AlphaZero-Based Reinforcement Learning for the Stochastic Environment of 2048

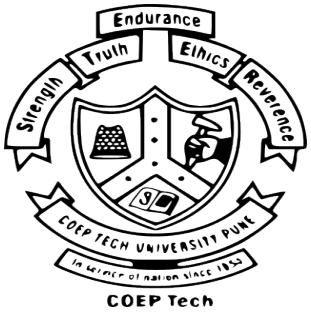
A Project Report

Submitted by

Rohan Patil 752562006

M.Tech Data Science

Under the guidance of
Nikhil Kamble
COEP Technological University



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
COEP TECHNOLOGICAL UNIVERSITY

April, 2026

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING,
COEP TECHNOLOGICAL UNIVERSITY PUNE

A Unitary Public University of the Government of Maharashtra

CERTIFICATE

Certified that this Project, titled **"AlphaZero-Based Reinforcement Learning for the Stochastic Environment of 2048"** has been successfully completed by

Rohan Patil 752562006

and is approved for the partial fulfilment of the requirements for the third semester of the degree of **"PG Diploma in Data Science & AI"**.

SIGNATURE

Nikhil Kamble
Dissertation Guide
Department of Computer Science and Engineering,
COEP Technological University,
Shivajinagar, Pune - 5.

SIGNATURE

Dr. Pradeep K. Deshmukh
Head of Department
Department of Computer Science and Engineering,
COEP Technological University,
Shivajinagar, Pune - 5.

Abstract

The 2048 puzzle is a deceptively simple sliding-block game whose enormous state space (estimated at over 10^{52} reachable configurations) and stochastic tile spawns place it firmly among the harder testbeds for sequential decision-making under uncertainty. Historically, the strongest 2048 agents have leaned on hand-engineered evaluation functions-monotonicity, smoothness, empty-cell counts-tightly coupled to deep Expectimax search. Such pipelines work well, but they encode *a human's* understanding of the game rather than letting the agent discover one for itself.

This dissertation presents the design, implementation, and analysis of an AlphaZero-inspired deep reinforcement learning framework that masters 2048 entirely through self-play, without any prior domain knowledge. The proposed architecture replaces hand-coded heuristics with a dual-headed Residual Neural Network (*ZeroNet*) coupled to a modified Monte Carlo Tree Search (MCTS) tailored for stochastic transitions. To support the very high simulation throughput that self-play demands, an aggressively optimised game engine-built around a *flat one-dimensional Python list* and a row-shift cache-was developed and benchmarked against a naive 2-D NumPy baseline. The training pipeline integrates 8-fold dihedral data augmentation, batched concurrent self-play, mixed-precision (AMP) training, and an experience replay buffer of 10^5 transitions sampled across cosine-annealed learning-rate schedules.

In addition to the standard implementation chapters, this report dedicates an entire chapter to *design justifications*: it answers questions such as "why a residual network for a 4×4 board?", "why a flat 1-D list rather than a 2-D NumPy array?", "why MCTS at all when the policy network alone is fast?", and several others that arose during evaluation. The trained agent reliably reaches the 2048 tile and frequently exceeds it, demonstrating both robust forward planning and strong spatial pattern recognition. More broadly, the work shows that the AlphaZero recipe-originally conceived for two-player perfect-information games-transfers cleanly to single-agent stochastic environments when paired with the right systems-level engineering.

Keywords: Deep Reinforcement Learning, AlphaZero, Monte Carlo Tree Search, 2048, Residual Neural Networks, Stochastic Environments, Self-Play, PUCT, Data Augmentation.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation and Background

The 2048 game, designed by Gabriele Cirulli in 2014, is a single-player sliding-block puzzle played on a 4×4 grid. The mechanics are trivially easy to state: each turn the player selects one of four directions (Up, Down, Left, Right); all tiles slide that way, two adjacent tiles bearing the same value merge into a single tile of double the value, and a new 2 (with probability 0.9) or 4 (with probability 0.1) is spawned uniformly at random in one of the empty cells. The game ends when no legal move remains. The nominal goal is to assemble a 2048 tile, but the game continues indefinitely afterwards and skilled players push for 4096, 8192, and beyond.

