Learning to Play 2048 with Reinforcement Learning: A Comparative Study of Learning Behaviors and Strategies

Shardul Junagade, Tejas Lohia, Umang Shikarvar, Zainab Kapadia
Department of Computer Science and Engineering,
Indian Institute of Technology Gandhinagar, India
Email: {shardul.junagade, tejas.lohia, umang.shikarvar, zainab.kapadia}@iitgn.ac.in

***Abstract*—We built a shared 2048 environment and tried a range of agents: tabular Q-learning, Double Q-learning, several SARSA variants, PPO, a lightweight MuZero, and a hand-crafted Expectimax planner, plus a random baseline. Our goal was to see how each method actually learned (or failed to) under the same reward shaping and episode budget rather than to chase leaderboard scores. The Q-based agents improved steadily; Double Q smoothed some spikes. SARSA (especially the classical n-tuple) and PPO edged out tabular Q on average score. MuZero Lite was constrained by shallow search and small networks. Expectimax, with domain heuristic and lookahead, was far ahead in both average score (> 33k) and win rate ($\approx 74\%$). We outline the concrete update rules we used (including shaping terms) so results are reproducible.**

***Index Terms*—Reinforcement learning, 2048, Q-learning, Double Q-learning, Expectimax, game playing**

I. Introduction

Our starting point was simple: build one clean 2048 environment and put several learning and planning approaches side by side to see how they actually behaved. 2048 gives us a small action set, stochastic tile spawns, and enough combinatorial state variety to expose differences in exploration and value estimation. We wrote our own environment and a minimal Tkinter UI so that every agent interacted with identical game logic.

In this project, we added following agents: random baseline, Expectimax (heuristic search), tabular Q-learning (multi α sweep), Double Q-learning, SARSA afterstate value, SARSA n-tuple, PPO, and a compact MuZero variant. We also made brief attempts at vanilla DQN and a simple actor-critic; those two stalled early given our hardware and time budget, so we redirected effort to PPO and MuZero.

The code is publicly available at https://github.com/Umang-Shikarvar/2048-agent.

II. Objective

Our main objectives in this project were:

- to implement a clean and reusable environment for the 2048 game that could be shared by multiple learning algorithms;
- to design and implement several agents, including tabular Q-learning, Double Q-learning and a heuristic Expectimax agent;
- to train the value-based agents on the same environment and compare their learning behaviour over episodes;
- to evaluate all agents on final performance metrics such as average score, best score and highest tile distribution;

III. Algorithms Used

In this section we describe the main algorithms we used. Throughout, we modelled the 2048 game as a Markov Decision Process (MDP) with state space $\mathcal{S}$ given by the grid configurations, action space $\mathcal{A} = \{\text{up}, \text{right}, \text{down}, \text{left}\}$, discount factor $\gamma \in [0, 1]$ and reward signal $r \in \mathbb{R}$ defined by score changes and shaping terms.

A. Random Baseline

As a simple baseline we included a random policy that selected a legal move uniformly at random. This policy provided a lower bound on performance and helped validate the environment implementation.

B. Heuristic Expectimax Agent

In addition to learning-based agents, we implemented an Expectimax search agent that did not learn from data but instead planned ahead using a hand-crafted evaluation function. For a given depth d, the agent simulated action nodes (our moves) and chance nodes (random tile insertions) in a lookahead tree and chose the action with the highest expected heuristic value.

Formally, for a state s we defined the value of a max node at depth d as

$$V_{\text{max}}(s, d) = \begin{cases} h(s), & d = 0, \\ \max\limits_{a \in \mathcal{A}(s)} \mathbb{E}\left[V_{\text{chance}}(s', d-1)\right], & d > 0, \end{cases} \quad (1)$$

where $h(s)$ was our heuristic and s' were successor states.

At chance nodes corresponding to a random tile spawn, we aggregated over all empty cells and tile values:

$$V_{\text{chance}}(s, d) = \sum_{i \in E(s)} \sum_{x \in \{2,4\}} p(i, x)\, V_{\text{max}}(s^{(i,x)}, d), \quad (2)$$

where $E(s)$ was the set of empty positions, $s^{(i,x)}$ was the state after inserting tile x at position i, and $p(i, x)$ was the probability of that event (we used 0.9 for the tile 2 and 0.1 for the tile 4, spread uniformly over empty cells).

