Reinforcement Learning in Match-3 Game

Ahmadsho Akdodshoev
a.akdodshoev@innopolis.university

Danil Andreev
d.andreev@innopolis.university

Vladimir Makharev
v.makharev@innopolis.university

Abstract

In this project, our objective was to solve the Match-3 game using Reinforcement Learning techniques. Our team started by reproducing the results of a well-cited paper [5] that proposed an environment implementation with a Gym interface. We encountered some issues in the original environment and successfully fixed them while replicating the paper's algorithms. Our reproduction is available on GitHub[1].

To enhance computational efficiency and scalability, we extended this work by re-implementing the environment using JAX, ensuring full compatibility with Gymnax. We rigorously tested the correctness of the environment by evaluating PPO, greedy, and random agents. The JAX-based implementation has been packaged as an open-source library on GitHub[2], facilitating accessibility for further experimentation and research. Additionally, we developed an interactive demo (included in the repository) that enables users to compete against trained agents, demonstrating the practical application of our work.

1 Introduction

This project delves into Match-3 tile-matching games, where players manipulate tiles to make them disappear based on specific matching rules, such as aligning three identical tiles in a row or column. These games are popular due to their simple gameplay and short sessions, making them perfect for casual play. Developers face challenges in designing levels that balance difficulty and player engagement, often relying on time-consuming human testing or simulations to assess level complexity.

To address these challenges, inspired by the paper "Deep Reinforcement Learning Methods in Match-3 Game" [5], we first reproduce this paper and then implement a new environment with a Gymnax interface. Therefore, we contribute to applying Reinforcement Learning on Match-3 game allowing agents to play numerous game sessions quickly and achieve more human-like results through imitation learning.

The main contribution of this work is the creation of an open-source JAX-based environment, making it accessible and extendable for the latest reinforcement learning research. The environment supports classic configurations from games like "Bejeweled" (see example on Fig. 1) and includes options for the number of tiles, grid size, and holes configuration. The work also compares the performance of popular RL algorithms, such as Advantage Actor-Critic and Proximal Policy Optimization, against random and greedy action policies, providing insights into their effectiveness in a model-free setting.

Figure 1. Bejeweled game board example.

2 Related Work

In scarce recent studies on Match-3 games, Kamaldinov & Makarov [5] introduced an open-source business-oriented Match-3 environment and evaluated the performance of deep reinforcement learning algorithms, while Dukkancı [3] developed an automated pipeline that combines genetic algorithms for procedural level generation with reinforcement learning to test difficulty, reducing manual design time significantly. Park & Lee [4] implemented a PPO-based reinforcement learning agent using Unity3D ML-Agents, achieving a 44% performance improvement and human-like strategic decision-making. Napolitano [8] used a dueling DQN approach to automate testing in the Jelly Juice game, achieving success rates comparable to human testers and adapting strategies as level difficulty increased. Additionally, Shin et al.

[1] https://github.com/simply-pleb/RL-match-three
[2] https://github.com/InnoRL/match-three-env

[11] formalized five strategic play types within an advantage actor-critic framework for automatic playtesting, achieving performance within 5% of human testers on complex missions and establishing standardized testing methodologies for Match-3 games.

3 Methodology

The methodology includes three stages:

1. Replication of results using the environment found in [5].
2. Creation of a JAX-based environment for match-3 game.
3. Replication and improvement of [5] results using the JAX-based environment.

All agents were tasked with solving a variant of the match-3 game with the following constraints:

- The maximum number of steps in the environment is set to 100. Each action by the agent, whether it matches or not, adds 1 to the step counter.
- The size of the game grid is set to 9×9 with four unique symbol types to match.
- All symbols on the game grid are symbols that could produce a match, i.e., there are no symbols that cannot be displaced.

The exact network architectures of agents are available on our GitHub repositories.

3.1 Original environment

We reproduced the original environment from GitHub[3]. This required debugging and resolving issues with running the environment, as well as fixing the reward function. Due to the lack of agent implementations in their repository, we replicated the Asynchronous Advantage Actor-Critic (A3C) [7] and Random agents as closely as possible to validate paper's results.

3.2 JAX-based environment

We implemented our version of the environment using JAX framework[2]. We adopted the interface of Gymnax, which is a JAX-based library for reinforcement learning environments, designed to leverage JAX's capabilities for high-performance, parallelized, and hardware-accelerated computations.

3.2.1 Main environment procedures. The main logic of the environment can be described using the following procedures.

Grid initialization. The grid is initialized randomly, then the matches are replaced with random elements until the grid stabilizes.