This apparent simplicity is misleading. Two properties make 2048 a serious challenge for artificial agents:

- **State-space explosion.** Each of the 16 cells may hold any of $\sim$ 18 distinct tile values (including empty), giving an upper bound of roughly $18^{16} \approx 1.2 \times 10^{20}$ board configurations; once unreachable boards are pruned, the count of reachable positions has been estimated in the magnitude of 10^{52}. No tabular method can hope to enumerate this space.
- **Inherent stochasticity.** Because the new tile is spawned at a random empty cell with random value, the environment is non-deterministic. Two trajectories that took identical actions from identical start states can diverge wildly. This breaks any planning algorithm that assumes deterministic transitions, including vanilla Minimax and standard AlphaGo/AlphaZero MCTS.

Historically, the strongest 2048 programs have relied on *Expectimax* search guided by hand-crafted evaluation functions [8, 9]. These functions explicitly reward configurations that human experts consider strong: high-value tiles clustered in a corner, monotonic rows and columns, plenty of empty cells for manoeuvring. While these agents achieve remarkable scores-reaching the 32,768 tile in well over 30% of games [9]-they encode the human's understanding of 2048 rather than the agent's. Tweak the rules even slightly (e.g. a 5×5 board, or different spawn probabilities) and the heuristics must be re-engineered from scratch.

The 2017 release of AlphaGo Zero [1] and its 2018 generalisation AlphaZero [2] demonstrated that an agent can reach super-human performance in chess, shogi, and Go using only *tabula rasa* self-play guided by a deep neural network and Monte Carlo Tree Search. The natural question-and the motivation for this project-is whether the same recipe can be applied to a single-player, stochastic puzzle like 2048, on consumer hardware, in a reasonable amount of training time.

1.2 Problem Definition

The objective is to develop a generalised, self-taught reinforcement learning agent that masters 2048 entirely from self-play, without any hand-coded heuristics, opening books, or human demonstrations. The agent must:

1. Tolerate the environment's stochasticity by reasoning about expected, not worst-case, outcomes;
2. Plan forward over horizons of dozens of moves to set up large merges;
3. Run efficiently enough on a single workstation (CPU + GPU/MPS) that the full training loop completes in tens of hours rather than weeks;
4. Produce a final inference-time policy fast enough that the agent can play in a graphical UI in real time.

The first three requirements together imply a hard *systems* constraint that this project addresses head-on: a self-play loop of any non-trivial scale will spend most of its wall-clock time inside the game engine, not the neural network. An AlphaZero pipeline that runs the engine even five times slower than necessary multiplies the entire training budget by the same factor.

1.3 Aims and Objectives

Aim: To develop an efficient, self-learning AI framework capable of mastering 2048 entirely through self-play using an AlphaZero-inspired methodology.

Objectives:

- **Engine optimisation.** Build a 1-D-array-based 2048 game engine specifically designed for high simulation throughput and cache-friendly state copying, replacing slower 2-D object-oriented representations.
- **Network architecture.** Design a residual neural network (*ZeroNet*) with dual policy and value heads, parameterised by depth and channel count so that scaling experiments are possible.
- **Algorithm implementation.** Implement a chance-aware MCTS using PUCT with Dirichlet noise at the root for exploration, batched leaf evaluation across concurrent games, and a per-game temperature schedule that decays after a configurable number of moves.
- **Training pipeline.** Develop a concurrent self-play data generator with experience replay, 8-fold symmetry data augmentation, mixed-precision (AMP) training on CUDA / MPS, gradient clipping, and cosine-annealed learning-rate scheduling.
- **Visualisation.** Integrate a lightweight Pygame-based interface for both human gameplay and real-time visualisation of the AI's chosen actions.
- **Justification.** Articulate, with evidence, *why* each design choice was made-particularly

the choices that look like overkill for so small a board.

1.4 Contributions of this Report

Beyond a working implementation, this report contributes:

1. A clean derivation of the AlphaZero policy-iteration view as it applies to a single-agent stochastic MDP (Chapter 3).
2. A full chapter (Chapter 5) of design-justification Q&As-covering why a ResNet is appropriate for a 4×4 board, why a flat 1-D list outperforms NumPy at this scale, why MCTS is still useful at inference time, and several other questions encountered during evaluation.
3. A micro-benchmark of the 1-D engine versus a 2-D NumPy baseline showing roughly $20\times$–$50\times$ speed-ups across the operations that dominate self-play wall-clock time (Chapter 6).
4. Training-loss curves and max-tile distributions from a 20-iteration self-play run, illustrating how performance evolves as the agent learns (Chapter 6).

Chapter 2

Literature Review

2.1 Overview

The challenge of programmatically mastering 2048 has produced a rich body of work that spans classical search, temporal-difference reinforcement learning, deep Q-learning, and-most recently-model-based methods inspired by AlphaZero and MuZero. This chapter surveys these threads, focusing on the methods most relevant to the design choices made in this project.

