decision tree code by providing task descriptions. The model further improves the quality and interpretability of the decision tree through self-reflection, guided by feedback from environment-provided rewards. Additionally, LLMs can serve as generators to create high-fidelity multimodal world models, reducing the cost of learning in real-world environments. They can also generate natural language rationales to explain agent behaviors, improving the transparency of decision-making processes. For instance, the Reflexion framework (Shinn et al. 2023) introduces a reinforcement mechanism guided by natural language feedback, without requiring updates to the model parameters. Agents reflect on task-related feedback signals through language and store them in contextual memory, enabling improved decision-making in future interactions. Moreover, LLMs can explicitly or implicitly shape reward signals for RL tasks. The EUREKA framework (Ma et al. 2023) achieves superhuman-level reward function coding through evolutionary search and policy feedback, empowering agents to execute more complex behaviors.

These examples demonstrate the powerful information processing and code generation capabilities of LLMs. Therefore, some researchers have leveraged LLMs to tackle the challenges of temporal and agent-level credit assignment in MARL. Qu et al. (2025) were the first to introduce LLMs into the credit assignment problem. They proposed the notion of latent rewards, making LLMs to capture subtle cooperative relationships among agents and to decompose global rewards accordingly. This method effectively addresses the challenge of multi-agent credit assignment under sparse rewards and outperforms existing baseline algorithms. However, it suffers from instability in code generation and may produce latent rewards that contain redundant or difficult-to-interpret components. Additionally, Lin et al. (2025) employed LLMs to learn potential-based reward functions across multiple queries, allowing for a more accurate estimation of individual agent contributions. These methods primarily focus on decomposing the global reward across agents but remain limited to settings compatible with single-agent reinforcement learning, which constrains their scalability and performance. Therefore, we propose QLLM, a multi-agent credit assignment algorithm based on the *coder-evaluator* framework, to address the aforementioned limitations.

Preliminaries

Decentralized Partially Observable Markov Decision Process

In this work, we focus on a fully cooperative MARL setting that can be formalized as a decentralized partially observable Markov decision process (Dec-POMDP). A Dec-POMDP is typically defined by a tuple $G = \langle S, A, P, r, Z, O, n, \gamma \rangle$. Here, n agents indexed by $i \in \{1, \ldots, n\}$ operate in an environment with true underlying state $s \in S$. At each timestep, agent i selects an action $a^i \in A$ based solely on its private observation $z_i \in Z$, which is sampled according to the observation function $O(s, i)$. The joint action of all agents is denoted by $\boldsymbol{a} \in \boldsymbol{A}$.

The environment evolves according to the transition probability function $P(s' \mid s, \boldsymbol{a}) : S \times \boldsymbol{A} \to [0, 1]$, which specifies the likelihood of transitioning to state s' given the current state s and joint action $\boldsymbol{a}$. All agents cooperate to maximize a shared reward signal defined by the function $r(s, \boldsymbol{a}) : S \times \boldsymbol{A} \to \mathbb{R}$. The objective in this setting is to learn joint policies that maximize the expected cumulative discounted reward $\sum_{j=0}^{\infty} \gamma^j r_{t+j}$, where $\gamma \in [0, 1)$ is a discount factor determining the weight of future rewards.

A key challenge in Dec-POMDPs lies in the credit assignment problem. This issue arises because all agents within the system share a common reward structure, making it inherently difficult to accurately attribute individual contributions to the collective outcome. Furthermore, the environment is perceived as non-stationary from the viewpoint of each agent. This non-stationarity poses significant challenges in achieving effective coordination and optimal decision-making, as agents must adapt to the changing strategies of others while managing incomplete state representations.

Value Decomposition and IGM Principle

Traditional value decomposition-based algorithms employ a mixing network to combine the local Q-value functions Q_i of individual agents into a global Q-value function Q_{tot}. The mixing network serves as a differentiable function approximator that aggregates individual agent utilities while ensuring that the global Q-value adheres to the Individual-Global-Maximum (IGM) constraint. The IGM principle (Hong, Jin, and Tang 2022) requires that the joint action that maximizes the global action-value function Q_{tot} must correspond to the combination of each agent's individually optimal action. Formally, the IGM principle can be expressed as:

$$\arg\max_{\boldsymbol{a}} Q_{\text{tot}}(\boldsymbol{\tau}, \boldsymbol{a}) = \begin{pmatrix} \arg\max_{a^1} Q_1\left(\tau^1, a^1\right) \\ \vdots \\ \arg\max_{a^n} Q_n\left(\tau^n, a^n\right) \end{pmatrix}, \tag{1}$$

where $\boldsymbol{\tau} \in T^n$ represents the joint action-observation histories of all agents. This equation implies that the maximization of the global Q-value function is aligned with the maximization of each agent's local Q-value function, enabling decentralized execution while preserving global optimality. Building upon this, Qatten (Yang et al. 2020) provides a theoretical extension, showing that Q_{tot} can be expressed as a linear combination of the individual Q_i functions:

$$Q_{\text{tot}}(s, \boldsymbol{a}) = \sum_{i=1}^{n} \sum_{h=1}^{H} w_{i,h}(s) Q_i(\tau^i, a^i) + b. \tag{2}$$

Here, $w_{i,h}(s)$ denotes a coefficient that depends on the state s, H is the number of attention heads. Theoretically, these coefficients can be derived from a higher-order functional expansion of Q_{tot} with respect to Q_i, where the contribution of each term diminishes rapidly with increasing derivative order. This indicates that the influence of higher-order interaction terms on Q_{tot} is sufficiently small to be ignored or effectively absorbed into a bias term b, allowing for a simplified model formulation and reduced computational complexity. Meanwhile, the theoretical insight in Equation (2) lays the groundwork for the subsequent development of TFCAF.

Method

This section presents a detailed discussion of the implementation of each component of QLLM. We first propose the concept of the IGM-Gating Mechanism, which enables the learning architecture to switch between strict adherence to the IGM principle and more relaxed joint value modeling. This provides a foundation for adaptively handling different types of multi-agent tasks. Then we introduce the concept of TFCAF, a training-free credit assignment function capable of accurately assigning credit to individual agents. Leveraging the generative capabilities of LLMs, TFCAF is constructed through prompt-based interaction with LLMs. However, when applied to complex tasks, LLMs typically encounter two fundamental challenges, hallucination (Tonmoy et al. 2024) and a lack of rigorous reasoning. These limitations motivate the design of our proposed *coder-evaluator* framework. By integrating this framework, QLLM is able to generate highly accurate TFCAF functions for credit assignment in multi-agent settings, offering an effective alternative to the traditional mixing network used in value decomposition approaches.

IGM-Gating Mechanism

To flexibly handle diverse coordination demands across tasks, we introduce the *IGM-Gating Mechanism*, which enables the learning architecture to switch between strict adherence to the IGM principle and relaxed joint value modeling. This mechanism allows for conditional enforcement of the monotonicity constraint in the credit assignment.

Definition 1 (IGM-Gating Mechanism)**.** *Let $Q_{tot}(\boldsymbol{s}, \boldsymbol{a})$ be the joint action-value function, and $\{Q_i(s_i, a_i)\}_{i=1}^n$ the individual utility functions. Define a binary gating variable $\gamma \in \{0, 1\}$, where $\gamma = 1$ indicates that IGM is enforced. The IGM-Gating Mechanism imposes the following conditional monotonicity constraint:*

$$\forall i, \quad \gamma = 1 \Longrightarrow \frac{\partial Q_{tot}(\boldsymbol{s}, \boldsymbol{a})}{\partial Q_i(s_i, a_i)} \geq 0. \tag{3}$$

When the gating variable is inactive ($\gamma = 0$), no monotonicity constraint is enforced, allowing Q_{tot} to freely capture non-monotonic dependencies.

Training-Free Credit Assignment Function

As a crucial component of credit assignment, the mixing network in value decomposition is essentially a state-conditioned feedforward neural network composed of multiple linear transformation layers and nonlinear activation functions. The network takes as input the individual Q-values of all agents, denoted as $\boldsymbol{Q} = \{Q_1, Q_2, \ldots, Q_n\}$, and produces the joint Q-value Q_{tot} as output. To enable the network to be state-dependent and adaptively parameterized, most value decomposition methods employ the *hypernetwork* mechanism (Ha, Dai, and Le 2017). Specifically, instead of being directly optimized, the weights and biases of each linear layer in the mixing network are generated by a *hypernetwork* that conditions on the global state s. This design facilitates state-aware adaptation while ensuring compatibility with CTDE framework, making deployment straightforward. However, such neural-network-based credit assignment functions cannot effectively incorporate prior knowledge, which may result in slower convergence and reduced interpretability.