Match detection. The match detection is done by applying roll in both axis. The exact implementation can be found in the environment repository. Initially, there was also an implementation using predefined masks for matches, but after comparing both approaches in terms of execution time, it was abandoned.

Grid collapse. When the matches are removed, elements are moved from top to bottom column-wise, filling the empty cells. New elements are generated randomly from the top.

Cascade match processing. If there are matches after collapsing the grid, these matches are removed and grid is collapsed once again. This procedure continues until the grid stabilizes.

3.3 Reinforcement Learning in the JAX-based Environment

3.3.1 Reward and return. Reward R is a signal that an agent receives after interacting with the environment by applying an action from a state. The goal of the agent is to maximize the cumulative reward $R = \sum_{t=0}^{\infty} r_t$. Reinforcement learning agents use return G_t from step t, also known as discounted return, to maximize the cumulative reward. Return is the sum of discounted rewards

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1},$$

where γ is the discount rate[12].

The reward in the JAX-based environment is calculated as

$$r_t = \sum_{k=1}^{k \leq K} C \cdot m_t^{(k)} \cdot \log_2(k),$$

where t is the current time-step, m_t are the match cascades on step t ($m_t^{(k)}$ is the k-th cascade which stores the number of symbols that matched on step t on k-th cascade), K is the maximum number of cascades allowed and C is some constant. It should be noted that the original environment has another reward function.

We used actor-critic [6] paradigm which can be viewed as a variant of REINFORCE with a *learned* baseline [13]. Actor-critic approaches require finding both an optimal value function and an optimal policy.

3.3.2 Value function. Value is the expected return that an agent can receive from a state

$$V(s) = \mathbb{E}[G_t | s_t = s]$$

It is expressed as a function that maps state to expected return (value) [12]. A neural network can be used to learn a value function by using MSE loss

$$\mathcal{L}_v = \mathbb{E}\left[(\text{Target Value} - \text{Predicted Value})^2\right]$$

[3] https://github.com/kamildar/gym-match3

We use Huber loss to learn the value function

$$\mathcal{L}_v = \begin{cases} \frac{1}{2}(v-\hat{v})^2 & \text{if } |v-\hat{v}| \le \delta \\ \delta\left(|v-\hat{v}| - \frac{1}{2}\delta\right) & \text{otherwise} \end{cases},$$

where δ is a bound that is set to 1.

3.3.3 Policy. Policy maps state to probability distribution of actions $\pi(a_t|s_t)$. Improving a policy could be viewed as finding a strategy to solve an environment. Policy gradient is a method to optimize a policy approximated by a neural network [13]. The policy gradient loss is expressed as

$$\mathcal{L}_\pi = -\mathbb{E}\left[\sum_{t=0}^{T} \log \pi(a_t|s_t) A_t\right],$$

where A_t is an advantage function helps to quantify how much better an action is compared to the average action at a given state

$$A(s_t, a_t) = Q(s_t, a_t) - V(s_t),$$

where $Q(s_t, a_t)$ is the expected return for taking action a_t in state s_t, as compared to $V(s_t)$ which is the expected return in state s_t. In pratice, advantage function is replaced by TD error[12]

$$\delta_t = r + \gamma V(s_{t+1}) - V(s_t)$$

where $Q(s_t, a_t)$ is substituted by $r + \gamma V(s_{t+1})$. We have used generalized advantage estimation (GAE) [9] which is a more accurate estimation of the advantage function

$$A_t^{\text{GAE}} = \sum_{k=0}^{\infty} (\gamma\lambda)^k \delta_{t+k},$$

where γ is the discount rate and λ can be viewed as a parameter that control bias-variance tradeoff.

3.3.4 PPO and value clipping. PPO [10] is a technique that improves the training stability of the policy gradient which can be expressed as

$$f = \frac{\pi(a_t|s_t)}{\pi_{\text{old}}(a_t, s_t)}$$

$$\mathcal{L}_\pi^{\text{CLIP}} = \mathbb{E}\left[\min\left(f A_t^{\text{GAE}}, \text{clip}\left(f, 1-\epsilon, 1+\epsilon\right) A_t^{\text{GAE}}\right)\right]$$

Value clipping is an analogous technique that improves the training stability of the value function

$$\hat{v}_{\text{clip}} = \text{clip}(\hat{v}, \hat{v}_{old} - \epsilon, \hat{v}_{old} + \epsilon)$$

$$\mathcal{L}_v^{\text{CLIP}} = \mathbb{E}\left[\max\left((\hat{v} - v)^2, \left(\hat{v}_{\text{clip}} - v\right)^2\right)\right]$$

3.3.5 Pretraining CNN. CNN is pretrained on $1,000,000$ grid examples on auto-encoder task. The encoder is used to find latent representations of game grids.