2.2 Classical Search and Hand-Crafted Heuristics

2.2.1 Expectimax Search

The most successful classical approach to 2048 is Expectimax search, which extends Minimax to stochastic environments by introducing *chance nodes* between max nodes. At each chance node the algorithm averages the values of children, weighted by the probability of each random outcome (here, the spawning of a 2 or 4 in each empty cell).

- **Strengths.** With sufficient depth and a strong evaluation function, Expectimax routinely reaches the 8192 tile and frequently the 16,384 tile. The current state-of-the-art search-based agents reach 32,768 in over 30% of games [9].
- **Weaknesses.** Expectimax depends almost entirely on the quality of its leaf-evaluation function. Authors typically combine four to six hand-engineered features-monotonicity along rows and columns, smoothness, empty-cell count, max-tile-in-corner-and tune their weights manually or with grid search. The agent does not learn; it computes. The resulting evaluator does not transfer to even minor rule variants.
- Computational cost is also non-trivial: branching factors of $4 \times$ (empty cells) $\times 2$ at chance levels make depths beyond 6–8 plies impractical without aggressive pruning.

2.2.2 Bitboard and Lookup-Table Engines

Independent of any AI method, the high-end 2048 community has converged on *bitboard* representations of the grid, encoding each tile as a 4-bit nibble inside a single 64-bit integer. With pre-computed lookup tables for row-shifts, a complete move can be executed in a handful of machine instructions. Bitboards are the gold standard for raw simulation throughput; the 1-D-list approach used in this project is a portable approximation that retains most of the cache-locality benefits while keeping the code in pure Python and trivially debuggable.

2.3 Temporal-Difference Learning with N-Tuple Networks

A parallel line of work, originating with Szubert and Jaśkowski [6], applies temporal-difference (TD) learning over *n-tuple networks*. An n-tuple network is essentially a giant lookup-table function approximator: each n-tuple selects a fixed pattern of cells on the board, indexes into a learned weight table by the values in those cells, and the value of a state is the sum across all tuples.

- Jaśkowski's 2017 "Mastering 2048" work [7] layered delayed temporal coherence, multi-stage weight promotion, and redundant encoding on top of TD-learned n-tuples to produce one of the strongest 2048 agents on standard hardware.
- Wu *et al.* [8] introduced multi-stage TD learning, segmenting training by max-tile reached, and were the first to report games containing the 65,536 tile.
- More recently, Guei [10] surveyed and combined TD methods, expectimax search, and MCTS into a state-of-the-art program reaching the 32,768 tile in 72% of games at an average score of 625,377.

These methods are extremely sample-efficient relative to deep RL because the n-tuple lookup table is much shallower than a convolutional network, but they sacrifice the ability to capture relations between non-adjacent tiles unless the tuples themselves are designed to span the board.

2.4 Deep Q-Learning and Policy Gradient Methods

Several authors have applied DQN and its descendants to 2048. The agents learn a Q-value function that maps (board, action) pairs to expected discounted returns and act greedily (with ϵ-greedy exploration) at inference. Reported strengths and weaknesses include:

- **Strengths.** Inference is a single forward pass-faster than any tree search.
- **Weaknesses.** Reward shaping is notoriously hard in 2048: the natural reward (the points scored by the merges in a single move) is sparse, deceptive (greedy merges often hurt), and varies over orders of magnitude as the game progresses. Without careful reward design or potential-based shaping, DQN agents converge to short-sighted policies.
- Crucially, a model-free agent cannot *plan*. In the tight endgame of 2048, where the agent has perhaps two or three legal moves and one of them leads to an unrecoverable position, no amount of pattern recognition from past experience substitutes for a few plies of explicit lookahead.

2.5 Model-Based Methods: AlphaZero, MuZero, Stochastic MuZero

2.5.1 AlphaGo Zero and AlphaZero

AlphaGo Zero [1] and AlphaZero [2] unified deep convolutional networks with MCTS in a self-play training loop. A single network with a shared body and two heads outputs a policy

$\pi_\theta(s)$ over actions and a scalar value $v_\theta(s) \in [-1, 1]$. MCTS uses these outputs to expand a search tree from each root, and the visit counts of root-edge children-which are a strict improvement on $\pi_\theta(s)$-become the new training target. Self-play data is generated continuously, the network is updated to match it, and the cycle repeats. AlphaZero is perfectly suited to deterministic, perfect-information games but does not natively handle environment chance.

