Integration of Multiplicative Weight Update Algorithm in Reinforcement Learning

Kartikesh Mishra
MIT - 18.S096 Probability Seminar

April 2024

Abstract

Reinforcement Learning (RL) algorithms are powerful techniques for training agents to make effective decisions in complex environments. However, they often face challenges such as balancing exploration and exploitation, slow convergence rates, and inefficient policy updates. This paper provides an expository overview of RL concepts and explores the potential integration of the Multiplicative Weight Update (MWU) Algorithm, a well-established online learning technique, into RL frameworks to address these challenges. We investigate different approaches for integrating MWU into RL algorithms, including treating policies as experts, grid-based experts, and reward strategies.

1 Introduction

Imagine a robot navigating through a complex grid maze, faced with the challenge of deciding which direction to turn at every intersection. Its goal is to reach the destination efficiently, but the optimal path is unknown. This scenario captures the essence of reinforcement learning (RL) (1), a powerful paradigm in machine learning (ML) that enables an agent to learn sequential decision-making through trial-and-error interactions with an environment. Unlike supervised learning, which relies on predefined datasets, RL agents learn from experience, continuously adapting their behavior to achieve better performance. This makes RL particularly well-suited for domains where trial-and-error learning is essential, such as robotics, autonomous systems, resource allocation, and recommendation systems.

Despite its success, traditional RL algorithms often face challenges such as the exploration-exploitation trade-off, slow convergence rates, and the need for efficient policy updates. In this paper, we explore the integration of the Multiplicative Weight Update (MWU) Algorithm (2) into RL frameworks to address these challenges and enhance learning efficiency.

The MWU algorithm is a powerful online learning technique that maintains a set of weights corresponding to available options (e.g., actions or policies) and adaptively adjusts these weights based on feedback received over time. Its robustness, efficiency, and close relation to RL make it a promising candidate for enhancing RL algorithms.

In Section 2, we provide an overview of the fundamental concepts and algorithms in RL, including Markov Decision Processes (MDPs), value functions, Q-learning, and

actor-critic methods. Section 3 introduces the Multiplicative Weight Update Algorithm and its variants. Motivated by the similarities between MWU and RL, we explore various ways to integrate these two paradigms in Section 4.

2 Reinforcement Learning

Reinforcement Learning (RL) is a machine learning paradigm that focuses on training agents to make sequential decisions in uncertain environments. The agent learns a policy (which guides an agent on what action to take in the given situation/state) by interacting with the environment, receiving feedback in the form of rewards or penalties, and adapting its behavior to maximize the cumulative reward over time.

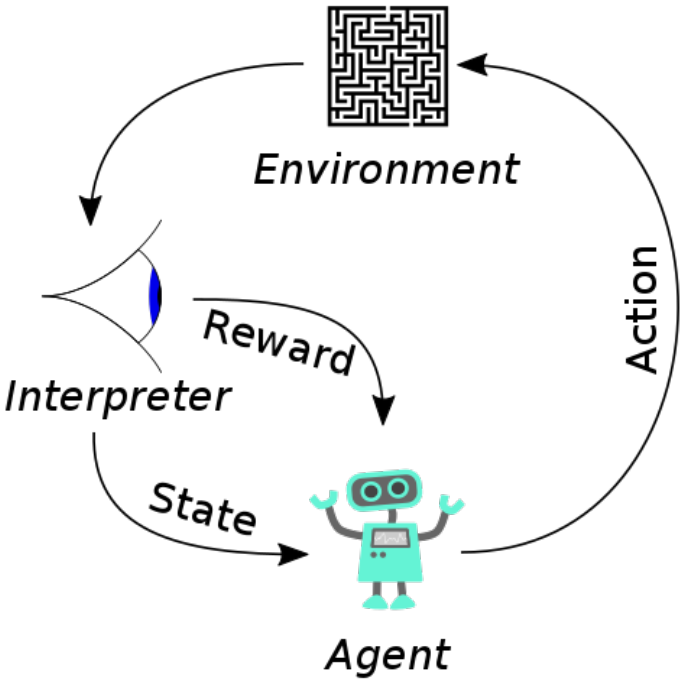

Figure 1: Reinforcement learning diagram (3).