Our heuristic $h(s)$ combined several intuitive features that we found important in 2048: the number of empty tiles, whether the maximum tile was kept in a corner, monotonicity along rows or columns, and the smoothness of neighbouring tiles. We used a fixed linear combination

$$h(s) = w_1 \cdot \text{empty}(s) + w_2 \cdot \text{corner}(s) + w_3 \cdot \text{mono}(s) + w_4 \cdot \text{smooth}(s) \tag{3}$$

with manually chosen weights $w_1, \ldots, w_4$.

C. Tabular Q-Learning

We kept a table $Q(s, a)$ and trained for 20,000 episodes per learning rate $\alpha \in \{0.05, 0.10, 0.15, 0.25\}$. Each episode started from two tiles. At a non-terminal state we formed a canonical representation by reflecting horizontally (as in the code) and taking the lexicographically smaller variant to reduce symmetry duplicates.

Action selection used ϵ-greedy with ϵ initialised to 1.0, multiplicatively decayed by 0.9993 every step, and floored at 0.10.

After choosing a_t we executed the move. The environment returned a merge reward r_t^{merge} (sum of the merged tile values) or -1 if the board did not change. The shaping we actually used in code was:

$$\tilde{r}_t = \begin{cases} -10, & \text{if move invalid} \\ r_t^{\text{merge}} + 2\,\text{score}(s_{t+1}) + 50\,\text{empty}(s_{t+1}), & \text{otherwise} \end{cases} \tag{4}$$

Then we spawned a new tile and checked termination. The update was standard off-policy Q-learning

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\big[\tilde{r}_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)\big], \tag{5}$$

with $\gamma = 0.98$. On terminal transitions we used a constant penalty of -200 (as in code) for the bootstrap term.

D. Double Q-Learning

Double Q-learning used two tables Q_1, Q_2, trained for the same 20,000 episodes per α with ϵ decay 0.99935 (slightly different) and floor 0.10. Policy selection used ϵ-greedy over the sum $Q_1 + Q_2$.

Reward shaping matched the code:

$$\tilde{r}_t = \begin{cases} -20, & \text{if move invalid} \\ r_t^{\text{merge}} + 5\,\Delta\text{empty}_t - 1, & \text{otherwise} \end{cases} \tag{6}$$

where $\Delta\text{empty}_t = \text{empty}(s_{t+1}) - \text{empty}(s_t)$. A new tile was spawned only if the move changed the board.

At each step we flipped a fair coin to decide which table to update. Updating Q_1:

$$Q_1(s_t, a_t) \leftarrow Q_1(s_t, a_t) + \alpha\big[\tilde{r}_t + \gamma Q_2(s_{t+1}, a^*) - Q_1(s_t, a_t)\big], \tag{7}$$

with $a^* = \arg\max_{a'} Q_1(s_{t+1}, a')$. Symmetric update for Q_2 swaps indices. This separation curbed optimistic bias we saw in early tabular runs.

E. SARSA with Afterstate Value Function

In addition to the off-policy methods above, we implemented an on-policy SARSA agent that used an afterstate value function. In 2048 the afterstate is the board configuration immediately after applying the agent's move but before the environment spawns a random tile. We denoted afterstates by s'_t.

We maintained a scalar value $V(s')$ for each afterstate. At time step t we started from a pre-move state s_t, considered each legal action a, and looked ahead to the corresponding afterstate $s'(s_t, a)$. Our ϵ-greedy policy operated on afterstate values:

$$a_t = \begin{cases} \text{uniform random legal action}, & \text{with probability } \epsilon, \\ \arg\max\limits_{a \in \mathcal{A}(s_t)} V\big(s'(s_t, a)\big), & \text{with probability } 1 - \epsilon. \end{cases} \tag{8}$$

After choosing a_t we applied the move to obtain $s'_t = s'(s_t, a_t)$, observed the merge reward and then let the environment add a random tile, leading to the next pre-move state s_{t+1}. Using the policy again on s_{t+1}, we obtained the next action a_{t+1} and corresponding afterstate $s'_{t+1} = s'(s_{t+1}, a_{t+1})$. The SARSA(0) update on afterstates was

$$V(s'_t) \leftarrow V(s'_t) + \alpha\left[r_t + \gamma V(s'_{t+1}) - V(s'_t)\right], \tag{9}$$

where r_t was the shaped reward assigned to the transition.

