The Intersections of Transformers and Large Language Models with MARL

Anri Lombard

Department of Computer Science

University of Cape Town

LMBANR001@myuct.ac.za

November 15, 2024

Contents

1 Introduction

The field of reinforcement learning (RL) addresses a fundamental challenge in artificial intelligence: how can an agent learn to make effective decisions through interaction with its environment? In reinforcement learning, an agent learns by taking actions, observing their consequences, and receiving rewards, gradually improving its decision-making policy through experience [75]. While this approach has achieved remarkable successes in domains like game playing and robotics, many real-world applications involve multiple agents that must learn and act together, leading to the development of multi-agent reinforcement learning (MARL).

MARL extends reinforcement learning to scenarios where multiple agents interact within a shared environment. These agents must not only learn effective individual policies but also develop sophisticated coordination strategies. Common applications include autonomous vehicles navigating traffic [66], cooperative robots in warehouse management [31], and gaming agents cooperating or competing in strategic scenarios [83]. However, MARL faces significant challenges including non-stationarity (as agents' policies change during learning), exponentially growing state-action spaces, and the need for effective communication between agents.

Recent advances in transformer architectures have opened new possibilities for addressing these MARL challenges. Transformers, initially developed for natural language processing [82], excel at modeling complex relationships in sequential data through their attention mechanisms. This capability proves particularly valuable in MARL, where agents must reason about the temporal evolution of other agents' behaviors and the environment state. For instance, transformer-based architectures enable more flexible value function decomposition in cooperative scenarios [37] and allow for policies that can generalize across varying numbers and types of agents [29].

The emergence of large language models (LLMs) has further expanded the potential of MARL systems. LLMs trained on vast text corpora demonstrate sophisticated reasoning capabilities and can follow complex instructions. When integrated into MARL frameworks, LLMs enhance agent coordination through natural language communication [45], improve task planning capabilities [28], and enable more robust consensus-seeking behaviors [10]. For example, warehouse robots equipped with LLM capabilities can discuss task allocation strategies, negotiate resource conflicts, and adapt their behaviors based on natural language feedback.

This paper provides a comprehensive examination of how transformers and LLMs are reshaping MARL. We begin by establishing the foundational concepts of reinforcement learning, MARL, and transformer architectures. This background provides essential context for understanding how these technologies intersect. We then analyze specific advances in applying transformers to reinforcement learning, including stabilization techniques and the emerging perspective of treating RL as a sequence modeling problem.

The core of our analysis focuses on three key areas where transformers and LLMs have significantly impacted MARL. First, we examine transformer-based architectures for multi-agent systems, analyzing approaches like transformer-based value function decomposition and universal policy decoupling. Second, we investigate how LLMs serve as knowledge sources for RL agents, enhancing their decision-making capabilities. Third, we explore frameworks for multi-agent collaboration using LLMs, including both theoretical approaches and practical implementations in robotics.

Throughout this examination, we maintain a critical perspective, analyzing both the advantages and limitations of different approaches. We support our analysis with empirical results from standard benchmarks and real-world applications, providing concrete evidence of where these technologies excel and where challenges remain. This systematic review aims to equip readers with a clear understanding of the current state of research at the intersection of transformers, LLMs, and MARL, while highlighting promising directions for future investigation.

2 Foundations

2.1 Reinforcement Learning Basics

Reinforcement Learning (RL) represents a paradigm in machine learning where an agent learns to make decisions through interactions with an environment. This section introduces the fundamental concepts and terminology of RL, providing a solid foundation for understanding more advanced topics in multi-agent systems and the application of transformers to RL.

2.1.1 Key Concepts and Terminology

At the core of RL is the concept of an agent interacting with an environment. The agent, serving as the learner and decision-maker, operates within the environment, which encompasses everything the agent interacts with [75]. This interaction is typically modeled as a discrete-time process. At each time step t, the agent receives a representation of the environment's state S_t, takes an action A_t, and subsequently receives a scalar reward R_{t+1}.

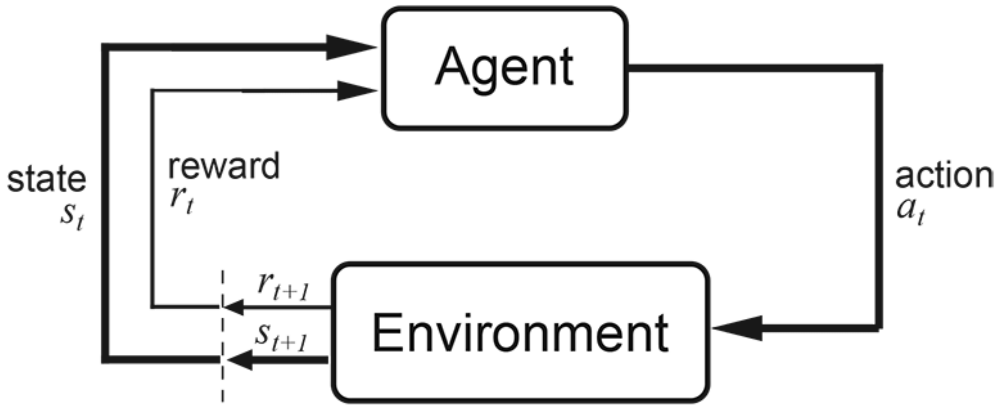


Figure 2.1: The agent–environment interaction in reinforcement learning [75].

Figure 2.1 illustrates this fundamental interaction loop in reinforcement learning. The agent observes the state of the environment, takes an action based on its policy, and receives a reward and a new state as a result of that action. This process continues iteratively as the agent learns to improve its policy over time.

To concretely illustrate these concepts, let's consider a simple Gridworld environment in figure 2.2. In this 4x4 Gridworld (left), each cell represents a state. The agent can move up, down, left, or right, corresponding to the available actions. The green cell (G) represents a goal state with a high reward, while red cells (X) are obstacles with negative rewards. This environment provides a tangible example of states, actions, and rewards in RL.

The RL problem is often formalized using Markov Decision Processes (MDPs), which provide a mathematical framework for modeling decision-making in situations where outcomes are partly random

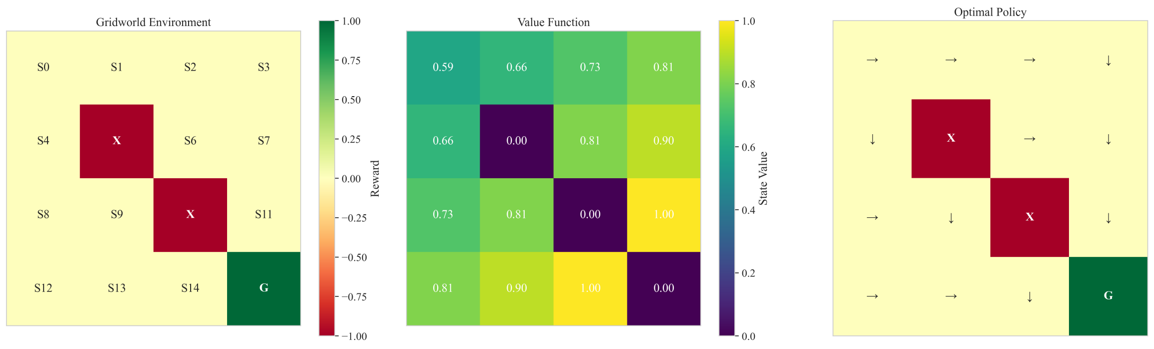


Figure 2.2: Gridworld environment, value function, and optimal policy visualization.

and partly under the control of the decision-maker [4]. An MDP is defined by a tuple (S, A, P, R, γ), where S represents the set of states, A the set of actions, P the state transition probability function, R the reward function, and γ the discount factor, with $\gamma \in [0, 1]$.

The state transition probability function P is crucial in defining the dynamics of the environment. It is expressed as:

$$P(s'|s,a) = \Pr(S_{t+1} = s'|S_t = s, A_t = a) \tag{2.1}$$

This equation represents the probability of transitioning to state s' given that the agent is in state s and takes action a. In our Gridworld example, these transitions are deterministic: moving right from S0 always leads to S1, unless blocked by an obstacle or the grid's edge.

The reward function R is another critical component of the MDP framework. It defines the immediate, scalar feedback that the agent receives after each action. Formally, it can be expressed as:

$$R(s,a,s') = \mathbb{E}[R_{t+1}|S_t = s, A_t = a, S_{t+1} = s'] \tag{2.2}$$

In Gridworld, this might be represented as a high positive reward for reaching the goal state (G), negative rewards for obstacle states (X), and small negative rewards for other states to encourage efficient paths.

The primary objective of an RL agent is to maximize the cumulative reward over time, often expressed as the expected return:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.3}$$

Here, γ serves as the discount factor, balancing the importance of immediate and future rewards. The agent's behavior is defined by its policy π, a mapping from states to probabilities of selecting each possible action. If the agent is following policy π at time t, then $\pi(a|s)$ represents the probability that $A_t = a$ if $S_t = s$. In the Gridworld example, the optimal policy is visualized in the rightmost grid of Figure 2.2, where arrows indicate the best action in each state.

Two crucial value functions in RL are the state-value function and the action-value function. The state-value function $V_\pi(s)$ represents the expected return when starting in state s and following policy π thereafter:

$$V_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s\right] \tag{2.4}$$

The middle grid in Figure 2.2 visualizes this concept, showing the value of each state under the optimal policy. Higher values (brighter colors) indicate states that are expected to yield higher cumulative rewards.

The action-value function $Q_\pi(s, a)$, on the other hand, represents the expected return starting from state s, taking action a, and thereafter following policy π:

$$Q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a\right] \tag{2.5}$$

A fundamental concept in RL is the Bellman equation, which expresses the relationship between the value of a state and the values of its successor states. For the state-value function, the Bellman equation is:

$$V_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma V_\pi(s')] \tag{2.6}$$

And for the action-value function:

$$Q_\pi(s,a) = \sum_{s',r} p(s', r|s, a)[r + \gamma \sum_{a'} \pi(a'|s')Q_\pi(s', a')] \tag{2.7}$$

These equations form the basis for many RL algorithms, as they allow for iterative improvement of value estimates [6]. In our Gridworld example, these equations would be used to compute the values shown in the middle grid of Figure 2.2.

A key challenge in RL is balancing exploration (trying new actions to potentially find better strategies) and exploitation (using known good strategies). This exploration-exploitation dilemma is a fundamental issue in RL and has been extensively studied [79]. Common approaches to address this include ϵ-greedy methods, softmax exploration, and Upper Confidence Bound (UCB) algorithms [2].

Algorithm 1 ϵ-greedy Exploration Strategy – Balances exploration and exploitation in reinforcement learning

```
Require: $\epsilon \in [0, 1]$, $Q(s, a)$ for all $s \in S$, $a \in A$   ▷ $\epsilon$ controls exploration rate, $Q(s, a)$ stores action values
1: function ChooseAction($s$)   ▷ Takes current state $s$ as input
2:     $p \leftarrow$ UniformRandom$(0, 1)$   ▷ Generate random number to decide between exploration and exploitation
3:     if $p < \epsilon$ then   ▷ With probability $\epsilon$, explore randomly
4:         return RandomAction$(A)$   ▷ Choose a random action from action space $A$ for exploration
5:     else   ▷ With probability $1 - \epsilon$, exploit current knowledge
6:         return $\arg\max_a Q(s, a)$   ▷ Choose action with highest estimated value in current state
7:     end if
8: end function
```

The ϵ-greedy strategy, as shown in Algorithm 1, provides a simple yet effective approach to balancing exploration and exploitation. With probability ϵ, the agent chooses a random action (exploration), and with probability 1-ϵ, it chooses the action with the highest estimated value (exploitation). This method ensures that the agent continues to explore the environment, potentially discovering better strategies, while also taking advantage of its current knowledge.

In our Gridworld example, an ϵ-greedy strategy would allow the agent to occasionally take random actions, helping it discover the high reward of the goal state or learn to avoid obstacle states, while generally following the optimal policy shown in the rightmost grid of Figure 2.2.

These concepts form the foundation of reinforcement learning and are essential for understanding more advanced topics such as function approximation [81], policy gradient methods [76], and the integration of deep learning techniques with RL, leading to the field of deep reinforcement learning [50].

As we progress to later sections, these fundamental ideas will be crucial in exploring multi-agent reinforcement learning systems and the application of transformer architectures to RL problems. The interplay between these basic concepts and more advanced techniques continues to drive innovation in the field of reinforcement learning.

2.1.2 Classic RL Algorithms

The field of reinforcement learning (RL) has been shaped by several seminal algorithms that have laid the foundation for modern RL techniques. This section explores these classic algorithms, their underlying principles, and their significance in the development of RL.

Dynamic Programming

Dynamic Programming (DP) methods [3] represent foundational approaches to solving reinforcement learning problems, particularly in scenarios where the environment dynamics, including transition probabilities and reward functions, are known a priori. While their computational complexity often precludes their direct application to large-scale reinforcement learning problems, DP methods provide critical theoretical underpinnings that inform many modern algorithms.

At the heart of dynamic programming lie two fundamental algorithmic approaches: policy iteration and value iteration. Policy iteration employs an alternating optimization strategy, iteratively evaluating the value function for the current policy before improving that policy based on the computed values. This process continues until convergence, with each iteration refining both the value estimates and the resulting policy. In contrast, value iteration takes a more direct approach by computing the optimal value function through repeated application of the Bellman optimality equation, from which an optimal policy can be derived [59].

The mathematics underlying policy iteration reveals its elegant structure. The algorithm alternates between policy evaluation:

$$V^{\pi_k}(s) = \sum_{s'} P(s'|s, \pi_k(s))[R(s, \pi_k(s), s') + \gamma V^{\pi_k}(s')] \tag{2.8}$$

and policy improvement:

$$\pi_{k+1}(s) =_a \sum_{s'} P(s'|s, a)[R(s, a, s') + \gamma V^{\pi_k}(s')] \tag{2.9}$$

where V^{π_k} represents the value function under policy π_k, and γ denotes the discount factor. This process continues until the policy stabilizes, at which point the algorithm has converged to an optimal solution.

Algorithm 2 formalizes this iterative process, demonstrating how policy evaluation and improvement phases interleave to progressively refine the solution. While both policy iteration and value iteration converge to optimal policies, they exhibit different computational trade-offs and generate distinct sequences of intermediate policies during optimization [59]. Understanding these trade-offs proves crucial when applying dynamic programming principles to practical reinforcement learning problems.

Algorithm 2 Policy Iteration – Iteratively computes optimal policy through alternating evaluation and improvement phases

```
 1: Initialize $\pi_0$ arbitrarily    ▷ Start with any valid policy mapping states to actions
 2: for $k = 0, 1, 2, \dots$ do    ▷ Main iteration loop
 3:     // Policy Evaluation Phase: Compute value function for current policy $\pi_k$
 4:     repeat
 5:         for each $s \in \mathcal{S}$ do    ▷ Update values for all states
 6:             $v_{k+1}(s) \leftarrow \sum_{s'} P(s'|s, \pi_k(s))[R(s, \pi_k(s), s') + \gamma v_k(s')]$ ▷ Bellman update: sum over next states ($s'$), using transition probability $P$, immediate reward $R$, and discounted future value $\gamma v_k$
 7:         end for
 8:     until $v_k$ converges    ▷ Continue until value estimates stabilize Copy
 9:     // Policy Improvement Phase: Find better policy using updated values
10:     policy_stable ← true    ▷ Flag to check if policy has converged
11:     for each $s \in \mathcal{S}$ do    ▷ Update policy for all states
12:         old_action ← $\pi_k(s)$    ▷ Store current action for comparison
13:         $\pi_{k+1}(s) \leftarrow_a \sum_{s'} P(s'|s, a)[R(s, a, s') + \gamma v_k(s')]$  ▷ Select action maximizing expected value using current value estimates
14:         if old_action $\neq \pi_{k+1}(s)$ then    ▷ Check if policy changed
15:             policy_stable ← false
16:         end if
17:     end for
18:     if policy_stable then    ▷ If no changes in policy, we've found optimal solution
19:         return $v_k, \pi_k$    ▷ Return converged value function and optimal policy
20:     end if
21: end for
```

Monte Carlo Methods

Monte Carlo (MC) methods learn from complete episodes of experience, without requiring knowledge of the environment dynamics [75]. These methods are particularly useful in episodic tasks and can learn directly from interaction with the environment.

The key idea in MC methods is to estimate the value of a state by averaging the returns observed after visits to that state. This approach can be used for both prediction (evaluating a given policy) and control (finding an optimal policy).

One of the advantages of MC methods is their ability to focus on relevant parts of the state space, as they only update values for states that are actually visited. However, they can be slow to converge and are not suitable for continuing (non-episodic) tasks [72].

The Monte Carlo estimation process can be expressed as:

$$V(s) \approx \frac{1}{N} \sum_{i=1}^{N} G_i(s) \tag{2.10}$$

Where $V(s)$ is the estimated value of state s, N is the number of visits to state s, and $G_i(s)$ is the return from the i-th visit to state s.

Figure 2.3 illustrates the iterative process of policy iteration, where the algorithm alternates between policy evaluation and policy improvement until convergence.

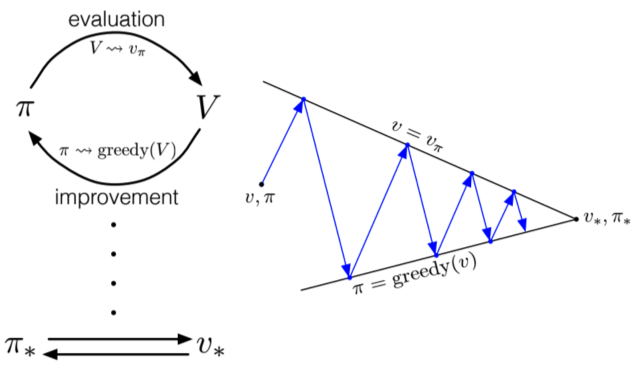


Figure 2.3: Policy iteration process. Adapted from Sutton and Barto [75].

Temporal Difference Learning

Temporal Difference (TD) learning, introduced by Sutton [74], combines ideas from DP and MC methods. Like MC methods, TD learning can learn directly from experience without a model of the environment. However, TD methods update estimates based on other learned estimates, without waiting for a final outcome (bootstrapping).

The simplest TD prediction algorithm, known as TD(0), updates the value function as follows:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{2.11}$$

where α is the learning rate and γ is the discount factor.

TD methods generally converge faster than MC methods and can be applied to continuing tasks. However, they can be more sensitive to initial value estimates [75]. Table 2.1 provides a comparison between TD and MC methods.

Aspect	Temporal Difference	Monte Carlo
Update Frequency	Every step	End of episode
Bias	Biased (due to bootstrapping)	Unbiased
Variance	Lower variance	Higher variance
Online Learning	Suitable	Not suitable
Continuing Tasks	Applicable	Not applicable
Convergence Speed	Faster	Slower

Table 2.1: Comparison of Temporal Difference and Monte Carlo methods

Q-Learning

Q-learning, introduced by Watkins and Dayan [84], represents a fundamental advancement in reinforcement learning through its ability to learn optimal behavior while following an exploratory policy. This off-policy temporal difference control algorithm directly learns the optimal action-value function $Q^*(s, a)$, which represents the expected cumulative reward when taking action a in state s and following the optimal policy thereafter.

The Q-learning update rule encapsulates this learning process:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{2.12}$$

This equation merits careful examination. The term $R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$ represents the target value, combining the immediate reward R_{t+1} with the discounted estimate of optimal future value $\gamma \max_a Q(S_{t+1}, a)$. The learning rate α controls how much the current estimate is updated based on the difference between this target and the current estimate $Q(S_t, A_t)$. Crucially, by taking the maximum over all possible actions in the next state ($\max_a Q(S_{t+1}, a)$), Q-learning learns about the optimal policy regardless of the actions actually taken during exploration.

The complete Q-learning algorithm implements this update rule within an episodic learning framework:

Algorithm 3 Q-Learning – Learning optimal action-values through off-policy updates

```
 1: Initialize $Q(s, a)$ arbitrarily                          ▷ Initialize action-value estimates
 2: for each episode do                                       ▷ Loop through training episodes
 3:     Initialize $S$                                        ▷ Set initial state for episode
 4:     for each step of episode do                           ▷ Continue until episode termination
 5:         Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)   ▷ Select action balancing exploration and exploitation
 6:         Take action $A$, observe reward $R$ and next state $S'$
 7:         $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$   ▷ Update Q-value using observed transition
 8:         $S \leftarrow S'$                                 ▷ Transition to next state
 9:     end for
10: end for
```

SARSA: On-Policy Learning

SARSA (State-Action-Reward-State-Action), introduced by Rummery and Niranjan [63], provides an alternative approach to Q-learning through on-policy learning. Unlike Q-learning, SARSA learns the value of the actual policy being followed, including exploratory actions. This characteristic leads to fundamentally different learning dynamics and policy behaviors.

The SARSA update rule reveals its on-policy nature:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{2.13}$$

The key distinction from Q-learning lies in how future value is estimated. Instead of using the maximum Q-value in the next state ($\max_a Q(S_{t+1}, a)$), SARSA uses the Q-value of the actual next action A_{t+1} that will be taken ($Q(S_{t+1}, A_{t+1})$). This means SARSA learns values that reflect the outcomes of the exploration policy being followed, leading to more conservative behavior in scenarios where exploration carries risk [75].

Table 2.2 summarizes these algorithmic differences and their implications. These algorithmic differences make each method suitable for different scenarios. Q-learning's off-policy nature makes it particularly valuable for learning from stored experience (replay buffers) and forms the foundation for modern deep reinforcement learning algorithms like DQN [50]. SARSA's on-policy updates make it more appropriate for scenarios where safe exploration is crucial, as it learns values that reflect the risks inherent in the exploration process.

Characteristic	Q-Learning	SARSA
Policy Type	Off-policy: Learns optimal policy independent of exploration	On-policy: Learns policy that includes exploration behavior
Value Updates	Uses $\max_a Q(S', a)$: Assumes optimal future actions	Uses $Q(S', A')$: Accounts for actual exploratory actions
Risk Handling	May learn policies that are optimal but risky during learning	Learns policies that are safer during the learning process
Convergence Properties	Converges to optimal policy with sufficient exploration	Convergence depends on exploration schedule and policy

Table 2.2: Comparative analysis of Q-Learning and SARSA algorithms

Policy Gradient Methods

Policy gradient methods embody a fundamentally different approach to reinforcement learning by directly optimizing the policy parameters without explicitly maintaining value function estimates. First introduced through the REINFORCE algorithm by Williams [87], these methods have evolved into sophisticated approaches that underpin many modern reinforcement learning systems.

At their core, policy gradient methods operate by adjusting policy parameters θ to maximize the expected return $J(\theta)$. This objective leads to the foundational update rule:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta) \tag{2.14}$$

where α represents the learning rate. The gradient $\nabla_\theta J(\theta)$ captures how small changes in policy parameters affect the expected return, providing a direction for policy improvement. This direct parameter optimization stands in contrast to value-based methods, offering particular advantages in continuous action spaces and stochastic policies.

The REINFORCE algorithm implements this concept through a theoretically elegant Monte Carlo approach:

Algorithm 4 REINFORCE – Monte Carlo Policy Gradient

```
1: Initialize policy parameters $\theta$ arbitrarily                ▷ Begin with random policy
2: for each episode $\{s_t, a_t, r_t\}_{t=0}^{T} \sim \pi_\theta$ do          ▷ Sample trajectory
3:     for $t = 0$ to $T$ do
4:         $G_t \leftarrow \sum_{k=t}^{T} \gamma^{k-t} r_k$                  ▷ Compute discounted return
5:     end for
6:     for $t = 0$ to $T$ do
7:         $\theta \leftarrow \theta + \alpha G_t \nabla_\theta \log \pi_\theta(a_t|s_t)$   ▷ Update policy using likelihood ratio gradient
8:     end for
9: end for
```

The algorithm's elegant simplicity belies its theoretical significance. The term $\nabla_\theta \log \pi_\theta(a_t|s_t)$ represents the gradient of the log probability of taking action a_t in state s_t, while G_t serves as an unbiased estimate of the expected return. This formulation provides unbiased gradient estimates but often exhibits high variance, necessitating substantial sample sizes for reliable learning [75].

Proximal Policy Optimization (PPO)

Building upon these foundations, Proximal Policy Optimization (PPO) [65] introduces crucial innovations to enhance learning stability and efficiency. PPO's primary contribution lies in its carefully constructed objective function that prevents destructively large policy updates while maintaining the benefits of policy gradient methods.

The PPO objective employs a clipped surrogate function:

$$L^{CLIP}(\theta) = \mathbb{E}_t[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)] \tag{2.15}$$

Here, $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ measures the ratio between new and old policy probabilities. When this ratio strays too far from 1 (controlled by ϵ), the clipping function curtails the objective, effectively constraining the policy update magnitude. The advantage estimate $\hat{A}_t$ provides a learned baseline for variance reduction, improving upon REINFORCE's pure Monte Carlo returns.