2.1 Markov Decision Processes

Reinforcement Learning (RL) problems are often modeled as Markov Decision Processes (MDPs), which are defined by a tuple $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$, where:

- $\mathcal{S}$ is the set of possible states that the agent can be in.
- $\mathcal{A}$ is the set of actions available to the agent.
- $P(s_{t+1}|s_t, a_t)$ is the transition probability function, which defines the probability of transitioning from state s_t to state s_{t+1} after taking action a_t.
- $R(s_t, a_t, s_{t+1})$ is the reward function, which specifies the immediate reward (or penalty) received by the agent after transitioning from state s_t to state s_{t+1} by taking action a_t.

- $\gamma \in [0, 1)$ is the discount factor, which determines the importance of future rewards relative to immediate rewards.

At each time step t, the agent observes the current state $s_t \in \mathcal{S}$, selects an action $a_t \in \mathcal{A}$ based on its deterministic policy $\pi : \mathcal{S} \to \mathcal{A}$, receives a reward $r_t = R(s_t, a_t, s_{t+1})$, and transitions to the next state s_{t+1}. The agent's goal is to learn an optimal policy π^* that maximizes the expected cumulative discounted reward:

$$\max_{\pi} \mathbb{E}_{\pi} \left[\sum_{t=0}^{\infty} \gamma^t r_t \right].$$

2.2 Value Functions

To evaluate the quality of a policy π, we define the value function $V^{\pi}(s)$, which represents the expected cumulative discounted reward from starting in state s and following policy π:

$$V^{\pi}(s) = \mathbb{E}_{\pi} \left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s \right].$$

Similarly, the action-value function $Q^{\pi}(s, a)$ represents the expected cumulative discounted reward from starting in state s, taking action a, and then following policy π:

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi} \left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, a_0 = a \right].$$

These value functions satisfy the Bellman equations(4) where a more generic term for policy is used i.e. indeterministic. In the case of deterministic policy, $\pi(a|s) = 1$ for a particular action and 0 otherwise.

$$V^{\pi}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} P(s'|s, a) \left[R(s, a, s') + \gamma V^{\pi}(s') \right]$$

$$Q^{\pi}(s, a) = \sum_{s' \in \mathcal{S}} P(s'|s, a) \left[R(s, a, s') + \gamma \sum_{a' \in \mathcal{A}} \pi(a'|s') Q^{\pi}(s', a') \right]$$

The optimal value functions, $V^*(s)$ and $Q^*(s, a)$, correspond to the maximum expected cumulative discounted reward achievable by any policy, and satisfy the Bellman optimality equations(4):

$$V^*(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} P(s'|s, a) \left[R(s, a, s') + \gamma V^*(s') \right],$$

$$Q^*(s, a) = \sum_{s' \in \mathcal{S}} P(s'|s, a) \left[R(s, a, s') + \gamma \max_{a' \in \mathcal{A}} Q^*(s', a') \right].$$

2.3 Q-Learning

Q-learning (5) is a popular model-free RL algorithm that learns the optimal action-value function $Q^*(s, a)$ by iteratively updating the Q-values based on the Bellman optimality equation:

$$Q(s_t, a_t) \leftarrow (1 - \alpha)Q(s_t, a_t) + \alpha \left(r_t + \gamma \max_a Q(s_{t+1}, a) \right).$$

Here, $\alpha \in (0, 1]$ is the learning rate, which determines the proportion of update based on the new experience (s_t, a_t, r_t, s_{t+1}). Q-learning is an off-policy algorithm, meaning that it can learn the optimal policy while following a different behavior policy. This allows for more efficient exploration of the state-action space and faster convergence to the optimal policy.

2.4 Actor-Critic Methods

Actor-critic methods (6) are a class of RL algorithms that separate the decision-making process into two components: the actor, which selects actions based on the current policy $\pi_\theta(a|s)$, and the critic, which evaluates the value of the current state or state-action pair, $V(s)$ or $Q(s, a)$.