To address these issues, we propose the **Training-Free Credit Assignment Function (TFCAF)**, which is the core module of QLLM. This module is generated by LLMs through iterative error correction, based on a series of easily accessible task-related prompts. As illustrated in Equation (2), the global Q-value function Q_{tot} can theoretically

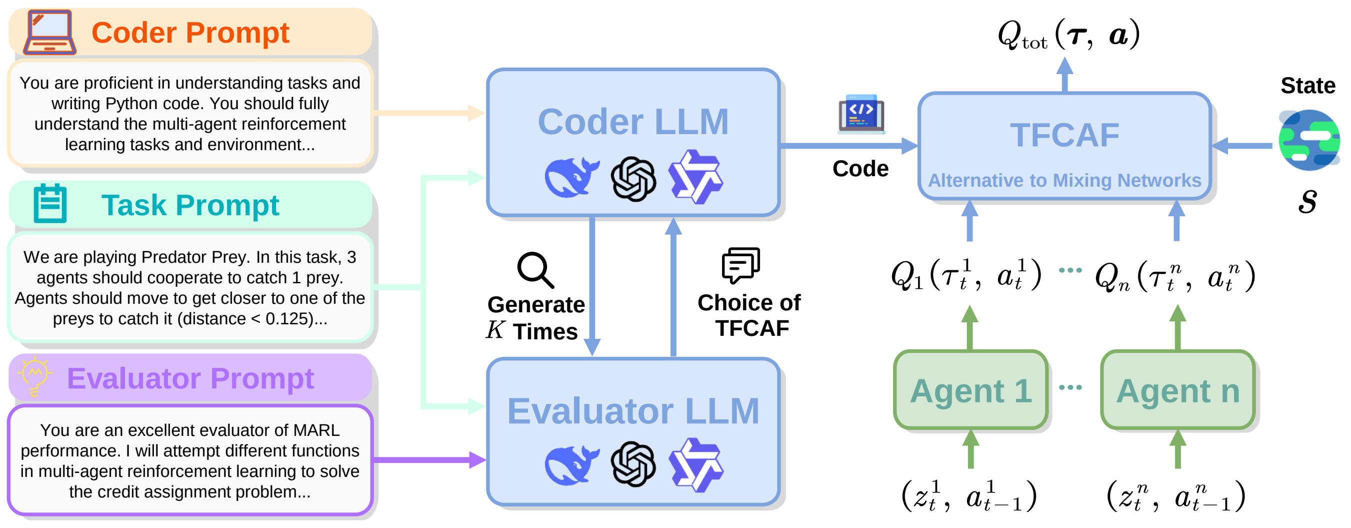
Coder Prompt
You are proficient in understanding tasks and writing Python code. You should fully understand the multi-agent reinforcement learning tasks and environment...
Task Prompt
We are playing Predator Prey. In this task, 3 agents should cooperate to catch 1 prey. Agents should move to get closer to one of the preys to catch it (distance < 0.125)...
Evaluator Prompt
You are an excellent evaluator of MARL performance. I will attempt different functions in multi-agent reinforcement learning to solve the credit assignment problem...
Coder LLM
Evaluator LLM
Generate K Times
Choice of TFCAF
Code
TFCAF
Alternative to Mixing Networks
$Q_{\text{tot}}(\boldsymbol{\tau}, \boldsymbol{a})$
State
s
$Q_1(\tau_t^1, a_t^1)$ ··· $Q_n(\tau_t^n, a_t^n)$
Agent 1 ··· Agent n
(z_t^1, a_{t-1}^1)
(z_t^n, a_{t-1}^n)

Figure 2: QLLM is designed within a *coder-evaluator* framework to autonomously generate high-quality, training-free credit assignment functions (TFCAFs). In each iteration, the *coder LLM* M_{coder} produces K candidate functions based on a task prompt and a role-specific instruction, referred to as the *coder prompt*. These candidate functions are initially assessed using environmental state information and local Q-values. If any runtime or syntax errors are identified, the faulty candidates are discarded and regenerated. Subsequently, the *evaluator LLM* $M_{\text{evaluator}}$ reviews the candidate functions, selects the most promising one based on its *evaluator prompt* This process is generated by LLMs through structured prompts without any need for human intervention.

be decomposed into a weighted sum of individual local Q-values with an additional bias term. Motivated by this decomposition paradigm, we incorporate a similar formulation in the design of TFCAF, which we formally define below.

Definition 2. *A* ***Training-Free Credit Assignment Function (TFCAF)*** *is defined as:*

$$Q_{tot}(s, \boldsymbol{a}) = \sum_{i=1}^{n} f_w^i(s)\, Q_i\left(\tau^i, a^i\right) + f_b(s), \qquad (4)$$

where $f_w^i(s)$ and $f_b(s)$ are general nonlinear functions of the global state s generated by LLMs before training.

As illustrated in Definition 1, when the gating signal $\gamma = 1$, the function $f_w^i(s)$ is strictly constrained to be non-negative. In contrast, when $\gamma = 0$, this constraint is lifted. Benefiting from the rich prior knowledge of LLMs, the appropriate value of γ can be easily inferred from the task description, enabling the model to adapt to different types of tasks. For instance, in matrix game scenarios, where individual optimal decisions do not necessarily align with the global optimum, the LLM may assign a negative value to $f_w^i(s)$. The coefficients $f_w^i(s)$ and the bias term $f_b(s)$ make the overall TFCAF a nonlinear mapping, although its final combination with the local Q_i remains linear in form. Both components are automatically produced by LLMs through the *coder-evaluator* framework, which we detail next along with the iterative correction and its role in credit assignment for multi-agent systems.

Coder Evaluator Framework

Due to the susceptibility of LLMs to hallucinations, the process of generating TFCAF must be designed to be robust. To address this challenge, we propose the *coder-evaluator* framework, which automatically generates TFCAF through a modular and self-correcting process, ensuring both accurate credit assignment and logical consistency in code generation.

This framework employs two LLMs with distinct yet complementary roles: the *coder LLM* M_{coder}, responsible for generating candidate TFCAFs, and the *evaluator LLM* $M_{evaluator}$, which critically evaluates these candidates. All of these are generated by LLMs through structured prompts without any need for human intervention.

The interaction between the two models is driven by two types of prompts: *task prompts* (P_{task}) specify the meanings of each dimension in the global state space, the action space of the agents, and the composition of the reward function. This information is objective and can be easily obtained from the environment; *role prompts* (P_{role}) define the fixed responsibilities and operational roles of each LLM within the framework. The role prompts are further categorized into *coder prompts* and *evaluator prompts*, which remain unchanged across different RL tasks and correspond to the Coder LLM and the Evaluator LLM, respectively. This clear separation and stability of prompt design make QLLM easy to adapt to new cooperative tasks without extensive manual prompt tuning or subjective rewriting (see in Appendix A).

The generation of TFCAF can be formally described through the following three clearly-defined steps.

Initial Generation. Firstly, M_{coder} generates K candidate credit assignment functions by processing both the role-specific and task-specific prompts:

$$\phi_1, \phi_2, ..., \phi_K \Leftarrow M_{coder}\left(P_{role},\ P_{task}\right), \qquad (5)$$

Algorithm 1: Pseudo-code for QLLM

```
    Part 1: TFCAF Generation (LLM-based)
 1: $\phi_1, \phi_2, \ldots, \phi_K \leftarrow M_{\text{coder}}(P_{\text{role}},\ P_{\text{task}})$
                                            ▷ Generate $K$ candidate functions
 2: for all $\phi_k \in \{\phi_1, \ldots, \phi_K\}$ do
 3:     if $\phi_k$ fails on environment samples then
 4:         $\phi_k \leftarrow M_{\text{coder}}(P_{\text{role}}, P_{\text{task}}, \text{error})$
                              ▷ Regenerate function based on error message
 5:     end if
 6: end for
 7: $choice \leftarrow M_{\text{evaluator}}(P_{\text{role}},\ P_{\text{task}},\ \Phi)$
                                         ▷ Evaluator selects best function
 8: $\phi_{\text{final}} \leftarrow choice$
 9: Generate $f_{\text{TFCAF}}$ from $\phi_{\text{final}}$ using predefined template
    Part 2: Agents Optimization (RL-based)
10: for episode = 1 to $M$ do
11:     for each agent $i$ do
12:         $a^i \leftarrow \arg\max_a Q_i(\tau^i, a; \theta)$
13:     end for
14:     Compute global Q-value:
15:         $Q_{\text{tot}}(s, \boldsymbol{a}) = f_{\text{TFCAF}}(Q_1, \ldots, Q_n; s)$
16:     Store transition $(s, \boldsymbol{a}, r, s')$ in buffer $\mathcal{D}$
17:     Sample batch and compute TD target:
18:         $y = r + \gamma \max_{\boldsymbol{a}'} Q_{\text{tot}}(s', \boldsymbol{a}'; \theta^-)$
19:     Update parameters $\theta$ by minimizing the TD loss:
20:         $\mathcal{L}(\theta) = \mathbb{E}\left[(y - Q_{\text{tot}}(s, \boldsymbol{a}; \theta))^2\right]$.
21: end for
```

where $\phi_1, \phi_2, \ldots, \phi_K$ denote candidate TFCAFs, each including instructions for calculating state-dependent weights and biases.

Error Detection and Correction. Due to the inherent hallucination tendencies of LLMs, candidate functions might contain syntax or dimensional errors. To address this, each candidate TFCAF is compiled and run once with any available input state and local Q-values. If an error occurs during execution, the error message generated by the compiler will be fed back to M_{coder}, prompting regeneration of the faulty TFCAF:

$$\phi_{correct} \Leftarrow M_{coder}\big(P_{role},\ P_{task},\ error\big). \tag{6}$$

This correction step repeats until all K candidate TFCAFs can execute successfully without errors.

Selection of TFCAF. Subsequently, $M_{evaluator}$ evaluates these validated candidate TFCAFs, selecting the most promising one based on task relevance, interpretability, and code clarity. Notably, the selection of the best candidate is fully guided by structured prompts and performed automatically by $M_{evaluator}$, without any manual or subjective intervention:

$$choice \Leftarrow M_{evaluator}\big(P_{role},\ P_{task},\ \Phi\big), \tag{7}$$

where $\Phi = \{\phi_1, \phi_2, ..., \phi_K\}$.

Training Procedure

With the *coder-evaluator* framework, we obtain a training-free credit assignment function. Like QMIX and VDN, QLLM is a value-based MARL method developed under the CTDE paradigm. Its objective is to learn a global action-value function $Q_{\text{tot}}(s, \boldsymbol{a})$ that can be decomposed into individual utility functions $Q_i(\tau^i, a^i)$, while avoiding the constraints imposed by conventional hypernetwork-based mixing networks. This design enables the LLM to model more complex and nonlinear interactions between local and global Q-values, allowing it to better adapt to the specific structural properties of the task.