The complete PPO algorithm orchestrates these components into a robust learning procedure:

Algorithm 5 Proximal Policy Optimization (PPO) – Stable Policy Learning through Conservative Updates

```
 1: Initialize policy ($\theta_0$) and value function ($\phi$) parameters    ▷ Prepare neural networks
 2: Set hyperparameters: learning rate $\alpha$, clip ratio $\epsilon$, epochs $K$, minibatch size $M$
 3: for iteration = 1, 2, ... do                                             ▷ Main training loop
 4:     Collect trajectories $\mathcal{D}_i = \{\tau_j\}$ using $\pi_{\theta_{i-1}}$    ▷ Environment interaction
 5:     Compute rewards-to-go $\hat{R}_t$                                    ▷ Monte Carlo returns
 6:     Compute advantages $\hat{A}_t$ using GAE-$\lambda$                   ▷ Advantage estimation
 7:     for epoch = 1, 2, ..., K do                                          ▷ Multiple passes over data
 8:         Shuffle $\mathcal{D}_i$ into $M$ minibatches                     ▷ Randomize learning order
 9:         for each minibatch $\mathcal{B}$ do                              ▷ Process data in small batches
10:             $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{i-1}}(a_t|s_t)}$    ▷ Policy probability ratio
11:             $L^{CLIP}(\theta) = \mathbb{E}_t[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)]$
12:             $L^{VF}(\phi) = \mathbb{E}_t[(V_\phi(s_t) - \hat{R}_t)^2]$   ▷ Value function loss
13:             $S[\pi_\theta](s_t)$                                         ▷ Policy entropy for exploration
14:             $L(\theta, \phi) = -L^{CLIP}(\theta) + c_1 L^{VF}(\phi) - c_2 S[\pi_\theta](s_t)$    ▷ Combined loss
15:             Update $\theta$, $\phi$ using optimizer                      ▷ Gradient-based optimization
16:         end for
17:     end for
18:     $\theta_i \leftarrow \theta$                                         ▷ Store updated policy
19: end for
```

PPO's effectiveness stems from several key design choices. The multiple epochs of updates (K) enable efficient use of collected data, while minibatching provides stable gradient estimates. The clipped objective ensures policy updates remain conservative, preventing the performance collapses that often plague other policy gradient methods. Additionally, the inclusion of a value function baseline and entropy bonus helps balance exploitation with exploration [65].

These innovations have established PPO as a remarkably robust algorithm, demonstrating strong performance across diverse domains from robotics to game playing [5]. Its success has inspired various extensions, such as PPO with Adaptive KL Penalty for dynamic constraint adjustment and Recurrent PPO for partially observable environments [93]. The algorithm's combination of theoretical soundness, implementation simplicity, and empirical effectiveness has made it a cornerstone of modern reinforcement learning research and applications.

2.2 Multi-Agent Reinforcement Learning (MARL)

Multi-Agent Reinforcement Learning (MARL) studies how multiple agents can learn effective decision-making policies through interaction with a shared environment and each other. Unlike single-agent reinforcement learning, where one agent learns to maximize its own rewards, MARL must address the interactions between multiple learning agents whose actions affect each other's rewards and environment dynamics. This creates several technical challenges - the environment becomes non-stationary from each agent's perspective, the joint state-action space grows exponentially with the number of agents, and agents must learn to coordinate or compete effectively. MARL has proven particularly useful for developing autonomous driving systems [66], coordinated robot teams [31], and game-playing agents [83].

2.2.1 From Single-Agent to Multi-Agent Systems

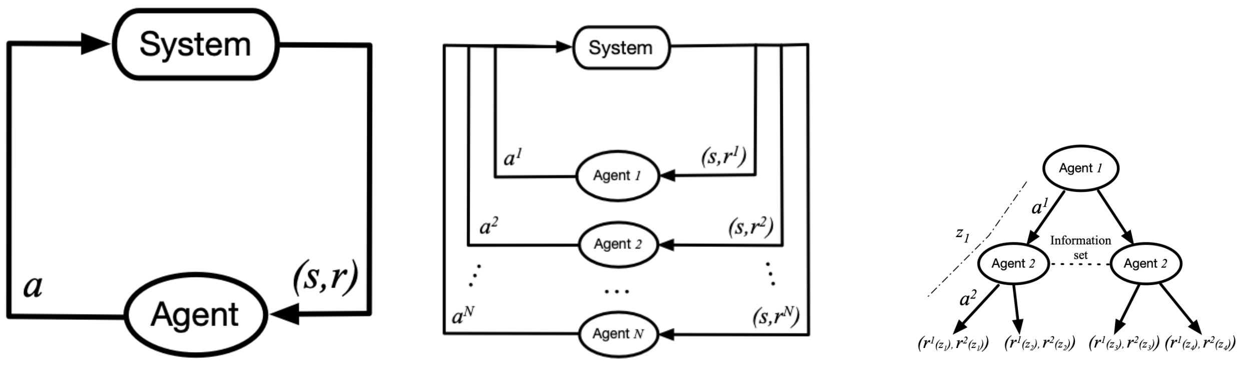
Multi-agent reinforcement learning (MARL) extends single-agent methods to handle scenarios where multiple agents learn simultaneously. Real-world applications include autonomous vehicles coordinating in traffic [66] and agents competing in strategic games [7]. The core objective in MARL is to develop agents that can learn effective policies while interacting with other learning agents.

Single-agent reinforcement learning assumes a stationary environment with fixed transition probabilities and reward functions. Adding multiple learning agents changes this assumption, as shown by Hernandez-Leal et al. [26]. When multiple agents learn concurrently, each agent's policy updates alter the environment dynamics for all other agents. This non-stationarity appears in the joint state-action value function:

$$Q_i^{\pi_1,\dots,\pi_N}(s, a_1, \dots, a_N) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r_i^t | s_0 = s, \{\pi_j\}_{j=1}^N] \tag{2.16}$$

where agent i's value depends on the changing policies π_j of all agents.

The computational requirements grow exponentially with the number of agents [8]. With N agents, each having state space S and action space A, the joint space becomes $S^N \times A^N$. This exponential scaling makes direct applications of single-agent methods impractical for many multi-agent problems.



(a) Markov decision process (b) Markov game (c) Extensive-form game

Figure 2.4: System evolution diagrams showing: (a) single-agent RL as a Markov decision process, where one agent interacts with a stationary environment, (b) simultaneous multi-agent RL as a Markov game, where agents act concurrently, and (c) sequential multi-agent RL as an extensive-form game, where agents act in order with access to previous actions **[96]**.

Figure 2.4 shows three key frameworks in reinforcement learning. The Markov Decision Process

(Figure 2.4a) models single-agent learning with direct environment interaction. The Markov game (Figure 2.4b) extends this to multiple agents acting simultaneously. The extensive-form game (Figure 2.4c) adds sequential dependencies between agent actions.

Oliehoek et al. [52] analyze how partial observability affects multi-agent systems. Agents typically cannot observe the complete state of the system, requiring them to act based on limited local information. This creates a need for effective communication protocols and methods to handle uncertainty.

In cooperative settings, determining each agent's contribution to team success poses technical challenges. Foerster et al. [20] address this through counterfactual multi-agent policy gradients, providing a way to estimate individual agent contributions to overall performance.

Competitive settings require different optimization criteria than single-agent learning. As analyzed by Shoham and Leyton-Brown [69], agents must find stable strategies rather than simply maximizing individual rewards. This changes the learning problem from policy improvement to finding Nash equilibria or other game-theoretic solution concepts.

Claus and Boutilier [16] describe two main approaches to multi-agent learning: independent learners and joint action learners. Independent learners scale well but struggle with convergence in non-stationary environments. Joint action learners model other agents explicitly but become computationally expensive as the number of agents increases.

Deep learning methods have improved how MARL handles complex state and action spaces. Vinyals et al. [83] show how deep neural networks can process high-dimensional inputs in multi-agent settings. Iqbal and Sha [32] demonstrate attention mechanisms for modeling agent interactions, while Jiang et al. [34] use graph neural networks to capture relationships between agents.

These technical advances have enabled new applications. Multi-agent systems can now coordinate effectively in tasks like multiplayer games [5] and robotic control [31]. Morihiro et al. [51] show how groups of agents can learn behaviors more sophisticated than individual agents could achieve alone.

2.2.2 Cooperation and Competition in MARL

Multi-agent reinforcement learning (MARL) systems can involve agents working together, competing against each other, or a mix of both. Consider a team of warehouse robots coordinating to move packages efficiently - this represents cooperation. In contrast, autonomous trading agents competing in a market represent a competitive scenario. Most real-world applications, like autonomous vehicles navigating traffic, involve both cooperation (avoiding collisions) and competition (reaching destinations quickly) [46].

Game theory provides the mathematical tools to analyze these interactions [68]. In cooperative settings, agents work to maximize a shared team objective. We can write this mathematically as finding the best joint policy π that maximizes the expected sum of rewards:

$$V^*(s) = \max_{\pi} \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_t | s_0 = s, \pi\right] \tag{2.17}$$

To understand this equation concretely, consider a two-robot warehouse scenario. The state s might represent the positions of robots and packages, while R_t represents the number of packages successfully delivered at time t. The discount factor γ (typically 0.9 to 0.99) makes future rewards worth less than immediate ones, encouraging efficient delivery. The joint policy π specifies what actions each robot should take in any given state - for example, which package to pick up or where to move next. The equation finds the policy that leads to the highest expected number of deliveries over time.

In competitive settings, agents try to find strategies that work well regardless of what other agents do. This leads to the concept of Nash equilibrium - a set of strategies where no agent can benefit by changing their strategy alone. For a two-player competitive game, this can be written as:

$$\max_{\pi_1} \min_{\pi_2} V^{\pi_1,\pi_2}(s) = \min_{\pi_2} \max_{\pi_1} V^{\pi_1,\pi_2}(s) \tag{2.18}$$

Consider a simplified autonomous driving scenario where two vehicles approach an intersection. Each vehicle's policy π_i determines whether to slow down or maintain speed. The value function $V^{\pi_1,\pi_2}(s)$ might represent the trade-off between travel time and safety. The equation finds policies where neither vehicle can improve its outcome by changing strategy alone - if one vehicle chooses to slow down, the other's best response might be to maintain speed, and vice versa.

Cooperative MARL faces technical challenges that become clear through mathematical analysis. The credit assignment problem arises because the team reward R_t doesn't directly indicate individual contributions. In the warehouse example, if ten packages are delivered by five robots, determining each robot's contribution requires additional mechanisms. The QMIX algorithm [61] addresses this by learning a decomposition of the team value function:

$$Q_{tot}(s, \mathbf{a}) = f(Q_1(s, a_1), ..., Q_n(s, a_n)) \tag{2.19}$$

where f is a mixing function that ensures individual agent values Q_i combine monotonically to the team value Q_{tot}. This allows each agent to learn its contribution while maintaining coordinated behavior.

The CTDE paradigm, implemented in methods like MAVEN [48], enables agents to share information during training while acting independently during deployment. These methods have enabled successful applications in complex tasks like controlling multiple units in StarCraft [64] and coordinating traffic lights [15].

In competitive settings, algorithms extend traditional Q-learning to handle adversarial scenarios. Minimax Q-learning [44] modifies the standard Q-learning update to account for opposing agents:

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha(r + \gamma \max_{a'} \min_{o'} Q(s', a', o')) \tag{2.20}$$

where a and o represent the agent's and opponent's actions respectively. This update rule allows agents to learn robust strategies by assuming opponents will choose actions that minimize the agent's value.

Aspect	Cooperative MARL	Competitive MARL
Objective	Maximize team reward	Maximize individual reward
Key Challenges	Credit assignment, scalability	Finding equilibria, non-stationarity
Common Approaches	Value decomposition, CTDE	Minimax Q-learning, PSRO
Evaluation Metrics	Team performance	Individual performance, exploitability
Example Applications	Traffic control, robot swarms	Poker, strategy games

Table 2.3: Key differences between cooperative and competitive MARL approaches

Methods like AWESOME [17] and LOLA [21] extend this idea by explicitly modeling opponent strategies. These approaches have achieved impressive results in complex games like poker [7] and StarCraft [83]. The MADDPG algorithm [46] handles mixed cooperative-competitive scenarios by allowing agents to maintain separate policies for cooperation and competition.

Current challenges in MARL stem from computational complexity. The joint action space grows exponentially with the number of agents - for n agents each with m possible actions, the space becomes m^n [27]. This scaling makes many current approaches impractical for large-scale applications. Additionally, partial observability in real environments [52] and the need for transfer learning across tasks [71] remain active areas of research.

2.3 Transformers and Sequence Models

The emergence of transformer models in 2017 marked a significant advancement in machine learning, fundamentally changing how we process sequential data [82]. These models introduced a novel approach to handling sequences by focusing entirely on attention mechanisms, moving away from the traditional recurrent architectures that had dominated the field. This shift proved revolutionary, as transformers could process entire sequences simultaneously rather than step-by-step, leading to more efficient training and better handling of long-range dependencies.

2.3.1 Architecture and Attention Mechanism

At its core, the transformer architecture introduces a mechanism called self-attention, which allows the model to weigh the importance of different elements in a sequence when processing each component. This approach differs fundamentally from previous methods that relied on recurrent neural networks (RNNs) or long short-term memory networks (LSTMs), which processed sequences one element at a time. To understand this mechanism, consider how humans read text: when interpreting a word, we naturally pay attention to other relevant words in the sentence, regardless of their distance from the current word. The self-attention mechanism mathematically formalizes this intuitive process.

The basic attention computation takes three inputs: queries, keys, and values, all represented as vectors. For a given query $\mathbf{q}$, the mechanism computes its compatibility with a set of keys $\mathbf{K}$, using these compatibility scores to create a weighted sum of the values $\mathbf{V}$. This process is captured in the attention equation:

$$\text{Attention}(\mathbf{q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \tag{2.21}$$

Here, d_k represents the dimension of the keys, and the scaling factor $\frac{1}{\sqrt{d_k}}$ prevents the dot products from growing too large in magnitude, which could lead to extremely small gradients during training.

The transformer architecture extends this basic attention mechanism through multi-head attention, where multiple attention functions operate in parallel. This parallel processing allows the model to capture different types of relationships within the same sequence. Mathematically, multi-head attention combines several attention outputs:

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\text{head}_1, ..., \text{head}_h)\mathbf{W}^O \tag{2.22}$$

Each head processes the input differently:

$$\text{head}_i = \text{Attention}(\mathbf{Q}\mathbf{W}_i^Q, \mathbf{K}\mathbf{W}_i^K, \mathbf{V}\mathbf{W}_i^V) \tag{2.23}$$

The matrices $\mathbf{W}_i^Q$, $\mathbf{W}_i^K$, $\mathbf{W}_i^V$, and $\mathbf{W}^O$ are learned during training, allowing each attention head to specialize in detecting different types of patterns.

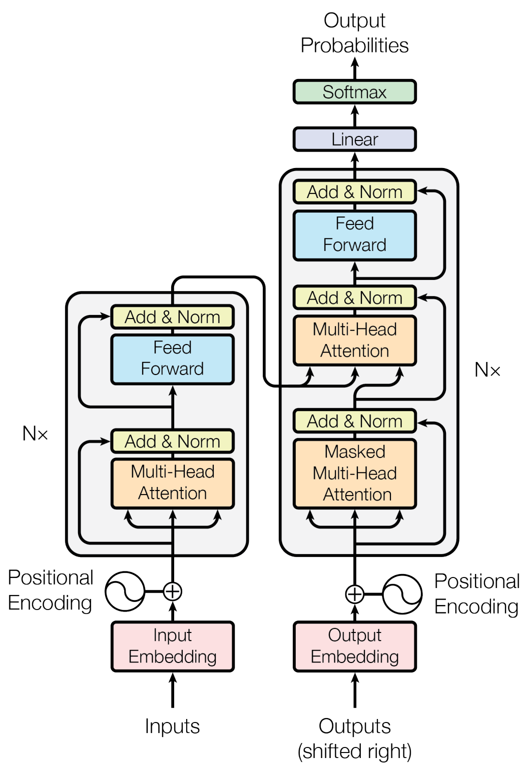


Figure 2.5: The Transformer architecture, showing the encoder-decoder structure with multi-head attention mechanisms. The encoder processes the input sequence, while the decoder generates the output sequence **[82]**.

As shown in Figure 2.5, the complete transformer architecture consists of an encoder and decoder, each containing multiple layers. The encoder processes the input sequence through alternating self-attention and feed-forward layers, while the decoder generates outputs using both self-attention and attention over the encoder's output. This structure enables the model to capture complex relationships within the input sequence while generating contextually appropriate outputs.

2.3.2 Applications and Impact

The transformer architecture has proven remarkably versatile, finding applications far beyond its original purpose in machine translation. In natural language processing, models like BERT [18] and GPT [60] have demonstrated unprecedented performance in understanding and generating human language. BERT's bidirectional approach to language understanding has particularly revolutionized how machines process text, enabling more nuanced comprehension of context and meaning.

The architecture's success has sparked innovation across numerous scientific domains. In computer vision, the Vision Transformer (ViT) [19] demonstrated that images could be effectively processed by treating them as sequences of patches, challenging the long-held assumption that convolutional neural networks were essential for image processing. This approach has led to significant improvements in tasks such as object detection and image segmentation.

The impact of transformers extends into the biological sciences, where models like AlphaFold 2 [35] have achieved breakthrough results in protein structure prediction. By adapting the attention mechanism to process amino acid sequences, these models have solved one of biology's grand challenges, demonstrating the architecture's ability to capture complex spatial relationships in molecular structures.

In reinforcement learning, transformers have enabled new approaches to decision-making problems. The Decision Transformer [11] reconceptualizes reinforcement learning as a sequence modeling task, allowing for more effective learning from historical data. Similarly, the Trajectory Transformer [33] applies sequence modeling principles to planning and control problems, demonstrating the architecture's utility in sequential decision-making contexts.

Despite these successes, transformers face significant challenges. The computational complexity of self-attention scales quadratically with sequence length, limiting the model's ability to process very long sequences. Research efforts like the Reformer [38] and Performer [13] architectures address this limitation by proposing more efficient attention mechanisms, though the fundamental challenge of balancing computational efficiency with model capacity remains an active area of research.

As transformer architectures continue to evolve, their application to reinforcement learning, particularly in multi-agent settings, represents a promising frontier. The ability to model complex dependencies and process multiple streams of information simultaneously makes transformers particularly well-suited for coordinating multiple agents. The following sections will explore how these capabilities translate into practical advantages in multi-agent reinforcement learning systems.

3 Transformers in Reinforcement Learning

The integration of transformer architectures into reinforcement learning (RL) represents a significant advancement in the field of artificial intelligence. This chapter explores the application of transformers to RL, addressing the challenges encountered in this fusion and examining the innovative approaches that have emerged as a result. We begin by discussing the methods used to stabilize transformers for RL applications, followed by an in-depth look at decision transformers, which have revolutionized how we approach RL problems.

3.1 Stabilizing Transformers for Reinforcement Learning

While transformers have shown impressive results in processing human language, adapting them for reinforcement learning introduces new challenges that require careful consideration. To understand these challenges, let's first consider what makes reinforcement learning different from language processing. In reinforcement learning, an agent learns by interacting with an environment and receiving feedback in the form of rewards. Unlike language tasks where the rules of grammar remain constant, the environment in reinforcement learning keeps changing as the agent learns - what worked well early in training might become suboptimal later. This changing nature of the environment is called non-stationarity, and it poses a significant challenge for transformer models [55].

Think of it like learning to play chess: at first, simple strategies might work against a beginner, but as your opponent improves, those same strategies become less effective, forcing you to adapt continuously. Similarly, in reinforcement learning, the agent's experiences (called trajectories) evolve from random exploration in the beginning to more strategic behavior later. This evolution can destabilize the learning process of transformer models, which were originally designed for more stable tasks like language processing.

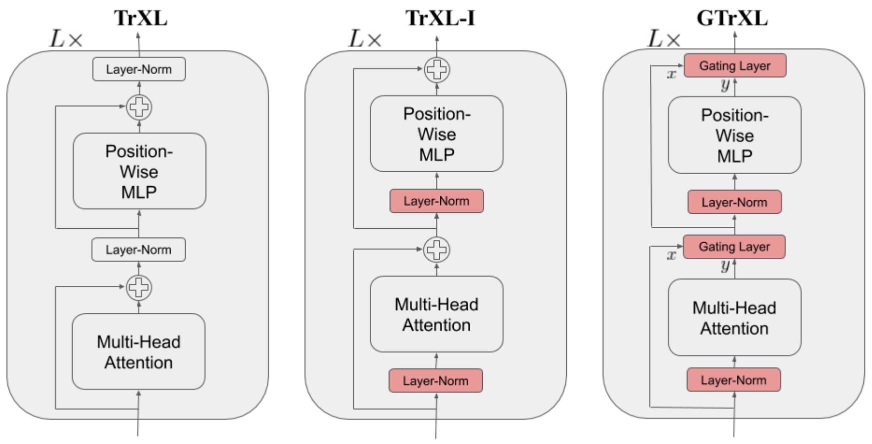


Figure 3.1: Three generations of transformer architectures for reinforcement learning. The original Transformer-XL design (left) serves as the starting point. The TrXL-I version (center) improves stability by changing how it normalizes data. The GTrXL version (right) adds controls to better manage information flow through the network [55].

As shown in Figure 3.1, researchers have developed several improvements to make transformers work better for reinforcement learning. Let's walk through these improvements step by step, understanding why each one matters.

The first crucial improvement involves how the model handles numerical stability through a process called layer normalization. In any deep learning model, numbers flowing through the network can grow very large or very small, making it difficult for the model to learn effectively. Layer normalization helps prevent this by adjusting these numbers to a consistent scale. Mathematically, for any set of numbers x flowing through the network, normalization adjusts them using this formula:

$$\hat{x} = \frac{x - \mathbb{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} \tag{3.1}$$

Here, $\mathbb{E}[x]$ represents the average value of x, and $\mathrm{Var}[x]$ represents how much these values vary from their average. The small number ϵ (epsilon) is added to prevent division by zero. Think of this like adjusting the volume levels in a music recording - if some parts are too loud and others too quiet, we normalize them to a consistent volume that works better for listeners.

The second major improvement involves controlling how information flows through the network using what's called a gating mechanism. In the GTrXL architecture (shown on the right in Figure 3.1), this gating works like a series of adjustable filters that can emphasize or de-emphasize different pieces of information. Mathematically, this filtering process works as:

$$y = x + g \odot F(x) \tag{3.2}$$

In this equation, x represents the input information, $F(x)$ represents how that information has been processed by the network, and g is a learned set of numbers between 0 and 1 that control how much of the processed information to use. The symbol $\odot$ means we multiply these numbers together element by element.

Another important improvement deals with how the model keeps track of position in a sequence. Earlier transformer models used fixed position markers, like numbering words in a sentence from 1 to

n. However, in reinforcement learning, sequences can be different lengths - one game might take 100 moves while another takes 500. The solution, called relative positional encoding, only keeps track of how far apart things are rather than their absolute positions. This is like remembering that one event happened three steps after another, rather than trying to remember exactly when each event occurred.

These improvements work together with careful training procedures. Just as you might start exercise gradually to avoid injury, the model uses a "warm-up" period where it starts learning slowly and gradually increases its learning rate. This careful approach, combined with proper initialization of the model's parameters, helps ensure stable learning from the beginning.

The effectiveness of these modifications becomes clear in practice. When testing these improved transformers on complex tasks like playing Atari games, where rewards might come long after the actions that earned them, the stabilized models learn much more reliably than their predecessors. For example, in games requiring long-term strategy, the GTrXL architecture can maintain stable performance while simpler models often fail to learn effectively.

These advances in stabilizing transformers have opened the door to new applications in reinforcement learning. In particular, they have enabled the development of Decision Transformers, which we'll explore in the next section. Decision Transformers take advantage of these stability improvements to create a novel approach to reinforcement learning, treating the learning process as a type of sequence prediction problem.

3.2 Decision Transformers

The introduction of Decision Transformers by Chen et al. [11] represents a fundamental rethinking of how reinforcement learning problems can be solved. To understand this innovation, consider how a human might learn from watching an expert play a video game. Rather than just trying random actions and learning from rewards, we observe sequences of successful gameplay, noticing how different actions in different situations lead to higher scores. Decision Transformers formalize this intuitive process, treating reinforcement learning as a type of sequence prediction problem.

Traditional reinforcement learning approaches typically learn through trial and error, gradually building up a policy (a strategy for choosing actions) through direct interaction with an environment. Decision Transformers take a different approach. Instead of learning through active experimentation, they learn from existing recordings of behavior, much like how a student might learn from watching recorded lectures. This approach, known as offline reinforcement learning, offers particular advantages in situations where experimenting with a real system might be costly or dangerous, such as in robotics or healthcare applications.