3.3.6 RL agent variations. Two types of agents are trained

- Agents with discrete actions that map an action to a specific tile and swap direction. The total number of actions is calculated $w(h-1) + (w-1)h$, where w is the width of a grid, and h is the height of a grid.
- Agents with continuous actions that output 3 values (h, w, d) where $h, w, d \in [0, 1]$, where h is a standardized height, w is a standardized width, and d is a logit for swap direction (horizontal or vertical).

4 Experiments and Evaluation

4.1 Replicated Agents on Original Environment

We selected Scenario 1 from the original paper and used a random 9×9 game grid with four tiles and no holes. For reproduction in this Scenario 1, we implemented the agents detailed below. The obtained results are presented in Fig. 3 and discussed in the next section.

Random Agent. We replicated a random agent as a baseline. This agent selects actions uniformly from all possible movements, including illegal ones.

A3C Agent. We replicated the A3C agent since it demonstrated the best performance in the selected paper. Based on the architecture description of this agent in the selected paper, we implemented it along with the original environment.

4.2 Baseline Agents

As a baseline we implemented random and greedy agents.

Random Agent. Random agent chooses action uniformly from all possible movements, even illegal ones.

Greedy Agent. Greedy agent performs each action and gets returns from the environment. Then, the action with the biggest return is selected. Note, that the agent does not know the cascading rewards, the action is performed to get the return only from the first match after the move is done.

Table 1. Performance of the Uniformly Random Policy

Metric	Value
Number of episodes	100
Steps in episode	100
Mean return (± std)	**336.1** ± 141.9

Table 2. Performance of the Greedy Agent

Metric	Value
Number of episodes	100
Steps in episode	100
Mean return (± std)	**1843.8** ± 676.2

4.3 Demo

We implemented the demo for environment visualization and manual agent assessment. The demo is written in Python using FastAPI framework, the frontend is implemented using pure HTML, JS, and CSS.

The demo allows user to interact with an environment, and see the moves of random, greedy, and PPO agents on the copies of the same environment.

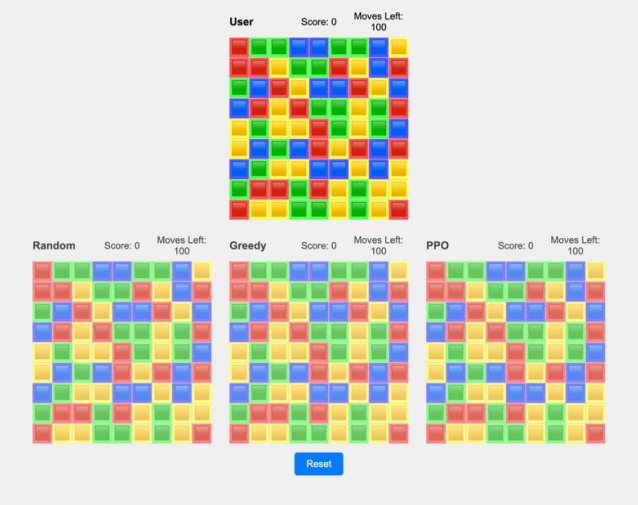

Figure 2. Interface of the demo.

5 Analysis and Observations

Results are reproducible and looks similar to paper. JAX-based models have similar performance to the original model without pretrained CNN. JAX-based model with discrete actions has better performance than the original model. JAX-based model with continuous actions did not show significant improvements. Greedy the best and was probably omitted due its high performance.

By solving the environment with the discrete JAX-based agent and the maximum number of steps of 10 we achieved a cumulative reward of 141.69. It extrapolates to 1416.9 for the maximum number of steps of 100. This demonstrates the need to conduct an experiment with curriculum learning[1] by gradually moving from solving 10 to 100 maximum number of steps.

5.1 Results

Fig. 3 shows that our reproduction of the A3C and Random agents on the original environment aligned with the results presented in the paper (see Fig. 3 in the paper). This alignment confirmed the lack of improvement of the A3C agent over the Random agent. Note that the reward function was slightly modified (mean cumulative reward differs) to enable the agents to learn.