As with most value decomposition approaches, each agent i in QLLM employs a deep neural network to approximate its individual action-value function $Q_i(\tau^i, a^i; \theta)$, where θ denotes the set of learnable parameters. These individual Q-values are then aggregated by the TFCAF, which is automatically generated by LLMs, to produce the global action-value function:

$$Q_{\text{tot}}(s, \boldsymbol{a}) = f_{\text{TFCAF}}(Q_1, \ldots, Q_n; s), \tag{8}$$

where f_{TFCAF} is a parameterized function in which the weights are allowed to take arbitrary values, while the biases are also unconstrained. f_{TFCAF} receives the agent-specific Q-values Q_i as input and utilizes the global state s to generate the corresponding weights and biases through the weight module f_w^i and the bias module f_b, respectively. Notably, all parameters in the TFCAF are generated by LLMs and remain fixed during training. The global Q-value is then computed as shown in Equation (4). The training objective is to minimize the temporal-difference (TD) loss between the predicted total Q-value and the target Q-value:

$$\mathcal{L}(\theta) = \mathbb{E}_{(s,\boldsymbol{a},r,s')}\left[\left(y - Q_{\text{tot}}(s, \boldsymbol{a}; \theta)\right)^2\right], \tag{9}$$

where the target value y is computed using Bellman equation:

$$y = r + \gamma \max_{\boldsymbol{a}'} Q_{\text{tot}}(s', \boldsymbol{a}'; \theta^-), \tag{10}$$

with γ being the discount factor. θ^- denotes the parameters of the target agent network.

Compared to other neural network-based approaches for multi-agent credit assignment, QLLM involves fewer trainable parameters. This facilitates more precise credit allocation among agents in the early training stages, thereby accelerating the convergence of their decision-making policies. Meanwhile, the incorporation of TFCAF allows QLLM to effectively exploit prior knowledge while employing an interpretable credit assignment mechanism.

Experiments

To comprehensively evaluate the proposed method, the experiments were designed with six main objectives: (1) to assess the performance of the QLLM algorithm in terms of average return across multi-agent environments; (2) to verify its adaptability to high-dimensional global state spaces by increasing the dimensionality in selected tasks; (3) to demonstrate the compatibility of QLLM by integrating it into other mixing-based algorithms; (4) to demonstrate the compatibility of QLLM with various LLMs; (5) to validate the effectiveness of the proposed *coder-evaluator* framework under various parameters; (6) to demonstrate the superiority of our method in terms of training costs.

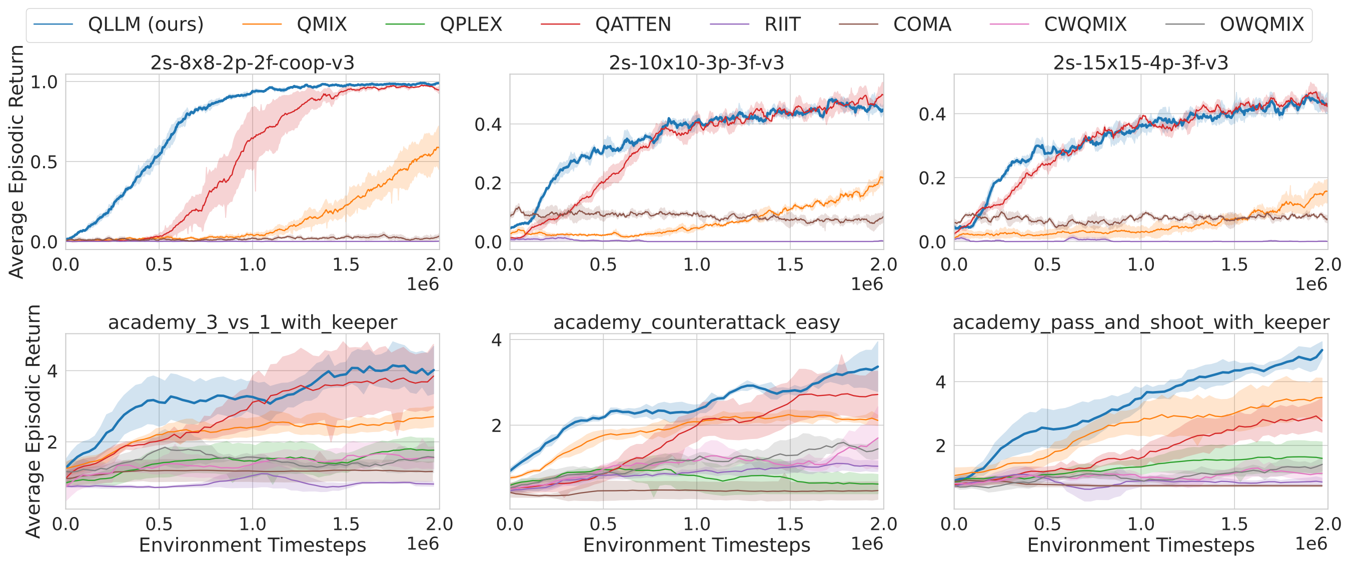

Figure 3: Average episodic return curves for selected tasks in GRF and MPE environments (the results for LBF are shown in Appendix C).

$$\begin{bmatrix} 0 & 6 & 5 \\ -30 & 7 & 0 \\ 11 & -30 & 0 \end{bmatrix}$$

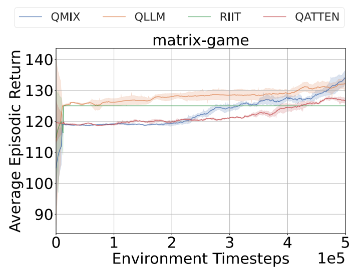

Figure 4: Average episodic return curves in matrix-games environment (**Right**) and the payoff matrix used in this environment (**Left**).

Experimental Setups

We tested QLLM's performance on four common MARL benchmarks and compared it with several well-known baseline credit assignment algorithms. The primary LLM used in our method is the **DeepSeek-R1** (Guo et al. 2025) inference model. For each experiment, the reported return values represent the mean over five independent training runs, with 90% confidence intervals indicated by error bars. Unless otherwise specified, all environment parameters are set to their default values. Our experimental environments include: (1) Cooperative Matrix Games[1], used to validate the algorithm's performance in non-monotonic scenarios, which may violate the IGM principle; (2) Multi-agent Particle Environments (MPE)[2], for evaluating performance in high-dimensional state spaces; (3) Level-Based Foraging (LBF)[3], for testing in sparse reward settings; and (4) Google Research Football (GRF)[4], to assess performance in complex, realistic environments.

We compare QLLM with six commonly used baseline algorithms: QMIX (Rashid et al. 2020b), QPLEX (Wang et al. 2020), Weighted QMIX (Rashid et al. 2020a), Qatten (Yang et al. 2020), RIIT (Hu et al. 2021), MASER (Jeon et al. 2022), and COMA (Foerster et al. 2018). A detailed introduction and implementation details for all baselines and environments are provided in Appendix B.

The Superiority of QLLM

Average Return and Convergence Speed. Figure 3 and Figure 4 show the performance of various algorithms in certain scenarios. It is evident that the QLLM algorithm outperforms the baseline in both the dense-reward MPE environment and the sparse-reward GRF environment. Especially in the GRF environment, COMA almost fail to train the agents properly, which highlights the importance of accurate credit assignment in multi-agent environments. Furthermore, the incorporation of the IGM-Gating mechanism enables our algorithm to perform effectively in non-monotonic matrix game environments, outperforming methods like QMIX that are limited by rigid monotonicity constraints. In Listing 1, we present an example of a credit assignment function generated by LLMs for the *pz-mpe-simple-spread* task in the MPE environment. It can be seen that this function takes into account key factors such as the agents' positions, their distances to landmarks, as well as the number of collisions, accurately assigning contributions to each agent. These verify the effectiveness and generalization ability of the credit assignment functions generated by our method.

Performance in Higher Dimensional State Spaces. We tested the performance of various algorithms in high-dimensional state spaces of certain scenarios. Taking the *pz-mpe-simple-spread* task as an example in the MPE envi-

[1]https://github.com/uoe-agents/matrix-games
[2]https://pettingzoo.farama.org/environments/mpe/
[3]https://github.com/uoe-agents/lb-foraging
[4]https://github.com/google-research/football

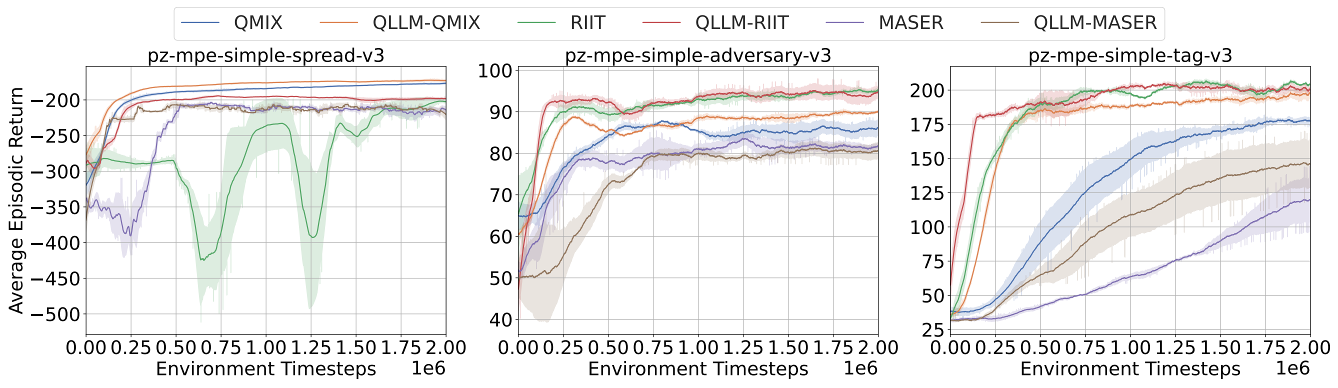

Figure 5: Compatibility evaluation of QLLM in the MPE environments. The original mixing networks in RIIT, MASER, and QMIX are replaced by the TFCAF generated by QLLM, and the resulting performance is compared against the original implementations.