At the end of an episode, when no legal moves remained, we set the bootstrap term to zero, so the final update became

$$V(s'_t) \leftarrow V(s'_t) + \alpha\left[r_t - V(s'_t)\right]. \tag{10}$$

F. SARSA n-Tuple Function Approximation

We also experimented with a classical SARSA agent combined with an n-tuple function approximator over state-action pairs. The idea was to represent the action-value function as a linear combination of features extracted from several fixed index tuples on the board.

We defined a set of tuples $T_1, \ldots, T_K$, where each T_k was a small ordered subset of board indices (for example, a row, column or diagonal). For each tuple and action we allocated a table of parameters, and for a given (s, a) we mapped the tiles on T_k to an index into that table. Concatenating all such lookups gave us a sparse feature vector $\phi(s, a)$.

The approximate action-value function took the form

$$Q(s, a; \mathbf{w}) = \mathbf{w}^\top \phi(s, a), \tag{11}$$

where $\mathbf{w}$ stacked all n-tuple parameters. At each on-policy transition $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$ we computed the temporal-difference error

$$\delta_t = r_t + \gamma Q(s_{t+1}, a_{t+1}; \mathbf{w}) - Q(s_t, a_t; \mathbf{w}), \tag{12}$$

and updated the weights by a gradient step

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha\, \delta_t\, \phi(s_t, a_t). \tag{13}$$

Action selection again used an ϵ-greedy policy over the approximate values $Q(s, a; \mathbf{w})$.

G. Proximal Policy Optimization (PPO)

We implemented PPO (clipped surrogate objective) as one of the main policy-gradient methods. PPO updates a parameterised policy π_θ by maximising a clipped surrogate objective to avoid large policy updates in a single step.

Given a batch of advantage estimates $\hat{A}_t$ (computed with a value baseline), PPO used the ratio

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} \tag{14}$$

and optimised the clipped loss

$$L^{\text{CLIP}}(\theta) = -\mathbb{E}_t \left[\min \left(r_t(\theta)\hat{A}_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right]. \tag{15}$$

In practice we used mini-batches and multiple epochs of stochastic gradient descent on the surrogate objective per data batch. Our implementation followed the standard actor-critic variant where the value function was updated separately with an ℓ_2 loss and the policy used the clipped surrogate above. PPO provided good empirical stability for the continuous updates we required in the 2048 domain.

H. MuZero (Lite)

We also experimented with a lightweight MuZero-style agent (model-based planning with learned dynamics) adapted to 2048 under tight compute constraints. We used three learned components similarly to the original formulation: a representation function h_ϕ, a dynamics function g_ψ, and a prediction function f_ω.

a) Latent Inference and Unrolling.: From an observed board o_0 we produced a latent state

$$s_0 = h_\phi(o_0). \tag{16}$$

After selecting an action a_k, we applied the learned dynamics to obtain the next latent and immediate reward prediction

$$(s_{k+1}, \hat{r}_{k+1}) = g_\psi(s_k, a_k). \tag{17}$$

Each latent s_k was passed through the prediction head to get a value estimate $\hat{v}_k$ and a prior policy $\hat{p}_k$ over actions:

$$(\hat{p}_k, \hat{v}_k) = f_\omega(s_k). \tag{18}$$

b) MCTS with PUCT.: Planning used a limited number of simulations per move. At each search node the action was chosen via the PUCT rule

$$a^* = \arg\max_a \left[Q(s,a) + U(s,a) \right], \tag{19}$$

with exploration term

$$U(s,a) = c_{\text{puct}}\, \hat{p}(a \mid s) \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)}, \tag{20}$$

where $N(s,a)$ was the visit count and $Q(s,a)$ the running mean of backed-up value estimates.

c) Targets.: For a trajectory we formed training targets for reward, value and policy. The (truncated) n-step return for value targets was

$$v_t^{\text{target}} = \sum_{i=0}^{n-1} \gamma^i r_{t+i} + \gamma^n \hat{v}_{t+n}, \tag{21}$$

and the policy target p_t^{target} was the normalised visit count distribution from the root after MCTS.

d) Loss.: We optimised a combined objective

$$\mathcal{L} = \sum_t \left[\ell_v(\hat{v}_t, v_t^{\text{target}}) + \ell_r(\hat{r}_t, r_t) + \ell_p(\hat{p}_t, p_t^{\text{target}}) \right] + \lambda \|\Theta\|_2^2, \tag{22}$$

where Θ collected all parameters and λ was a small weight decay. We used mean-squared error for ℓ_v, cross-entropy for ℓ_p and MSE for ℓ_r.