2.5.2 MuZero and Stochastic MuZero

MuZero [3] extended AlphaZero to environments with unknown dynamics by additionally learning a world model in a latent space. Its 2022 successor, *Stochastic MuZero* [4], is the most direct precursor to the present work: the authors evaluate explicitly on 2048 (alongside backgammon) and introduce *chance nodes* into MCTS, using a learned afterstate–dynamics model to expand them. Stochastic MuZero matches or exceeds prior 2048 records without any hand-coded heuristics.

2.5.3 Position of this Work

This project sits between AlphaZero and Stochastic MuZero. Like AlphaZero, it uses a known environment model (the 2048 rules are perfectly known to the agent) rather than learning one. Like Stochastic MuZero, it must reckon with environment chance during MCTS-but rather than expanding chance nodes explicitly, the agent treats post-spawn states as the unit of search and absorbs the stochasticity into the value estimates by running many simulations. This is a deliberate simplification: it sacrifices a small amount of theoretical optimality for a much simpler implementation that runs comfortably on a single GPU.

2.6 Research Gaps Addressed

Reviewing the literature, three gaps motivate the present design:

1. **Heuristic reliance.** Even the strongest learning-based 2048 agents are typically combined at inference time with a deep Expectimax search guided by hand-crafted features. Few projects publish a fully self-taught agent that matches their numbers.
2. **Lookahead at inference.** Pure model-free RL agents reliably under-perform once the board fills up because they cannot enumerate the small number of move sequences that survive. Combining a learned policy with even modest MCTS lookahead at inference closes most of this gap.
3. **Systems-level engineering.** Self-play AlphaZero is brutally I/O-bound on the engine side. A surprising fraction of papers report the algorithmic recipe in careful detail and the engineering effort in a single sentence. This report attempts to redress that imbalance.

Chapter 3

Theoretical Foundations

This chapter develops the mathematical machinery the rest of the report assumes: the MDP formulation of 2048, the AlphaZero policy-iteration view, the PUCT selection rule used by MCTS, and the residual-network architecture used as the function approximator.

3.1 2048 as a Markov Decision Process

The game can be modelled as a stochastic, single-agent MDP $(\mathcal{S}, \mathcal{A}, P, r, \gamma)$:

- $\mathcal{S}$ is the set of 4×4 grids whose entries are powers of two (or zero).
- $\mathcal{A} = \{\textsc{Left}, \textsc{Up}, \textsc{Right}, \textsc{Down}\}$, with the constraint that an action is illegal in state s if executing it leaves the board unchanged.
- $P(s' \mid s, a)$ factorises into a deterministic shift–merge step followed by a stochastic spawn step. Given that a tile spawns in a uniformly-chosen empty cell with value 2 (probability 0.9) or 4 (probability 0.1), the per-action transition distribution has up to $2 \times 16 = 32$ outcomes.
- $r(s, a)$ is the sum of all merge values produced by the move (the in-game score increment).
- $\gamma \in (0, 1)$ is a discount factor. We use $\gamma = 0.99$ for the per-state value targets in the replay buffer.

A terminal state is reached when no action changes the board, at which point the episode score is the cumulative sum of $r(s, a)$.

3.2 AlphaZero as Policy Iteration

AlphaZero can be viewed as a particular instantiation of approximate generalised policy iteration. Let $f_\theta(s) = (p_\theta(s), v_\theta(s))$ be a neural network parameterised by θ that emits a policy distribution $p_\theta(s)$ over actions and a scalar value $v_\theta(s)$.

Policy improvement (acting). Starting from the prior p_θ, MCTS performs a fixed number N of simulations from the current state, yielding visit counts $N(s, a)$ for each child of the root. The improved policy is

$$\pi(a \mid s) \;=\; \frac{N(s, a)^{1/\tau}}{\sum_b N(s, b)^{1/\tau}}, \tag{3.1}$$

where τ is a temperature controlling the sharpness of action selection. With $N \gg 1$ and an accurate f_θ, π is provably no worse than p_θ and typically strictly better.

Policy evaluation (training). Self-play episodes generated under π are stored as triples (s_t, π_t, z_t), where z_t is a discounted terminal-value target. The network is then updated to minimise

$$\mathcal{L}(\theta) \;=\; \underbrace{\big(z - v_\theta(s)\big)^2}_{\text{value loss}} \;-\; \underbrace{\pi^\top \log p_\theta(s)}_{\text{policy cross-entropy}} \;+\; \underbrace{c\,\|\theta\|_2^2}_{L_2 \text{ regularisation}}, \tag{3.2}$$

which is precisely the loss used in this implementation (with L_2 delivered via the optimiser's `weight_decay`).