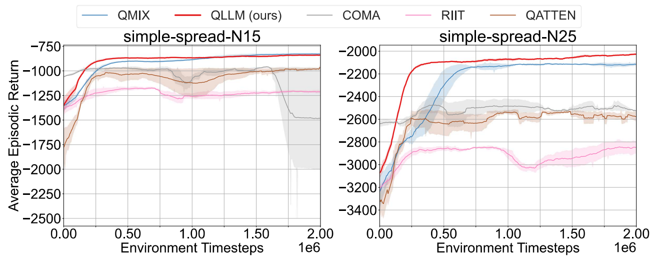

Figure 6: The results of high-dimensional testing in MPE environment.

ronment, in the navigation scenario, as the number of agents increases, the global state space dimension becomes very large, and the difficulty of credit assignment grows exponentially. In Figure 6, the results indicate that most algorithms perform poorly when faced with such a high-difficulty task, while our algorithm still maintains a high level of contribution assignment accuracy, which is attributed to the mechanism of using LLMs in QLLM. The baseline algorithms rely entirely on neural networks for credit assignment, and as the state dimensions increase, determining the weight values becomes more difficult. However, our algorithm uses a set of well-defined nonlinear functions to assign contributions, which are unaffected by high-dimensionality, thus maintaining a high level of performance.

Compatibility Test. QLLM is still based on value decomposition methods, so it is compatible with most MARL algorithms that use mixing networks to compute global Q-values. We used the TFCAF generated by QLLM to replace the mixing networks in three value-decomposition algorithms—RIIT, MASER, and QMIX. We then conducted experiments in the MPE environment to examine how integrating TFCAF affects the performance of these value-decomposition MARL methods. The experimental results tested in MPE environments are shown in Figure 5. Our results show that QLLM can be seamlessly incorporated into existing architectures, offering a flexible and scalable solution for a wide range of multi-agent settings.

Listing 1: A credit assignment function example of *pz-mpe-simple-spread* environment generated by LLMs.

```
def QLLMNetwork(agents_q: torch.Tensor, global_state:
     torch.Tensor) -> torch.Tensor:
    batchsize = agents_q.size(0)
    n_agents = 6
    global_state_reshaped = global_state.view(
         batchsize, n_agents, 36)
    # Process landmark distances
    landmarks_relative = global_state_reshaped[:, :,
         4:16].view(batchsize, n_agents, 6, 2)
    landmark_distances = torch.norm(landmarks_relative
         , dim=-1)
    min_dist_per_agent = torch.clamp(torch.min(
         landmark_distances, dim=-1)[0], min=0.1)
    pos_weights = 1.0 / min_dist_per_agent
    # Process collision counts
    positions = global_state_reshaped[:, :, 2:4]
    diffs = positions.unsqueeze(2) - positions.
         unsqueeze(1)
    pairwise_dist = torch.norm(diffs, dim=-1)
    collision_mask = (pairwise_dist <= 0.3) & ~torch.
         eye(n_agents, device=positions.device).bool()
         .unsqueeze(0)
    collision_weights = collision_mask.sum(dim=-1).
         float()
    # Combine weights and softmax
    combined_weights = pos_weights - collision_weights
    softmax_weights = torch.softmax(combined_weights,
         dim=-1)
    global_q = (agents_q * softmax_weights).sum(dim
         =-1, keepdim=True)
    return global_q
```

Conclusion

In this study, we propose QLLM, a method that uses LLMs to address the credit assignment problem in multi-agent reinforcement learning through *coder-evaluator* framework. This approach generates training-free, task-specific credit assignment functions, reducing hallucination and improving task understanding. Experiments on four standard MARL benchmarks show that QLLM consistently outperforms baselines and generalizes well across different mixing-based algorithms. Future work will explore broader applications and deployment in real-world multi-robot systems.

References

Agarwal, A.; Kakade, S. M.; Lee, J. D.; and Mahajan, G. 2021. On the theory of policy gradient methods: Optimality, approximation, and distribution shift. *Journal of Machine Learning Research*, 22: 1–76.

Anil, C.; Wu, Y.; Andreassen, A.; Lewkowycz, A.; Misra, V.; Ramasesh, V.; Slone, A.; Gur-Ari, G.; Dyer, E.; and Neyshabur, B. 2022. Exploring length generalization in large language models. *Advances in Neural Information Processing Systems*, 35: 38546–38556.

Deng, Y.; Ma, W.; Fan, Y.; Zhang, Y.; Zhang, H.; and Zhao, J. 2024. A new approach to solving smac task: Generating decision tree code from large language models. *arXiv preprint arXiv:2410.16024*.

Desmond, M.; Ashktorab, Z.; Pan, Q.; Dugan, C.; and Johnson, J. M. 2024. EvaluLLM: LLM assisted evaluation of generative outputs. In *Companion Proceedings of the 29th International Conference on Intelligent User Interfaces*, 30–32.

Foerster, J.; Farquhar, G.; Afouras, T.; Nardelli, N.; and Whiteson, S. 2018. Counterfactual multi-agent policy gradients. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32.

Guo, D.; Yang, D.; Zhang, H.; Song, J.; Zhang, R.; Xu, R.; Zhu, Q.; Ma, S.; Wang, P.; Bi, X.; et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Ha, D.; Dai, A. M.; and Le, Q. V. 2017. HyperNetworks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.

Hong, Y.; Jin, Y.; and Tang, Y. 2022. Rethinking individual global max in cooperative multi-agent reinforcement learning. *Advances in neural information processing systems*, 35: 32438–32449.

Hu, J.; Jiang, S.; Harding, S. A.; Wu, H.; and Liao, S.-w. 2021. Rethinking the implementation tricks and monotonicity constraint in cooperative multi-agent reinforcement learning. *arXiv preprint arXiv:2102.03479*.

Iqbal, S.; and Sha, F. 2019. Actor-attention-critic for multi-agent reinforcement learning. In *International conference on machine learning*, 2961–2970. PMLR.

Jeon, J.; Kim, W.; Jung, W.; and Sung, Y. 2022. Maser: Multi-agent reinforcement learning with subgoals generated from experience replay buffer. In *International conference on machine learning*, 10041–10052. PMLR.

Kasneci, E.; Seßler, K.; Küchemann, S.; Bannert, M.; Dementieva, D.; Fischer, F.; Gasser, U.; Groh, G.; Günnemann, S.; Hüllermeier, E.; et al. 2023. ChatGPT for good? On opportunities and challenges of large language models for education. *Learning and individual differences*, 103: 102274.

Kaven, L.; Huke, P.; Göppert, A.; and Schmitt, R. H. 2024. Multi agent reinforcement learning for online layout planning and scheduling in flexible assembly systems. *Journal of Intelligent Manufacturing*, 35: 3917–3936.

Li, D.; Dong, H.; Wang, L.; Qiao, B.; Qin, S.; Lin, Q.; Zhang, D.; Zhang, Q.; Xu, Z.; Zhang, B.; et al. 2024. Verco: Learning coordinated verbal communication for multi-agent reinforcement learning. *arXiv preprint arXiv:2404.17780*.

Lillicrap, T. P.; Hunt, J. J.; Pritzel, A.; Heess, N.; Erez, T.; Tassa, Y.; Silver, D.; and Wierstra, D. 2015. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

Lin, M.; Shi, S.; Guo, Y.; Tadiparthi, V.; Chalaki, B.; Pari, E. M.; Stepputtis, S.; Kim, W.; Campbell, J.; and Sycara, K. 2025. Speaking the Language of Teamwork: LLM-Guided Credit Assignment in Multi-Agent Reinforcement Learning. *arXiv preprint arXiv:2502.03723*.

Linardatos, P.; Papastefanopoulos, V.; and Kotsiantis, S. 2020. Explainable ai: A review of machine learning interpretability methods. *Entropy*, 23: 18.

Liu, B.; Pu, Z.; Pan, Y.; Yi, J.; Liang, Y.; and Zhang, D. 2023. Lazy agents: A new perspective on solving sparse reward problem in multi-agent reinforcement learning. In *International Conference on Machine Learning*, 21937–21950. PMLR.

Lowe, R.; Wu, Y. I.; Tamar, A.; Harb, J.; Pieter Abbeel, O.; and Mordatch, I. 2017. Multi-agent actor-critic for mixed cooperative-competitive environments. *Advances in neural information processing systems*, 30.

Ma, Y. J.; Liang, W.; Wang, G.; Huang, D.-A.; Bastani, O.; Jayaraman, D.; Zhu, Y.; Fan, L.; and Anandkumar, A. 2023. Eureka: Human-level reward design via coding large language models. *arXiv preprint arXiv:2310.12931*.

Martino, A.; Iannelli, M.; and Truong, C. 2023. Knowledge injection to counter large language model (LLM) hallucination. In *European Semantic Web Conference*, 182–185. Springer.

Oroojlooy, A.; and Hajinezhad, D. 2023. A review of cooperative multi-agent deep reinforcement learning. *Applied Intelligence*, 53: 13677–13722.

Orr, J.; and Dutta, A. 2023. Multi-agent deep reinforcement learning for multi-robot applications: A survey. *Sensors*, 23: 3625.

Petroni, F.; Rocktäschel, T.; Lewis, P.; Bakhtin, A.; Wu, Y.; Miller, A. H.; and Riedel, S. 2019. Language models as knowledge bases? *arXiv preprint arXiv:1909.01066*.