The actor updates the policy parameters θ to maximize the expected cumulative discounted reward, typically using a form of policy gradient ascent:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta} \left[\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t|s_t) A(s_t, a_t) \right]$$

where $A(s_t, a_t) = Q(s_t, a_t) - V(s_t)$ is the advantage function. The log term is used here to transform the product of probabilities into a sum, simplifying gradient computation and stabilizing updates by scaling them according to the probability of taking action a_t in state s_t.

The critic estimates the value function $V(s)$ or $Q(s, a)$ and provides feedback to the actor, allowing for more efficient policy updates. The critic's objective is to minimize the mean squared error (MSE) between the estimated value and the actual return, which can be expressed as:

$$L(\phi) = \mathbb{E}_{\pi_\theta} \left[(Q_\phi(s_t, a_t) - (r_t + \gamma Q_\phi(s_{t+1}, a_{t+1})))^2 \right]$$

where ϕ are the parameters of the value function $Q_\phi(s, a)$.

Popular actor-critic algorithms include Advantage Actor-Critic (A2C) (7), Deep Deterministic Policy Gradient (DDPG) (8), and Proximal Policy Optimization (PPO) (9). These algorithms differ primarily in how they update the actor and critic, the type of policies they handle (e.g., deterministic or stochastic), and how they manage exploration.

3 Multiplicative Weight Update Algorithm (MWU)

The Multiplicative Weight Update (MWU) Algorithm is a powerful online learning technique originally proposed by Littlestone and Warmuth (10). It is commonly used in scenarios where a decision maker (e.g., an RL agent) must repeatedly choose from a set of N options (e.g., actions or policies) and adaptively adjust their choices based on feedback received over time. At its core, MWU maintains a weight vector $\mathbf{w}_t =$

$(w_t(1), w_t(2), \ldots, w_t(N))$, where $w_t(i)$ represents the weight associated with the i-th option at time t. Initially, these weights are typically set uniformly, i.e., $w_1(i) = 1/N$ for all i. After each decision, MWU updates the weights based on the feedback received, with the aim of increasing the weights of good options and decreasing the weights of poor options. MWU has several variants, including the Halving Algorithm and the Weighted Majority Algorithm. We will start by presenting the Halving Algorithm, which is a simple variant of MWU.

3.1 Halving Algorithm

The Halving Algorithm is a special case of MWU where the weights are updated as follows:

$$w_{t+1}(i) = \begin{cases} w_t(i) & \text{if the } i\text{-th option was correct in round } t \\ \frac{1}{2} w_t(i) & \text{otherwise} \end{cases}$$

In other words, the weight of an option is halved if it is incorrect, and remains the same if it is correct. This algorithm is called "Halving" because the sum of the weights of the incorrect options is halved in each round.

3.2 Weighted Majority Algorithm

The Weighted Majority Algorithm is a more general variant of MWU, where the weights are updated as follows, which is taken from the MWU survey paper by Arora et al (2) and slightly modified for incorporating RL terms.

$$w_{t+1}(j) \leftarrow w_t(j) \cdot (1 - \eta)$$

Here, $\eta \in (0, 1)$ is the learning rate. The algorithm can be summarized as follows:

Algorithm 1 Weight Majority Algorithm (2)

1: Initialize weights $W_0(i) = 1$ for all experts i
2: **for** each round $t = 1, 2, \ldots, T$ **do**
3: Get action $A_t(i)$ from every expert i
4: Sum the weights for every action and choose one, a with a maximum weight
5: Update weights: $w_{t+1}(j) \leftarrow w_t(j) \cdot (1 - \eta)$ if $A_t(j)$ is not correct action.
6: **end for**

The Weighted Majority Algorithm can be further generalized to the Meta-Algorithm for Multiplicative Weight Updates (2):

Algorithm 2 Meta-Algorithm for Multiplicative Weight Updates (2)