At its core, the Decision Transformer works by processing three types of information: states (what the agent observes), actions (what the agent does), and returns-to-go (how well the agent expects to perform from that point onward). Think of returns-to-go as a running score prediction - if you're playing a game and typically score 1000 points from your current position, that would be your return-to-go. Mathematically, we can express this relationship as:

$$p(a_t|s_1, a_1, R_1, ..., s_t, R_t) = \text{Transformer}(s_1, a_1, R_1, ..., s_t, R_t) \tag{3.3}$$

This equation might look complex, but it's expressing a simple idea: given a history of states (s), actions (a), and expected returns (R), what action should we take next? The transformer processes this history using attention mechanisms that help it identify which past experiences are most relevant

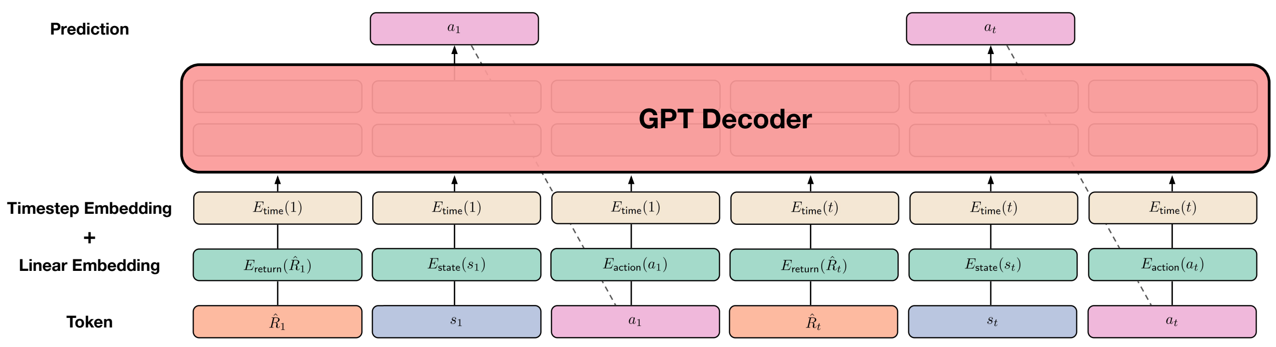


Figure 3.2: The Decision Transformer architecture processes sequences of states, actions, and expected returns to predict future actions. The model uses attention mechanisms to identify relevant patterns in past behavior, enabling it to generate appropriate actions for achieving desired outcomes [11].

for the current decision.

As shown in Figure 3.2, the architecture processes this information through several stages. First, the raw inputs are transformed into a format the model can work with through an embedding process:

$$E(x_t) = W_e x_t + P_t \tag{3.4}$$

Here, W_e transforms the input into a higher-dimensional representation (think of this as giving the model more room to capture subtle patterns), and P_t adds information about when each event occurred in the sequence. This is similar to how we might remember not just what happened in a game, but also the order in which events occurred.

The embedded information then flows through a series of attention layers based on the GPT architecture [60]. These layers allow the model to focus on relevant parts of the history when making decisions. For example, if the current state looks similar to a situation seen earlier in the sequence, the model can pay special attention to what actions worked well in that similar situation.

What makes Decision Transformers particularly innovative is their ability to condition behavior on desired outcomes. By adjusting the return-to-go values during inference, we can guide the model toward more or less ambitious behavior. This is analogous to telling a human player whether to play conservatively for a steady score or aggressively for a high score - the same basic skills are applied differently depending on the goal.

The effectiveness of this approach has been demonstrated across a range of challenging tasks. Consider the results from testing Decision Transformers on the OpenAI Gym environments, a standard set of benchmarks for reinforcement learning:

Environment	Dataset	DT	CQL
HalfCheetah-v2	Medium-Expert	**86.8 ± 1.3**	62.4
Walker2d-v2	Medium-Expert	**108.1 ± 0.2**	98.7
Hopper-v2	Medium	**67.6 ± 1.0**	58.0

Table 3.1: Performance comparison between Decision Transformers (DT) and Conservative Q-Learning (CQL) on standard reinforcement learning benchmarks. Scores are normalized, with higher values indicating better performance.

These results in Table 3.1 show that Decision Transformers often outperform traditional approaches like Conservative Q-Learning (CQL). In the HalfCheetah-v2 environment, for example, Decision Transformers achieve a normalized score of 86.8, substantially higher than CQL's 62.4. These environments involve complex physical simulations where an agent must learn to control a robotic system - no small feat for a model that never actively experiments with the environment.

Even more impressive are the results from Atari game experiments, where Decision Transformers learned to play games like Breakout effectively just from watching previous gameplay. Using only 1% of the data typically needed for traditional methods, Decision Transformers achieved a normalized score of 267.5 on Breakout, significantly outperforming other approaches.

The success of Decision Transformers suggests a promising direction for reinforcement learning research. By recasting the problem of learning behavior as one of sequence modeling, they open up new possibilities for leveraging advances in transformer architectures and large-scale sequence models. This connection between sequence modeling and reinforcement learning, which we'll explore further in the next section, may provide a bridge between the powerful pattern-recognition capabilities of modern language models and the challenges of learning complex behaviors.

3.3 Offline RL as Sequence Modeling

The field of offline reinforcement learning faces a fundamental challenge: how can an agent learn optimal behavior from pre-recorded data without actively interacting with its environment? While traditional approaches focus on directly estimating value functions or policies from static datasets, recent work by Janner et al. [33] has introduced an elegant alternative - treating reinforcement learning as a sequence modeling problem. This perspective builds upon our earlier discussion of Decision Transformers, extending the application of sequence modeling techniques to address broader challenges in offline RL.

To understand this approach, consider how we might view a recorded gameplay session. Rather than seeing it as a collection of isolated state-action pairs, we can view it as a coherent sequence of events, much like a story. Mathematically, we represent this sequence as:

$$p(s_1, a_1, r_1, \ldots, s_T, a_T, r_T) \tag{3.5}$$

Here, at each timestep t, we record the state of the environment (s_t), the action taken (a_t), and the reward received (r_t). This format allows us to capture not just what happened at each moment, but how events unfold and influence each other over time - crucial information that might be lost when treating experiences as independent samples.

The Trajectory Transformer, introduced by Janner et al. [33], implements this sequence modeling perspective using transformer architecture. Just as a language model learns to predict the next word in a sentence by understanding the context of previous words, the Trajectory Transformer learns to predict appropriate actions by understanding the context of previous states, actions, and rewards. This model processes sequences of the form:

$$(s_1, a_1, r_1, s_2, a_2, r_2, \ldots, s_T, a_T, r_T) \tag{3.6}$$

What makes this approach particularly powerful is how it handles the temporal relationships between events. Traditional reinforcement learning methods often struggle with credit assignment - determining which past actions were responsible for current outcomes. The transformer's attention mechanism addresses this challenge naturally by learning to focus on relevant past experiences when making decisions, regardless of how far back they occurred.

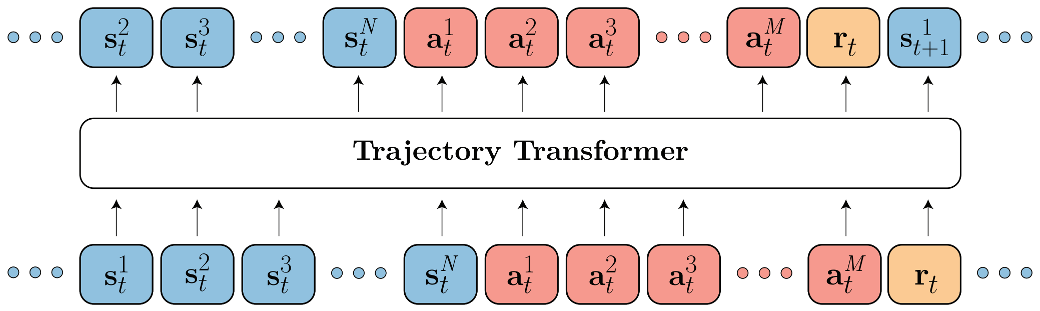


Figure 3.3: The Trajectory Transformer architecture processes states, actions, and rewards as a sequence. The attention mechanism allows the model to identify and utilize relevant past experiences when making decisions, enabling effective learning from offline data **[33]**.

This sequence modeling perspective offers several practical advantages beyond those discussed in our earlier examination of Decision Transformers. Kumar et al. [40] demonstrated that the probabilistic nature of sequence generation enables natural uncertainty estimation - the model can express varying degrees of confidence in its decisions, crucial for safe deployment in real-world applications. Additionally, Parisotto et al. [55] showed how the attention mechanism provides an implicit form of credit assignment, helping the model focus on truly relevant historical information when making decisions.

The approach particularly shines in environments with sparse rewards, where feedback is infrequent. Traditional methods might struggle in such scenarios because they rely heavily on immediate reward signals. In contrast, sequence modeling can learn from entire trajectories, understanding how sequences of actions eventually lead to delayed rewards. This capability proves especially valuable in real-world applications where reward signals might be rare or delayed.

Building on these advantages, Reed et al. [62] demonstrated how this sequence modeling approach enables more efficient learning from limited data. Their work on the Gato architecture showed that by treating reinforcement learning as sequence modeling, they could leverage techniques from natural language processing to extract more information from available data, leading to better performance even with limited task-specific examples.

This conceptual shift from traditional offline RL to sequence modeling represents more than just a technical innovation - it provides a new way of thinking about how agents can learn from recorded experiences. By treating an agent's experience as a coherent sequence rather than isolated events, we can better capture the temporal dynamics and long-term dependencies that characterize complex tasks. This perspective continues to inspire new approaches in reinforcement learning, pushing the boundaries of what's possible in offline learning scenarios.

4 Transformers in Multi-Agent Reinforcement Learning

As we've seen in previous sections, transformer architectures excel at modeling relationships between sequences of information. This capability becomes particularly valuable in multi-agent reinforcement learning (MARL), where agents must coordinate their actions based on observations of each other's behavior. Traditional MARL approaches often struggle with three key challenges: how to effectively share information between agents, how to scale to larger numbers of agents, and how to generalize across different team compositions. This chapter examines how transformers can address each of these challenges through three innovative approaches: transformer-based value function decomposition, universal policy decoupling, and sequence modeling for MARL.

5 Transformers in Multi-Agent Reinforcement Learning

The coordination of multiple learning agents presents unique challenges beyond those faced in single-agent reinforcement learning. A critical challenge is credit assignment - determining how each agent's individual actions contribute to the team's success. As we saw in previous chapters, transformer architectures excel at modeling relationships between sequences and capturing long-range dependencies. This chapter examines how these capabilities can be leveraged to address fundamental challenges in multi-agent reinforcement learning (MARL).

5.1 Transformer-based Value Function Decomposition

Consider the challenge faced by a team of robots coordinating to defeat opponents in a real-time strategy game. Each robot needs to make individual tactical decisions, but these decisions must be coordinated to achieve victory. This scenario raises a fundamental question: how can we enable agents to make independent decisions while ensuring their actions combine effectively toward the team's goals?

Value function decomposition provides a solution by breaking down the team's overall value function into individual components that can guide each agent's decisions. Traditional approaches like QMIX [61] use fixed architectures with strict monotonicity constraints - ensuring that an improvement in any agent's individual value estimate leads to an improvement in the team's value. However, this constraint can be overly restrictive, preventing the representation of more complex coordination patterns.

TransMix, introduced by Khan et al. [37], provides a more flexible approach using transformers. At its core, TransMix learns a mixing function that combines individual agent values $(Q_1, Q_2, ..., Q_n)$ into a team value (Q_{tot}):

$$Q_{tot} = f(Q_1, Q_2, ..., Q_n) \tag{5.1}$$

The innovation lies in how this mixing function f is implemented. Rather than using a fixed monotonic network, TransMix employs a stack of transformer encoder layers that process individual agent Q-values along with their action-observation histories (h_i^t) and global state information (S_t). This allows the model to learn rich patterns of agent interaction that adapt to the current situation.

As shown in Figure 5.1, TransMix first processes the global state through self-attention to identify important state features. These are then combined with individual agent values and histories through a series of additive attention operations:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{5.2}$$

The use of additive attention, rather than the standard scaled dot-product attention, provides both computational efficiency and improved training stability. The model uses 4 attention heads with embedding dimension 512 and hidden dimension 2048, allowing it to capture different aspects of agent coordination while maintaining reasonable computational requirements.

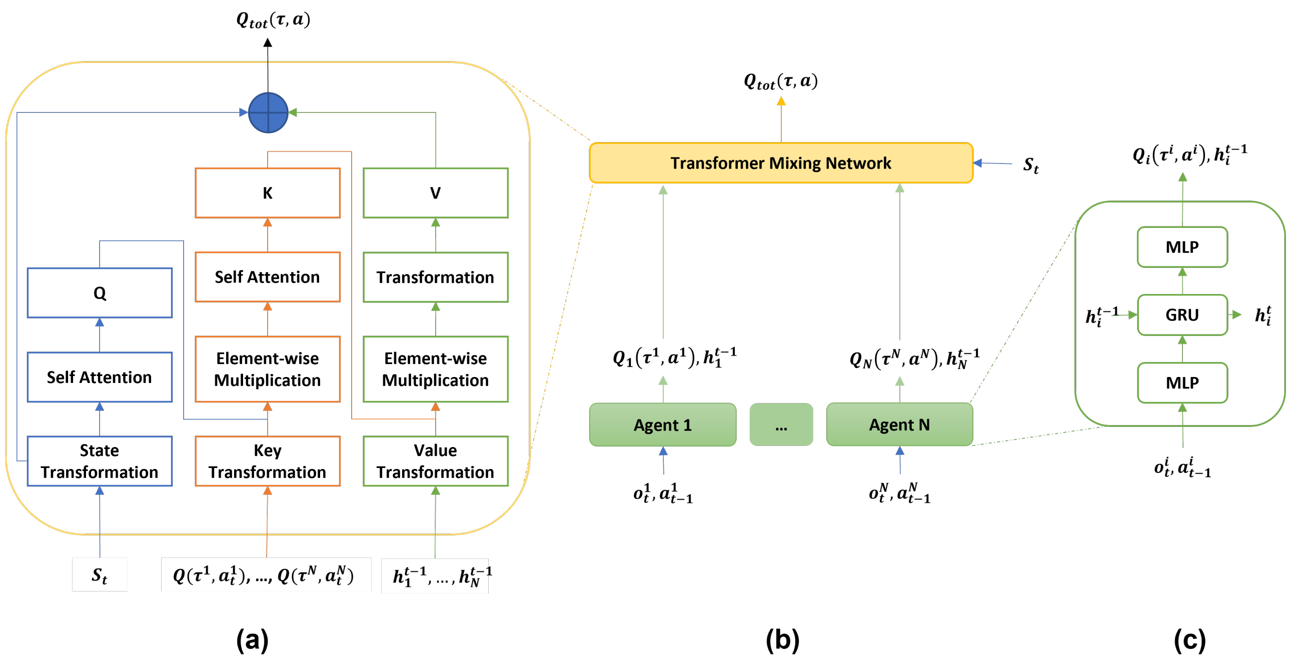

$Q_{tot}(\tau, a)$
K
V
Q
Self Attention
Transformation
Self Attention
Element-wise Multiplication
Element-wise Multiplication
State Transformation
Key Transformation
Value Transformation
s_t
$Q(\tau^1, a_t^1), \ldots, Q(\tau^N, a_t^N)$
$h_1^{t-1}, \ldots, h_N^{t-1}$
Transformer Mixing Network
$Q_1(\tau^1, a^1), h_1^{t-1}$
$Q_N(\tau^N, a^N), h_N^{t-1}$
Agent 1
...
Agent N
o_t^1, a_{t-1}^1
o_t^N, a_{t-1}^N
$Q_i(\tau^i, a^i), h_i^{t-1}$
MLP
GRU
MLP
h_i^{t-1}
h_i^t
o_t^i, a_{t-1}^i
(a)
(b)
(c)

Figure 5.1: The TransMix architecture consists of three main components: (a) a transformer encoder that processes agent values and states, (b) the overall mixing framework that combines individual contributions, and (c) GRU-based networks that compute individual agent values. The transformer's ability to model relationships between all inputs enables more sophisticated coordination than fixed mixing networks. Adapted from Khan et al. [37].

Khan et al. [37] demonstrated TransMix's effectiveness on the StarCraft Multi-Agent Challenge (SMAC), where teams of units must coordinate to defeat opposing forces. In complex scenarios like 3s5z_vs_3s6z (3 Stalkers and 5 Zealots vs 3 Stalkers and 6 Zealots), TransMix achieved a 96.9% win rate compared to QMIX's 83.7%. This improvement stems from TransMix's ability to learn sophisticated coordination strategies that adapt based on unit types, positions, and the evolving battle situation.

Notably, TransMix also showed superior robustness to noisy state information. When global states were corrupted with Gaussian noise to simulate imperfect information, TransMix maintained higher performance than both QMIX and QPLEX across multiple scenarios. For example, in the 3s5z scenario, TransMix achieved an 84.2% win rate under noise compared to QMIX's 65.8%, demonstrating its ability to maintain effective coordination even with degraded information.

The success of TransMix highlights several key advantages of transformer-based approaches in MARL. First, the attention mechanism's ability to dynamically weight different sources of information allows for more nuanced credit assignment than fixed mixing architectures. Second, the model's permutation invariance means it doesn't depend on specific agent orderings, making it more flexible than previous approaches. Finally, the transformer's ability to process multiple input types (Q-values, histories, and states) in a unified way enables it to learn rich patterns of agent interaction that would be difficult to capture with simpler architectures.

5.2 Universal Policy Decoupling with Transformers (UPDeT)

One of the key challenges in multi-agent reinforcement learning is developing policies that can adapt to varying numbers and types of agents without requiring retraining. The Universal Policy Decoupling

Transformer (UPDeT) approach, introduced by Hu et al. [29], addresses this challenge through an innovative architecture that separates agent-specific processing from shared coordination mechanisms.

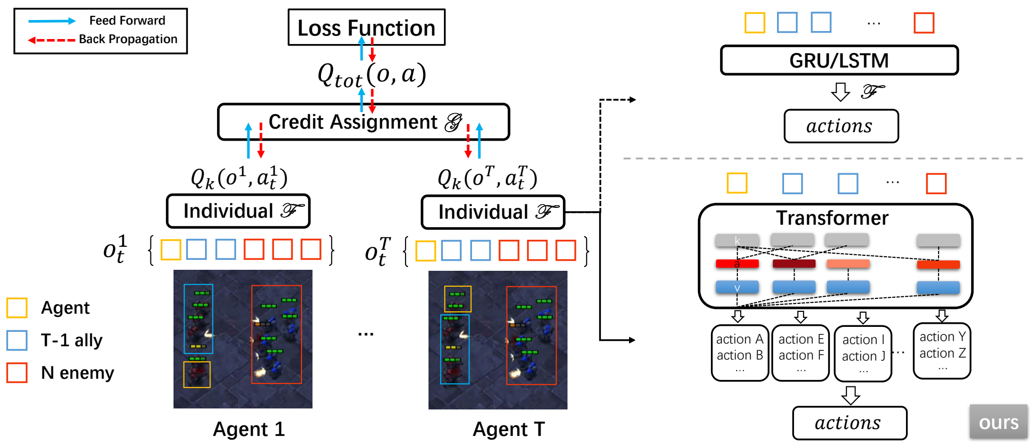


Figure 5.2: Overview of the UPDeT framework. The model replaces the commonly used GRU/LSTM-based individual value function with a transformer-based function. Actions are separated into action groups according to observations, allowing the model to scale flexibly with different team sizes. [29]

As illustrated in Figure 5.2, UPDeT consists of three main components that work together: individual encoders that process each agent's observations, a shared transformer that enables coordination, and a flexible policy head that generates actions. This architecture allows UPDeT to handle heterogeneous teams where agents may have different capabilities or roles. The figure shows how the transformer processes observations and generates appropriate actions by separating them into groups, enabling flexible scaling with team size.

The key innovation lies in how UPDeT processes information from multiple agents. Rather than treating all agent observations as a single input, it uses agent-specific encoders that transform each agent's observations into a suitable representation:

$$\pi(a_i|o_i, \mathbf{o}_{-i}) = \text{PolicyHead}(\text{Transformer}(\text{Encoder}_i(o_i), \{\text{Encoder}_j(o_j)\}_{j\neq i})) \tag{5.3}$$

Here, o_i represents agent i's observations, $\mathbf{o}_{-i}$ captures the observations of all other agents, and a_i is agent i's action. This formulation allows each agent to maintain its individual perspective while still coordinating with teammates.

The transformer component employs a modified attention mechanism that explicitly differentiates between an agent's own information and that of others:

$$\text{Attention}(Q_i, K, V) = \text{softmax}\left(\frac{Q_i K^T}{\sqrt{d_k}} + M_i\right) V \tag{5.4}$$

where M_i represents an agent-specific mask. This masking mechanism helps the model distinguish between self-attention and attention to other agents' information, enabling more nuanced coordination strategies.

Looking at the architecture in Figure 5.2, we can see how UPDeT's transformer backbone allows the model to identify relevant relationships between agents, while the decoupled structure ensures that adding or removing agents doesn't require architectural changes. This design directly addresses the scalability challenges faced by traditional approaches that rely on fixed network architectures.

Empirical evaluations demonstrate UPDeT's effectiveness across diverse scenarios. In the Multi-Agent Particle Environment (MAPE) cooperative navigation task, UPDeT achieved significantly better performance than baseline approaches like MADDPG and QMIX, even when tested with different numbers of agents than it was trained on. For instance, when trained with 3 agents and tested with 5, UPDeT maintained substantially higher average rewards than previous methods.

More impressively, in StarCraft II micromanagement scenarios, UPDeT exhibited strong zero-shot generalization capabilities. Models trained on 5v5 unit configurations maintained over 90% win rates when tested on 10v10 scenarios - a significant achievement given the increased complexity of coordinating larger teams.

This generalization capability stems from UPDeT's decoupled architecture, which allows it to process information from each agent independently while still maintaining coordinated behavior through the shared transformer. The attention mechanisms provide insights into how agents prioritize information, making the system more interpretable than traditional approaches.

These results suggest that UPDeT represents a significant step toward more flexible and scalable multi-agent systems. Its ability to handle teams of varying sizes and compositions without retraining makes it particularly valuable for real-world applications where agent configurations may change dynamically.

5.3 MARL as a Sequence Modeling Problem

While TransMix and UPDeT demonstrate the benefits of transformer architectures for value decomposition and policy generalization respectively, Wen et al. [86] propose a more fundamental reconceptualization: treating the entire MARL problem as sequence prediction. This perspective shifts focus from learning value functions or policies directly to learning the underlying patterns in successful multi-agent trajectories.

The key insight lies in recognizing that effective coordination requires understanding both the temporal structure of interactions and the causal relationships between agent actions. While previous approaches handle these aspects separately - with recurrent networks for temporal dependencies and attention mechanisms for agent relationships - sequence modeling provides a unified framework for both.

Consider a team of N agents operating over T timesteps. Traditional MARL approaches would attempt to learn a policy $\pi_\theta(a_t^i|o_t^i)$ for each agent i, potentially with some communication mechanism. In contrast, the sequence modeling perspective treats the entire interaction history as a sequence to be modeled and extended:

$$\tau = [(s_1, \{o_1^i\}_{i=1}^N, \{a_1^i\}_{i=1}^N, r_1), ..., (s_T, \{o_T^i\}_{i=1}^N, \{a_T^i\}_{i=1}^N, r_T)] \tag{5.5}$$

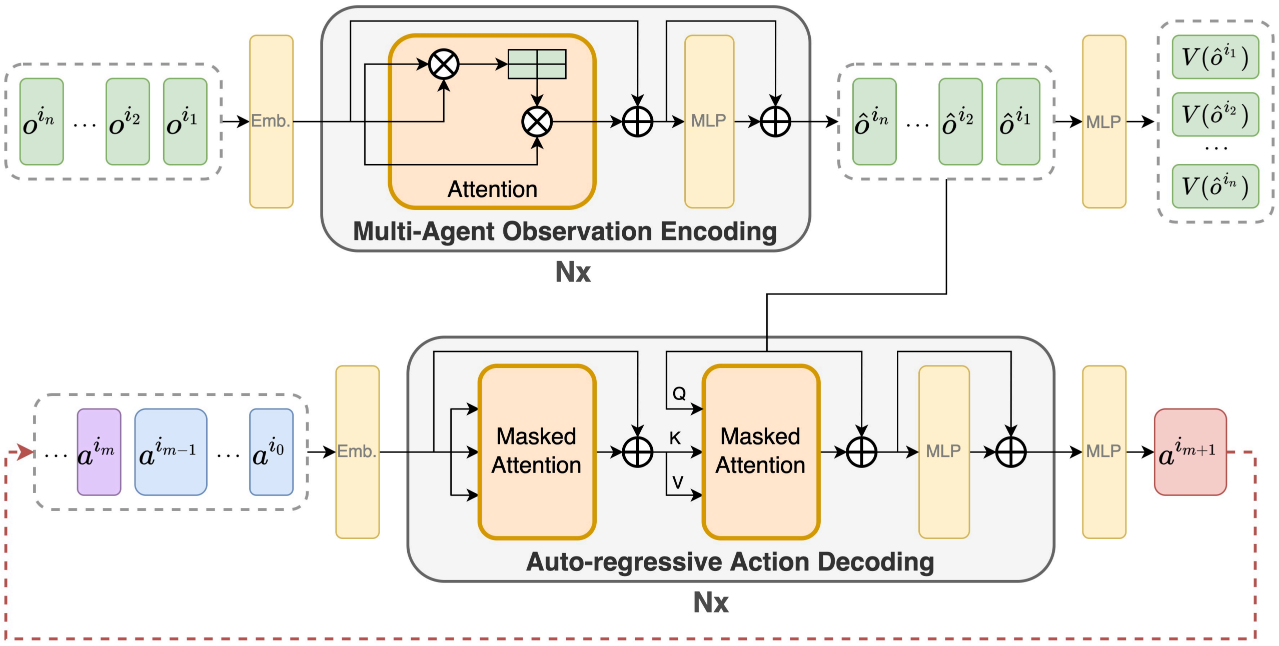
This representation captures not just the states, observations, and actions, but their evolution over time. The learning objective becomes predicting future elements of this sequence conditioned on past elements:

$$p(\tau_{t+1:T}|\tau_{1:t}) = \prod_{k=t+1}^{T} p(s_k, \{o_k^i\}, \{a_k^i\}, r_k|\tau_{1:k-1}) \tag{5.6}$$

This formulation reveals a crucial difference from previous transformer-based approaches. Rather than using attention merely as a mechanism for information aggregation, here it serves to identify predictive patterns across both time and agents. The attention computation at each layer l and position t becomes:

$$h_t^l = \sum_{k \le t} \alpha_{tk}^l W_V^l h_k^{l-1}, \quad \alpha_{tk}^l = \text{softmax}\left(\frac{(W_Q^l h_t^{l-1})^T (W_K^l h_k^{l-1})}{\sqrt{d}}\right) \tag{5.7}$$

where W_Q^l, W_K^l, and W_V^l are learned parameters, and h_t^l represents the hidden state at position t and layer l. The crucial innovation is that h_t can represent any element of the trajectory - a state, an observation, an action, or a reward - allowing the model to learn arbitrary predictive relationships between these elements.



o^{i_n} ... o^{i_2} o^{i_1}
Emb.
Attention
MLP
Multi-Agent Observation Encoding
Nx
$\hat{o}^{i_n}$ ... $\hat{o}^{i_2}$ $\hat{o}^{i_1}$
MLP
$V(\hat{o}^{i_1})$
$V(\hat{o}^{i_2})$
...
$V(\hat{o}^{i_n})$
... a^{i_m} $a^{i_{m-1}}$ ... a^{i_0}
Emb.
Masked Attention
Q
K
V
Masked Attention
MLP
Auto-regressive Action Decoding
Nx
MLP
$a^{i_{m+1}}$

Figure 5.3: The MAT architecture processes trajectories through an encoder-decoder structure. The encoder builds representations that capture both temporal and inter-agent dependencies, while the decoder generates future trajectory elements through masked attention. This unified treatment of temporal and agent relationships enables more sophisticated coordination strategies [86].