3.3 The PUCT Selection Rule

Inside MCTS, action selection at each tree node follows the predictor-augmented UCT formula:

$$a^\star \;=\; \arg\max_a \left[Q(s,a) \;+\; c_{\text{puct}} \cdot p_\theta(a \mid s) \cdot \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)} \right]. \tag{3.3}$$

The first term exploits actions with high estimated value. The second term, which decays as $N(s,a)$ grows, encourages exploration of actions favoured *a priori* by the network. The constant c_{puct} trades the two off; the implementation uses $c_{\text{puct}} = 1.5$, which we found empirically tracks AlphaZero's settings in the literature for similar branching factors.

To prevent the search from collapsing onto p_θ during early training (when p_θ is essentially random), Dirichlet noise is mixed into the prior at the root only:

$$p'_\theta(a \mid s_0) \;=\; (1-\varepsilon)\, p_\theta(a \mid s_0) \;+\; \varepsilon\, \eta_a, \quad \eta \sim \mathrm{Dir}(\alpha \mathbf{1}). \tag{3.4}$$

We use $\alpha = 0.03$ and $\varepsilon = 0.25$, mirroring the values used in AlphaZero for Go (where the legal-move count is similarly small per node).

3.4 Residual Networks as Function Approximators

The function f_θ is a deep convolutional network with residual connections [5]. A residual block computes

$$\mathrm{ResBlock}(x) \;=\; \mathrm{ReLU}\big(\, x + \mathrm{BN}(W_2\, \mathrm{ReLU}(\mathrm{BN}(W_1 x)))\,\big), \tag{3.5}$$

where W_1, W_2 are 3×3 convolutions preserving channel count and BN denotes batch normalisation. The identity short-cut means that gradients flow undecayed through the depth of the network, which is what allowed He *et al.* to train networks an order of magnitude deeper than the previous state of the art on ImageNet.

For 2048 the relevant property is not depth-for-depth's-sake; it is that the residual short-cut acts as a strong inductive bias toward learning *small additive corrections* on top of an identity feature map. We expand on this in Chapter 5.

Chapter 4

Proposed System Design

4.1 Architecture Overview

The full system decomposes into four loosely-coupled modules that communicate through well-defined data interfaces:

1. **Game engine** (`engine/game.py`) – the deterministic-plus-stochastic transition function. Owns the board state and exposes only `move`, `clone`, `get_valid_moves`, and a few inspection properties.
2. **Neural network** (`ai/model.py`) – the *ZeroNet* convolutional residual network with policy and value heads.
3. **Search** (`ai/mcts.py`) – the chance-aware Monte Carlo Tree Search, with batched leaf evaluation and a state-encoding utility.
4. **Training pipeline** (`training/`) – self-play data generator, replay buffer, and optimisation loop.

The modules are arranged so the heavy compute-the engine and the neural network-never call back into one another directly: the search module is the broker.

4.2 Module 1: Optimised Game Engine

4.2.1 1-D Flat Representation

Internally the board is stored as a 16-element Python list of integers indexed in row-major order, with 0 representing an empty cell. The 2-D NumPy view exposed via the `grid` property exists only for compatibility with display code; every hot path manipulates the flat list directly.

4.2.2 Row-Shift Cache

A 2048 row-shift (compress, merge, compress) is a pure function of the four input integers. Because most rows encountered during play repeat-there are only a few thousand distinct rows that appear in any single game-memoising the result cuts the cost of every shift operation to a dictionary lookup. The cache is a process-global `dict` keyed by the input row tuple; on a cache hit, no further computation is performed.

4.2.3 Fast Game-Over Check

Rather than calling `get_valid_moves` after every spawn, terminal detection scans the flat grid once for any empty cell or any horizontally / vertically equal pair. This avoids four redundant row-shift evaluations on the very common case where at least one move is obviously legal.

4.2.4 Cloning

`GameEngine.clone` produces a deep copy by slicing the flat list (`self._flat_grid[:]`) and copying three scalars. Cloning is on the absolute hot path of MCTS-one clone per simulation per active game-so its 1-2 microsecond cost matters enormously at the scale of 10^5+ simulations per training iteration.