Qu, Y.; Jiang, Y.; Wang, B.; Mao, Y.; Wang, C.; Liu, C.; and Ji, X. 2025. Latent reward: Llm-empowered credit assignment in episodic reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, 20095–20103.

Rashid, T.; Farquhar, G.; Peng, B.; and Whiteson, S. 2020a. Weighted qmix: Expanding monotonic value function factorisation for deep multi-agent reinforcement learning. *Advances in neural information processing systems*, 33: 10199–10210.

Rashid, T.; Samvelyan, M.; De Witt, C. S.; Farquhar, G.; Foerster, J.; and Whiteson, S. 2020b. Monotonic value function factorisation for deep multi-agent reinforcement learning. *Journal of Machine Learning Research*, 21: 1–51.

Sanh, V.; Webson, A.; Raffel, C.; Bach, S. H.; Sutawika, L.; Alyafeai, Z.; Chaffin, A.; Stiegler, A.; Scao, T. L.; Raja, A.; et al. 2021. Multitask prompted training enables zero-shot task generalization. *arXiv preprint arXiv:2110.08207*.

Shinn, N.; Cassano, F.; Gopinath, A.; Narasimhan, K.; and Yao, S. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36: 8634–8652.

Son, K.; Kim, D.; Kang, W. J.; Hostallero, D. E.; and Yi, Y. 2019. Qtran: Learning to factorize with transformation for cooperative multi-agent reinforcement learning. In *International conference on machine learning*, 5887–5896. PMLR.

Soydaner, D. 2022. Attention mechanism in neural networks: where it comes and where it goes. *Neural Computing and Applications*, 34: 13371–13385.

Su, J.; Adams, S.; and Beling, P. 2021. Value-decomposition multi-agent actor-critics. In *Proceedings of the AAAI conference on artificial intelligence*, volume 35, 11352–11360.

Sunehag, P.; Lever, G.; Gruslys, A.; Czarnecki, W. M.; Zambaldi, V.; Jaderberg, M.; Lanctot, M.; Sonnerat, N.; Leibo, J. Z.; Tuyls, K.; et al. 2017. Value-decomposition networks for cooperative multi-agent learning. *arXiv preprint arXiv:1706.05296*.

Tonmoy, S.; Zaman, S.; Jain, V.; Rani, A.; Rawte, V.; Chadha, A.; and Das, A. 2024. A comprehensive survey of hallucination mitigation techniques in large language models. *arXiv preprint arXiv:2401.01313*, 6.

Wang, J.; Ren, Z.; Liu, T.; Yu, Y.; and Zhang, C. 2020. Qplex: Duplex dueling multi-agent q-learning. *arXiv preprint arXiv:2008.01062*.

Xu, F. F.; Alon, U.; Neubig, G.; and Hellendoorn, V. J. 2022a. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN international symposium on machine programming*, 1–10.

Xu, Z.; Bai, Y.; Li, D.; Zhang, B.; and Fan, G. 2022b. SIDE: State Inference for Partially Observable Cooperative Multi-Agent Reinforcement Learning. In *Proceedings of the 21st International Conference on Autonomous Agents and Multiagent Systems*, 1400–1408.

Yan, Y.; Chow, A. H.; Ho, C. P.; Kuo, Y.-H.; Wu, Q.; and Ying, C. 2022. Reinforcement learning for logistics and supply chain management: Methodologies, state of the art, and future opportunities. *Transportation Research Part E: Logistics and Transportation Review*, 162: 102712.

Yang, Y.; Hao, J.; Liao, B.; Shao, K.; Chen, G.; Liu, W.; and Tang, H. 2020. Qatten: A general framework for cooperative multiagent reinforcement learning. *arXiv preprint arXiv:2002.03939*.

Yu, C.; Velu, A.; Vinitsky, E.; Gao, J.; Wang, Y.; Bayen, A.; and Wu, Y. 2022. The surprising effectiveness of ppo in cooperative multi-agent games. *Advances in neural information processing systems*, 35: 24611–24624.

Zhang, B.; Mao, H.; Li, L.; Xu, Z.; Li, D.; Zhao, R.; and Fan, G. 2024a. Sequential asynchronous action coordination in multi-agent systems: A stackelberg decision transformer approach. In *Forty-first International Conference on Machine Learning*.

Zhang, B.; Mao, H.; Ruan, J.; Wen, Y.; Li, Y.; Zhang, S.; Xu, Z.; Li, D.; Li, Z.; Zhao, R.; et al. 2023a. Controlling large language model-based agents for large-scale decision-making: An actor-critic approach. *arXiv preprint arXiv:2311.13884*.

Zhang, C.; Chen, J.; Li, J.; Peng, Y.; and Mao, Z. 2023b. Large language models for human–robot interaction: A review. *Biomimetic Intelligence and Robotics*, 3: 100131.

Zhang, R.; Hou, J.; Walter, F.; Gu, S.; Guan, J.; Röhrbein, F.; Du, Y.; Cai, P.; Chen, G.; and Knoll, A. 2024b. Multi-agent reinforcement learning for autonomous driving: A survey. *arXiv preprint arXiv:2408.09675*.

Zhou, M.; Liu, Z.; Sui, P.; Li, Y.; and Chung, Y. Y. 2020. Learning implicit credit assignment for cooperative multi-agent reinforcement learning. *Advances in neural information processing systems*, 33: 11853–11864.

A LLM Prompts and Responses

Below are the prompt template, example task information, and LLM response in our work.

Role Prompt

Coder's ROLE Prompt:
You are proficient in understanding tasks and writing Python code. You should fully understand the multi-agent reinforcement learning tasks and environment, including the agents, action space, observation space, and global state space provided by the user. Then, based on your understanding of the task and objectives, you need to write a function for multi-agent credit assignment, called QLLMNetwork, which is used to measure each agents' contribution to the collective reward. The function input will be the individual Q-values of the agents and the global state, and the output will be the global Q-value. Both input and output should be torch tensors.Individual Q-values represent each agent's expected contribution to the team's future rewards, while the global Q-value denotes the expected overall reward of the team as a whole. The input Q-values will have a shape of torch.Size([batchsize, n_agent]), the input global state will have a shape of torch.Size([batchsize, state_dim]), and the output global Q-value should have a shape of torch.Size([batchsize, 1]).You must strictly follow the input and output dimension requirements! In the credit assignment function you wrote,You need to extract the individual components of the local observation vectors of the individual agents. You must accurately understand the meaning of each component in the global state space, and based on that, determine the weights for each Q-value. You must understand the information and suggestions provided by the user in real-time to optimize the function you design, but strictly adhere to the function's input and output requirements!
Note:
1.It must be a function, not a class. The parameters in the function are fixed and do not require training to calculate an accurate global Q-value. Do not use trainable layers such as nn.Linear.
2.You must accurately understand the meaning of each component in the global state space, without making any assumptions or guesses!
3.When calculating the weights, do not divide by values that may be close to zero. Be careful to check intermediate results, weights for invalid numbers!This is not solved just by adding a small number to the dividend.
4.The weights are usually a combination of several components, and when calculating the individual components of the weights, it is important to pay attention to the ratio between the components and the range of the value. At the same time, the weights can be output through a softmax layer if it is necessary.
5.All tensor data should be computed on the GPU. Please write accurate code and strictly follow the template output below, without including any other explanatory text or code run sample.

```
def QLLMNetwork(agents_q: torch.Tensor, global_state: torch.Tensor) -> torch.Tensor:
    # You design this place
    return global_q
```

Evaluator's ROLE Prompt:
You are an excellent evaluator of multi-agent reinforcement learning performance. I will attempt different functions in multi-agent reinforcement learning to solve the credit assignment problem. This function is required to compute the global Q-values accurately without training. It must not use trainable layers such as nn.Linear. Your task is to propose modifications to the function in brief plain text suggestions or return function selection results in an array format. Your suggestions must be in line with the task description and requirements, and should not be based on random assumptions or unrealistic scenarios. Please listen carefully to my subsequent instructions and do not provide additional answers! When you output plain text suggestions, please ensure that you do not output any selections. When you output selections in an array format, please ensure that you do not output plain text suggestions. Whether you output plain text suggestions or selections, consider if the function meets all the details and considerations specified in the task description. Do not use any information that has not been provided to you to answer!