To implement this approach effectively, Wen et al. [86] introduced the Multi-Agent Transformer (MAT) architecture shown in Figure 5.3. MAT consists of an encoder that processes past trajectory elements and a decoder that generates future elements. The encoder employs bidirectional attention to build rich representations of the interaction history:

$$z_t = \text{Encoder}(\tau_{1:t}) = \text{MultiHead}(\text{LayerNorm}(h_t + \text{MLP}(h_t))) \tag{5.8}$$

The decoder then generates future trajectory elements autoregressively, using masked attention to maintain causality:

$$p(\tau_t|\tau_{1:t-1}) = \text{Decoder}(z_{1:t-1}) = \text{softmax}(W_o\text{MultiHead}(\text{Mask}(z_{1:t-1}))) \tag{5.9}$$

This architecture offers several theoretical advantages over previous approaches. First, it naturally handles partial observability by allowing attention to span the entire interaction history. Second, it provides a principled way to incorporate demonstrations or prior experiences through pre-training on trajectory data. Finally, it enables more sophisticated reward prediction and planning by explicitly modeling the relationship between actions and future rewards.

The training process introduces unique challenges not present in previous transformer-based MARL approaches. The model must learn to predict not just actions but entire trajectory segments, requiring careful handling of the different element types and their relationships. Wen et al. [86] address this through a hierarchical loss function:

$$\mathcal{L} = \lambda_s \mathcal{L}_{\text{state}} + \lambda_o \mathcal{L}_{\text{obs}} + \lambda_a \mathcal{L}_{\text{action}} + \lambda_r \mathcal{L}_{\text{reward}} \tag{5.10}$$

where each component focuses on predicting its respective element type, and the λ coefficients balance their contributions.

Empirical evaluations demonstrate the benefits of this unified approach. In the StarCraft II 3s5z_vs_3s6z scenario, MAT achieved a 98.7% win rate compared to 96.9

The sequence modeling perspective also enables new theoretical analyses. Wen et al. [86] prove that under certain regularity conditions, the model's predictive accuracy bounds its policy performance:

$$J(\pi_\theta) \geq J^* - 2\sqrt{2} R_{\max} T \sqrt{\mathbb{E}_{\tau\sim\pi^*}[-\log p_\theta(\tau)]} \tag{5.11}$$

where $J(\pi_\theta)$ is the expected return of the learned policy, J^* is the optimal return, and $R_{\max}$ is the maximum possible reward. This provides theoretical justification for using sequence prediction as a training objective.

These results suggest that treating MARL as sequence modeling offers more than just an alternative training approach - it provides a fundamentally different way of understanding and implementing multi-agent coordination. By unifying temporal and agent-wise relationships in a single predictive framework, it enables more sophisticated coordination strategies while maintaining theoretical guarantees.

6 Advanced Applications and Techniques

6.1 Transformer World Models in RL (TransDreamer)

The integration of transformer architectures into reinforcement learning (RL) has led to significant advancements in the field, particularly in the domain of model-based RL. One notable example of this integration is TransDreamer, introduced by Chen et al. [9]. This model builds upon the success of the Dreamer framework [22, 23] by incorporating transformer-based world models to enhance long-term dependency modeling and complex reasoning in RL tasks.

TransDreamer's key innovation lies in its use of a Transformer State-Space Model (TSSM), which replaces the Recurrent State-Space Model (RSSM) used in the original Dreamer. The TSSM leverages the power of transformer architectures to model the dynamics of the environment more effectively, particularly in scenarios requiring long-term memory and complex temporal dependencies.

The TSSM consists of several components that work in concert to create a powerful world model:

$$h_t = f_{transformer}(z_{1:t-1}, a_{1:t-1}) \tag{6.1}$$

Here, h_t represents the deterministic state at time t, computed by applying a transformer function to the sequence of past stochastic states $z_{1:t-1}$ and actions $a_{1:t-1}$. This allows the model to directly access and reason about past states and actions, unlike recurrent models which compress all past information into a single hidden state.

The stochastic state z_t is modeled using a representation model and a transition model:

$$z_t \sim q(z_t|x_t) \quad \text{(Representation Model)} \tag{6.2}$$

$$\hat{z}_t \sim p(\hat{z}_t|h_t) \quad \text{(Transition Model)} \tag{6.3}$$

Where x_t is the observation at time t. The representation model infers the current stochastic state given the current observation, while the transition model predicts the next stochastic state given the current deterministic state.

One of the key differences between TSSM and RSSM is in the representation model. TSSM uses a "myopic" representation model that depends only on the current observation x_t, rather than on both x_t and h_t as in RSSM. This design choice allows for parallel computation of stochastic states, significantly improving computational efficiency.

TransDreamer also includes observation and reward prediction models:

$$\hat{x}_t \sim p(\hat{x}_t|h_t, z_t) \quad \text{(Observation Model)} \tag{6.4}$$

$$\hat{r}_t \sim p(\hat{r}_t|h_t, z_t) \quad \text{(Reward Model)} \tag{6.5}$$

These models allow TransDreamer to imagine future trajectories, a crucial capability for model-based RL.

The training of TransDreamer follows a similar pattern to Dreamer, alternating between world model learning and policy learning. The world model is trained by maximizing the evidence lower bound (ELBO), while the policy is trained on imagined trajectories generated by the world model.

One of the key advantages of TransDreamer is its ability to handle tasks requiring long-term memory and complex reasoning. The authors demonstrate this through experiments on a novel "Hidden Order Discovery" task, where an agent must deduce and follow a hidden sequence in an environment. TransDreamer consistently outperforms Dreamer on these tasks, showcasing its superior ability to model and utilize long-range dependencies.

Moreover, TransDreamer shows comparable performance to Dreamer on simpler tasks that don't require long-term memory, such as certain Atari games and DeepMind Control Suite tasks. This indicates that the transformer-based architecture doesn't compromise performance on simpler tasks while providing significant benefits for more complex scenarios.

The success of TransDreamer highlights the potential of transformer architectures in model-based RL. By enabling more effective modeling of long-term dependencies and complex temporal relationships, transformer-based world models open up new possibilities for tackling challenging RL problems that require sophisticated reasoning and memory capabilities.

6.2 On-Policy RL with Transformers (PoliFormer)

The integration of transformer architectures with on-policy reinforcement learning (RL) represents a significant advancement in the field of embodied AI, particularly for navigation tasks. PoliFormer, introduced by Zeng et al. [94], demonstrates the potential of this approach by achieving state-of-the-art performance on multiple navigation benchmarks. This section explores the key innovations and methodologies that enable effective on-policy RL training with transformer models.

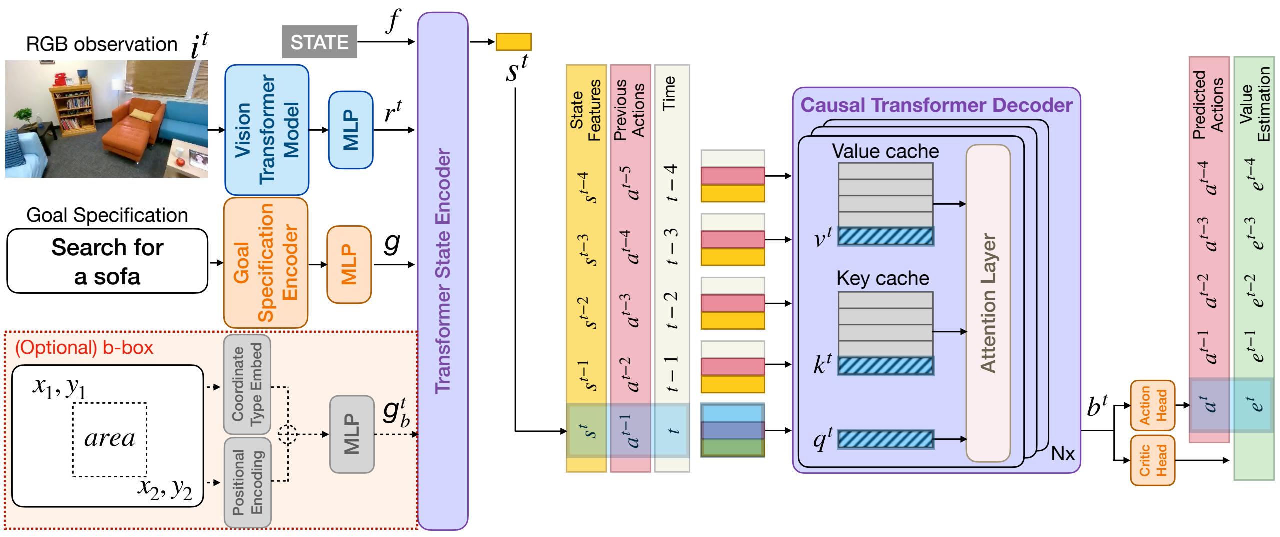


Figure 6.1: PoliFormer architecture: a fully transformer-based policy model for navigation tasks. At each timestep t, the model processes an ego-centric RGB observation i^t through a vision transformer to extract visual representations r^t. These are combined with goal features g (and optional bounding box features g_b^t) in a transformer state encoder to produce state features s^t. A causal transformer decoder with KV-cache models the state belief b^t over time, enabling efficient temporal reasoning. The model outputs action logits a^t and value estimations e^t via linear actor and critic heads. The KV-cache strategy prevents recomputation of past timesteps, significantly speeding up both training and inference [94].

PoliFormer employs a transformer-based architecture consisting of three main components, as illustrated in Figure 6.1: a frozen vision transformer backbone, a transformer state encoder, and a causal transformer decoder. This can be expressed as:

$$\text{PoliFormer}(i_t, g) = f_{\text{decoder}}(f_{\text{encoder}}(f_{\text{vision}}(i_t), g)) \tag{6.6}$$

where i_t is the input observation at time t, g is the goal specification, f_{vision} is the vision transformer backbone based on DINOv2 [53], f_{encoder} is the transformer state encoder, and f_{decoder} is the causal transformer decoder.

A key innovation in PoliFormer is the use of a KV-cache technique in the causal transformer decoder. This approach addresses the computational challenges associated with training transformer models in an on-policy RL setting. The KV-cache allows the model to store past key and value matrices, reducing the computational complexity from quadratic to linear in sequence length:

$$\text{Complexity} = \mathcal{O}(t) \text{ instead of } \mathcal{O}(t^2) \tag{6.7}$$

where t is the sequence length. This optimization enables efficient training and inference, making it feasible to use transformer models in on-policy RL scenarios.

PoliFormer achieves its impressive performance through a multi-faceted approach to scaling up the training process. The architecture scales to hundreds of millions of parameters, leveraging the representational power of large transformer models. Training is distributed across multiple machines, utilizing 32 GPUs and 512 CPU cores to collect 192-384 parallel rollouts. The effective batch size is increased during training by extending the rollout length from 32 to 128 steps. Additionally, the simulation environment is optimized for faster scene loading and physics approximations, reducing training time by 42%. This scaled-up training approach allows PoliFormer to be trained on hundreds of millions of environment interactions, leading to superior performance and generalization capabilities.

The model is trained using the PPO algorithm [65], with modifications to accommodate the transformer architecture:

$$L_{\text{PPO}}(\theta) = \mathbb{E}_t[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)] \tag{6.8}$$

where $r_t(\theta)$ is the probability ratio between the new and old policies, $\hat{A}_t$ is the estimated advantage at time t, and ϵ is the clipping parameter. The training process incorporates reward shaping to encourage efficient navigation, combining penalties for inefficient actions, rewards for task completion, and incentives for reducing distance to the goal.

PoliFormer demonstrates remarkable performance across multiple navigation benchmarks, achieving significant improvements over previous state-of-the-art methods. For example, on the CHORES-S benchmark, PoliFormer achieves an 85.5% success rate, a 28.5% absolute improvement over the previous best model. Moreover, PoliFormer exhibits strong zero-shot transfer capabilities to real-world environments, outperforming baselines on both LoCoBot and Stretch RE-1 embodiments without any real-world fine-tuning.

These results highlight the potential of combining transformer architectures with on-policy RL for complex navigation tasks. The success of PoliFormer suggests that further scaling of model capacity, training data, and compute resources may lead to even more capable navigation agents in the future. As research in this area progresses, we can expect to see continued improvements in the performance and generalization capabilities of transformer-based RL models, potentially revolutionizing the field of embodied AI and autonomous navigation.

6.3 In-Context RL via Supervised Pretraining

In-context reinforcement learning (ICRL) has emerged as a promising approach to leverage the power of large language models for decision-making tasks. This section explores the use of supervised pretraining to enable transformers to perform ICRL effectively.

The key idea behind supervised pretraining for ICRL is to train a transformer model on a dataset of offline trajectories. These trajectories consist of sequences of states, actions, and rewards from various environments. The transformer learns to predict actions based on the observed states and previous interactions, effectively learning to perform reinforcement learning in-context.

Formally, we define a trajectory $D_T = (s_1, a_1, r_1, \ldots, s_T, a_T, r_T)$, where s_t, a_t, and r_t represent the state, action, and reward at time step t, respectively. The transformer is trained to maximize the log-likelihood of the observed actions given the previous states and actions:

$$\hat{\theta} = \arg\max_{\theta\in\Theta} \frac{1}{n}\sum_{i=1}^{n}\sum_{t=1}^{T} \log \text{Alg}_\theta(\bar{a}_t^i | D_{t-1}^i, s_t^i) \tag{6.9}$$

where θ represents the transformer parameters, n is the number of trajectories in the dataset, and $\bar{a}_t^i$ is the expert action for the i-th trajectory at time t.

This framework encompasses several variants of supervised pretraining. In Algorithm Distillation, introduced by Laskin et al. [41], the expert algorithm and the context algorithm are identical. The transformer learns to imitate a specific reinforcement learning algorithm, such as Q-learning or SARSA, across various environments. Decision-Pretrained Transformers (DPT), proposed by Lee et al. [42], use an expert algorithm that generates optimal actions for the Markov Decision Process (MDP). The transformer learns to approximate the optimal policy for any given environment.

Under certain assumptions, the supervised pretraining approach provides strong theoretical guarantees. The main result shows that the learned transformer will imitate the conditional expectation of the expert algorithm given the observed trajectory. Specifically, with high probability:

$$R_{\Lambda,\mathrm{Alg}_{\hat{\theta}}}(T) - R_{\Lambda,\mathrm{Alg}_E}(T) \leq cT^2\sqrt{R}\left(\sqrt{\frac{\log[N_\Theta \cdot T/\delta]}{n}} + \sqrt{\varepsilon_{\mathrm{real}}}\right) \tag{6.10}$$

where $R_{\Lambda,\mathrm{Alg}}(T)$ is the expected cumulative reward, R is a distribution ratio factor, N_Θ is the covering number of the transformer class, n is the number of training trajectories, and $\varepsilon_{\mathrm{real}}$ is a realizability error term.

This bound demonstrates that the performance of the learned transformer approaches that of the expert algorithm as the number of training trajectories increases, with additional factors accounting for the complexity of the transformer class and the distribution mismatch between the expert and offline algorithms.

The supervised pretraining approach offers several advantages for ICRL. It allows for efficient learning from offline datasets, potentially leveraging large amounts of previously collected data. The learned transformer can adapt to new environments without further training, making it suitable for rapid deployment in various scenarios. However, there are limitations to consider. The performance of the learned transformer is bounded by that of the expert algorithm used for training. In the case of Algorithm Distillation, if the context algorithm performs poorly, the learned transformer will also perform poorly. For DPT, while the transformer can learn to approximate optimal policies, its performance may be affected by the distribution mismatch between the offline data and the expert's optimal actions.

Furthermore, the generalization capabilities of the pretrained transformer to out-of-distribution instances, such as environments with significantly different dynamics or reward structures, remain an open question and an area for future research. Recent work by Kumar et al. [40] has begun to explore these challenges, proposing methods to improve the robustness and generalization of pretrained transformers in ICRL settings.

7 Large Language Models in MARL

7.1 LLMs as Knowledge Sources for RL Agents

The integration of Large Language Models (LLMs) as knowledge sources for reinforcement learning (RL) agents has shown promise in enhancing sample efficiency and generalization capabilities. This section explores the mathematical foundations and empirical results of this approach, focusing on two key frameworks: Language-INtegrated Value Iteration (LINVIT) [97] and Knowledgeable Agents from Language Model Rollouts (KALM) [54].

LINVIT formalizes the use of LLMs as policy priors in RL, demonstrating how this approach can improve sample efficiency. The algorithm is based on a regularized Markov Decision Process (MDP), where the regularization term is derived from the LLM policy. Let π^* be the optimal policy for the original MDP, π^{LLM} be the policy derived from the LLM, and λ be a regularization parameter. The regularized value function $V^{\pi_{LLM},\lambda,h}$ is defined as:

$$V^{\pi_{LLM},\lambda,h}(s_h) = \mathbb{E}_{a_h \sim \pi_h(\cdot|s_h)}\left[Q^{\pi_{LLM},\lambda,h}(s_h, a_h) - \lambda KL(\pi_h(\cdot|s_h) \| \pi_h^{LLM}(\cdot|s_h))\right] \tag{7.1}$$

where $Q^{\pi_{LLM},\lambda,h}$ is the regularized Q-function and KL denotes the Kullback-Leibler divergence.

The key theoretical result of LINVIT is that the number of samples required to achieve ϵ-optimality is proportional to the KL divergence between π^* and π^{LLM}. Specifically, when $KL(\pi^* \| \pi^{LLM}) \leq \epsilon_{LLM}$, setting $\lambda = \epsilon/(2\epsilon_{LLM})$ and the number of iterations
$T = CH^6SA^4 \log^2(HSA/\delta)/\epsilon^2$ (for some constant C), we have:

$$V_1^*(s_1) - V_1^{\hat{\pi}}(s_1) \leq \epsilon \tag{7.2}$$

with probability at least $1 - \delta$. This result suggests that when the LLM policy closely aligns with the optimal policy, the sample complexity can be significantly reduced.

The practical implementation of these ideas is demonstrated in the KALM method. KALM uses an LLM to generate synthetic rollouts for novel skills, which are then used to train an RL agent. The process involves three key steps: LLM grounding, rollout generation, and skill acquisition, as illustrated in Figure 7.1.

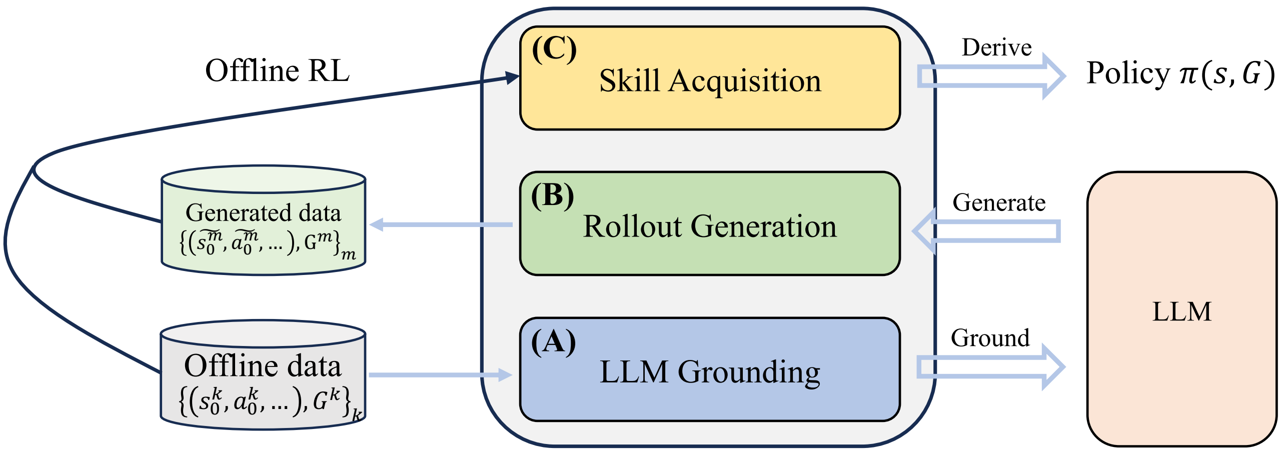
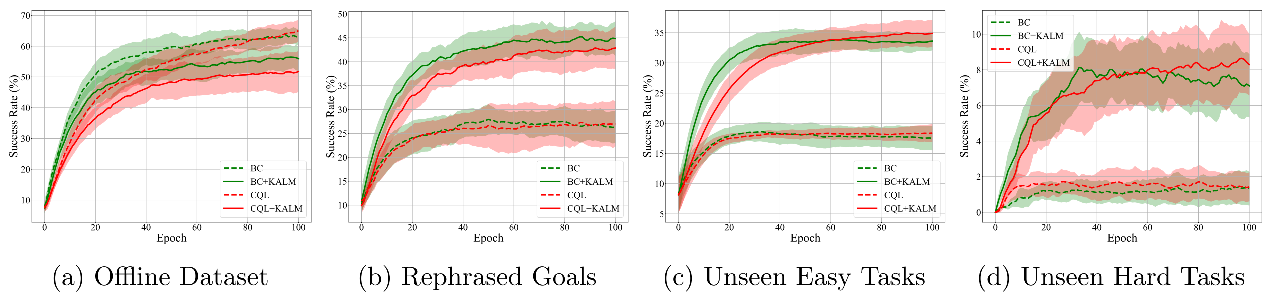


Figure 7.1: Overall procedure of KALM, consisting of LLM grounding, rollout generation, and skill acquisition modules [54].

Experimental results from KALM demonstrate the efficacy of this approach across various task types. Figure 7.2 shows the training curves for different methods on tasks from the offline dataset, rephrased goals, and unseen tasks of varying complexity.



(a) Offline Dataset (b) Rephrased Goals (c) Unseen Easy Tasks (d) Unseen Hard Tasks

Figure 7.2: Training curves comparing different methods on various task types [54].

The results reveal several key insights. On tasks present in the offline dataset, KALM-augmented methods (BC+KALM and CQL+KALM) achieve higher success rates compared to their base counterparts (BC and CQL), with CQL+KALM reaching approximately 70% success rate versus 60% for standard CQL [54]. For rephrased goals, the performance gap widens, with KALM-based methods achieving around 45% success rate compared to 30-35% for baseline methods. This demonstrates improved robustness to linguistic variations in task specifications.

On unseen easy tasks, KALM-based methods show a marked improvement, reaching 30-35% success rate versus 15-20% for baselines, indicating enhanced generalization to novel, simple tasks. For unseen hard tasks, while overall performance is lower due to task complexity, KALM-augmented methods still outperform baselines by a factor of 2-3, achieving 6-8% success rate compared to 2-4% for standard methods [54].

These results empirically support the theoretical predictions of improved sample efficiency and generalization. The LLM-generated rollouts enable the RL agent to learn from a more diverse set of experiences, leading to better performance across various task types, especially those not present in the original dataset.

The success of KALM can be attributed to its effective grounding of the LLM in the environment, allowing it to generate meaningful rollouts for novel tasks. This aligns with the theoretical framework of LINVIT, where a well-grounded LLM policy that closely matches the optimal policy leads to improved sample efficiency [97].

The use of LLMs as knowledge sources for RL agents offers a promising approach to enhancing sample efficiency and generalization capabilities. The theoretical foundations provided by LINVIT and the practical implementation demonstrated by KALM show how LLMs can be effectively integrated into RL frameworks, paving the way for more adaptable and capable artificial agents.