Due to limited simulations per move and modest network sizes this "MuZero Lite" did not reach the performance of our hand-crafted Expectimax but it provided a useful comparison between learned and hand-coded planning.

I. Notes on DQN and Actor-Critic Attempts

We attempted a vanilla DQN and a simple actor-critic implementation. These two approaches did not converge reliably within our compute budget: after roughly 5,000 training epochs they still showed high variance and unstable value estimates. Given the limited GPU/CPU time available, we focused our effort on PPO and MuZero where results were more promising in this setting.

IV. Methodology

A. Environment and State Representation

We implemented a unified environment for the 4×4 version of 2048. The environment maintained the board as a flat array of length 16, stored tile values in log-space (for example, a tile with value 2^k was represented as the integer k), and provided helper functions for moving, merging and spawning tiles.

The action set consisted of four actions: up, right, down and left. For each step the environment first applied the chosen action, computed the merge reward and updated the score. If the move changed the board, it then spawned a new tile (either a 2 or a 4) at a random empty position.

We exposed this environment both as a non-visual simulator for training and as a simple graphical interface built with Tkinter. All agents interacted with the same 'Game2048' core logic so that the comparison across algorithms remained fair.

B. Reward Design

The base environment returns r_t^{merge} (sum of newly created tile values) for a valid move, and -1 for an invalid/no-change move before shaping. Our shaping formulas were intentionally simple and reflect the actual code paths:

$$\tilde{r}_t^Q = \begin{cases} -10, & \text{invalid} \\ r_t^{\text{merge}} + 2\,\text{score}(s_{t+1}) + 50\,\text{empty}(s_{t+1}), & \text{valid} \end{cases} \tag{23}$$

$$\tilde{r}_t^{DQ} = \begin{cases} -20, & \text{invalid} \\ r_t^{\text{merge}} + 5\,\Delta\text{empty}_t - 1, & \text{valid} \end{cases} \tag{24}$$

Notes:

- In Q-learning the term $2\,\text{score}(s_{t+1})$ uses the cumulative score (not the incremental delta). This inflates rewards later in an episode and accelerates value scale; we kept it for simplicity but it biases towards long episodes.
- The $50\,\text{empty}(s_{t+1})$ bonus was computed before spawning the new random tile, so one empty is immediately lost after shaping.
- Double Q uses a change term Δempty_t which rewards creating space via merges and slightly penalises fills; the constant -1 dampens unbounded growth.
- SARSA variants reused the Double Q pattern (delta empties plus merge reward) with minor coefficient tweaks (not shown here as those scripts are analogous).
- PPO and MuZero used the raw environment merge reward (invalid moves treated as small negatives) inside their advantage / value target calculations; we did not layer extra emptiness bonuses there.
- A terminal penalty of -200 appears only in Q-learning when bootstrapping from a terminal state (see Section II), not in Double Q where we set future to zero.

C. Training Procedure

We trained the tabular Q-learning and Double Q-learning agents offline, using many self-play episodes without any human interaction. Each training episode started from an empty board with two initial tiles and ended when no legal moves were left.

We tuned the main hyperparameters empirically. For both algorithms we used a discount factor of $\gamma \approx 0.98$. We swept over several learning rates α in the range $[0.05, 0.25]$ and trained separate agents for each value. The exploration rate ϵ started at 1.0 and decayed multiplicatively by a factor slightly smaller than one until it reached a minimum value around 0.10.

For each choice of α we trained for approximately $20{,}000$ episodes. To monitor learning we maintained a moving window over recent episode scores and periodically logged the average score over that window.

We summarised representative learning curves in Fig. 1. They show the moving average score versus training episodes for different algorithm families.

For evaluation we used a separate script without any training updates. This script loaded the learned tables, ran each agent for a fixed number of episodes and recorded summary statistics.

D. Evaluation Protocol

To compare agents fairly we evaluated all of them under the same conditions. For each agent we ran a large number of independent games and recorded the total score and maximum tile reached in each episode. From these samples we computed

- the mean score across episodes;
- the best score observed;
- the empirical distribution of the maximum tile (for example, how often the agent reached 512, 1024, 2048 and beyond);
- an approximate "win rate", defined as the fraction of games in which the agent reached at least the 2048 tile.