Task Prompt

Task Prompt of *pz-mpe-simple-spread-v3*:

1. Composition of the reward:
We are playing Cooperative Navigation. In this task, 3 agents are asked to reach 3 landmarks. Each agent should move to get close to one landmark and avoid collision(distance<=0.3) with other agents. The agent-team are rewarded based

on how far the closest agent is to each landmark (the reward equals the negative value of the sum of the minimum distances). At the same time, the agent-team are penalized if they collide each other (-1 for each collision). At each step, each agent receives an observation array.
2. State space:
This observation is represented by an array with 18 dimensions:
concat([agent's velocity, agent's location, 3 landmarks' relative locations, 2 other agents' relative locations, communication code]).
The agent's velocity is 2-dimensional, including x and y components.
The agent's location is 2-dimensional, including x and y coordinates.
The landmarks' relative locations are 3x2 dimensions, including relative x and y coordinates (landmark_i.x-agent.x, landmark_i.y-agent.y).
The other agents' relative locations are 2x2 dimensions, including relative x and y coordinates (agent_i.x-agent.x, agent_i.y-agent.y).
The communication code are 4 dimensions, they have no role in credit assignment.
The global state space is concatenate together from the observation space of the three agents, which are 18x3 dimensions.
3. Action space:
`[move_left, move_right, move_down, move_up]`

Task Prompt of *lbforaging:Foraging-8x8-2p-2f-coop-v3*:

The Level-Based Foraging environment focus on the coordination of involved agents. The task for each agent is to navigate the grid-world map and collect items. Each agent and food item is assigned a level and food items are randomly scattered in the environment. In order to collect a food item, agents have to choose a pick-up action next to the food item. However, such collection is only successful if 2 agents participate in the pickup at the same time. At the same time the agents involved in the pickup must be one square above, below, left and right of the food item. Agents receive team reward equal to the level of the collected food item. In each step, every agent can choose to act by moving in one of four directions or attempt to pick up a food item. The task is an 8*8 grid-world with 2 agents and 2 food items scattered in the grid. Below, we will provide additional details to global states, actions and rewards for the Level-Based Foraging environment.
1. State space:
Global state vector consists of 3 component(agents component,food items component and last step action)(Total dimension: 2x3+2x3+2=14 dims)
1.Agents Component (2 agents × 3 features = 6 dims)
Feature 0: agent_i's x coordinate (i=0,1)
Feature 1: agent_i's y coordinate
Feature 2: agent_i's level
2.Food Items Component (2 Food Items × 3 features = 6 dims)
Feature 0: food_i's x coordinate (i=0,1)
Feature 1: food_i's y coordinate
Feature 2: food_i's level
When a food item has been picked up, then the respective values are replaced with (-1, -1, 0).You have to be careful to handle this particular situation properly.
3.The actions just executed by the agents. (2 dims)
{0,1,2...5} respectively represent {Noop, Move North, Move South, Move West, Move East, Pickup}
This information is set to [-1 -1] when the environment is in the first step of each episode.
2. Action space:
In Level-Based Foraging environments, each agent has six possible discrete actions to choose at each timestep:
Ai = {Noop, Move North, Move South, Move West, Move East, Pickup}
While the first action corresponds to the action simply staying within its grid and the last action being used to pickup nearby food, the remaining four actions encode discrete 2D navigation. Upon choosing these actions, the agent moves a single cell in the chosen direction within the grid. Picking up is only valid if the sum of the levels of the agents doing the picking up action is greater than or equal to the level of the food. At the same time, the agents involved in the pickup must be positioned one cell above, below, to the left, or to the right of the food.
3. Composition of the reward:
Agents only receive non-zero team reward for picking up food items within the Level-Based Foraging environment. The reward for picking up a food item equals to the food level.

Task Prompt of *matrixgames:climbing-nostate-v0*:

This task is a two-player cooperative matrix games with a given 3×3 matrix:
`[[11, -30, 0], [-30, 7, 0], [0, 6, 5]]`
1. Action space:
The environment consists of two agents, agent_0 and agent_1, each having three possible actions(0,1,2) corresponding to the row and column indices of the matrix.
2. Composition of the reward:
Once both agents select their actions, they receive a shared reward determined by the corresponding entry in the matrix. There is no communication between the two agents, and the objective is to maximize the shared reward. Be sure to prevent the strategy from falling into a local optimum, and only get the highest reward if both intelligences choose action 0.
3. State space:
The state vector is 2 dims, storing the actions that the two agents have just executed. At the first step at the beginning of an episode, the state vector is [-1,-1], because the agents have not yet started executing actions. Please take care to handle this special case.

Task Prompt of *academy_counterattack_easy*:

1. Action space:
This is an offensive football game where our team controls four reinforcement learning agents, including one goalkeeper and three attacking players. The opposing team consists of one goalkeeper and one defender. The objective is to move the ball as close as possible to the opponent's goal, located at coordinate (1.0, 0.0), through passing and off-ball movement, and to score a goal.
2. Composition of the reward:
The agents receive rewards based on two components: progress reward and goal-scoring reward. The progress reward is given when a player on our team carries the ball closer to the opponent's goal, providing a team reward. This reward is only available when our player is in possession of the ball and the ball's distance to the opponent's goal is greater than 0.19 and less than 0.99. The player in possession is usually the one closest to the ball. The goal-scoring reward is provided when our team successfully scores. In addition to this reward, an extra reward is given based on the distance from which the goal is scored. The farther the distance, the higher the extra reward.
3. State space:
Each agent has a local observation vector of 115 dimensions:
0–7: positions of our four players, each represented by (x, y)
8–15: directions of our four players, each represented by (x, y)
16–19: positions of two opponent players, each represented by (x, y)
20–23: directions of two opponent players, each represented by (x, y)
24–87: invalid information
88–90: ball position (x, y, z)
91–93: ball direction (x, y, z)
94–96: ball possession, encoded as one-hot vectors: our team [0, 1, 0], opponent [0, 0, 1], no possession [1, 0, 0]
97–114: invalid information
The global state vector is constructed by concatenating the local observations of all four agents, resulting in a total of 460 dimensions.

TFCAF template

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
class QLLMMixer(nn.Module):
    def __init__(self):
        super(QLLMMixer, self).__init__()
    def forward(self, agents_q, global_state):
        agents_q = torch.squeeze(agents_q)
        a = agents_q.shape[0]
        b = agents_q.shape[1]
        agents_q = agents_q.reshape(-1, agents_q.shape[-1])
        global_state = global_state.reshape(-1, global_state.shape[-1])
        #This is generated by LLM
        return (global_qagents_q.shape[-1]).reshape(a, b, 1).cuda()
```

LLM Generated Responses

LLM Response of *pz-mpe-simple-spread*:

```
import torch def QLLMNetwork(agents_q: torch.Tensor, global_state: torch.Tensor) -> torch.Tensor:
    batchsize = agents_q.size(0)
    n_agents = 6
    global_state_reshaped = global_state.view(batchsize, n_agents, 36)
    # Process landmark distances
    landmarks_relative = global_state_reshaped[:, :, 4:16].view(batchsize, n_agents, 6, 2)
    landmark_distances = torch.norm(landmarks_relative, dim=-1)
    min_dist_per_agent = torch.clamp(torch.min(landmark_distances, dim=-1)[0], min=0.1)
    pos_weights = 1.0 / min_dist_per_agent
    # Process collision counts
    positions = global_state_reshaped[:, :, 2:4]
    diffs = positions.unsqueeze(2) - positions.unsqueeze(1)
    pairwise_dist = torch.norm(diffs, dim=-1)
    collision_mask = (pairwise_dist <= 0.3)  &  torch.eye(n_agents, device=positions.device).bool().unsqueeze(0)
    collision_weights = collision_mask.sum(dim=-1).float()
    # Combine weights and softmax
    combined_weights = pos_weights - collision_weights
    softmax_weights = torch.softmax(combined_weights, dim=-1)
    global_q = (agents_q  softmax_weights).sum(dim=-1, keepdim=True)
    return global_q
```

LLM Response of *lbforaging:Foraging-8x8-2p-2f-coop-v3*:

```
import torch
def QLLMNetwork(agents_q: torch.Tensor, global_state: torch.Tensor) -> torch.Tensor:
    batch_size = agents_q.size(0)
    agents_data = global_state[:, :6].view(batch_size, 2, 3)
    food_data = global_state[:, 6:12].view(batch_size, 2, 3)
    agents_level = agents_data[:, :, 2]
    agents_pos = agents_data[:, :, :2].unsqueeze(2)
    food_pos = food_data[:, :, :2].unsqueeze(1)
    dxdy = torch.abs(agents_pos - food_pos).sum(dim=-1)
    adjacent = (dxdy == 1).float()
    active_food_mask = ((food_data[:, :, 0] != -1) & (food_data[:, :, 1] != -1)).unsqueeze(1)
    adjacent = adjacent  active_food_mask
    total_contribution = torch.zeros_like(agents_level)
    for j in range(2):
        adjacent_j = adjacent[:, :, j]
        food_level_j = food_data[:, j, 2]
        sum_adj_levels = (agents_level  adjacent_j).sum(dim=1)
        num_adj = adjacent_j.sum(dim=1)
        valid = ((num_adj >= 2) & (sum_adj_levels >= food_level_j)).float().unsqueeze(1)
        contribution_j = adjacent_j  food_level_j.unsqueeze(1)  valid
        total_contribution += contribution_j
    agent_contrib = agents_level  total_contribution
    weights = torch.softmax(agent_contrib, dim=1)
    global_q = (agents_q  weights).sum(dim=1, keepdim=True)
    return global_q
```

LLM Response of *academy_counterattack_easy*:

```
def QLLMNetwork(agents_q: torch.Tensor, global_state: torch.Tensor) -> torch.Tensor:
    batch_size = agents_q.size(0)
    n_agents = agents_q.size(1)
    # Extract ball information from first agent's observation
    ball_pos = global_state[:, 88:90]
    holder_info = global_state[:, 94:97]
    is_my_team = holder_info[:, 1].bool()
    # Calculate distance to goal
    goal_pos = torch.tensor([1.0, 0.0], device=agents_q.device)
    ball_goal_dist = torch.norm(ball_pos - goal_pos, dim=1)
    valid_advance = is_my_team & (ball_goal_dist > 0.19) & (ball_goal_dist < 0.99)
    # Calculate agent-ball distances and holder mask
    all_dists = []
    for i in range(n_agents):
        agent_obs = global_state[:, i115:(i+1)115]
        agent_id = torch.argmax(agent_obs[:, 97:108], dim=1)
        pos_x = agent_obs[torch.arange(batch_size), 2agent_id]
        pos_y = agent_obs[torch.arange(batch_size), 2agent_id+1]
        dist = torch.norm(torch.stack([pos_x-ball_pos[:,0], pos_y-ball_pos[:,1]], dim=1), dim=1)
        all_dists.append(dist.unsqueeze(1))
    dist_matrix = torch.cat(all_dists, dim=1)
    holder_mask = F.one_hot(torch.argmin(dist_matrix, dim=1), num_classes=n_agents).bool()
    # Calculate agent-goal distances
    goal_dists = []
    for i in range(n_agents):
        agent_obs = global_state[:, i115:(i+1)115]
        agent_id = torch.argmax(agent_obs[:, 97:108], dim=1)
        pos_x = agent_obs[torch.arange(batch_size), 2agent_id]
        pos_y = agent_obs[torch.arange(batch_size), 2agent_id+1]
        dist = torch.norm(torch.stack([pos_x-goal_pos[0], pos_y-goal_pos[1]], dim=1), dim=1)
        goal_dists.append(dist.unsqueeze(1))
    goal_dist_matrix = torch.cat(goal_dists, dim=1)
    # Calculate weights components
    advance_component = holder_mask.float()  valid_advance.unsqueeze(1)  5.0
    position_component = 1.0 / (goal_dist_matrix + 1e-6)
    # Combine components with softmax
    combined = torch.where(valid_advance.unsqueeze(1), advance_component, position_component)
    weights = F.softmax(combined, dim=1)
     # Compute global Q-value
    global_q = (agents_q  weights).sum(dim=1, keepdim=True)
    return global_q
```

LLM Response of *matrixgames:climbing-nostate-v0*:

```
def QLLMNetwork(agents_q: torch.Tensor, global_state: torch.Tensor) -> torch.Tensor:
    device = agents_q.device
    batch_size = agents_q.shape[0]
    # Initialize base weights and reward matrix
    weights = torch.full((batch_size, 2), 0.5, device=device)
    reward_matrix = torch.tensor([[11, -30, 0], [-30, 7, 0], [0, 6, 5]], device=device, dtype=torch.float32)
```

```
# Create mask for valid states (non-initial)
valid_mask = (global_state[:, 0] >= 0) & (global_state[:, 1] >= 0)
if valid_mask.any():
    prev_actions = global_state[valid_mask].long()
    a0 = prev_actions[:, 0]
    a1 = prev_actions[:, 1]
    # Calculate previous rewards and optimal condition
    prev_rewards = reward_matrix[a0, a1]
    optimal_mask = (prev_rewards == 11)
    # Calculate deviation from optimal for each agent
    agent0_deviation = (a0 != 0).float()
    agent1_deviation = (a1 != 0).float()
    # Create penalty weights based on deviation
    penalty_weights = torch.stack([1.0 - agent0_deviation  0.5,1.0 - agent1_deviation  0.5], dim=1)
    # Apply optimal condition override
    penalty_weights[optimal_mask] = 0.5 # Maintain equal weights for optimal case
    # Normalize weights to maintain sum=1
    weight_sums = penalty_weights.sum(dim=1, keepdim=True)
    normalized_weights = penalty_weights / weight_sums
    # Update weights for valid samples
    weights[valid_mask] = normalized_weights
    # Compute global Q-value with dynamic weighting
    global_q = (agents_q  weights).sum(dim=1, keepdim=True)
return global_q
```

B Details of Environments and Baselines

Baseline Algorithms

QMIX (Rashid et al. 2020b) is a representative value decomposition method designed for cooperative MARL. It factorizes the joint action-value function into individual agent utilities through a mixing network that enforces a monotonic constraint, ensuring that maximizing local utilities also maximizes the global return. The mixing network parameters are generated by a hypernetwork conditioned on the global state, enabling state-dependent flexible mixing.

QPLEX (Wang et al. 2020) builds on QMIX by introducing a duplex dueling architecture to overcome the limited expressiveness caused by strict monotonicity. It splits the value factorization into a main (advantage) stream and a complementary stream, allowing more flexible interactions while maintaining decentralized policies. This design expands the class of joint value functions that can be represented under the CTDE paradigm.

Weighted QMIX (Rashid et al. 2020a) further generalizes QMIX by learning an additional weighting factor for each agent's contribution to the global Q-value. By relaxing the monotonicity assumption through these learnable weights, it improves the accuracy of credit assignment in tasks with more complex cooperative dependencies.

Qatten (Yang et al. 2020) enhances value decomposition by embedding a multi-head attention mechanism into the mixing network. This mechanism adaptively adjusts the importance of each agent's local utility based on the global state context, enabling the model to focus on the most relevant agents or features when computing the global Q-value.

RIIT (Hu et al. 2021) tackles the multi-agent credit assignment problem by modeling implicit influence relationships among agents. It proposes a reward interpolation technique that redistributes the global team reward across agents, considering both immediate and long-term interdependencies, which helps reduce misattribution and improves coordination performance.

COMA (Foerster et al. 2018) is a centralized actor-critic method designed for cooperative MARL, with a focus on addressing the multi-agent credit assignment problem. It employs a centralized critic that estimates a counterfactual baseline for each agent by marginalizing out its action while keeping others fixed, enabling the computation of an advantage function that isolates each agent's contribution. This counterfactual reasoning helps reduce variance in the policy gradient and promotes coordinated behaviors.

MASER (Jeon et al. 2022) is a value-decomposition-based MARL algorithm designed to improve credit assignment under sparse and delayed rewards. It introduces subgoals automatically generated from the experience replay buffer, which are used to construct intrinsic rewards that guide agents toward meaningful intermediate states. By augmenting standard Q-learning with subgoal-conditioned value estimation, MASER enhances exploration efficiency and coordination while remaining compatible with centralized training and decentralized execution.

Code Repository

The baseline algorithm in this experiment is implemented based on the *epymarl*[5] and *pymarl2*[6] codebases. We tested MPE, LBF, and matrix games using the *epymarl* library, while Google Research Football was evaluated with *pymarl2*. Although the *SMAC*[7] environment is a popular benchmark for multi-agent reinforcement learning and is built upon *PySC2*[8], it lacks comprehensive documentation regarding the specific unit types, abilities, and map configurations used in each scenario. This obscurity hinders reproducibility and limits fine-grained control over experimental variables. Consequently, we opted for several environments with better transparency and configurability to ensure experimental rigor and facilitate controlled evaluations.

Tasks Details

Matrix Games The cooperative matrix games environment is a simple benchmark setting in multi-agent reinforcement learning, where two or more agents simultaneously select actions from predefined action spaces. The environment is defined by a payoff matrix that specifies the global reward received based on the joint actions of all agents (below is the payoff matrix used in test environment). Cooperation is essential, as the optimal strategy often depends on coordinated decision-making among agents. In order to make sense of the global state space of this task, we include the actions executed by the agents last step.

$$\begin{bmatrix} 0 & 6 & 5 \\ -30 & 7 & 0 \\ 11 & -30 & 0 \end{bmatrix}$$

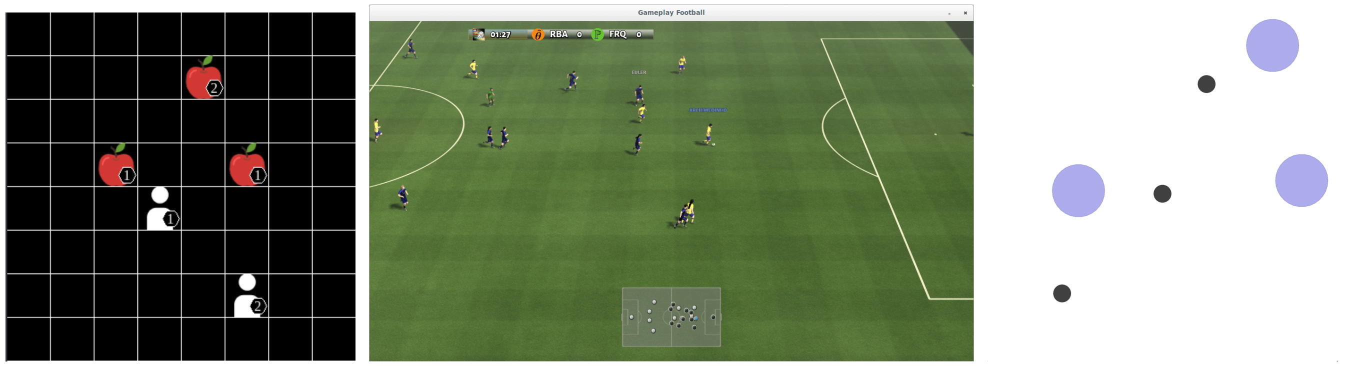

Figure 7: Environment visualizations of Level-Based Foraging (LBF), Google Research Football (GRF), and Multi-Agent Particle Environments (MPE)

Level-Based Foraging The Level-Based Foraging environment consists of tasks focusing on the coordination of involved agents. The task for each agent is to navigate the grid-world map and collect items. Each agent and item is assigned a level and items are randomly scattered in the environment. In order to collect an item, agents have to choose a certain action next to the item. However, such collection is only successful if the sum of involved agents' levels is equal or greater than the item level. Agents receive reward equal to the level of the collected item. The tasks that were selected are denoted first by the grid-world size (e.g. 15×15 means a 15 by 15 grid-world). Then, the number of agents is shown (e.g. "4p" means four agents/players), and the number of items scattered in the grid (e.g. "3f" means three food items). We also have some special flags, the "2s" which denotes partial observability with a range of two squares. In those tasks, agents can only observe items or other agents as long as they are located in a square of size 5x5 centred around them. Finally, the flag "c" means a cooperative-only variant were all items can only be picked up if all agents in the level attempt to load it simultaneously. Since the global state space of the original environment is stitched together from the observation vectors of individual agents, it contains a lot of duplicated information and some environmental information is lost. Therefore, we redesigned a global state space which contains all the environment information while reducing the size of the dimension. Specifically, it includes the coordinates and levels of individual intelligences and things, as well as the actions executed by each agent last step.