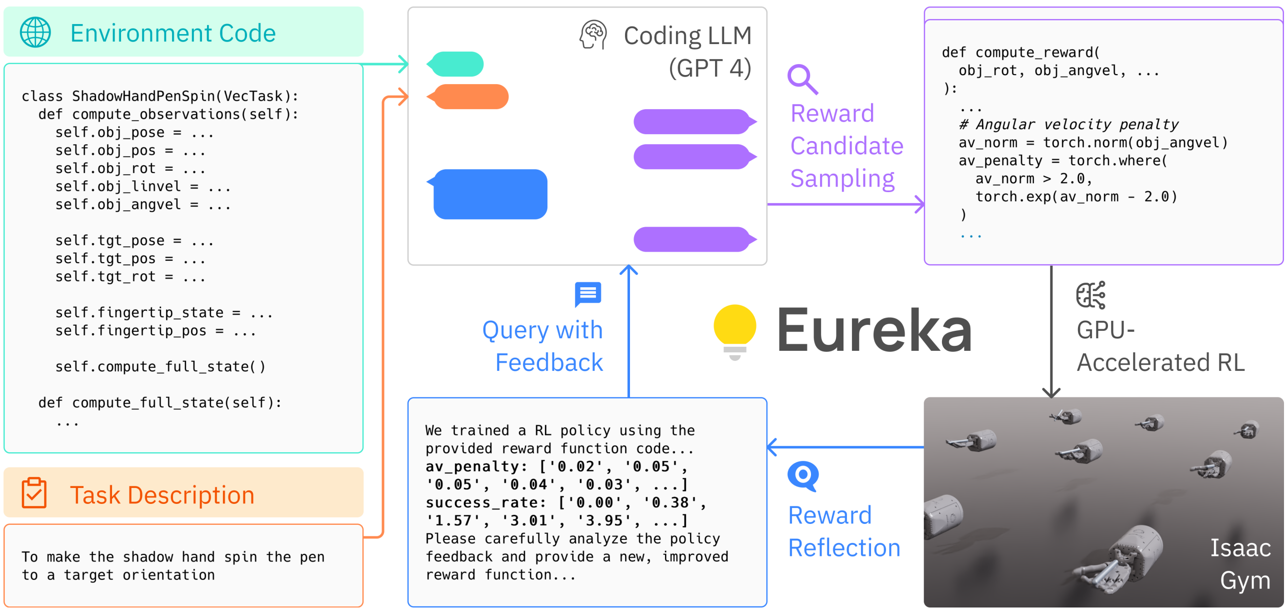
7.2 LLM-guided Reward Design (Eureka)

Reward design is a critical challenge in reinforcement learning (RL), often requiring significant human expertise and trial-and-error. Recent advancements in large language models (LLMs) have opened new avenues for automating this process. EUREKA (Evolution-driven Universal REward Kit for Agent) stands out as a pioneering approach that leverages LLMs to generate human-level reward functions across a diverse range of robotic tasks [47].

EUREKA frames the reward design problem as follows: given a world model $M = (S, A, T)$ with state space S, action space A, and transition function T, the goal is to find a reward function $R \in \mathcal{R}$ such that the policy $\pi := A_M(R)$ that optimizes R achieves the highest fitness score $F(\pi)$. Here, $A_M(\cdot)$ is a learning algorithm that outputs a policy optimizing the given reward in the MDP (M, R), and F is a task-specific fitness function.

The EUREKA framework operates through three key components: environment as context, evolutionary search, and reward reflection. The environment as context component feeds the raw environment code directly into the LLM, allowing it to understand the task's state and action spaces without requiring task-specific prompts. The evolutionary search implements an iterative improvement process within the LLM's context window, enabling in-context learning to refine reward functions. The reward reflection component provides automated feedback on the reward function's effectiveness, guiding the LLM in generating improved versions.

Figure 7.3 provides an overview of EUREKA's architecture. The process begins by generating an initial reward function based on the environment code and task description. This function is then



```
class ShadowHandPenSpin(VecTask):
  def compute_observations(self):
    self.obj_pose = ...
    self.obj_pos = ...
    self.obj_rot = ...
    self.obj_linvel = ...
    self.obj_angvel = ...

    self.tgt_pose = ...
    self.tgt_pos = ...
    self.tgt_rot = ...

    self.fingertip_state = ...
    self.fingertip_pos = ...

    self.compute_full_state()

  def compute_full_state(self):
    ...
```

```
To make the shadow hand spin the pen
to a target orientation
```

```
def compute_reward(
  obj_rot, obj_angvel, ...
):
  ...
  # Angular velocity penalty
  av_norm = torch.norm(obj_angvel)
  av_penalty = torch.where(
    av_norm > 2.0,
    torch.exp(av_norm - 2.0)
  )
  ...
```

```
We trained a RL policy using the
provided reward function code...
av_penalty: ['0.02', '0.05',
'0.05', '0.04', '0.03', ...]
success_rate: ['0.00', '0.38',
'1.57', '3.01', '3.95', ...]
Please carefully analyze the policy
feedback and provide a new, improved
reward function...
```

Figure 7.3: Overview of EUREKA's architecture, illustrating the iterative process of reward function generation and improvement [47].

evaluated in the RL environment, and the resulting performance metrics are used to create a reward reflection. The LLM uses this reflection to propose new, potentially improved reward functions. This process is repeated iteratively, allowing for continuous refinement of the reward function.

Algorithm 6 EUREKA Algorithm for LLM-driven Reward Function Design

```
Require: Task description $l$, environment code $M$, coding LLM LLM, fitness function $F$, initial
    prompt prompt
Ensure: Optimized reward function $R_{\text{Eureka}}$
1: for $N$ iterations do                                          ▷ Iterate for N generations to evolve reward functions
2:     Sample $K$ reward codes: $R_1, ..., R_K \sim \text{LLM}(l, M, \texttt{prompt})$    ▷ Generate K candidate reward
    functions
3:     Evaluate reward candidates: $s_1 = F(R_1), ..., s_K = F(R_K)$           ▷ Evaluate each candidate's
    performance
4:     Update prompt with reflection:
    $\texttt{prompt} \leftarrow \texttt{prompt} : \textsc{Reflection}(R^n_{\text{best}}, s^n_{\text{best}})$      ▷ Add reflection on best performer
5:     Update EUREKA reward:
    $R_{\text{Eureka}}, s_{\text{Eureka}} = (R^n_{\text{best}}, s^n_{\text{best}})$, if $s^n_{\text{best}} > s_{\text{Eureka}}$      ▷ Track best reward found
6: end for
7: return $R_{\text{Eureka}}$                                    ▷ Return the best reward function discovered
```

EUREKA has shown remarkable success across a diverse suite of 29 robotic environments, including 10 distinct robot morphologies. It outperformed human-engineered rewards on 83% of the tasks, with an average normalized improvement of 52%. Figure 7.4 illustrates EUREKA's performance compared to baselines across various tasks.

One of EUREKA's most significant achievements is its ability to solve complex dexterous manipulation tasks that were previously infeasible with manual reward engineering. A prime example is the pen spinning task, where EUREKA, combined with curriculum learning, enabled a simulated anthropomorphic hand to perform rapid pen spinning maneuvers for the first time.

EUREKA's effectiveness stems from its ability to generate novel, interpretable reward functions that often differ significantly from traditional human-designed rewards. Analysis of the correlation

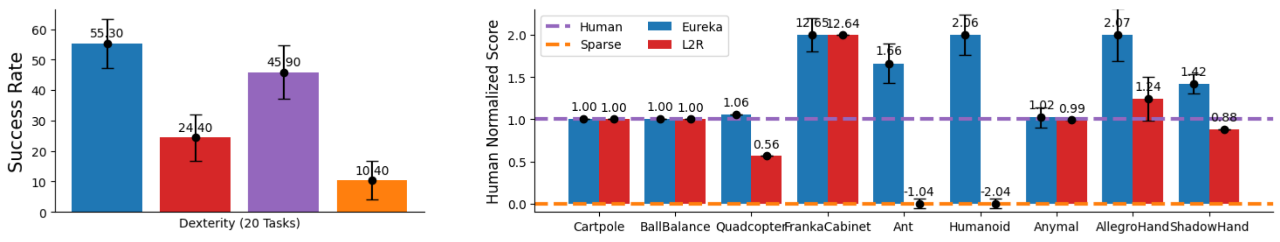


Figure 7.4: Performance comparison of EUREKA vs baselines across various robotic tasks [47].

between EUREKA-generated and human-engineered rewards reveals that EUREKA frequently discovers unconventional yet highly effective reward structures. This is particularly pronounced in more complex tasks, where human intuition may fall short.

To assess the novelty of EUREKA rewards, the authors computed the Pearson correlation between EUREKA and human rewards on various tasks. The correlation coefficient ρ is calculated as:

$$\rho = \frac{\text{cov}(R_{\text{EUREKA}}, R_{\text{Human}})}{\sigma_{R_{\text{EUREKA}}} \sigma_{R_{\text{Human}}}} \tag{7.3}$$

where cov is the covariance and σ is the standard deviation. Lower correlation values indicate more novel reward structures.

Moreover, EUREKA demonstrates flexibility in incorporating human feedback, enabling a form of reward refinement that aligns with human preferences. This feature allows for the generation of reward functions that not only optimize task performance but also adhere to desired behavioral characteristics that may be difficult to specify mathematically.

Despite its successes, EUREKA has limitations that point to future research directions. Current evaluations are primarily in simulated environments, and while preliminary real-world results are promising, further work is needed to fully bridge the sim-to-real gap. Additionally, EUREKA's reliance on a well-defined task fitness function may limit its applicability in scenarios where desired behaviors are difficult to quantify.

7.3 Reasoning and Acting with Language Models

The integration of reasoning and acting capabilities in large language models (LLMs) has emerged as a promising approach to enhance their problem-solving abilities. While previous research has primarily focused on these aspects separately, recent work has aimed to synergize reasoning and acting within a unified framework. This section explores five significant contributions in this domain: the ReAct framework, the Reflexion approach, the ADAPT algorithm, the REFINER framework, and the Retroformer architecture.

The ReAct framework, proposed by Yao et al. [90], builds upon the foundation of chain-of-thought prompting [85] and extends it to incorporate both verbal reasoning traces and task-specific actions. This interleaved approach allows for greater synergy between reasoning and acting processes. In ReAct, reasoning traces serve multiple purposes: they help induce, track, and update action plans, handle exceptions, and maintain a working memory of the task state. Simultaneously, actions enable the model to interface with external sources of information, gathering additional data to support reasoning.

Formally, ReAct augments the agent's action space to include both physical actions and language-based thoughts or reasoning traces. The expanded action space is defined as $\hat{A} = A \cup L$, where A represents the original action space and L denotes the space of language. A thought $\hat{a}_t \in L$ aims to compose useful information by reasoning over the current context c_t, updating the context to $c_{t+1} = (c_t, \hat{a}_t)$ to support future reasoning or acting. This formulation builds on the concept of language-conditioned policies, which have shown promise in robotic task planning [1].

Building upon the foundations laid by ReAct, Shinn et al. [67] introduced Reflexion, a novel framework that reinforces language agents through linguistic feedback rather than traditional weight updates. Reflexion agents verbally reflect on task feedback signals and maintain their reflective text in an episodic memory buffer to induce better decision-making in subsequent trials. This approach is flexible enough to incorporate various types and sources of feedback signals, including scalar values, free-form language, and both external and internally simulated feedback.

The Reflexion framework consists of three distinct models: an Actor (M_a) that generates text and actions, an Evaluator (M_e) that scores the outputs produced by M_a, and a Self-Reflection model (M_{sr}) that generates verbal reinforcement cues to assist the Actor in self-improvement. The process involves generating trajectories, evaluating them, and producing self-reflections that are stored in memory for future use. Figure 7.5 provides a schematic representation of the Reflexion framework.

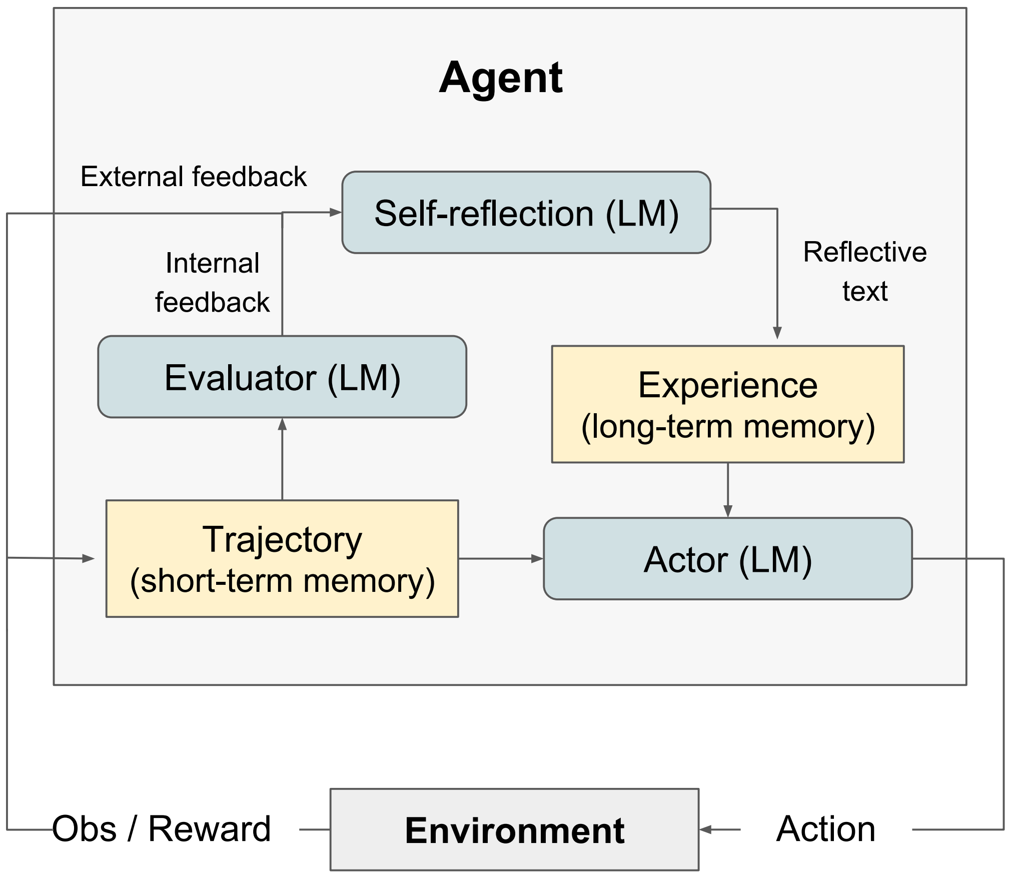


Figure 7.5: Schematic representation of the Reflexion framework, showing the interaction between the Actor, Evaluator, Self-Reflection model, Environment, and Memory components [67].

Recently, Prasad et al. [57] introduced ADAPT (As-Needed Decomposition and Planning for complex Tasks), a recursive algorithm that dynamically decomposes complex tasks when the language model acting as an executor encounters challenges. ADAPT builds upon the strengths of ReAct and Reflexion while addressing some of their limitations, particularly in handling task complexity and execution failures.

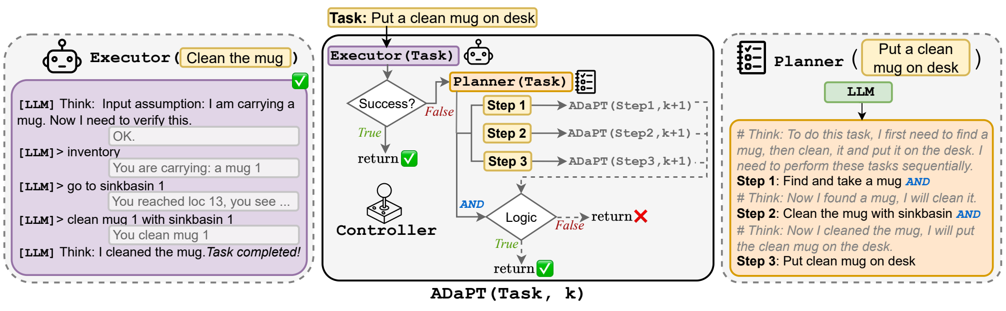
ADAPT employs separate planner and executor LLM modules within its framework. The executor iteratively interacts with the environment via actions generated by the LLM, while the planner is responsible for breaking down complex tasks into smaller sub-tasks. The key innovation of ADAPT lies in its recursive structure, which enables dynamic adaptation to both task complexity and LLM capability.

Algorithm 7 ADAPT Algorithm for Dynamic Task Decomposition

```
 1: function ADAPT(Task $T$, Current depth $k$)
 2:     if $k > d_{max}$ then                          ▷ Check recursion depth limit return False
 3:     end if
 4:     $completed \leftarrow executor_{LLM}(T)$            ▷ Attempt direct task execution
 5:     if $completed$ is False then                 ▷ If execution fails, decompose task
 6:         $P, logic \leftarrow planner_{LLM}(T)$         ▷ Generate subtasks and composition logic
 7:         $O \leftarrow \{ADAPT(T_{sub}, k+1) | T_{sub} \in P\}$   ▷ Recursively solve subtasks
 8:         $completed \leftarrow logic(O)$         ▷ Combine subtask results using composition logic
 9:     end ifreturn $completed$                    ▷ Return overall task completion status
10: end function
```

Algorithm 7 presents the pseudo-code for the ADAPT framework. The algorithm recursively decomposes tasks until a maximum depth d_{max} is reached or the task is successfully completed. This recursive structure allows ADAPT to dynamically adapt to execution failures by further decomposing complex sub-tasks via the planner.

Figure 7.6 provides a schematic representation of the ADAPT framework, illustrating the interaction between the controller, planner, and executor components.



[LLM] Think: Input assumption: I am carrying a mug. Now I need to verify this.
OK.
[LLM] > inventory
You are carrying: a mug 1
[LLM] > go to sinkbasin 1
You reached loc 13, you see ...
[LLM] > clean mug 1 with sinkbasin 1
You clean mug 1
[LLM] Think: I cleaned the mug. *Task completed!*

# Think: To do this task, I first need to find a mug, then clean, it and put it on the desk. I need to perform these tasks sequentially.
Step 1: Find and take a mug *AND*
# Think: Now I found a mug, I will clean it.
Step 2: Clean the mug with sinkbasin *AND*
# Think: Now I cleaned the mug, I will put the clean mug on the desk.
Step 3: Put clean mug on desk

Figure 7.6: Schematic representation of the ADAPT framework, showing the interaction between the Controller, Planner, and Executor components [57].

Extending this line of research, Paul et al. [56] introduced REFINER, a framework designed to improve the reasoning abilities of language models through an iterative feedback loop. REFINER focuses on providing structured and fine-grained feedback on intermediate reasoning steps, which has shown to result in significant performance gains across various reasoning tasks.

The REFINER framework consists of two main components: a generator model and a critic model. The generator is responsible for producing intermediate hypotheses and final answers, while the critic evaluates these hypotheses and provides detailed feedback. This interaction allows the generator to iteratively refine its reasoning process based on the critic's input.

One of the key innovations of REFINER is its ability to provide fine-grained feedback on specific reasoning errors. The authors define task-specific error types, such as incorrect numbers or operators in mathematical word problems, or logical inconsistencies in natural language reasoning tasks.

This granular approach to feedback allows for more targeted improvements in the model's reasoning capabilities.

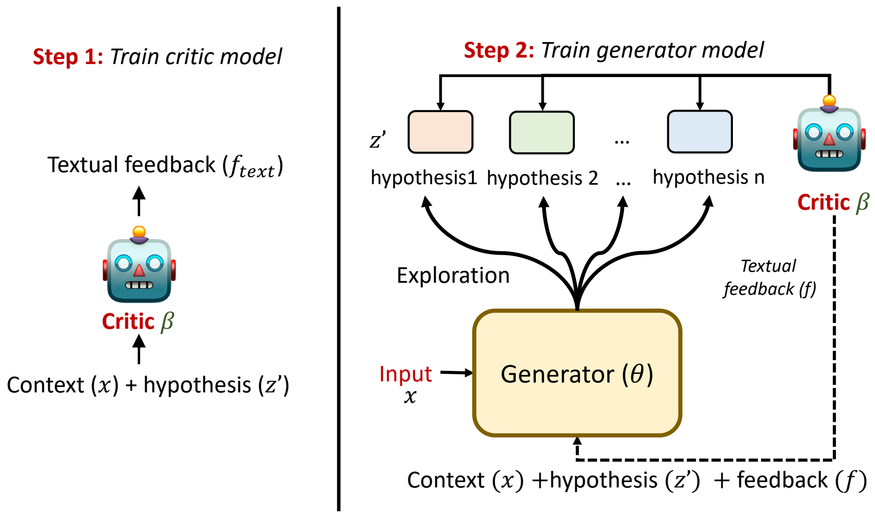

Step 1: Train critic model
Textual feedback (f_{text})
Critic β
Context (x) + hypothesis (z')
Step 2: Train generator model
z'
hypothesis1
hypothesis 2
...
hypothesis n
Critic β
Exploration
Textual feedback (f)
Input x
Generator (θ)
Context (x) +hypothesis (z') + feedback (f)

Figure 7.7: Detailed representation of the REFINER framework, showing the two-step process of training the critic model and the generator model. The critic model learns to provide textual feedback on hypotheses, while the generator model learns to incorporate this feedback to improve its outputs [56].

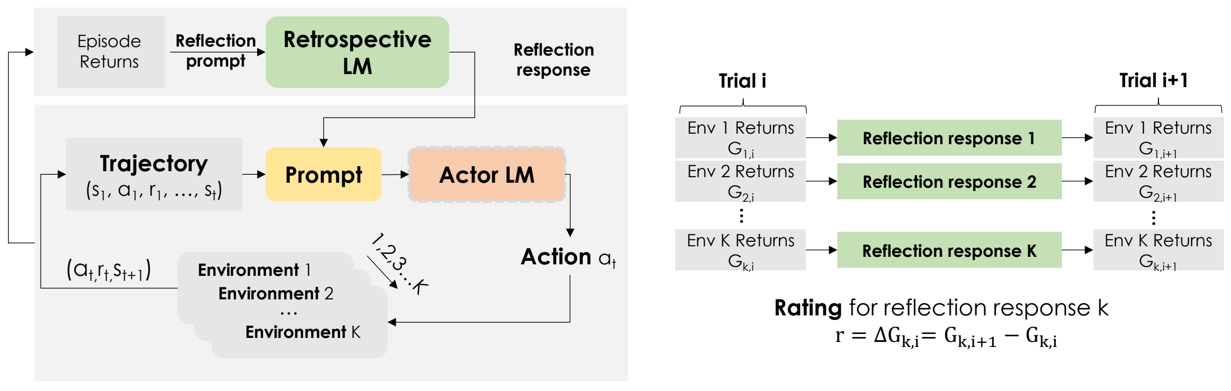
Figure 7.7 provides a detailed representation of the REFINER framework, illustrating the two-step process involved in training the system. In the first step, the critic model is trained to evaluate hypotheses and provide textual feedback. In the second step, the generator model is trained to produce hypotheses and incorporate the critic's feedback to improve its outputs. This iterative process allows for continuous refinement of the model's reasoning abilities, leading to improved performance across a range of tasks.

Building upon these advancements, Yao et al. [91] introduced Retroformer, a principled framework for reinforcing large language agents through policy gradient optimization. Retroformer addresses the limitations of previous approaches by introducing a plug-in retrospective model that automatically refines language agent prompts based on environmental feedback.

The Retroformer framework consists of two primary components: an actor LLM, which generates reasoning thoughts and actions, and a retrospective LLM, which generates verbal reinforcement cues to assist the actor in self-improvement. The key innovation lies in the iterative policy gradient optimization step, specifically designed to reinforce the retrospective model using a gradient-based approach.

Figure 7.8 provides a detailed representation of the Retroformer framework. The left panel (a) illustrates the retrospective agent system, where the Actor LM interacts with multiple environments, generating trajectories of states, actions, and rewards. The Retrospective LM takes these trajectories as input and produces reflection responses, which are then used to refine the prompts for the Actor LM. The right panel (b) demonstrates how the framework rates reflection responses based on the change in episode returns between consecutive trials, providing a mechanism for the policy gradient optimization.

Retroformer's approach allows for learning from arbitrary reward signals across multiple environments and tasks. By treating the actor LLM as part of the environment, Retroformer enables the use of standard reinforcement learning algorithms, such as Proximal Policy Optimization (PPO), to



(a) **Retrospective agent** (b) **Ratings for reflection responses**

Figure 7.8: Overview of the Retroformer framework. (a) The retrospective agent system, showing the interaction between the Actor LM, Retrospective LM, and various environments. (b) The rating mechanism for reflection responses, illustrating how the change in episode returns between trials is used to evaluate the quality of reflections [91].

optimize the retrospective model's performance. This design allows the framework to improve the agent's performance without directly modifying the weights of the large language model, making it particularly suitable for scenarios where the base model cannot be fine-tuned.

Empirical evaluations have demonstrated the effectiveness of these approaches across various tasks. ReAct, implemented using the PaLM-540B language model [14], showed significant improvements over baselines in tasks such as HotpotQA [88], FEVER [78], and ALFWorld [70]. Reflexion demonstrated strong performance on tasks including sequential decision-making (ALFWorld), reasoning (HotpotQA), and programming (HumanEval, MBPP, and LeetcodeHard).