We used the same random seed handling across agents where possible to reduce variance. The Expectimax agent did not require any training, so for this agent the evaluation phase was simply repeated planning on fresh games.

TABLE I
SUMMARY OF EVALUATION METRICS (1000 EPISODES PER AGENT)

Agent	Avg Score	Max Score	Win Rate (% $\geq$2048)
RandomAgent	1,102.70	4,604	0.0
Q-Learning	1,897.65	7,236	0.0
Double Q-Learning	1,893.87	6,788	0.0
SARSA N-tuple (afterstate)	2,791.53	8,168	0.0
SARSA (classical)	**3,098.31**	8,876	0.0
PPO	3,079.03	**11,684**	0.0
MuZero (Lite)	2,208.42	6,776	0.0
Expectimax	**33,427.56**	**114,624**	**73.9**

V. Results and Discussion

A. Quantitative Results

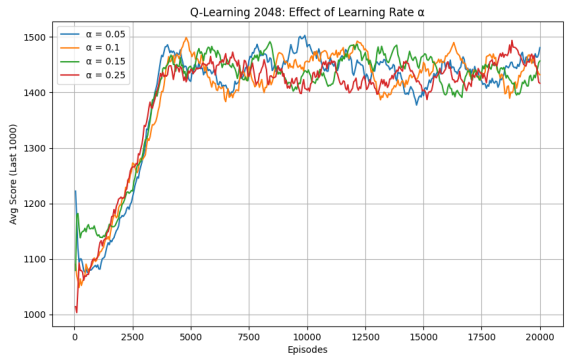
Table I summarises the main evaluation metrics we recorded for each agent over 1000 evaluation episodes (these values were produced by the 'inference.py' script and are stored in 'results.txt').

From Table I we observed that the heuristic Expectimax agent substantially outperformed all learning agents in average score, maximum score and win rate (bolded). Among learning methods that did not rely on full search, classical SARSA produced the highest average score while PPO achieved the highest max single-episode score. None of the learning agents (excluding Expectimax) reached the 2048 tile frequently enough to register a nonzero win rate in this evaluation window. MuZero Lite delivered moderate scores given its limited planning budget and small networks.

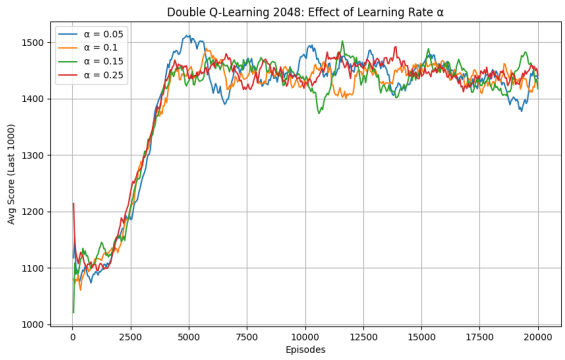
B. Maximum Tile Distribution

Beyond aggregate scores, the distribution of the highest tile reached per episode reveals qualitative differences in space management and merge planning. Table II reports the empirical percentage of episodes (out of the same 1000 evaluation runs) that ended with a given maximum tile. For agents that never reached a tile level, the entry is 0.0.

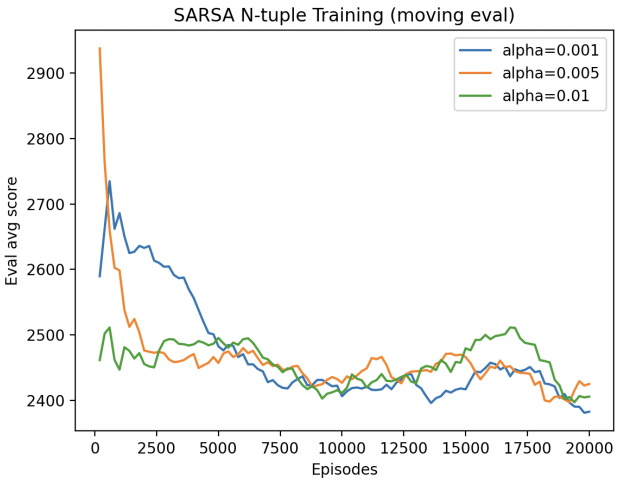
The value-based agents concentrate mass in mid-range tiles (128–512), reflecting limited long-horizon planning. PPO shifts some probability to 1024, indicating better sustained space creation. MuZero Lite resembles Double Q/SARSA due to shallow search. Expectimax's heavy tail beyond 2048 (including 4096 and occasional 8192) demonstrates its advantage in deliberate corner consolidation and monotonic shaping guided by the heuristic.