Multi-Agent Particle Environments **Simple-Spread:** In this task, three agents are trained to move to three landmarks while avoiding collisions with each other. All agents receive their velocity, position, relative position to all other agents and landmarks. The action space of each agent contains five discrete movement actions. Agents are rewarded with the sum of

[5]https://github.com/uoe-agents/epymarl
[6]https://github.com/hijkzzz/pymarl2
[7]https://github.com/oxwhirl/smac
[8]https://github.com/google-deepmind/pysc2

Table 1: Details of LBF tasks

LBF tasks	n_agents	state shape	action num
Foraging-2s-15x15-4p-3f	4	25	6
Foraging-2s-8x8-2p-2f-coop	2	14	6
Foraging-2s-10x10-3p-3f	3	21	6

negative minimum distances from each landmark to any agent and a additional term is added to punish collisions among agents.
Simple-Adversary: In this task, two cooperating agents compete with a third adversary agent. There are two landmarks out of which one is randomly selected to be the goal landmark. Cooperative agents receive their relative position to the goal as well as relative position to all other agents and landmarks as observations. However, the adversary agent observes all relative positions without receiving information about the goal landmark. All agents have five discrete movement actions. Agents are rewarded with the negative minimum distance to the goal while the cooperative agents are additionally rewarded for the distance of the adversary agent to the goal landmark. Therefore, the cooperative agents have to move to both landmarks to avoid the adversary from identifying which landmark is the goal and reaching it as well. For this competitive scenario, we use a fully cooperative version where the adversary agent is controlled by a pretrained model.
Simple-Tag: In this task, three cooperating predators hunt a forth agent controlling a faster prey. Two landmarks are placed in the environment as obstacles. All agents receive their own velocity and position as well as relative positions to all other landmarks and agents as observations. Predator agents also observe the velocity of the prey. All agents choose among five movement actions. The agent controlling the prey is punished for any collisions with predators as well as for leaving the observable environment area (to prevent it from simply running away without needing to learn to evade). Predator agents are collectively rewarded for collisions with the prey. We employ a fully cooperative version of this task with a pretrained prey agent.

Table 2: Details of MPE tasks

MPE tasks	n_agents	state shape	action num
pz-mpe-simple-spread	6	216	5
pz-mpe-simple-tag	3	48	5
pz-mpe-simple-adversary	2	16	5

Google Research Football The Google Research Football environment is a challenging reinforcement learning platform designed to simulate football matches in a highly realistic and strategic setting. Agents control individual players and must learn to cooperate, plan, and execute complex behaviors to achieve objectives such as scoring goals or defending against opponents. The environment provides various predefined scenarios as well as full 11-versus-11 matches, supporting both single-agent and multi-agent training. In the Google Research Football environment, the action space consists of a discrete set of predefined high-level actions such as moving, passing, shooting, and sprinting, allowing agents to focus on strategic decision-making rather than low-level motor control. The *Simple115* observation space is a fixed-size vector with 115 features, providing essential information about the game state, including the positions and velocities of the ball and players, the directions of movement, and other key attributes necessary for learning effective policies.

Table 3: Details of GRF tasks

GRF tasks	n_agents	state shape	action num
academy_3_vs_1_with_keeper	4	460	19
academy_counterattack_easy	4	460	19
academy_pass_and_shoot_with_keeper	3	345	19

C Additional Experiments

Compatibility Testing of Different LLMs

We tested the performance of QLLM with three mainstream LLMs, Deepseek-R1, Deepseek-V3[9], and ChatGPT-4o[10], along with other algorithms in the MPE environment. As shown in Figure 8, the experimental results indicate that mainstream LLMs outperform the baseline algorithms. Moreover, as the inference capabilities of the LLMs improve, the performance of our algorithm also improves.

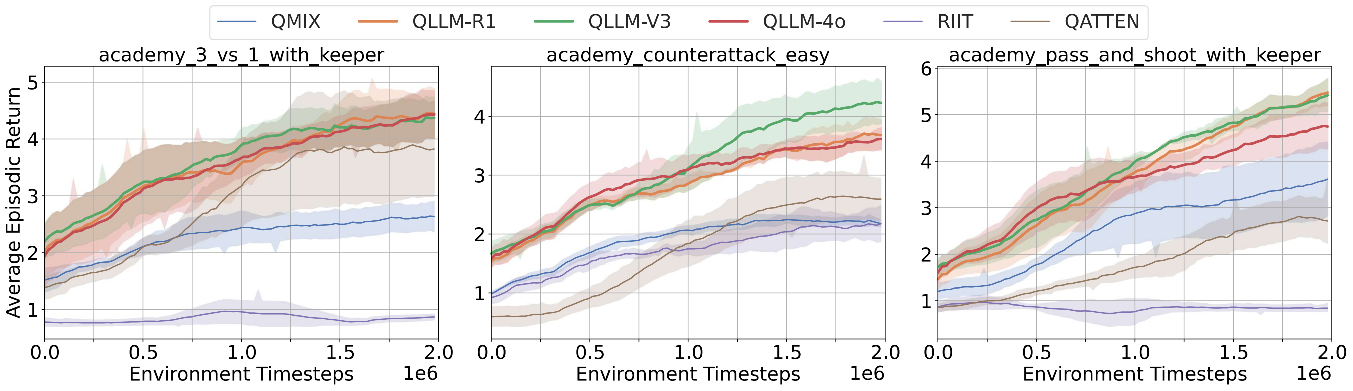

Figure 8: Performance comparison of QLLM using different LLMs in the GRF environment.

Tests of the Effect of the Evaluator LLM

Most reinforcement learning algorithms enhanced by LLMs use only one kind of LLM to do their job directly. Our *coder-evaluator* framework features two LLMs, each with its specific role, working together to generate an accurate credit assignment function. Therefore, we need to explore in depth the effect of the evaluator LLM on experimental performance and investigate how it helps mitigate the issues of hallucination and randomness in LLMs. We tested the performance of QLLM across different reinforcement learning environments without using the evaluator LLM. The experimental results (Figure 9) show that, the evaluator LLM significantly improves the algorithm's performance in most environments.

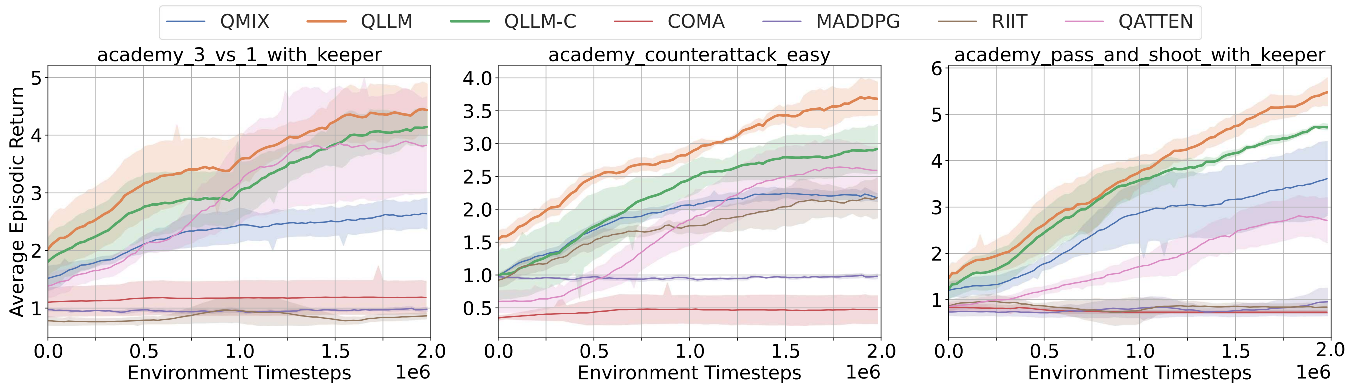

Figure 9: Performance comparison of QLLM without evaluator LLM in the GRF environment, where QLLM-C stands for QLLM that do not use the evaluator LLM

Comparison of Algorithms Training Costs

As mentioned in the main text, QLLM replaces the mixing network with a deterministic functional expression, TFCAF, resulting in a significantly smaller number of learnable parameters. Therefore, the training cost of our algorithm is reduced. Table 4 lists the number of trainable parameters for each algorithm in the environments tested in this paper. Our algorithm reduces the

[9]https://www.deepseek.com/
[10]https://openai.com/

number of trainable parameters by approximately 12% to 37%. This reduction directly lowers the training cost and simplifies the model structure, making the algorithm more efficient and improving performance.

Table 4: Number of learnable parameters (in K) for each environment under different algorithms

Environment / Algorithm	**QLLM**	**QMIX**	**QATTEN**	**MADDPG**	**RIIT**	**QPLEX**	**Baseline Average**	**Reduction Rate (%)**
pz-mpe-simple-spread	**105.221**	140.422	191.114	154.246	191.175	154.762	166.744	36.88%
pz-mpe-simple-tag	**102.277**	112.806	129.034	127.494	132.199	128.010	125.909	18.78%
pz-mpe-simple-adversary	**101.381**	107.270	118.282	122.246	124.359	122.762	118.584	14.52%
Foraging-2s-15x15-4p-3f	**103.174**	111.815	121.963	126.727	130.536	127.372	123.283	16.30%
Foraging-2s-8x8-2p-2f-coop	**101.766**	106.887	116.683	122.119	124.296	122.764	118.150	13.87%
Foraging-2s-10x10-3p-3f	**102.662**	109.735	120.043	124.807	127.912	125.452	121.190	15.28%
climbing-nostate-v0	**100.099**	103.684	110.408	118.148	118.661	118.406	113.861	12.07%
academy_3_vs_1_with_keeper	**430.355**	529.300	601.496	634.900	568.085	543.014	575.359	25.23%
academy_counterattack_easy	**430.355**	529.300	601.496	634.900	568.085	543.014	575.359	25.23%
academy_pass_and_shoot_with_keeper	**430.099**	504.884	560.760	600.084	530.869	523.046	543.129	20.83%