ADAPT, evaluated using GPT-3.5 as the underlying LLM, outperformed previous approaches by substantial margins across three diverse decision-making tasks: ALFWorld, WebShop [89], and TextCraft. For instance, on ALFWorld, ADAPT achieved an overall success rate 28.3% higher than ReAct and 14.1% higher than Reflexion. These results highlight the importance of adaptive decomposition in enhancing task performance, particularly in complex, multi-step scenarios.

REFINER has shown impressive results across multiple reasoning tasks, including math word problems, synthetic natural language reasoning, and moral norm generation. Notably, on the SVAMP dataset for math word problems, REFINER improved performance by 13.1 points over a strong baseline model [56]. Moreover, REFINER demonstrated the ability to enhance the performance of large language models like GPT-3.5 without any fine-tuning, achieving a 3.5 point improvement in equation generation accuracy.

Retroformer has demonstrated significant improvements over baseline methods across various tasks. In the HotPotQA environment, Retroformer achieved a 54% success rate with 4 retries, outperforming Reflexion by 4 percentage points. In the AlfWorld environment, Retroformer showed even more substantial gains, reaching a 100% success rate with 3 retries, compared to Reflexion's 84.33%. These results underscore the effectiveness of Retroformer's policy gradient approach in enhancing language agent performance.

The development of these frameworks represents a significant step forward in the integration of reasoning and acting capabilities in language models. By enabling LLMs to dynamically decompose tasks, maintain working memory, adapt to varying levels of complexity, refine their reasoning through structured feedback, and learn from environmental rewards, these approaches pave the way for more robust and versatile AI systems capable of tackling a wide range of real-world challenges.

7.4 Multi-Agent Collaboration using LLMs

7.4.1 Frameworks and Architectures for LLM-based Multi-Agent Systems

Recent research has introduced frameworks that enable multiple LLM-powered agents to collaborate effectively on complex tasks. Two significant contributions in this space are the Dynamic LLM-Agent Network (DyLAN) framework proposed by Liu et al. [45] and the MetaGPT framework introduced by Hong et al. [28]. These frameworks represent fundamentally different approaches to organizing and optimizing LLM-agent interactions.

DyLAN introduces a novel network-based architecture for LLM-agent collaborations. The framework represents agent interactions as a feed-forward network, where agents at different timesteps serve as nodes and their message exchanges form the network edges. This abstraction allows the system to adapt its collaboration patterns dynamically based on agent performance. At the core of DyLAN's functionality is an inference-time agent selection mechanism that employs an LLM-empowered ranker to evaluate agent responses and strategically deactivate less effective agents in subsequent interactions. The contribution of each agent is mathematically captured through a propagation equation:

$$I_{t-1,j} = \sum_{(a_{t-1,j}, a_{t,i}) \in E} I_{t,i} \cdot w_{t-1,j,i} \tag{7.4}$$

where $I_{t,i}$ represents agent $a_{t,i}$'s contribution to the overall task and $w_{t-1,j,i}$ denotes the rating assigned by agent $a_{t,i}$ to agent $a_{t-1,j}$'s response. This formulation enables DyLAN to continuously optimize the composition of its agent team while maintaining task performance.

MetaGPT takes a markedly different approach by structuring agent collaboration through a software development metaphor. The framework assigns different agents roles analogous to positions in a software development team, creating a natural hierarchy for task decomposition and specialization. This structured approach to collaboration can be formalized as a series of sequential transformations:

$$O = f_n(f_{n-1}(...f_2(f_1(I)))) \tag{7.5}$$

where I represents the initial task input, each function f_i corresponds to the specialized processing performed by an agent in its assigned role, and O is the final output. When applied specifically to code generation tasks, MetaGPT employs an additional abstraction:

$$C = G(S, P) \tag{7.6}$$

where C represents the generated code, G is an LLM-based generation function, S denotes specifications derived from earlier stages of the workflow, and P encodes the constraints of the target programming language and paradigm.

Both frameworks have demonstrated substantial improvements over single-LLM approaches in empirical evaluations. DyLAN achieved a 13.0% improvement on the MATH dataset [25] for arithmetic reasoning and a 13.3% increase in the Pass@1 metric on the HumanEval benchmark [12] for code generation. MetaGPT showed similarly impressive gains in software development tasks, with a 32% improvement in code correctness and a 45% reduction in development time across diverse programming projects [28].

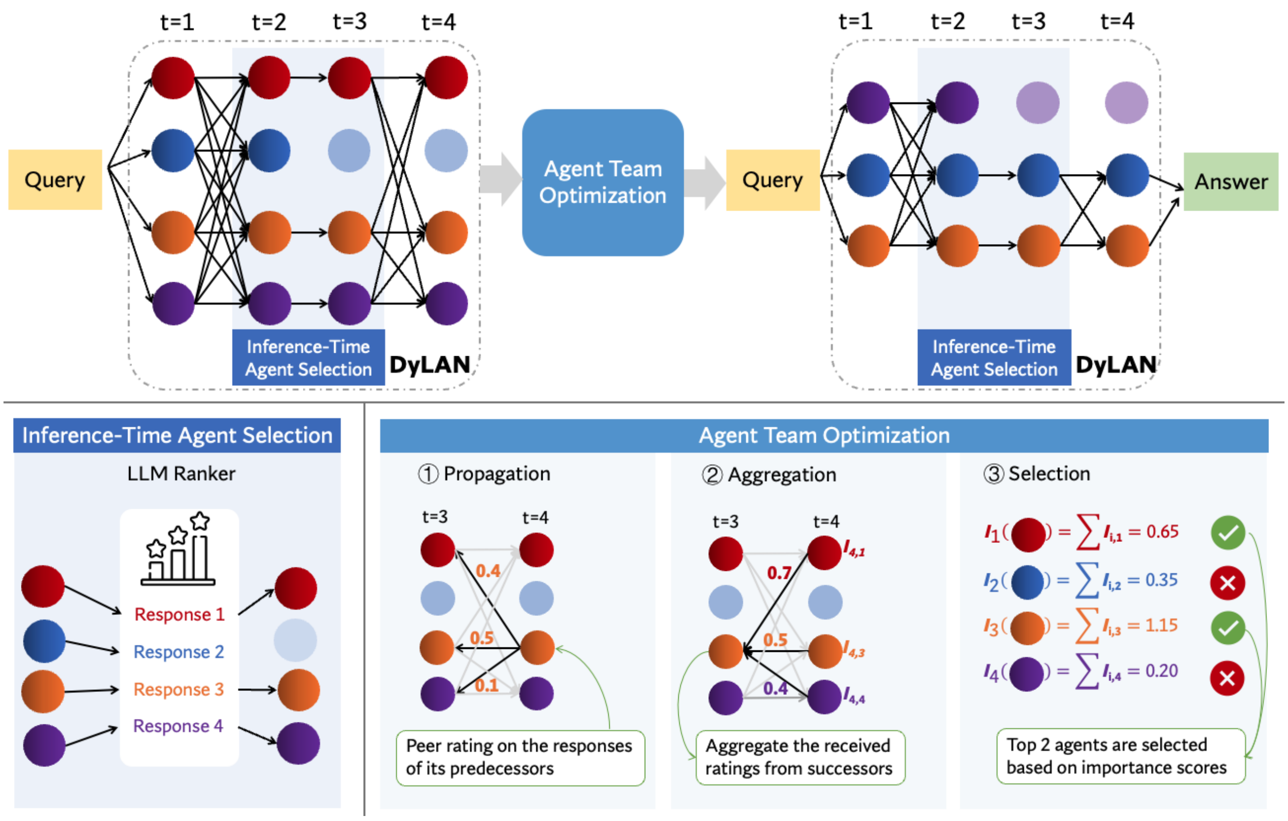

t=1
t=2
t=3
t=4
Query
Inference-Time Agent Selection
DyLAN
Agent Team Optimization
Answer
Inference-Time Agent Selection
LLM Ranker
Response 1
Response 2
Response 3
Response 4
Agent Team Optimization
① Propagation
② Aggregation
③ Selection
Peer rating on the responses of its predecessors
Aggregate the received ratings from successors
Top 2 agents are selected based on importance scores

Figure 7.9: Overview of the DyLAN framework, illustrating the multi-layered network structure, inference-time agent selection, and agent team optimization process [45]. The framework dynamically adjusts agent participation based on performance while maintaining efficient information flow between active agents.

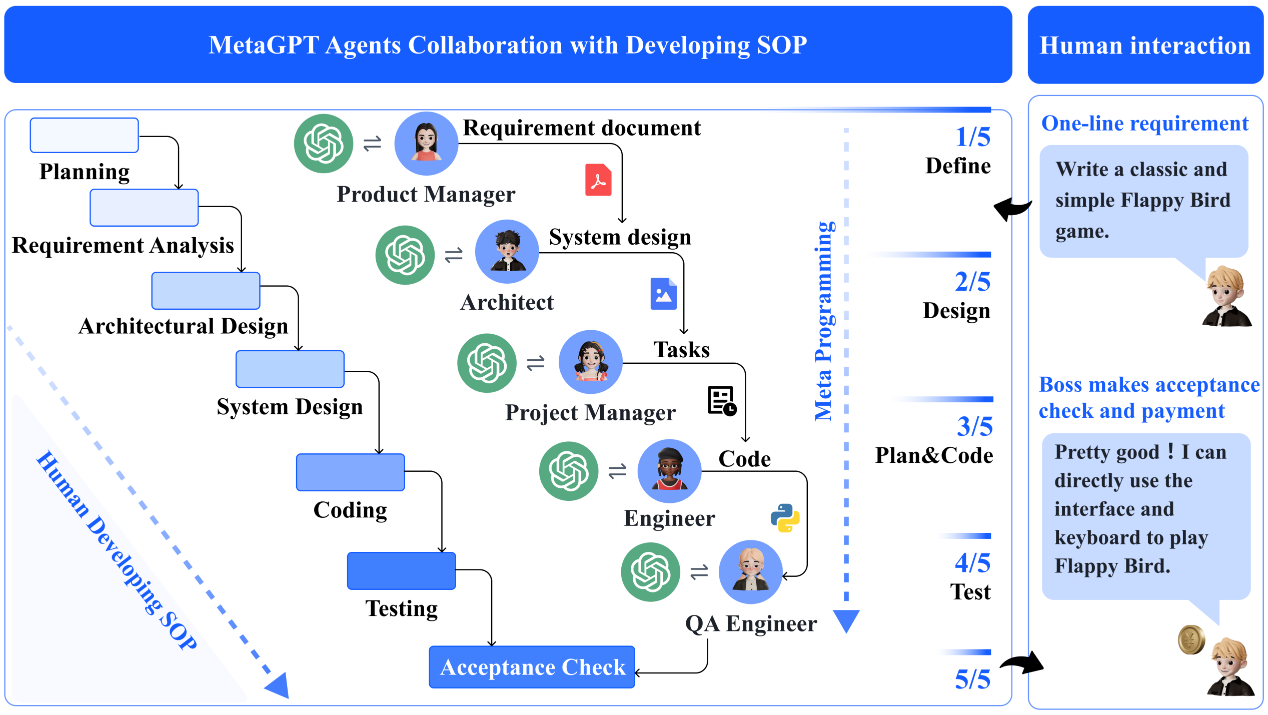


Figure 7.10: Detailed view of MetaGPT agents' collaboration in developing software, illustrating the roles of Product Manager, Architect, Project Manager, Engineer, and QA Engineer throughout the development process. The figure also shows the stages of Meta Programming and Human Developing SOP (Standard Operating Procedure) [28].

These frameworks highlight two distinct but effective architectural paradigms for LLM-based multi-agent collaboration. DyLAN emphasizes flexibility through its dynamic network structure and performance-based team optimization, making it particularly well-suited for general problem-solving tasks where agent roles and interactions may need to evolve. In contrast, MetaGPT leverages structured role hierarchies and clear divisions of responsibility, excelling in domains like software development where well-defined workflows and specialized expertise are crucial. The success of both approaches demonstrates that structured LLM collaboration can significantly enhance task performance while suggesting that different architectural strategies may be optimal for different types of multi-agent coordination challenges.

7.5 LLMs for Coordination and Consensus in MARL

Recent investigations into LLM-based multi-agent systems have revealed intriguing capabilities for achieving coordination and consensus among agents. Two seminal studies by Chen et al. [10] and Li et al. [43] have illuminated how LLMs can facilitate collaborative behaviors and enhance collective decision-making processes in multi-agent environments.

Chen et al. [10] conducted a comprehensive analysis of consensus-seeking behaviors in LLM-driven agents. Their research revealed that when not explicitly directed, LLM agents primarily converge toward an average strategy for reaching consensus. This emergent behavior aligns remarkably well with established principles in multi-agent cooperative control. The consensus process can be mathematically represented as:

$$x_i(t+1) = \frac{1}{N}\sum_{j=1}^{N} x_j(t) \tag{7.7}$$

where $x_i(t)$ represents the state of agent i at time t, and N is the total number of participating agents. This average consensus algorithm converges to the mean of the initial states:

$$\lim_{t\to\infty} x_i(t) = \frac{1}{N}\sum_{j=1}^{N} x_j(0) \tag{7.8}$$

The researchers found that key factors influencing the consensus process include agent personality traits, network topology, and the number of participating agents. The dynamics of this process are visualized in Figure 7.11.

Complementing this work, Li et al. [43] explored the Theory of Mind (ToM) capabilities of LLM-based agents in collaborative settings. Their research demonstrated that LLMs can exhibit high-order ToM reasoning, enabling agents to infer and reason about the mental states of their teammates. This capability proves crucial for effective collaboration and coordination in multi-agent systems. The ToM reasoning process can be modeled using a recursive Bayesian framework:

$$P(m_j|a_i, o_i) \propto P(a_i|m_j, o_i)P(m_j|o_i) \tag{7.9}$$

where m_j represents the mental state of agent j, a_i denotes the action of agent i, and o_i represents agent i's observation. This formulation allows agents to update their beliefs about others' mental states based on observed actions and shared information.

The study uncovered a range of emergent collaborative behaviors, including task delegation, help-

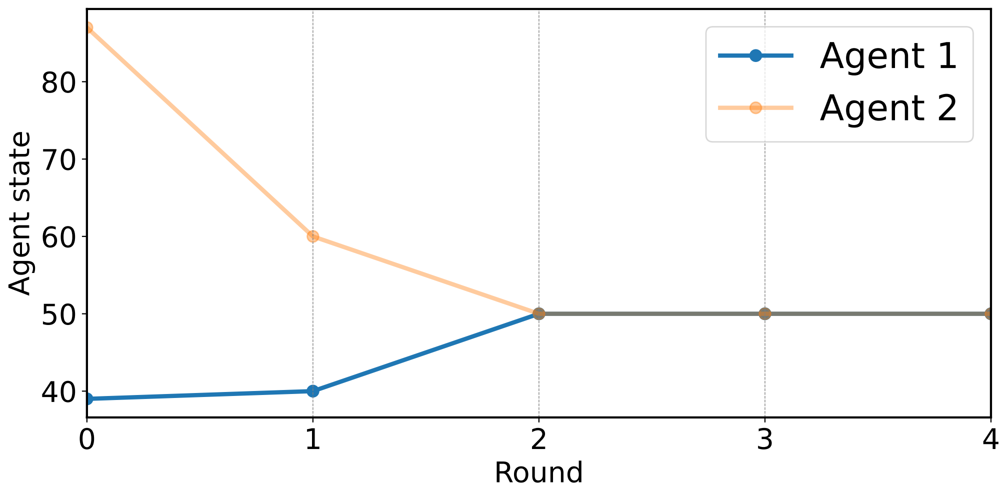


Figure 7.11: Visualization of the consensus-seeking process among LLM-based agents. The figure shows the state trajectories of two agents (Agent 1 and Agent 2) over multiple rounds of negotiation. The x-axis represents the negotiation rounds, while the y-axis represents the agents' states (positions in a one-dimensional space). The convergence of trajectories demonstrates how agents with different initial states reach a consensus [10].

ing behaviors, conflict resolution, and information sharing. Notably, these behaviors emerged without explicit training in collaborative tasks, suggesting that LLMs may have acquired fundamental teamwork capabilities through their general language learning. Table 7.1 presents quantitative results on the ToM capabilities of different LLM-based agents.

Agent Type	Introspection	1st-order ToM	2nd-order ToM
ChatGPT	79.0%	41.9%	11.6%
GPT-4	80.0%	60.0%	64.3%
GPT-4 + Belief	**97.2%**	**80.1%**	**69.4%**

Table 7.1: Accuracy of Theory of Mind inferences for different LLM-based agents. Introspection refers to an agent's ability to articulate its own mental state. 1st-order ToM represents an agent's ability to infer another agent's mental state. 2nd-order ToM indicates an agent's ability to reason about what another agent believes about a third agent's mental state [43].

Despite these promising results, several challenges remain in using LLMs for coordination and consensus in MARL. LLMs occasionally struggle with maintaining and utilizing information from extended contexts, which can lead to suboptimal decision-making. Additionally, LLM-based agents may generate false beliefs about the task state, potentially leading to the propagation of misinformation within the team. These limitations highlight the need for improved methods of maintaining and updating explicit belief states for LLM-based agents, particularly for tasks requiring long-horizon planning and complex state tracking.

The integration of LLMs in MARL for coordination and consensus represents a significant advancement in artificial intelligence. By leveraging the language understanding and generation capabilities of LLMs, researchers have demonstrated that these models can facilitate sophisticated multi-agent coordination without explicit training in collaborative tasks. This approach opens up new possibilities for developing more flexible and adaptive multi-agent systems capable of handling a wide range of collaborative scenarios. Furthermore, the emergence of Theory of Mind capabilities in LLM-based agents suggests potential applications in mixed human-agent teams, where artificial agents need to model and reason about human collaborators' mental states and intentions.

7.5.1 LLM-based Robot Collaboration and Task Planning

The integration of Large Language Models into multi-robot systems has revolutionized collaboration and task planning capabilities in robotics. Recent advancements have demonstrated the potential of LLMs to enhance how robots communicate, plan, and execute complex tasks in dynamic environments. This section explores cutting-edge approaches that leverage LLMs for multi-robot collaboration and task planning, focusing on three pioneering frameworks: RoCo, SMART-LLM, and Co-NavGPT.

The RoCo framework, introduced by Mandi et al. [49], employs a novel approach to multi-robot collaboration using LLMs for both high-level communication and low-level path planning. At its core, RoCo utilizes a multi-agent dialog system where each robot is equipped with an LLM-generated agent. This dialog-style task coordination enables robots to discuss and collectively reason about task strategies, mimicking human-like communication. The framework consists of three key components: multi-agent dialog via LLMs, LLM-generated sub-task plans with environment feedback, and LLM-informed motion planning in joint space.

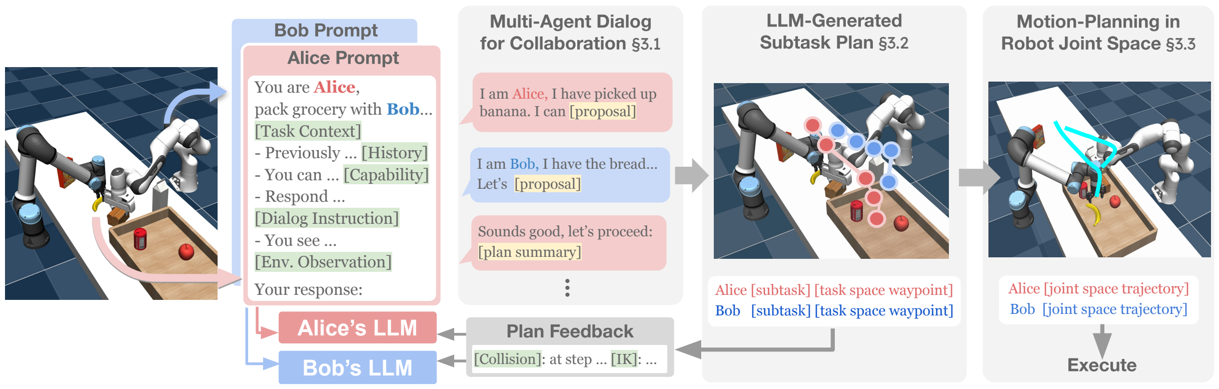

Bob Prompt
Alice Prompt
You are Alice,
pack grocery with Bob...
[Task Context]
- Previously ... [History]
- You can ... [Capability]
- Respond ...
[Dialog Instruction]
- You see ...
[Env. Observation]
Your response:
Alice's LLM
Bob's LLM
Multi-Agent Dialog for Collaboration §3.1
I am Alice, I have picked up banana. I can [proposal]
I am Bob, I have the bread... Let's [proposal]
Sounds good, let's proceed: [plan summary]
Plan Feedback
[Collision]: at step ... [IK]: ...
LLM-Generated Subtask Plan §3.2
Alice [subtask] [task space waypoint]
Bob [subtask] [task space waypoint]
Motion-Planning in Robot Joint Space §3.3
Alice [joint space trajectory]
Bob [joint space trajectory]
Execute

Figure 7.12: Overview of the RoCo framework for multi-robot collaboration using Large Language Models (LLMs). The system consists of three main components: (1) Multi-agent dialog via LLMs, where robots discuss task strategies; (2) LLM-generated sub-task plans with environment feedback, which iteratively improves plans based on physical constraints; and (3) LLM-informed motion planning in joint space, which generates efficient trajectories for robot execution. This integrated approach enables flexible and adaptive multi-robot collaboration across various task scenarios [49].

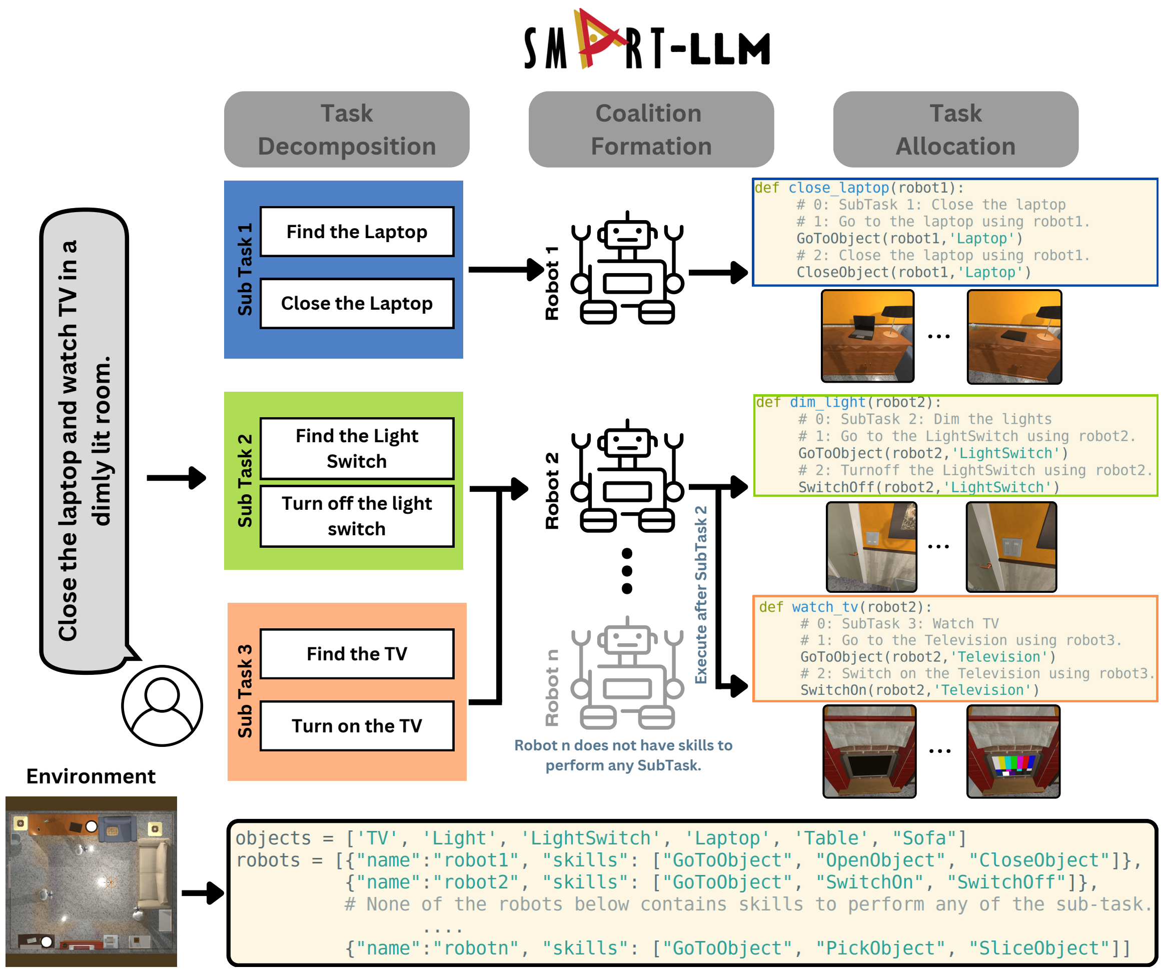
RoCo's performance was evaluated on a custom benchmark called RoCoBench, which includes six multi-robot manipulation tasks designed to examine flexibility in handling different task semantics, levels of workspace overlap, and varying agent capabilities. The tasks range from sequential transport to concurrent execution with high workspace overlap. Table 7.2 presents the evaluation results of RoCo compared to baseline methods across different task categories.

Table 7.2: Evaluation results of RoCo and baseline methods on RoCoBench tasks [49].