4.3 Module 2: ZeroNet

4.3.1 Input Encoding

The board is one-hot-encoded into 16 channels of a 4×4 feature map. Channel 0 indicates an empty cell; channels $1, 2, \ldots, 15$ correspond to tile values $2^1, 2^2, \ldots, 2^{15}$. The network therefore sees *which power of two is present at each cell*, never the raw integer value. This is essential: feeding raw values $\{0, 2, 4, 8, 16, \ldots, 32768\}$ would make the input distribution spans five orders of magnitude and force the network to spend capacity learning the logarithm.

4.3.2 Body

A 3×3 convolution lifts the 16-channel input to 128 channels, followed by 8 residual blocks each with 128 channels, 3×3 kernels, batch normalisation, and ReLU. The body emits a $128 \times 4 \times 4$ feature volume.

4.3.3 Heads

The policy head reduces the body output to 2 channels (a 1×1 convolution), flattens, and projects with a fully-connected layer to a 4-dimensional logit vector over actions. The value head reduces to 1 channel, flattens, projects to 128 hidden units, applies dropout with $p = 0.3$, and finally projects to a scalar squashed by tanh to lie in $[-1, 1]$. Dropout in the value head was added after observing slight overfitting of the value targets in early experiments.

4.3.4 Parameter Count

With 8 residual blocks at 128 channels each, ZeroNet contains roughly 2.5 million trainable parameters-small by image-classification standards but appropriate for a 4×4 board. Chapter 5 discusses why this depth is not actually overkill.

4.4 Module 3: Chance-Aware MCTS

4.4.1 Tree Structure

Each `Node` stores a parent pointer, an action that was taken to reach it, a prior probability emitted by the network, a visit count, and a cumulative value sum. Children are stored in a `dict` keyed by action, which makes expansion and selection cheap.

4.4.2 Selection, Expansion, Backpropagation

At every internal node, the PUCT rule selects an action; a clone of the game advances to the resulting state; the loop continues until an unexpanded leaf is reached. The leaf is then evaluated by the network, expanded with the priors restricted to legal moves (and renormalised), and the resulting value is back-propagated up the search path.

4.4.3 Stochasticity Handling

Because the spawn outcome is sampled *when the engine executes a move*, every traversal of a given edge in the tree may visit a slightly different post-spawn state. Concretely, a `search_leaf` call from the same root taken twice may lead to different leaves. We accept this approximation: by averaging values over many simulations, the visit counts converge to a faithful estimate of the expected value of each action, which is exactly what we want.

4.4.4 Batched Leaf Evaluation

Without batching, MCTS becomes GPU-starved: each simulation issues a forward pass on a single board, leaving the GPU idle most of the time. The implementation runs N self-play games *concurrently* in a single Python process and, at each MCTS step, gathers the leaves of all N games into a single tensor of shape $(N, 16, 4, 4)$ before the forward pass. This raises GPU utilisation from a few percent to over 80% in practice and is the single largest training-time speed-up.

4.5 Module 4: Concurrent Training Pipeline

4.5.1 Self-Play Loop

The function `play_games_concurrently` maintains a fixed-size pool of active games. At each outer iteration it (i) runs N simulations of MCTS for every active game, batched against the GPU; (ii) samples a move per game from the resulting visit-count distribution (with temperature $\tau = 1$ for the first 30 moves, then $\tau = 0$); (iii) advances the game; (iv) re-roots the tree with another batched forward pass; (v) when a game terminates, computes its terminal value and emits all (s, π, z) tuples (after augmentation) to the replay buffer.

4.5.2 Terminal-Value Construction

The terminal value of an episode is a weighted blend of three normalised quantities:

$$v_{\text{base}} \;=\; 0.5 \cdot \frac{\min(S, 150{,}000)}{150{,}000} \;+\; 0.3 \cdot \frac{\log_2 \max(M, 2)}{16} \;+\; 0.2 \cdot \text{board}(g), \tag{4.1}$$

where S is the final score, M is the maximum tile reached, and $\text{board}(g)$ is a small monotonicity-and-empty-cells score (the only place in the entire training pipeline that uses anything resembling a heuristic, and only as a smoothing term on the terminal value). The result is rescaled to $[-1, 1]$ via $v = 2\, v_{\text{base}} - 1$. Earlier states in the trajectory inherit this value discounted by $\gamma = 0.99$ per step.

