QMIX Algorithm: Theory and Implementation Report

---

1. QMIX Algorithm Overview

What is QMIX?

QMIX is a multi-agent reinforcement learning algorithm where agents learn to work together. It helps agents learn coordinated behavior during training while still allowing them to act independently when the model is used.

The Core Problem

In multi-agent reinforcement learning, we need to balance two competing requirements:

1. **Centralized Training:** During training, agents can benefit from global information (full state) to learn coordinated strategies.
2. **Decentralized Execution:** During execution, each agent must act based only on its local observation, without communication.

QMIX's Solution

- Learning individual agent Q-functions $Q_i(o_i, a_i)$ that depend only on local observations
- Learning a mixing network that combines these Q-values into a joint Q-value $Q_{tot}(s, a_1, ..., a_n)$
- Enforcing a monotonicity constraint: $\partial Q_{tot} / \partial Q_i \geq 0$ for all agents i

This constraint ensures that:

- If an agent's Q-value increases, the joint Q-value also increases
- The optimal joint action can be found by each agent independently selecting $argmax_{a_i} Q_i(o_i, a_i)$
- Decentralized execution is valid while benefiting from centralized training

---

2. How QMIX Works

Architecture Components

QMIX consists of three main components:

* **Agent Q-Networks**
  - Each agent i has its own Q-network: $Q_i(o_i, a_i)$
  - Takes local observation o_i as input
  - Outputs Q-values for each action a_i
  - Decentralized: Each agent's network only sees its own observation

* **Mixing Network**
  - Takes individual agent Q-values $[Q_1, Q_2, ..., Q_n]$ and global state s
  - Combines them into joint Q-value: $Q_{tot}(s, a_1, ..., a_n)$
  - Uses hypernetworks to generate state-dependent mixing weights
  - Centralized: Has access to global state during training

* **Centralized Replay Buffer**
  - Stores transitions: $(s, o_1, ..., o_n, a_1, ..., a_n, r, s', o'_1, ..., o'_n, done)$
  - Includes both local observations and global state
  - Enables off-policy learning with experience replay

The Monotonicity Constraint

The key innovation of QMIX is the monotonicity constraint:

Mathematical Formulation:

$$\partial Q_{tot} / \partial Q_i \geq 0 \quad \text{for all agents i}$$

What this means:

- If agent i's Q-value increases, the joint Q-value cannot decrease
- This ensures that maximizing each Q_i independently will maximize Q_{tot}

Why this matters:

- Enables decentralized execution: Each agent can greedily select $argmax_{a_i} Q_i(o_i, a_i)$
- The resulting joint action will be optimal for Q_{tot}
- No need for communication or centralized action selection during execution

Hypernetworks

QMIX uses hypernetworks to enforce monotonicity:

- Hypernetworks are neural networks that generate weights for another network
- In QMIX, hypernetworks take the global state s as input
- They generate positive weights for the mixing network
- Positive weights ensure monotonicity: if all weights are positive, increasing any Q_i increases Q_{tot}

What the weights do

Think of the mixing network as computing:

$$Q_{tot} = w_1 \cdot Q_1 + w_2 \cdot Q_2 + \text{bias}$$

- Q_1, Q_2: how good each agent's action is
- w_1, w_2: how much each agent's Q-value matters
- Q_{tot}: how good the team action is

The weights decide whether an agent's action helps or hurts the team.

Case 1: Weights are positive (QMIX rule)

Example:

$Q_1 = 5$ (agent 1 is doing well)
$Q_2 = 3$ (agent 2 is doing okay)
$w_1 = +2$
$w_2 = +1$

Total: $Q_{tot} = 2 \cdot 5 + 1 \cdot 3 = 13$

Now agent 1 improves:

$Q_1 = 6$
$Q_{tot} = 2 \cdot 6 + 1 \cdot 3 = 15$
* Q_1 increased
* Q_{tot} increased

☑ Improving an agent never hurts the team.