1: Initialize weights $w_1(i) = 1$ for all options $i \in 1, \ldots, N$
2: **for** $t = 0, 1, 2, \ldots, T$ **do**
3: Receive actions $A_t(i)$ from all options i
4: Choose action a_t with probability $p_t(i) = \frac{w_t(i)}{\sum_{j=1}^{N} w_t(j)}$
5: Observe feedback $M(i, a_t)$ for all options i
6: Update weights: $w_{t+1}(i) = w_t(i) \cdot (1 - \eta M(i, a_t))$
7: **end for**

In this meta-algorithm, $M(i, a_t) \in [-1, 1]$ is a general feedback function that can encode various types of feedback, such as losses, regrets, or other performance measures. The update rule adjusts the weights based on this feedback, allowing for a flexible and general framework for online learning. The bound $[-1, 1]$ is from the proof of the main Theorem from (2) as presented below.

Theorem 1 (Convergence of MWU) *Let $\epsilon \leq \frac{1}{2}$. After T rounds, for any expert i, we have:*

$$\sum_t \sum_i p_t(i) M(i, a_t) \leq \frac{\ln n}{\epsilon} + (1+\epsilon) \sum_{\geq 0} M(i, a_t) + (1-\epsilon) \sum_{<0} M(i, a_t),$$

where the subscripts ≥ 0 and < 0 refer to the rounds t where $M(i, a_t)$ is ≥ 0 and < 0, respectively.

The practical implication of this theorem is that the regret of the MWU algorithm grows at most as $O(\sqrt{T \log N})$. Therefore, the average regret per round, defined as:

$$\text{Average Regret} = \frac{1}{T} \left(\sum_{t=1}^{T} \sum_{i=1}^{N} p_t(i) M(i, a_t) - \min_i \sum_{t=1}^{T} M(i, a_t) \right),$$

converges to zero as T increases, ensuring that the algorithm's performance approaches that of the best expert in the long run.

4 Integrating MWU into Reinforcement Learning

Having reviewed the RL framework and the MWU algorithm, we now explore various ways to integrate the MWU algorithm into Reinforcement Learning (RL) frameworks. This integration aims to address challenges such as the exploration-exploitation trade-off, policy learning, and efficient updates.

4.1 Every Deterministic Policy as an Expert

One approach is to treat every possible deterministic policy as an expert and apply the MWU algorithm to update the weights of these policies based on their performance. Specifically, we initialize a weight vector as described in algorithm (2), where the number of experts, $N = |A|^{|S|}$, corresponds to the total number of possible policies π_i.

At each time step t, we choose a policy π from the set of all policies π_i uniformly, i.e., with probability $p_t(i) = \frac{w_t(i)}{\sum_{j=1}^{N} w_t(j)}$, similar to algorithm (2). We then simulate some iterations (say n_t times) to focus on long-term goals rather than short-term ones. Finally, we update the weights for each expert as follows:

$$w_{t+1}(i) = w_t(i) \cdot (1 - \eta \ell_t(i))$$

where $\ell_t(i)$ is the loss (or regret) suffered by the i-th policy in round t. This value must be in $[-1, 1]$ for all i. We use max-min scaling on expected rewards depending on the policy followed. Suppose the agent was at states $s_0, s_1, \ldots, s_{n_t - 1}$ during round t. We can calculate the expected reward for policy π_i during state s_j as follows:

$$R^E_{i,j} = \sum_{s \in S} R(s_j, \pi_i(s_j), s) P(s_j, \pi_i(s_j), s).$$

Similarly, we can calculate the minimum and maximum possible rewards when the agent is at state s_j as follows:

$$R_j^{min} = \min_{a \in A} \sum_{s \in S} R(s_j, a, s) P(s_j, a, s)$$

and

$$R_j^{max} = \max_{a \in A} \sum_{s \in S} R(s_j, a, s) P(s_j, a, s).$$

Finally, using these three computations, we can calculate the feedback for each policy as follows:

$$\ell_t(i) = 1 - \frac{2}{n_t} \sum_{j=0}^{n_t - 1} \frac{R_{i,j}^E - R_j^{min}}{R_j^{max} - R_j^{min}}.$$

While this approach is theoretically sound, it suffers from computational issues due to the exponential number of policies, making it intractable for large state spaces.