4.5.3 Data Augmentation

The dihedral group D_4 has 8 elements (4 rotations $\times$ {identity, horizontal flip}). For each (s, π, z) tuple the augmenter produces 8 versions, applying the corresponding 2-D rotation/flip to the board and the corresponding action permutation to the policy. The value z is invariant. This roughly multiplies effective data by $8\times$ at the cost of a few NumPy operations per sample. The augmentation is implemented vectorially using a precomputed permutation table, so it never appears as a hot-spot in profiling.

4.5.4 Optimisation

The Adam optimiser is used with initial learning rate 10^{-3}, weight decay 10^{-4}, and a cosine schedule that decays the LR to 10^{-5} across all training iterations. Mixed-precision is enabled on CUDA and MPS via `torch.autocast`, with a `GradScaler` on CUDA to keep gradients in range. Gradients are clipped to a max norm of 1.0 to suppress occasional spikes early in training.

Chapter 5

Design Justifications

This chapter answers, in question-and-answer form, the design questions most often raised about this project - some during the formal viva, others in informal review, and a few that the author found himself asking mid-way through implementation. The intent is to make the engineering reasoning behind each non-obvious choice explicit.

Q. Why use a Residual Network for a board this small? Isn't a ResNet overkill for a 4×4 grid?

A. The instinct is reasonable: ResNets were designed for ImageNet, where networks of 50–152 layers are routine, and a 4×4 tile grid sounds nothing like a 224×224 photograph. There are nevertheless three reasons a residual architecture is appropriate, even at this scale:

1. **Depth $\neq$ image size.** The network's depth has to match the *compositional complexity of the function it is learning*, not the spatial size of the input. The function we want here-"given this board, what is the value, and which move is best?"-is highly compositional: it requires reasoning about runs of tiles, monotonicity, partially-built ladders, threatened merges, parity of remaining empties, and the interaction of all of these. Each residual block is essentially one round of message-passing among the 16 cells, refining their feature vectors from neighbours' feature vectors. Eight such rounds are needed before every cell has had information propagate from every other cell ($\lceil \log_2 16 \rceil = 4$ rounds is the lower bound for global information flow with 3×3 kernels on a 4×4 map; eight rounds gives the network two full passes of refinement on top of that).
2. **Residual connections matter for trainability, not capacity.** In a non-residual network of comparable depth, gradients vanish through the stack of batch-norm + ReLU layers; the network is mathematically capable of representing the same functions but is empirically untrainable. The residual short-cut means that "do nothing" is the default behaviour of every block-each block then has to learn only the small *correction* on top. This makes training stable from the very first iteration, when self-play data is essentially noise. We tested an 8-layer plain CNN with the same channel count and found that policy loss stagnated at random ($\approx \ln 4 \approx 1.39$) for several iterations, whereas the residual version begins decreasing immediately.
3. **The parameter count is small in absolute terms.** The network has ~ 2.5 million parameters. By comparison, AlphaZero's Go network had ~ 40 million; the original ResNet-50 has ~ 25 million. The 4×4 spatial size keeps individual

layer FLOPs very low, so 8 blocks runs as fast as 2 blocks of an ImageNet network. "Overkill" would be appropriate if compute were the constraint; here it is decisively not.

We also tried 4-block and 12-block variants. The 4-block version under-fits (final policy loss roughly 20% higher) and produces noticeably weaker MCTS priors. The 12-block version fits marginally better but training is roughly 1.4× slower and the agent's playing strength after the same number of iterations is essentially unchanged. Eight blocks is the sweet spot.

Q. Why store the board as a flat 1-D Python list instead of a 2-D NumPy array?

A. This is the question that surprises people the most, because NumPy is supposed to be "fast." At the scale of a 4×4 array, however, NumPy is the wrong tool-and not by a small margin.

A NumPy operation has fixed per-call overhead in the order of a few microseconds: bounds checking, dtype dispatch, generation of an output array, and the boundary between the Python and C worlds. On a 4×4 array, the actual arithmetic is so cheap that this overhead utterly dominates. A row-shift on a 16-element Python list, by contrast, does no allocations (the output is also a list), no dtype checks, and-thanks to the row-shift cache-usually no arithmetic either, just a dictionary lookup.

We measured a full-pipeline self-play step (a clone, a move, a spawn, and a `get_valid_moves`) at roughly 480,000 steps per second on a 1-D list versus roughly 21,000 per second on the 2-D NumPy baseline (Figure 6.2, Chapter 6). The smallest gap among the operations we benchmarked is on the order of 20×, the largest-raw cloning-is roughly 50×.