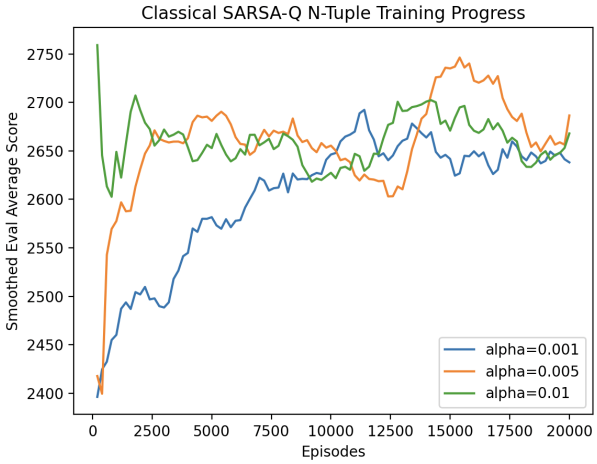
(a) Q-Learning (multi α)



(b) Double Q-Learning (multi α)



(c) SARSA afterstate n-tuple



(d) Classical SARSA n-tuple

(e) PPO

(f) MuZero Lite

Fig. 1. Learning curves for all agents: Q-Learning, Double Q-Learning, SARSA variants, PPO, and MuZero Lite.

C. Learning Curves

The training curves for Q-learning showed the expected qualitative pattern: starting from low scores dominated by exploration, the average score rose steadily as ϵ decayed and the agent started exploiting its learned value function. We observed that too small a learning rate (for example $\alpha = 0.05$) led to slow progress, while too large a value sometimes caused instability. In our runs a moderate learning rate around 0.15 gave the best trade-off between speed and stability.

For Double Q-learning the learning curves were generally

TABLE II
MAXIMUM TILE DISTRIBUTION: PERCENTAGE OF EPISODES ENDING WITH EACH HIGHEST TILE (1000 EVALUATION EPISODES PER AGENT). PERCENTAGES MAY NOT SUM TO EXACTLY 100 DUE TO ROUNDING.

oprule Agent	16	32	64	128	256	512	1024	2048	4096	8192
RandomAgent	0.30	6.70	37.60	47.50	7.80	0.10	0.00	0.00	0.00	0.00
Q-Learning	0.30	2.90	20.10	49.50	25.70	1.50	0.00	0.00	0.00	0.00
Double Q-Learning	0.00	3.60	21.40	47.00	26.40	1.60	0.00	0.00	0.00	0.00
SARSA N-tuple (afterstate)	0.00	0.20	3.60	40.80	46.20	9.20	0.00	0.00	0.00	0.00
SARSA (classical)	0.00	0.00	4.00	33.20	50.20	12.60	0.00	0.00	0.00	0.00
PPO	0.20	1.00	9.40	28.20	43.60	16.80	0.80	0.00	0.00	0.00
MuZero (Lite)	0.00	2.60	12.00	41.80	41.00	2.60	0.00	0.00	0.00	0.00
Expectimax	0.00	0.00	0.00	0.10	2.90	5.10	18.00	51.70	21.80	0.40

smoother. Even when individual runs of tabular Q-learning exhibited occasional spikes due to overestimated values, the Double Q agent tended to progress more steadily. For some values of α the final performance of Double Q-learning slightly exceeded that of vanilla Q-learning.

D. Final Performance

When we compared agents at the end of training, both Q-learning variants clearly outperformed the random baseline. The random policy rarely reached tiles above 256 and achieved relatively low scores. In contrast, the trained Q-learning agent often reached 512 and 1024 tiles and occasionally achieved 2048. The Double Q-learning agent achieved similar or slightly better average scores and had a more favourable distribution of maximum tiles for the best learning rates.

The heuristic Expectimax agent remained a strong benchmark. Because it explicitly planned ahead using the evaluation function, it often produced boards that looked more "structured", with high tiles concentrated in one corner and monotonic rows or columns. In some runs it matched or exceeded the scores of the learned agents, which reminded us that good domain knowledge can still be very competitive in small games.

E. Qualitative Behaviour

Besides the numerical metrics, we also observed how the agents actually played by watching games in the graphical interface. After training, the Q-learning and Double Q-learning agents learned several intuitive behaviours that strong human players also use. They tended to keep the largest tile in a fixed corner, avoided unnecessary moves that broke monotonic rows, and preferred moves that created space on the board.