4.2 Grid-Based Experts

To address the computational issue of the previous approach, we can exploit the structure of the RL problem by considering experts for each state-action pair instead of complete policies. This reduces the number of experts from $|A|^{|S|}$ to $|S| \times |A|$, making the problem more manageable, though experts are now restricted to particular pre-defined corresponding states.

In this approach, the weight initialization remains uniform, but weights are in a matrix form, i.e., $\mathbf{w} = (w(s,a))_{s \in S, a \in A}$, where $w(s,a) = 1$ initially. The weight update needs enhancement. Note that not every expert is competing at once. At iteration t, suppose the agent is in state s_t; then only experts associated with state s_t are competitors, and thus only their weights need to be updated.

At every iteration, given state s_t, to continue the simulation, an action a_t is selected from A uniformly with the weights, i.e., each action a_j has probability $\frac{w(s_t, a_j)}{\sum_{a \in A} w(s_t, a)}$. Note, here, we have three kinds of experts after the selection of action. The expert associated with (s_t, a_t) is called a believer, and other experts associated with state s_t are called non-believers. The rest of the experts are not applicable experts for this particular iteration and thus their weights do not get updated for this iteration.

The weights for the current state s_t and for all actions $a \in A$ are updated using the MWU update rule:

$$w(s_t, a) \leftarrow \begin{cases} w(s_t, a) \cdot (1 - \eta \ell_t) & \text{if } a = a_t \text{ (believer)} \\ w(s_t, a) \cdot (1 + \eta \ell_t) & \text{otherwise (non-believers)} \end{cases}.$$

where $\ell_t \in [-1, 1]$ is the loss (or regret) suffered by taking action a in state s_t during round t. We again use max-min scaling and a hyperparameter n for controlling exploration-exploitation. This can be done by collecting the rewards r_t in each round but updating the weights after n rounds with the sum of rewards. The max-min scaling can be done by keeping the history of minimum and maximum cumulative rewards corresponding to each state s as $R_{min}^{cumm}(s)$ and $R_{max}^{cumm}(s)$.

The loss ℓ_t can be calculated as:

$$\ell_t = 1 - 2 \cdot \frac{\sum_{j=t}^{t+n} r_t - R_{min}^{cumm}(s_t)}{R_{max}^{cumm}(s_t) - R_{min}^{cumm}(s_t)}.$$

Here, the minimum and maximum cumulative rewards get updated with the following rules:

$$R_{min}^{cumm}(s_t) \leftarrow \min \left(R_{min}^{cumm}(s_t), \sum_{j=t}^{t+n} r_t \right)$$

and

$$R_{max}^{cumm}(s_t) \leftarrow \max \left(R_{max}^{cumm}(s_t), \sum_{j=t}^{t+n} r_t \right).$$

This approach does not require knowledge of transition probabilities and is similar to Q-learning but with the addition of the MWU update rule for adaptively adjusting the weights of state-action pairs. By dynamically updating the weights based on performance, this method ensures a more efficient learning process that balances exploration and exploitation.

4.3 Reward Strategy Experts with Human Feedback

Another challenge in RL is designing appropriate reward functions or strategies. Typically, a reward function must consider multiple objectives, such as energy consumption, task completion time, and safety constraints. Balancing these often conflicting goals requires a nuanced approach.

To address this, we propose treating different reward strategies as "experts" and employing the Multiplicative Weights Update (MWU) algorithm to combine their outputs into a single, optimal reward function. This approach leverages the strengths of each strategy while adapting dynamically based on their performance.

Consider $R_1(s, a), R_2(s, a), \ldots, R_K(s, a)$ as K different reward strategies for each state-action pair (s, a). We initialize a weight vector, $\mathbf{w}_0$, where each element is set to one, representing equal importance initially.