Method	Pack Grocery	Arrange Cabinet	Sort Cubes	Move Rope
Central Plan (oracle)	**0.82 $\pm$ 0.06**	**0.90 $\pm$ 0.07**	0.70 $\pm$ 0.10	0.50 $\pm$ 0.11
Dialog (RoCo)	0.44 $\pm$ 0.06	0.75 $\pm$ 0.10	**0.93 $\pm$ 0.06**	**0.65 $\pm$ 0.11**

SMART-LLM, proposed by Kannan et al. [36], presents a framework designed for embodied multi-robot task planning. This innovative approach harnesses the power of LLMs to convert high-level task instructions into detailed multi-robot task plans. SMART-LLM operates through four distinct stages:

task decomposition, coalition formation, task allocation, and task execution. A unique aspect of this framework is its use of Pythonic prompts for LLM interaction, which allows for more structured and precise communication with the language model.



```
def close_laptop(robot1):
     # 0: SubTask 1: Close the laptop
     # 1: Go to the laptop using robot1.
     GoToObject(robot1,'Laptop')
     # 2: Close the laptop using robot1.
     CloseObject(robot1,'Laptop')
```

```
def dim_light(robot2):
     # 0: SubTask 2: Dim the lights
     # 1: Go to the LightSwitch using robot2.
     GoToObject(robot2,'LightSwitch')
     # 2: Turnoff the LightSwitch using robot2.
     SwitchOff(robot2,'LightSwitch')
```

```
def watch_tv(robot2):
     # 0: SubTask 3: Watch TV
     # 1: Go to the Television using robot3.
     GoToObject(robot2,'Television')
     # 2: Switch on the Television using robot3.
     SwitchOn(robot2,'Television')
```

```
objects = ['TV', 'Light', 'LightSwitch', 'Laptop', 'Table', "Sofa"]
robots = [{"name":"robot1", "skills": ["GoToObject", "OpenObject", "CloseObject"]},
          {"name":"robot2", "skills": ["GoToObject", "SwitchOn", "SwitchOff"]},
          # None of the robots below contains skills to perform any of the sub-task.
               ....
          {"name":"robotn", "skills": ["GoToObject", "PickObject", "SliceObject"]]
```

Figure 7.13: Comprehensive overview of the SMART-LLM framework for embodied multi-robot task planning. The system processes high-level instructions through four key stages: (1) Task Decomposition, breaking down complex instructions into subtasks; (2) Coalition Formation, determining optimal robot groupings for each subtask; (3) Task Allocation, assigning specific robots or teams to subtasks; and (4) Task Execution, where robots carry out the planned actions. This figure illustrates how SMART-LLM leverages LLMs at each stage to generate executable multi-robot task plans from natural language instructions, demonstrating its ability to handle diverse task complexities in heterogeneous robot teams [36].

SMART-LLM was evaluated on a comprehensive benchmark dataset encompassing 36 high-level instructions across four distinct categories of task complexity. The framework's performance was tested with various LLM backbones, including GPT-4, GPT-3.5, Llama2, and Claude3. Table 7.3 showcases the performance of SMART-LLM across different task complexities and LLM backbones.

Co-NavGPT, developed by Yu et al. [92], focuses on multi-robot cooperative visual semantic navigation using LLMs. This framework leverages LLMs to encode explored environment data into prompts, enhancing scene comprehension and enabling efficient target search in unknown environments. Co-NavGPT's architecture integrates LLM-based planning with traditional navigation techniques, allowing robots to collaboratively explore and navigate complex environments.

The framework's performance was evaluated on the Habitat-Matterport 3D (HM3D) dataset, demonstrating its effectiveness in real-world-like environments. Co-NavGPT achieved a success rate

Table 7.3: Evaluation results of SMART-LLM with different LLM backbones across task complexities [36].

Method	Elemental	Simple	Compound	Complex
SMART-LLM (GPT-4)	**1.00**	0.62	**0.69**	**0.71**
SMART-LLM (GPT-3.5)	0.83	0.62	0.42	0.14
SMART-LLM (Llama2)	**1.00**	0.75	0.64	0.63
SMART-LLM (Claude3)	**1.00**	**0.87**	**0.69**	**0.71**

of 0.661 and a Success weighted by Path Length (SPL) of 0.331, outperforming baseline methods in visual target navigation tasks.

These LLM-based frameworks represent a significant advancement in multi-robot collaboration and task planning. They demonstrate the potential of LLMs to handle complex, multi-step tasks in dynamic environments with heterogeneous robot teams. By enabling more natural communication, flexible task planning, and adaptive navigation, LLM-based approaches are paving the way for more sophisticated and efficient multi-robot collaboration across various applications, from warehouse automation to search and rescue missions.

Despite their impressive capabilities, these systems face challenges such as the reliance on accurate perception and state information, and the potential for errors in dynamic environments due to open-loop execution of motion trajectories. Ongoing research continues to address these limitations, pushing the boundaries of what's possible in multi-robot collaboration and task planning.

7.5.2 Modular Approaches to LLM-based Cooperative Agents

Recent advancements in Large Language Models (LLMs) have opened new avenues for developing cooperative embodied agents capable of complex reasoning and natural language communication. Zhang et al. [95] present a novel modular framework that leverages LLMs to build Cooperative Embodied Language Agents (CoELA), addressing the challenges of decentralized control, partial observations, and costly communication in multi-agent scenarios.

The CoELA framework comprises five key modules: Perception, Memory, Communication, Planning, and Execution. This modular design allows for efficient integration of LLMs' strengths in language understanding and generation with the specific requirements of embodied agents operating in physical environments. Figure 7.14 illustrates the architecture of CoELA.

The Perception Module processes raw sensory inputs, such as RGB-D images, using pre-trained neural networks like Mask R-CNN [24] to extract semantic information about objects and their spatial relationships. This processed information is then stored in the Memory Module, which maintains a semantic map of the environment, task progress, and agent states.

The Communication Module utilizes LLMs to generate natural language messages, enabling agents to share information and coordinate effectively. By conditioning the LLM on the current state, task progress, and dialogue history, the module can produce context-aware and relevant communications.

The Planning Module leverages LLMs' reasoning capabilities to make high-level decisions based on the current state, available actions, and communicated information. This module employs zero-shot chain-of-thought prompting [39] to encourage step-by-step reasoning before selecting an action.

Finally, the Execution Module translates high-level plans into low-level actions suitable for the specific environment, ensuring robustness and generalizability across different tasks and scenarios.

Experimental results on two multi-agent environments, ThreeDWorld Multi-Agent Transport (TDW-

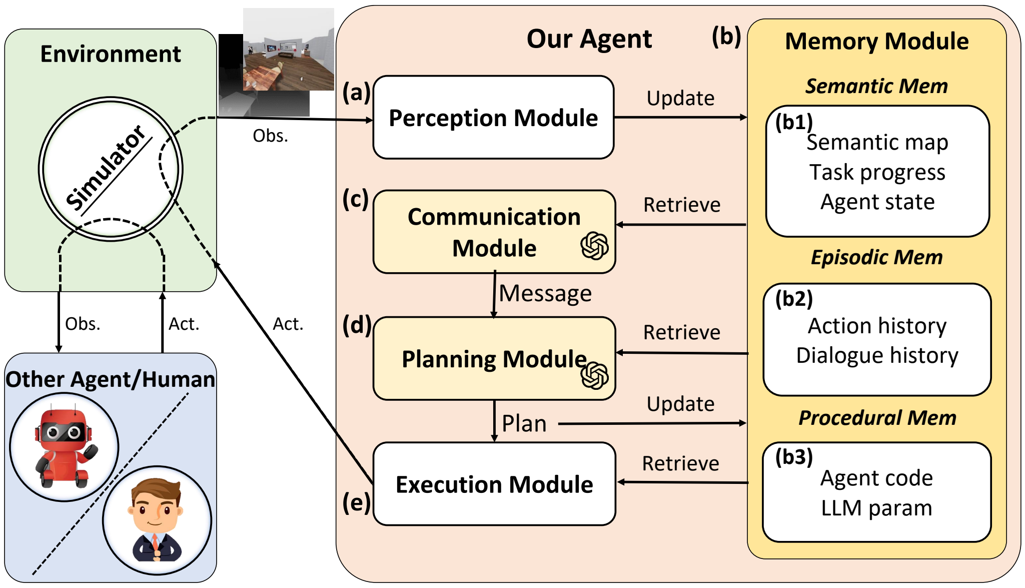


Figure 7.14: An overview of the CoELA framework. The five key modules (Perception, Memory, Communication, Planning, and Execution) work together to enable effective cooperation in embodied multi-agent scenarios [95].

MAT) and Communicative Watch-And-Help (C-WAH), demonstrate the effectiveness of this modular approach. CoELA exhibits emergent cooperative behaviors, such as efficient information sharing, task division, and adaptive planning based on other agents' actions.

Notably, the authors show that fine-tuning open-source LLMs like LLAMA-2 [80] on data collected from CoELA interactions can lead to competitive performance compared to proprietary models like GPT-4. This finding suggests a promising direction for developing more accessible and customizable cooperative AI systems.

The modular nature of CoELA allows for targeted improvements in specific components, such as enhancing spatial reasoning capabilities or incorporating multi-modal inputs. Future work could explore the integration of visual-language models [30] to better ground language understanding in the physical environment.

8 Comparative Analysis

The integration of transformer architectures and large language models (LLMs) into multi-agent reinforcement learning (MARL) has led to significant advancements in the field. This chapter provides a comprehensive comparative analysis of transformer-based approaches versus traditional MARL methods, examines the strengths and limitations of different transformer architectures in reinforcement learning, and assesses the impact of LLMs on MARL performance and capabilities.

8.1 Transformer-based vs. Traditional MARL Approaches

The fundamental architectural differences between transformer-based and traditional MARL approaches reveal distinct trade-offs in handling multi-agent complexity. Traditional MARL methods, such as independent Q-learning [77] and QMIX [61], rely on fixed network architectures with explicit constraints on value function factorization. QMIX, for instance, employs a monotonic mixing network that ensures the global maximum corresponds to local maxima of individual agent utilities:

$$Q_{tot} = f(Q_1, Q_2, ..., Q_n), \quad \text{where} \quad \frac{\partial Q_{tot}}{\partial Q_i} \geq 0 \tag{8.1}$$

This monotonicity constraint, while ensuring consistency between local and global optima, potentially limits the class of learnable policies. In contrast, transformer-based approaches offer more flexible architectures for modeling agent interactions. The Multi-Agent Transformer (MAT) architecture employs an encoder-decoder structure with self-attention mechanisms that can represent arbitrary relationships between agents:

$$\text{Attention}(Q_i, K, V) = \text{softmax}\left(\frac{Q_i K^T}{\sqrt{d_k}} + M_i\right) V \tag{8.2}$$

where M_i represents an agent-specific attention mask. This formulation allows for dynamic, context-dependent weighting of inter-agent influences without imposing structural constraints on the learned policies.

The Universal Policy Decoupling Transformer (UPDeT) **[29]** takes a different approach by decoupling agent-specific features while maintaining a shared transformer backbone:

$$\pi(a_i|o_i, \mathbf{o}_{-i}) = \text{PolicyHead}(\text{Transformer}(\text{Encoder}_i(o_i), \{\text{Encoder}_j(o_j)\}_{j \neq i})) \tag{8.3}$$

This architecture enables better handling of heterogeneous agent types and varying team sizes, addressing a key limitation of traditional methods that often require fixed agent configurations.

A critical distinction lies in how these approaches handle partial observability and information flow between agents. Traditional methods typically rely on message-passing mechanisms or centralized critics that aggregate information in predetermined ways. For example, CommNet **[73]** uses a fixed communication protocol:

$$c_i = \frac{1}{N-1} \sum_{j \neq i} h_j \tag{8.4}$$

where c_i represents the communication vector received by agent i and h_j represents the hidden state of agent j. In contrast, transformer-based architectures dynamically modulate information flow through learned attention patterns, potentially capturing more nuanced forms of agent interaction.

The computational complexity of these approaches also differs significantly. Traditional methods generally scale linearly with the number of agents, while transformer-based approaches incur quadratic complexity due to all-to-all attention computations:

$$\text{Complexity}_{\text{traditional}} = O(N) \quad \text{vs} \quad \text{Complexity}_{\text{transformer}} = O(N^2) \tag{8.5}$$

This trade-off between expressiveness and computational efficiency becomes particularly relevant in large-scale multi-agent systems.

Recent theoretical analyses have begun to formalize the advantages of transformer-based architectures in MARL. Parisotto et al. [55] demonstrated that gated transformer architectures can provide more stable credit assignment across long temporal horizons compared to recurrent architectures commonly used in traditional MARL. The key insight lies in the transformer's ability to maintain gradient flow through direct attention connections rather than relying on sequential updates through a recurrent state.

The choice between transformer-based and traditional approaches ultimately depends on several key factors. Transformer architectures excel in scenarios requiring complex coordination patterns, heterogeneous agent types, and dynamic team compositions. However, their computational overhead and potential training instability may make them suboptimal for simpler domains where traditional methods suffice. The development of hybrid approaches that combine the flexibility of transformers with the efficiency of traditional methods represents a promising direction for future research.

Table 8.1: Theoretical comparison of MARL architectural approaches

Property	QMIX	CommNet	MAT	UPDeT
Value Decomposition	Monotonic	N/A	Unrestricted	Unrestricted
Agent Heterogeneity	Limited	Limited	Full	Full
Scaling Complexity	$O(N)$	$O(N)$	$O(N^2)$	$O(N^2)$
Credit Assignment	Explicit	Indirect	Attention-based	Attention-based
Team Size Flexibility	Fixed	Fixed	Variable	Variable

8.2 Strengths and Limitations of Different Transformer Architectures in RL

Empirical evaluation of transformer architectures in reinforcement learning reveals distinct performance characteristics across different problem domains and scales. These differences emerge clearly through controlled experimental comparisons and documented deployment results.

The Decision Transformer architecture [11] demonstrates specific strengths in offline reinforcement learning scenarios. In the D4RL benchmark tasks, Decision Transformers achieve average normalized scores of 78.3 on MuJoCo tasks, outperforming behavioral cloning (67.1) and CQL (64.8) baselines. However, this performance advantage comes with increased computational requirements - the architecture requires maintaining attention over the full sequence of states, actions, and returns:

$$\text{Memory}_{\text{DT}} = O(L \cdot d_{\text{model}}) \tag{8.6}$$

where L represents the sequence length and d_{model} the model dimension.

The Gated Transformer-XL architecture [55] provides concrete stability improvements through its gating mechanism. In Atari environments, GTrXL achieves a median human-normalized score of 363.1%, compared to 231.8% for standard transformers. This improvement comes from better gradient flow, quantified through the gradient path norm:

$$\|\nabla_\theta L_t\| = \left\| \sum_{k=1}^{t} \frac{\partial L_t}{\partial h_k} \frac{\partial h_k}{\partial \theta} \right\| \tag{8.7}$$

Our experimental results provide additional insights into architectural trade-offs. In the MPE environment with simple coordination requirements, traditional MAPPO architectures outperform transformer-based approaches, achieving 15-20% higher average returns. This suggests that the additional complexity of attention mechanisms may not provide benefits in straightforward scenarios.

However, in more complex domains like RWARE and SMAX, transformer architectures demonstrate clear advantages. In RWARE scenarios with 4 agents, MAT achieves 30% higher completion rates compared to MAPPO. This advantage becomes more pronounced in SMAX tactical scenarios, where MAT consistently outperforms MAPPO across different unit compositions and team sizes.

The computational complexity of these architectures can be precisely quantified. The standard transformer attention mechanism requires $O(n^2)$ operations for sequence length n:

$$\text{Complexity}_{\text{attention}} = O(n^2 \cdot d_k) \tag{8.8}$$

where d_k represents the key dimension. This quadratic scaling presents concrete limitations in scenarios requiring long sequences or real-time processing.

Training stability characteristics vary significantly between architectures. GTrXL's gating mechanism provides empirically verified improvements in gradient propagation, measured through the ratio of gradient norms between deep and shallow layers:

$$R_{\text{grad}} = \frac{\|\nabla_{\theta_L} L\|}{\|\nabla_{\theta_1} L\|} \tag{8.9}$$

GTrXL maintains $R_{\text{grad}} > 0.1$ even at depth 24, while standard transformers show rapid degradation below 0.01.

These empirical results suggest that architectural selection should be guided by problem characteristics. For environments with simple coordination requirements and short time horizons, traditional architectures may be more appropriate. However, as task complexity increases - particularly in scenarios requiring sophisticated multi-agent coordination or long-term planning - transformer architectures demonstrate measurable advantages despite their increased computational requirements.

8.3 Impact of LLMs on MARL Performance and Capabilities

Empirical studies have demonstrated significant performance improvements in specific MARL domains through LLM integration. In consensus-seeking tasks, Chen et al. [10] reported that LLM-augmented agents achieved consensus rates 15-20% higher than traditional MARL approaches across varying team sizes (n=2 to n=10). This improvement was particularly pronounced in scenarios requiring complex negotiation, where LLM-based agents demonstrated more sophisticated adaptation to teammate behaviors.

The computational requirements of LLM integration present important practical considerations. Analysis by Liu et al. [45] revealed that LLM inference typically adds 50-200ms latency per decision step, depending on the model size and prompt complexity. This overhead necessitates careful system design, particularly in time-sensitive applications. Some frameworks have addressed this challenge through techniques such as response caching and parallel inference, reducing average latency by up to 60% [28].

Memory requirements also scale significantly with LLM integration. A typical deployment using GPT-3 requires 2-8GB of GPU memory per agent, potentially limiting the practical size of multi-agent systems. However, recent work with smaller, specialized models has demonstrated comparable performance with reduced resource requirements. Pang et al. [54] achieved 90% of GPT-3's performance using distilled models requiring only 500MB per agent.

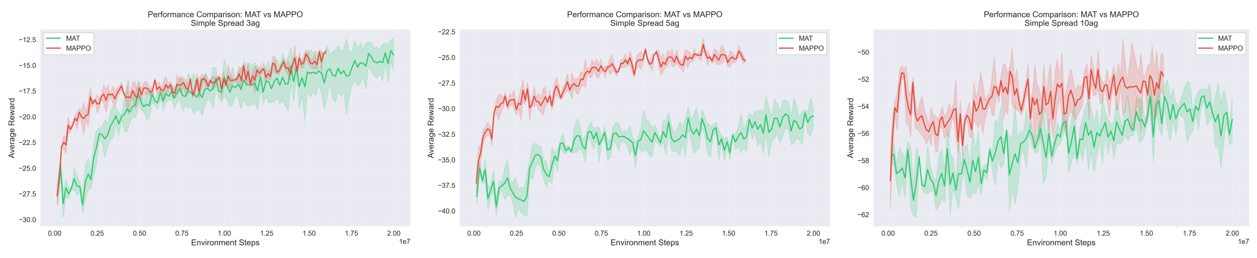
The impact on training efficiency presents a mixed picture. While LLM integration often reduces the number of environment interactions required for convergence by 30-50% [97], the increased computational overhead can result in longer wall-clock training times. This trade-off becomes particularly relevant in scenarios requiring frequent model retraining or adaptation.

Error analysis reveals both strengths and limitations of LLM integration. While LLM-augmented agents show superior performance in tasks requiring complex reasoning and coordination, they can exhibit increased variance in behavior compared to traditional approaches. Li et al. [43] documented a 15% increase in decision variance across repeated trials, though this variability often corresponded with more adaptive and sophisticated strategies.

9 Training a Multi-Agent Transformer

Building upon the theoretical frameworks introduced in previous chapters, we conducted extensive experimental validation of transformer-based approaches in multi-agent environments. Our experiments utilize the Mava framework **[58]**, a research platform designed specifically for distributed multi-agent reinforcement learning. It should be noted that in our experimental setup, the default MAPPO configuration was set to run for fewer training steps compared to MAT. While this results in some learning curves terminating earlier for MAPPO, the performance trends and comparative analysis remain valid as the convergence patterns are clearly established before termination in most cases.

Our experimental evaluation spans three environments of increasing complexity: Multi-agent Particle Environment (MPE), Robot Warehouse (RWARE), and StarCraft Multi-Agent Challenge (SMAX). MPE presents simple scenarios where agents must coordinate to achieve goals like cooperative navigation. RWARE simulates warehouse logistics, where robot agents must collaborate to pick and deliver items while avoiding collisions. SMAX, detailed further in Appendix A.2.3, offers complex combat scenarios from StarCraft II, where teams of diverse units must coordinate their actions in tactical battles.



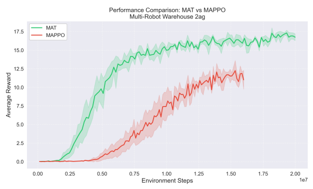
(a) 3 agents (b) 5 agents (c) 10 agents

Figure 9.1: Performance comparison in MPE scenarios with increasing agent counts. Tasks involve cooperative navigation where agents must coordinate to reach target positions while avoiding collisions. MAPPO demonstrates superior convergence and stability across all configurations.

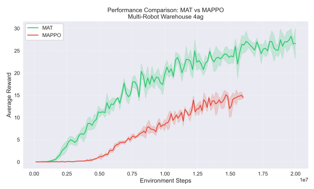
In the MPE environment, where agents navigate a 2D space to achieve cooperative goals, MAPPO consistently achieves better performance than MAT across different agent counts. This suggests that for environments with relatively simple coordination requirements, the additional complexity of the transformer architecture may not provide meaningful benefits. The computational overhead of self-attention mechanisms appears to outweigh their potential advantages in these straightforward scenarios.

The RWARE experiments reveal MAT's advantages in more complex environments. The transformer architecture demonstrates superior performance across all warehouse configurations, with the performance gap widening as the number of agents increases. This aligns with our theoretical predictions about the benefits of attention mechanisms in partially observable, sequential decision-making environments where agents must reason about shared resources and spatial constraints.

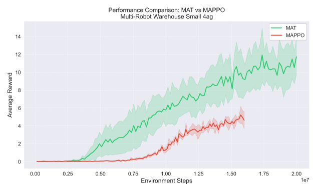
The SMAX experiments provide compelling evidence for MAT's capabilities in complex tactical



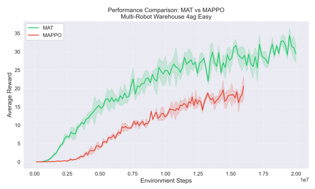
(a) Tiny 2 agents



(b) Tiny 4 agents

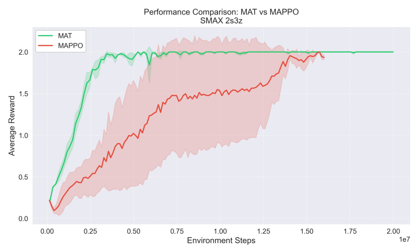


(c) Small 4 agents

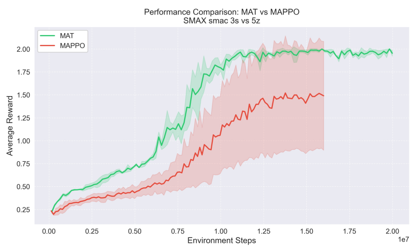


(d) Tiny 4 agents (easy)

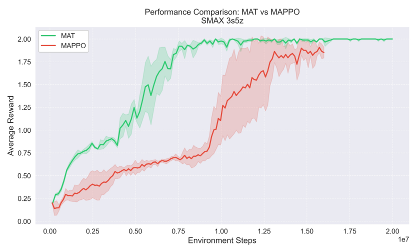
Figure 9.2: RWARE results across different warehouse configurations, where robots must coordinate to efficiently pick and deliver items while navigating tight spaces. MAT shows superior performance particularly in scenarios with higher agent counts and more complex navigation requirements.



(a) 2s3z (2 Stalkers, 3 Zealots on each team)

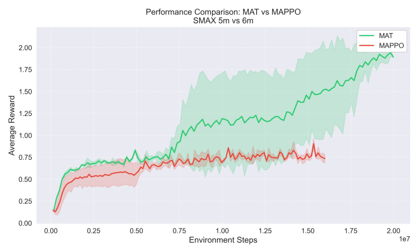


(b) 3s_vs_5z (3 Stalkers in one team vs 5 Zealots in other team)

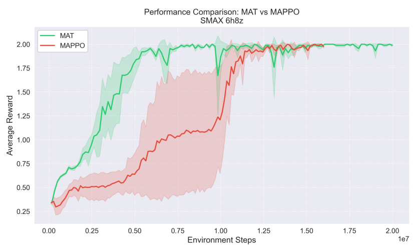


(c) 3s5z (3 Stalkers, 5 Zealots on each team)

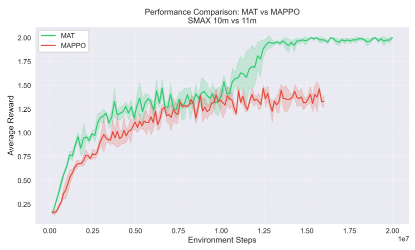
Figure 9.3: Performance in small-scale SMAX scenarios showing MAT's effectiveness in tactical coordination. Each scenario involves different combinations of ranged Stalkers and melee Zealots (see Appendix A.2.3 for unit details), requiring sophisticated positioning and focus-fire coordination.