Case 2: Weights are negative (what QMIX avoids)

Example:

$Q_1 = 5$
$Q_2 = 3$
$w_1 = -2$ (negative!)
$w_2 = +1$

Total: Q_tot = -2·5 + 1·3 = -7

Now agent 1 improves:

$Q_1 = 6$
$Q_{tot} = -2 \cdot 6 + 1 \cdot 3 = -9$
* Q_1 increased
* Q_{tot} got worse

⭕ Agent 1 doing better makes the team worse. This breaks coordination.

Training Process

1. Collect Experience: Agents interact with environment, storing transitions in centralized buffer
2. Compute Q_{tot}: For each transition, compute $Q_{tot}(s, a_1, \ldots, a_n)$ using mixing network
3. Compute Target: Use target networks to compute Q_tot_target(s', a'_1, ..., a'_n)
4. Update: Minimize loss L = (Q_{tot} - (r + γ * Q_tot_target))²
5. Update Targets: Periodically copy main networks to target networks

Key Insight: The loss is computed on Q_{tot}, but gradients flow back to both:

* Agent Q-networks (through mixing network)
* Mixing network itself

This allows agents to learn coordinated strategies while maintaining decentralized execution capability.

Action Selection
- During Training (**Exploration**):
    * Each agent uses epsilon-greedy: select random action with probability ε, else $argmax_{a_i}Q_i(o_i, a_i)$
- During Execution (**Exploitation**):
    * Each agent greedily selects: $a_i = argmax_{a_i}Q_i(o_i, a_i)$
    * No communication needed - each agent acts independently
    * Monotonicity ensures this joint action maximizes Q_{tot}

---

3. Implementation Details

Project Structure
* src/algos/**qmix.py**: Main QMIX coordinator class
    - Manages the whole algorithm, including setup, training, and updating the target networks.

* src/algos/**qmix_agent.py**: Individual agent Q-networks
    - **Main Q-network**: $Q_i(o_i, a_i)$ - estimates Q-values from local observations
    - **Target Q-network**: Used for stable target computation
    - **Action selection**: Epsilon-greedy policy
    - **select_action(obs, epsilon)**: Epsilon-greedy action selection
    - **update_target_network()**: Copy main network to target

* src/models/**mixing_network.py**: Mixing network with hypernetworks
    - Uses **F.softplus()** to ensure all weights are strictly positive
    - **softplus(x)** = $log(1 + exp(x))$ is always > 0
    - Better gradient flow than abs() for training stability

* src/utils/**qmix_replay_memory.py**: Centralized replay buffer
    - Global state s
    - Local observations o_i
    - Actions a_i
    - Joint reward r
    - Next state/observations
    - Done flag

* src/models/**qvalue_network.py**: Base Q-network architecture (shared with IQL/PS-DQN)

---

4. Key Differences from IQL and PS-DQN

Aspect	IQL	PS-DQN	QMIX
Network Structure	Separate Q-networks per agent	Single shared Q-network	Separate Q-networks + Mixing network
Training Signal	Individual Q_i losses	Shared Q-network loss	Joint Q_tot loss
State Information	Local observations only	Local observations only	Global state + local observations
Coordination	Implicit	Implicit	Explicit
Execution	Decentralized	Decentralized	Decentralized
Monotonicity	N/A	N/A	Enforced

QMIX Advantages:

- Explicit coordination through mixing network
- Can leverage global state during training
- Maintains decentralized execution

QMIX Challenges:

* More complex architecture
* Requires global state
* More hyperparameters
* Higher computation cost

---

5. Summary

QMIX enables centralized training while keeping decentralized execution. It uses a mixing network with monotonic constraints so agents can learn teamwork while acting independently.

The implementation applies QMIX to the Meeting Gridworld task, showing how coordinated multi-agent behavior can be learned without requiring communication during execution.