At each time step t, we compute the combined reward signal as the weighted sum of the individual strategies:

$$R^t(s, a) = \sum_{k=1}^{K} w_t(k) R_k(s, a).$$

The weights are then updated using the MWU rule:

$$w_{t+1}(k) = w_t(k) \cdot (1 - \eta \ell_t(k)).$$

Here, $\ell_t(k) \in [-1, 1]$ represents the loss or regret associated with the k-th reward strategy in round t. This loss is computed using human feedback, which provides qualitative assessment beyond automated metrics.

Human feedback, denoted $H_t \in [-1, 1]$, ranges from -1 (completely unsatisfied) to 1 (completely satisfied). We integrate this feedback to adjust the weights, encouraging strategies that perform well and discouraging those that do not. The loss $\ell_t(k)$ is computed as follows:

$$\ell_t(k) = \begin{cases} H_t & \text{if } \frac{w_t(k)}{\sum_{j=1}^{K} w_t(j)} < \frac{1}{K} \text{ (lower weights)} \\ -H_t & \text{if } \frac{w_t(k)}{\sum_{j=1}^{K} w_t(j)} > \frac{1}{K} \text{ (higher weights)} \\ 0 & \text{otherwise} \end{cases}$$

This method ensures that underperforming strategies are penalized more if they happen to have higher weights thus dictating the strategy, and rewarded more if they have lower weights and perform well, aligning with human preferences. In short, this can again be seen as the strategies with weights higher than the average are believers and thus get updated to higher weight if they perform well otherwise to lower weight. This adaptive mechanism allows the system to learn an optimal reward function that balances multiple objectives or constraints effectively.

Incorporating human feedback into RL is a well-explored concept, with numerous studies demonstrating its effectiveness (e.g., (11), (12)). By dynamically adjusting the weights through MWU, our approach enhances the robustness and adaptability of the reward function in complex environments.

5 Conclusion

In this paper, we have explored the integration of the Multiplicative Weight Update (MWU) Algorithm into traditional Reinforcement Learning (RL) frameworks. We discussed treating every possible policy as an expert and using MWU algorithm to update their weights based on performance and employing grid-based experts for more efficient updates.

Looking ahead, future research directions could include developing more advanced clustering techniques for merging experts, integrating MWU with other RL methods such as policy gradients, and exploring applications in safety-critical domains where robust and efficient learning is paramount. Additionally, theoretical analyses and empirical evaluations across various domains will be crucial in validating the performance and robustness of MWU-enhanced RL algorithms.

Overall, the integration of MWU algorithm into RL frameworks represents a promising step towards more efficient, adaptive, and robust decision-making systems, paving the way for exciting new developments in the field of reinforcement learning and its numerous applications.

6 Acknowledgements

While writing this paper, AI tools like Grammarly and Claude were used to improve the grammar and sentence flow.

References

[1] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[2] S. Arora, E. Hazan, and S. Kale, "The multiplicative weights update method: a meta-algorithm and applications," *Theory of computing*, vol. 8, no. 1, pp. 121–164, 2012.

[3] Wikipedia, "Reinforcement learning diagram." `https://en.wikipedia.org/wiki/Reinforcement_learning#/media/File:Reinforcement_learning_diagram.svg`, 2023. Accessed: 21 April 2024.

[4] R. Bellman, "The theory of dynamic programming," *Bulletin of the American Mathematical Society*, vol. 60, no. 6, pp. 503–515, 1954.

[5] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, pp. 279–292, 1992.

[6] V. Konda and J. Tsitsiklis, "Actor-critic algorithms," *Advances in neural information processing systems*, vol. 12, 1999.

[7] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*, pp. 1928–1937, PMLR, 2016.

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

[9] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[10] N. Littlestone and M. K. Warmuth, "The weighted majority algorithm," *Information and computation*, vol. 108, no. 2, pp. 212–261, 1994.

[11] Z. Li, Z. Yang, and M. Wang, "Reinforcement learning with human feedback: Learning dynamic choices via pessimism," *arXiv preprint arXiv:2305.18438*, 2023.

[12] B. Zhu, M. Jordan, and J. Jiao, "Principled reinforcement learning with human feedback from pairwise or k-wise comparisons," in *International Conference on Machine Learning*, pp. 43037–43067, PMLR, 2023.