The random agent, as expected, frequently moved tiles back and forth without any plan, quickly filling the board and ending the game. The Expectimax agent showed deliberate long chains of moves that set up merges several steps ahead. In comparison, the learned value-based agents sometimes made short-sighted moves, especially in difficult positions, but overall they showed a clear improvement over random play.

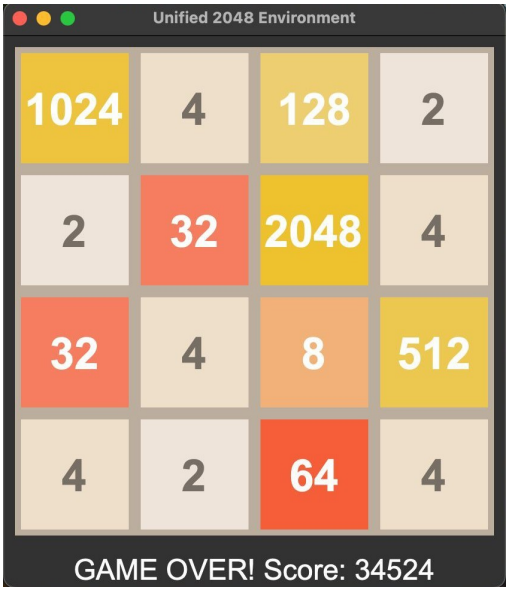


Fig. 2. Example board state reaching the 2048 tile. This snapshot comes from a successful Expectimax run.

VI. STATEMENT OF CONTRIBUTIONS

We worked on this project as a team and split the work according to our interests while still discussing design decisions together. The main contributions of each member were as follows:

- Umang implemented the tabular Q-learning agent, the Double Q-learning agent and the Expectimax agent, and ran most of the training and evaluation experiments for these methods.
- Zainab implemented two SARSA-based agents: a value function over afterstates and an n-tuple SARSA-Q agent that followed the classical SARSA update, and contributed to discussions on on-policy learning.
- Tejas implemented PPO and a lightweight MuZero variant; also attempted DQN and a basic actor-critic which were curtailed due to compute limits.
- Shardul researched on different games upon which RL can be applied, did the initial 2048 environment and game setup, and compiled results and wrote the report.

We coordinated our work through the shared Git repository and met regularly to look at intermediate results and decide which experiments to run next.

VII. Conclusion

In this project we implemented a common 2048 environment and several agents based on reinforcement learning and heuristic search. By training and evaluating these agents under the same conditions, we were able to compare how different algorithms learned and what kinds of strategies they discovered.

Our experiments confirmed that even relatively simple value-based methods such as tabular Q-learning can learn useful behaviours in 2048 when combined with sensible reward shaping. Double Q-learning reduced overestimation and gave slightly more stable learning curves. A well-designed Expectimax agent, although not a learning method, remained a strong baseline that sometimes matched or outperformed the learned policies.

If we had more time, we would like to explore deeper neural network based agents, more systematic hyperparameter searches and alternative state representations (for example, afterstate based value functions or symmetry-aware encodings). We are also interested in extending the environment to larger boards or variants of 2048 to further stress test the algorithms.

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. MIT Press, 2018.

[2] R. Bellman, "A Markovian decision process," *Journal of Mathematics and Mechanics*, vol. 6, no. 5, pp. 679-684, 1957.

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

[4] H. van Hasselt, "Double Q-learning," in *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 2613-2621, 2010.

[5] G. A. Rummery and M. Niranjan, "On-line Q-learning using connectionist systems," Univ. Cambridge, Tech. Rep. CUED/F-INFENG/TR 166, 1994.

[6] G. Szubert and W. Jaśkowski, "Temporal difference learning of N-tuple networks for the game 2048," in *IEEE Conference on Computational Intelligence and Games (CIG)*, 2014.

[7] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Pearson, 2010.

[8] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," arXiv:1707.06347, 2017.

[9] J. Schrittwieser *et al.*, "Mastering Atari, Go, Chess and Shogi by planning with a learned model," *Nature*, vol. 588, pp. 604-609, Dec. 2020.

[10] G. Cirulli, "2048," available at: https://play2048.co/, accessed Nov. 2025.

[11] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine Learning*, vol. 3, pp. 9-44, 1988.