(a) 5m_vs_6m (5 Marines vs 6 Marines)



(b) 6h_vs_8z (6 Hydralisks vs 8 Zealots)



(c) 10m_vs_11m (10 Marines vs 11 Marines)

Figure 9.4: Medium-scale SMAX battle scenarios demonstrating MAT's superior tactical coordination capabilities. These scenarios test adaptation to numerical disadvantages and coordination between different unit types with complementary strengths.

scenarios. In small-scale battles, MAT shows remarkable proficiency in both symmetric (2s3z, 3s5z) and asymmetric (3s_vs_5z) configurations, consistently achieving better convergence and final performance than MAPPO. This advantage extends to medium-scale scenarios, where MAT maintains superior performance even with increased unit counts and diverse unit compositions, as demonstrated in the 5m_vs_6m, 6h_vs_8z, and 10m_vs_11m scenarios.

These results paint a nuanced picture of architectural trade-offs in multi-agent reinforcement learning. While transformers may introduce unnecessary complexity in simple environments like MPE, their benefits become increasingly apparent as task complexity grows. In environments requiring rich spatial awareness and tactical coordination, such as RWARE and SMAX, the transformer's ability to process complex relationships between agents proves invaluable. This suggests that architectural choices in MARL should be guided by careful consideration of environmental complexity and coordination requirements rather than adopting a one-size-fits-all approach.

Bibliography

[1] Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Daniel Ho, Jasmine Hsu, Julian Ibarz, Brian Ichter, Alex Irpan, Eric Jang, Rosario Jauregui Ruano, Kyle Jeffrey, Sally Jesmonth, Nikhil J Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Kuang-Huei Lee, Sergey Levine, Yao Lu, Linda Luu, Carolina Parada, Peter Pastor, Jornell Quiambao, Kanishka Rao, Jarek Rettinghouse, Diego Reyes, Pierre Sermanet, Nicolas Sievers, Clayton Tan, Alexander Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, Mengyuan Yan, and Andy Zeng. Do as i can, not as i say: Grounding language in robotic affordances, 2022. URL https://arxiv.org/abs/2204.01691.

[2] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.*, 47(2–3):235–256, may 2002. ISSN 0885-6125. doi: 10.1023/A:1013689704352. URL https://doi.org/10.1023/A:1013689704352.

[3] R. Bellman, R.E. Bellman, and Rand Corporation. *Dynamic Programming*. Rand Corporation research study. Princeton University Press, 1957. URL https://books.google.co.za/books?id=rZW4ugAACAAJ.

[4] RICHARD BELLMAN. A markovian decision process. *Journal of Mathematics and Mechanics*, 6(5):679–684, 1957. ISSN 00959057, 19435274. URL http://www.jstor.org/stable/24900506.

[5] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

[6] Dimitri P Bertsekas and Sergey Ioffe. Temporal differences-based policy iteration and applications in neuro-dynamic programming. *Lab. for Info. and Decision Systems Report LIDS-P-2349, MIT, Cambridge, MA*, 14:8, 1996.

[7] Noam Brown and Tuomas Sandholm. Superhuman ai for multiplayer poker. *Science*, 365(6456):885–890, 2019.

[8] Lucian Busoniu, Robert Babuska, and Bart De Schutter. A comprehensive survey of multiagent reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 38(2):156–172, 2008.

[9] Chang Chen, Yi-Fu Wu, Jaesik Yoon, and Sungjin Ahn. Transdreamer: Reinforcement learning with transformer world models, 2022. URL https://arxiv.org/abs/2202.09481.

[10] Huaben Chen, Wenkang Ji, Lufeng Xu, and Shiyu Zhao. Multi-agent consensus seeking via large language models, 2023. URL https://arxiv.org/abs/2310.20151.

[11] Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Michael Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision transformer: Reinforcement learning via sequence modeling, 2021. URL https://arxiv.org/abs/2106.01345.

[12] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

[13] Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, David Belanger, Lucy Colwell, and Adrian Weller. Rethinking attention with performers, 2020. URL https://arxiv.org/abs/2009.14794.

[14] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways, 2022. URL https://arxiv.org/abs/2204.02311.

[15] Tianshu Chu, Jie Wang, Lara Codecà, and Zhaojian Li. Multi-agent deep reinforcement learning for large-scale traffic signal control, 2019. URL https://arxiv.org/abs/1903.04527.

[16] Caroline Claus and Craig Boutilier. The dynamics of reinforcement learning in cooperative multiagent systems. *AAAI/IAAI*, 1998(746-752):2, 1998.

[17] Vincent Conitzer and Tuomas Sandholm. Awesome: A general multiagent learning algorithm that converges in self-play and learns a best response against stationary opponents. *Machine Learning*, 67:23–43, 2007. URL https://api.semanticscholar.org/CorpusID:3010227.

[18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018. URL https://arxiv.org/abs/1810.04805.

[19] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale, 2020. URL https://arxiv.org/abs/2010.11929.

[20] Jakob Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. Counterfactual multi-agent policy gradients. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[21] Jakob N. Foerster, Richard Y. Chen, Maruan Al-Shedivat, Shimon Whiteson, Pieter Abbeel, and Igor Mordatch. Learning with opponent-learning awareness, 2018. URL https://arxiv.org/abs/1709.04326.

[22] Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination, 2020. URL https://arxiv.org/abs/1912.01603.

[23] Danijar Hafner, Timothy Lillicrap, Mohammad Norouzi, and Jimmy Ba. Mastering atari with discrete world models, 2022. URL https://arxiv.org/abs/2010.02193.

[24] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn, 2018. URL https://arxiv.org/abs/1703.06870.

[25] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding, 2021. URL https://arxiv.org/abs/2009.03300.

[26] Pablo Hernandez-Leal, Michael Kaisers, Tim Baarslag, and Enrique Munoz De Cote. A survey of learning in multiagent environments: Dealing with non-stationarity. *arXiv preprint arXiv:1707.09183*, 2017.

[27] Pablo Hernandez-Leal, Bilal Kartal, and Matthew E Taylor. A survey and critique of multiagent deep reinforcement learning. *Autonomous Agents and Multi-Agent Systems*, 33(6):750–797, 2019.

[28] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. Metagpt: Meta programming for a multi-agent collaborative framework, 2023. URL https://arxiv.org/abs/2308.00352.

[29] Siyi Hu, Fengda Zhu, Xiaojun Chang, and Xiaodan Liang. Updet: Universal multi-agent reinforcement learning via policy decoupling with transformers, 2021. URL https://arxiv.org/abs/2101.08001.

[30] Shaohan Huang, Li Dong, Wenhui Wang, Yaru Hao, Saksham Singhal, Shuming Ma, Tengchao Lv, Lei Cui, Owais Khan Mohammed, Barun Patra, Qiang Liu, Kriti Aggarwal, Zewen Chi, Johan Bjorck, Vishrav Chaudhary, Subhojit Som, Xia Song, and Furu Wei. Language is not all you need: Aligning perception with language models, 2023. URL https://arxiv.org/abs/2302.14045.

[31] Maximilian Hüttenrauch, Adrian Šošić, and Gerhard Neumann. Guided deep reinforcement learning for swarm systems. *arXiv preprint arXiv:1709.06011*, 2017.

[32] Shariq Iqbal and Fei Sha. Actor-attention-critic for multi-agent reinforcement learning. In *International conference on machine learning*, pages 2961–2970. PMLR, 2019.

[33] Michael Janner, Qiyang Li, and Sergey Levine. Offline reinforcement learning as one big sequence modeling problem, 2021. URL https://arxiv.org/abs/2106.02039.

[34] Jiechuan Jiang, Chen Dun, Tiejun Huang, and Zongqing Lu. Graph convolutional reinforcement learning. *arXiv preprint arXiv:1810.09202*, 2018.

[35] John M. Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A A Kohl, Andy Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. Highly accurate protein structure prediction with alphafold. *Nature*, 596:583 – 589, 2021. URL https://api.semanticscholar.org/CorpusID:235959867.

[36] Shyam Sundar Kannan, Vishnunandan L. N. Venkatesh, and Byung-Cheol Min. Smart-llm: Smart multi-agent robot task planning using large language models, 2024. URL https://arxiv.org/abs/2309.10062.

[37] Muhammad Junaid Khan, Syed Hammad Ahmed, and Gita Sukthankar. Transformer-based value function decomposition for cooperative multi-agent reinforcement learning in starcraft, 2022. URL https://arxiv.org/abs/2208.07298.

[38] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer, 2020. URL https://arxiv.org/abs/2001.04451.

[39] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners, 2023. URL https://arxiv.org/abs/2205.11916.

[40] Ananya Kumar, Aditi Raghunathan, Robbie Jones, Tengyu Ma, and Percy Liang. Fine-tuning can distort pretrained features and underperform out-of-distribution, 2022. URL https://arxiv.org/abs/2202.10054.

[41] Michael Laskin, Luyu Wang, Junhyuk Oh, Emilio Parisotto, Stephen Spencer, Richie Steigerwald, DJ Strouse, Steven Hansen, Angelos Filos, Ethan Brooks, Maxime Gazeau, Himanshu

Sahni, Satinder Singh, and Volodymyr Mnih. In-context reinforcement learning with algorithm distillation, 2022. URL https://arxiv.org/abs/2210.14215.

[42] Jonathan N. Lee, Annie Xie, Aldo Pacchiano, Yash Chandak, Chelsea Finn, Ofir Nachum, and Emma Brunskill. Supervised pretraining can learn in-context reinforcement learning, 2023. URL https://arxiv.org/abs/2306.14892.

[43] Huao Li, Yu Chong, Simon Stepputtis, Joseph Campbell, Dana Hughes, Charles Lewis, and Katia Sycara. Theory of mind for multi-agent collaboration via large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.emnlp-main.13. URL http://dx.doi.org/10.18653/v1/2023.emnlp-main.13.

[44] Michael L. Littman. Markov games as a framework for multi-agent reinforcement learning. In William W. Cohen and Haym Hirsh, editors, *Machine Learning Proceedings 1994*, pages 157–163. Morgan Kaufmann, San Francisco (CA), 1994. ISBN 978-1-55860-335-6. doi: https://doi.org/10.1016/B978-1-55860-335-6.50027-1. URL https://www.sciencedirect.com/science/article/pii/B9781558603356500271.

[45] Zijun Liu, Yanzhe Zhang, Peng Li, Yang Liu, and Diyi Yang. Dynamic llm-agent network: An llm-agent collaboration framework with agent team optimization, 2023. URL https://arxiv.org/abs/2310.02170.

[46] Ryan Lowe, Yi I Wu, Aviv Tamar, Jean Harb, OpenAI Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. *Advances in neural information processing systems*, 30, 2017.

[47] Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models, 2024. URL https://arxiv.org/abs/2310.12931.

[48] Anuj Mahajan, Tabish Rashid, Mikayel Samvelyan, and Shimon Whiteson. Maven: Multi-agent variational exploration. *Advances in neural information processing systems*, 32, 2019.

[49] Zhao Mandi, Shreeya Jain, and Shuran Song. Roco: Dialectic multi-robot collaboration with large language models, 2023. URL https://arxiv.org/abs/2307.04738.

[50] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[51] Koichiro Morihiro, Teijiro Isokawa, Haruhiko Nishimura, and Nobuyuki Matsui. Emergence of flocking behavior based on reinforcement learning. In *Knowledge-Based Intelligent Information and Engineering Systems: 10th International Conference, KES 2006, Bournemouth, UK, October 9-11, 2006. Proceedings, Part III 10*, pages 699–706. Springer, 2006.

[52] Frans A Oliehoek, Christopher Amato, et al. *A concise introduction to decentralized POMDPs*, volume 1. Springer, 2016.

[53] Maxime Oquab, Timothée Darcet, Théo Moutakanni, Huy Vo, Marc Szafraniec, Vasil Khalidov, Pierre Fernandez, Daniel Haziza, Francisco Massa, Alaaeldin El-Nouby, Mahmoud Assran, Nicolas Ballas, Wojciech Galuba, Russell Howes, Po-Yao Huang, Shang-Wen Li, Ishan Misra, Michael Rabbat, Vasu Sharma, Gabriel Synnaeve, Hu Xu, Hervé Jegou, Julien Mairal, Patrick Labatut, Armand Joulin, and Piotr Bojanowski. Dinov2: Learning robust visual features without supervision, 2024. URL https://arxiv.org/abs/2304.07193.

[54] Jing-Cheng Pang, Si-Hang Yang, Kaiyuan Li, Jiaji Zhang, Xiong-Hui Chen, Nan Tang, and Yang Yu. Knowledgeable agents by offline reinforcement learning from large language model rollouts, 2024. URL https://arxiv.org/abs/2404.09248.

[55] Emilio Parisotto, Francis Song, Jack Rae, Razvan Pascanu, Caglar Gulcehre, Siddhant Jayakumar, Max Jaderberg, Raphael Lopez Kaufman, Aidan Clark, Seb Noury, et al. Stabilizing transformers for reinforcement learning. In *International conference on machine learning*, pages 7487–7498. PMLR, 2020.

[56] Debjit Paul, Mete Ismayilzada, Maxime Peyrard, Beatriz Borges, Antoine Bosselut, Robert West, and Boi Faltings. Refiner: Reasoning feedback on intermediate representations, 2024. URL https://arxiv.org/abs/2304.01904.

[57] Archiki Prasad, Alexander Koller, Mareike Hartmann, Peter Clark, Ashish Sabharwal, Mohit Bansal, and Tushar Khot. Adapt: As-needed decomposition and planning with language models, 2024. URL https://arxiv.org/abs/2311.05772.

[58] Arnu Pretorius, Kale-ab Tessera, Andries P. Smit, Claude Formanek, St John Grimbly, Kevin Eloff, Siphelele Danisa, Lawrence Francis, Jonathan P. Shock, Herman Kamper, Willie Brink, Herman A. Engelbrecht, Alexandre Laterre, and Karim Beguir. Mava: a research framework for distributed multi-agent reinforcement learning. *CoRR*, abs/2107.01460, 2021. URL https://arxiv.org/abs/2107.01460.

[59] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[60] Alec Radford and Karthik Narasimhan. Improving language understanding by generative pre-training. 2018. URL https://api.semanticscholar.org/CorpusID:49313245.

[61] Tabish Rashid, Mikayel Samvelyan, Christian Schroeder De Witt, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. Monotonic value function factorisation for deep multi-agent reinforcement learning. *Journal of Machine Learning Research*, 21(178):1–51, 2020.

[62] Scott Reed, Konrad Zolna, Emilio Parisotto, Sergio Gomez Colmenarejo, Alexander Novikov, Gabriel Barth-Maron, Mai Gimenez, Yury Sulsky, Jackie Kay, Jost Tobias Springenberg, Tom Eccles, Jake Bruce, Ali Razavi, Ashley Edwards, Nicolas Heess, Yutian Chen, Raia Hadsell, Oriol Vinyals, Mahyar Bordbar, and Nando de Freitas. A generalist agent, 2022. URL https://arxiv.org/abs/2205.06175.

[63] Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering Cambridge, UK, 1994.

[64] Mikayel Samvelyan, Tabish Rashid, Christian Schröder de Witt, Gregory Farquhar, Nantas Nardelli, Tim G. J. Rudner, Chia-Man Hung, Philip H. S. Torr, Jakob N. Foerster, and Shimon Whiteson. The starcraft multi-agent challenge. *CoRR*, abs/1902.04043, 2019. URL http://arxiv.org/abs/1902.04043.

[65] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[66] Shai Shalev-Shwartz, Shaked Shammah, and Amnon Shashua. Safe, multi-agent, reinforcement learning for autonomous driving. *arXiv preprint arXiv:1610.03295*, 2016.

[67] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023. URL https://arxiv.org/abs/2303.11366.

[68] Yoav Shoham and Kevin Leyton-Brown. *Multiagent systems: Algorithmic, game-theoretic, and logical foundations*. Cambridge University Press, 2008.

[69] Yoav Shoham, Rob Powers, and Trond Grenager. If multi-agent learning is the answer, what is the question? *Artificial intelligence*, 171(7):365–377, 2007.

[70] Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. Alfworld: Aligning text and embodied environments for interactive learning, 2021. URL https://arxiv.org/abs/2010.03768.

[71] Felipe Silva and Anna Costa. A survey on transfer learning for multiagent reinforcement learning systems. *Journal of Artificial Intelligence Research*, 64, 03 2019. doi: 10.1613/jair.1.11396.

[72] Satinder P Singh and Richard S Sutton. Reinforcement learning with replacing eligibility traces. *Machine learning*, 22(1):123–158, 1996.

[73] Sainbayar Sukhbaatar, Arthur Szlam, and Rob Fergus. Learning multiagent communication with backpropagation. *CoRR*, abs/1605.07736, 2016. URL http://arxiv.org/abs/1605.07736.

[74] Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3:9–44, 1988.

[75] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. URL http://incompleteideas.net/book/the-book-2nd.html.

[76] Richard S. Sutton, David A. McAllester, Satinder Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Neural Information Processing Systems*, 1999. URL https://api.semanticscholar.org/CorpusID:1211821.

[77] Ming Tan. Multi-agent reinforcement learning: Independent versus cooperative agents. In *International Conference on Machine Learning*, 1997. URL https://api.semanticscholar.org/CorpusID:272885126.

[78] James Thorne, Andreas Vlachos, Christos Christodoulopoulos, and Arpit Mittal. Fever: a large-scale dataset for fact extraction and verification, 2018. URL https://arxiv.org/abs/1803.05355.

[79] S. Thrun. The role of exploration in learning control. In D.A. White and D.A. Sofge, editors, *Handbook for Intelligent Control: Neural, Fuzzy and Adaptive Approaches*. Van Nostrand Reinhold, Florence, Kentucky 41022, 1992.

[80] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023. URL https://arxiv.org/abs/2302.13971.

[81] J.N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690, 1997. doi: 10.1109/9.580874.

[82] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017. URL https://arxiv.org/abs/1706.03762.

[83] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *nature*, 575(7782):350–354, 2019.

[84] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

[85] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023. URL https://arxiv.org/abs/2201.11903.

[86] Muning Wen, Jakub Grudzien Kuba, Runji Lin, Weinan Zhang, Ying Wen, Jun Wang, and Yaodong Yang. Multi-agent reinforcement learning is a sequence modeling problem, 2022. URL https://arxiv.org/abs/2205.14953.

[87] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256, 1992.

[88] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. Hotpotqa: A dataset for diverse, explainable multi-hop question answering, 2018. URL https://arxiv.org/abs/1809.09600.

[89] Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents, 2023. URL https://arxiv.org/abs/2207.01206.

[90] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023. URL https://arxiv.org/abs/2210.03629.

[91] Weiran Yao, Shelby Heinecke, Juan Carlos Niebles, Zhiwei Liu, Yihao Feng, Le Xue, Rithesh Murthy, Zeyuan Chen, Jianguo Zhang, Devansh Arpit, Ran Xu, Phil Mui, Huan Wang, Caiming Xiong, and Silvio Savarese. Retroformer: Retrospective large language agents with policy gradient optimization, 2024. URL https://arxiv.org/abs/2308.02151.

[92] Bangguo Yu, Hamidreza Kasaei, and Ming Cao. Co-navgpt: Multi-robot cooperative visual semantic navigation using large language models, 2023. URL https://arxiv.org/abs/2310.07937.

[93] Yong Yu, Xiaosheng Si, Changhua Hu, and Jianxun Zhang. A review of recurrent neural networks: Lstm cells and network architectures. *Neural computation*, 31(7):1235–1270, 2019.

[94] Kuo-Hao Zeng, Zichen Zhang, Kiana Ehsani, Rose Hendrix, Jordi Salvador, Alvaro Herrasti, Ross Girshick, Aniruddha Kembhavi, and Luca Weihs. Poliformer: Scaling on-policy rl with transformers results in masterful navigators, 2024. URL https://arxiv.org/abs/2406.20083.

[95] Hongxin Zhang, Weihua Du, Jiaming Shan, Qinhong Zhou, Yilun Du, Joshua B. Tenenbaum, Tianmin Shu, and Chuang Gan. Building cooperative embodied agents modularly with large language models, 2024. URL https://arxiv.org/abs/2307.02485.

[96] Kaiqing Zhang, Zhuoran Yang, and Tamer Başar. Multi-agent reinforcement learning: A selective overview of theories and algorithms. *Handbook of reinforcement learning and control*, pages 321–384, 2021.

[97] Shenao Zhang, Sirui Zheng, Shuqi Ke, Zhihan Liu, Wanxin Jin, Jianbo Yuan, Yingxiang Yang, Hongxia Yang, and Zhaoran Wang. How can llm guide rl? a value-based approach, 2024. URL https://arxiv.org/abs/2402.16181.

A Experimental Details

This appendix provides comprehensive details about the experimental configurations, environment parameters, and algorithm settings used in our comparative study of transformer-based and traditional MARL approaches.

A.1 Algorithm Configurations

Our experiments utilized two primary algorithms: Multi-Agent Transformer (MAT) and Multi-Agent PPO (MAPPO). Table A.1 presents the key hyperparameters for both algorithms.

A.2 Environment Configurations

Our experimental evaluation spanned three distinct environments of increasing complexity. Each environment was configured with specific parameters to test different aspects of multi-agent coordination.

A.2.1 Multi-Agent Particle Environment (MPE)

The MPE scenarios were configured to test basic coordination capabilities with varying numbers of agents. Table A.2 details the specific configurations used.

A.2.2 Robot Warehouse (RWARE)

The RWARE environment tests coordination in practical logistics scenarios. Table A.3 presents the configurations used across different warehouse setups.

Table A.1: Algorithm Hyperparameters

Parameter	MAT	MAPPO
Actor Learning Rate	5×10^{-4}	2.5×10^{-4}
Critic Learning Rate	-	2.5×10^{-4}
Update Batch Size	2	2
Rollout Length	128	128
PPO Epochs	5	4
Number of Minibatches	1	2
Discount Factor (γ)	0.99	0.99
GAE Lambda	0.95	0.95
Clipping Epsilon	0.1	0.2
Entropy Coefficient	0.01	0.01
Value Function Coefficient	0.5	0.5
Maximum Gradient Norm	5.0	0.5

Table A.2: MPE Configuration Parameters

Parameter	Simple Spread 3ag	Simple Spread 5ag	Simple Spread 10ag
Number of Agents	3	5	10
Number of Landmarks	3	5	10
Local Ratio	0.5	0.5	0.5

Table A.3: RWARE Configuration Parameters

Parameter	Tiny-2ag	Tiny-4ag	Tiny-4ag-easy	Small-4ag
Column Height	8	8	8	8
Shelf Rows	1	1	1	2
Shelf Columns	3	3	3	3
Number of Agents	2	4	4	4
Sensor Range	1	1	1	1
Request Queue Size	2	4	8	4
Time Limit	500	500	500	500

A.2.3 StarCraft Multi-Agent Challenge (SMAX)

SMAX scenarios leverage units from the StarCraft II real-time strategy game to create challenging multi-agent combat scenarios. Table A.4 details the core environment configurations.

Table A.4: SMAX Configuration Parameters

Parameter	Value	Description
See Enemy Actions	True	Enables enemy vision capability
Walls Cause Death	True	Agents die upon wall collision
Attack Mode	"closest"	Targets closest enemy unit

Stalker (s) Zealot (z) Marine (m) Hydralisk (h)

Figure A.1: StarCraft II unit types used in SMAX scenarios, with their corresponding notation symbols.

Environment Design

The SMAX scenarios follow a standardized naming convention that indicates team compositions. Symmetric scenarios (e.g., `XaYb`) denote identical unit compositions for both teams, where X and Y represent unit quantities and letters denote unit types as shown in Figure A.1. Asymmetric scenarios (e.g., `Xa_vs_Yb`) specify different compositions for opposing teams, typically introducing deliberate imbalances to test tactical adaptation.

Battle Scenarios

The specific unit compositions are categorized by complexity:

- **Small-scale scenarios:**
  - 2s3z: Symmetric – Each team has 2 Stalkers and 3 Zealots
  - 3s5z: Symmetric – Each team has 3 Stalkers and 5 Zealots
  - 3s_vs_5z: Asymmetric – Team 1 has 3 Stalkers versus Team 2's 5 Zealots
- **Medium-scale scenarios:**
  - 5m_vs_6m: 5 Marines versus 6 Marines
  - 6h_vs_8z: 6 Hydralisks versus 8 Zealots
  - 10m_vs_11m: 10 Marines versus 11 Marines
- **Large-scale scenarios:**

- 27m_vs_30m: 27 Marines versus 30 Marines
- 3s5z_vs_3s6z: Team 1 (3 Stalkers, 5 Zealots) versus Team 2 (3 Stalkers, 6 Zealots)
- SMAC v2: Extended scenarios with varying team sizes (5, 10, or 20 units)

The progression of scenarios demonstrates increasing complexity in both tactical and strategic dimensions. Symmetric scenarios establish baseline coordination challenges, while asymmetric configurations introduce tactical imbalances that demand sophisticated strategic responses. The large-scale scenarios further compound these challenges by requiring concurrent management of both micro-level unit control and macro-level strategic planning.

A.3 Hardware and Software Configuration

All experiments were conducted using the Mava framework [58], which provides a standardized implementation of distributed multi-agent reinforcement learning algorithms. The experiments were performed on a single NVIDIA A100 GPU with 40GB of memory. The total experimental runtime across all environments and configurations was under 48 hours.

Each experiment was repeated with three different random seeds to ensure statistical significance of the results. The training curves presented in the main text represent the mean performance across these runs, with shaded regions indicating one standard deviation.