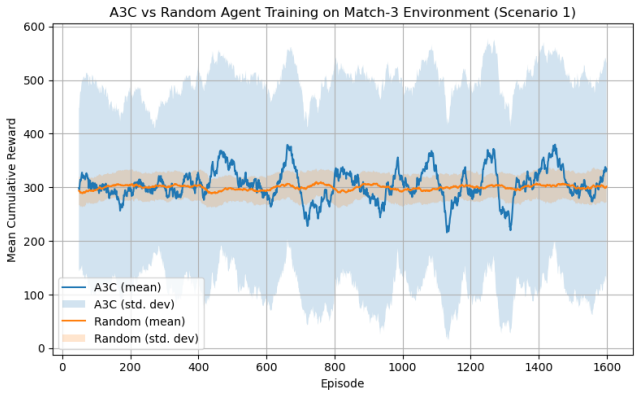

Figure 3. Comparison of reproduced A3C and Random agents solving the original environment in Scenario 1.

The Table 3 highlights significant performance differences among agents, with the extrapolated PPO agent with discrete actions achieving the highest average cumulative reward, though its large standard deviation suggests variability. The greedy agent outperforms basic PPO variants, while the random agent performs the weakest.

Table 3. Average Cumulative Reward of Agents in the JAX-based Environment

Model	Average Cumulative Reward
Random agent	336.1 ± 141.9
Greedy agent	1843.1 ± 676.2
PPO (continuous actions)	~ 342.6 ± 40.0
PPO (discrete actions)	~ 557.8 ± 200.0
PPO (discrete, 10-step)*	~ 1416.9 ± 300.0

*The results were extrapolated to 100 steps.

6 Conclusion

In conclusion, this project explores the application of reinforcement learning to Match-3 tile-matching games, addressing the challenge of balancing level difficulty and player engagement. Inspired by the paper "Deep Reinforcement Learning Methods in Match-3 Game", we reproduced the original work and developed a new, efficient JAX-based open-source environment that supports parallel learning for agents. This environment, compatible with classic configurations from games like "Bejeweled," offers flexibility in tile number, grid size, and holes configuration. Our findings indicate that pretraining CNN and using discrete actions significantly enhance agent performance. Additionally, curriculum learning shows promise for future work, suggesting the need for experiments that gradually increase the complexity of the task from 10 to 100 maximum steps. This work contributes to the field by providing a versatile platform for testing and

comparing popular RL algorithms, such as Advantage Actor-Critic and Proximal Policy Optimization, against random and greedy action policies in a model-free setting.

References

[1] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. 2009. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning (ICML '09)*. ACM. https://doi.org/10.1145/1553374.1553380

[2] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs*. http://github.com/jax-ml/jax

[3] Samet Alp Dukkancı. 2021. *Level generation using genetic algorithms and difficulty testing using reinforcement learning in match-3 game*. Master's thesis. Middle East Technical University.

[4] Dae geun Park and Wan-Bok Lee. 2021. Design and Implementation of Reinforcement Learning Agent Using PPO Algorithim for Match 3 Gameplay. *Journal of Convergence Information Technology* 11 (2021), 1–6. https://api.semanticscholar.org/CorpusID:236693866

[5] Ildar Kamaldinov and Ilya Makarov. 2019. Deep Reinforcement Learning Methods in Match-3 Game. In *International Conference on Analysis of Images, Social Networks and Texts*. Springer, 51–62.

[6] Vijay Konda and John Tsitsiklis. 1999. Actor-Critic Algorithms. In *Advances in Neural Information Processing Systems*, S. Solla, T. Leen, and K. Müller (Eds.), Vol. 12. MIT Press. https://proceedings.neurips.cc/paper_files/paper/1999/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf

[7] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous Methods for Deep Reinforcement Learning. *CoRR* abs/1602.01783 (2016). arXiv:1602.01783 http://arxiv.org/abs/1602.01783

[8] Nicholas Napolitano. 2020. Testing match-3 video games with Deep Reinforcement Learning. *CoRR* abs/2007.01137 (2020). arXiv:2007.01137 https://arxiv.org/abs/2007.01137

[9] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. 2015. High-dimensional continuous control using generalized advantage estimation. (2015). arXiv:1506.02438 [cs.LG]

[10] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. *CoRR* abs/1707.06347 (2017). arXiv:1707.06347 http://arxiv.org/abs/1707.06347

[11] Yuchul Shin, Jaewon Kim, Kyohoon Jin, and Young Bin Kim. 2020. Playtesting in Match 3 Game Using Strategic Plays via Reinforcement Learning. *IEEE Access* 8 (2020), 51593–51600. https://doi.org/10.1109/ACCESS.2020.2980380

[12] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction* (second ed.). The MIT Press. http://incompleteideas.net/book/the-book-2nd.html

[13] Ronald J. Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning* 8, 3–4 (May 1992), 229–256. https://doi.org/10.1007/bf00992696