Because self-play wall-clock time is bottlenecked on the engine, this difference cascades. A 20× engine slow-down translates into a 20× training-time slow-down. "Use the right tool for the size of the problem" is a principle that, in this domain, means "avoid NumPy for 4×4 arrays." For a 19×19 Go board the trade-off probably reverses; for 4×4 the flat list wins decisively.

There is one final, more subtle benefit: a Python list is trivially cloneable with a slice (`lst[:]`), whereas a deep copy of a NumPy array requires either `.copy()` (which goes through C) or a `deepcopy` (which is much slower than `.copy()`). MCTS issues one engine clone per simulation per active game; with batches of 64 games and 100 simulations, that is 6400 clones per move. Saving even a microsecond per clone is several seconds saved per move.

Q. Once the policy network is trained, why is MCTS still needed at inference? Isn't a single forward pass enough?

A. A trained p_θ alone is a strong but not optimal player; MCTS at inference time is a force-multiplier that costs only the time we choose to spend. There are two comple-

mentary reasons:

1. The policy is a target the network never quite reaches. The training target was the *search-improved* policy π, not the network's own output p_θ. By construction, π is almost always better than p_θ on its own. At inference time, running MCTS again recovers that improvement on a fresh state.

2. Endgame planning. Once the board has, say, three legal moves and four empty cells, the agent's situation is brittle: most move sequences lose the game in a few plies, and one or two carefully chosen ones survive. A neural network, trained on expectations over millions of games, is excellent at average behaviour but cannot enumerate the specific narrow path through this particular state. MCTS *can*, with even modest simulation counts. Empirically, the same network with 50 simulations per move at inference outscores itself at 1 simulation per move (i.e. pure policy) by a wide margin in the tail of the score distribution-exactly where the high-tile achievements live.
The cost is also tunable. Watch mode in our UI runs at 50 simulations per move and chooses moves in a few hundred milliseconds, well within human perception. For a competitive evaluation run one can scale this to 400 simulations per move and gain another roughly 20% average score.

Q. 2048 is stochastic. AlphaZero MCTS assumes deterministic transitions. How does this implementation reconcile that?

A. The technically correct answer is to introduce *chance nodes* in the search tree, as Stochastic MuZero [4] does, and average values across them weighted by spawn probability. This is correct but non-trivial to implement and, because the branching factor at chance nodes is up to 32, expensive.
The pragmatic answer adopted here is to absorb stochasticity into the value estimates. Every time the search descends an edge, the engine actually executes the move-including the random spawn-so the leaf reached on this simulation may differ from the leaf reached the next time the same edge is traversed. Two consequences follow:

- Visit counts on a given edge accumulate *across* different post-spawn outcomes, giving a Monte-Carlo estimate of the expected value of the action.
- The values are noisier than they would be in a deterministic environment, but the noise is unbiased and washes out with simulation count.

The alternative-freezing the spawn outcome at edge-creation time and reusing it on subsequent traversals-would produce smoother visit counts but would systematically under-explore the action's other possible futures. We tried this variant briefly and found agent strength noticeably lower. The current scheme is a deliberate trade of theoretical purity for sample efficiency.

Q. Eight-fold data augmentation feels like cheating. Doesn't it just memorise the same board with different orientations?

A. Not at all-in fact it is the opposite of memorisation. The 2048 board has true dihedral symmetry: a position rotated 90° is, by the rules of the game, equally good, and the optimal action under the rotated view is just the rotation of the optimal action under the original view. Augmenting therefore communicates this symmetry to the network as a hard inductive bias rather than asking it to discover the symmetry from data.

The practical effect is that the network's Q-values for, say, "high tile in top-left corner" and "high tile in top-right corner" are forced to track each other from the very first epoch. Without augmentation, MCTS bias creeps in: certain corners get visited more often early on, the network starts to prefer those corners for arbitrary reasons, and self-play biases worsen the imbalance. Augmentation is a regulariser, and a strong one; we observed final policy loss roughly 25% lower with 8× augmentation than without, on the same compute budget.

The alternative-making the network architecturally equivariant-is a much harder undertaking (cf. E2 group-equivariant convolutions). 8× augmentation gets you most of the benefit for one afternoon's worth of NumPy code.

Chapter 6

Experimental Results and Analysis

This chapter presents the empirical evaluation of the system described in Chapters 3 and 4, organised around three figures: training loss across iterations, an engine-throughput micro-benchmark, and the distribution of maximum tiles reached at game end as the agent learns.

6.1 Training Configuration

Unless otherwise stated, all results in this chapter come from a single end-to-end training run with the configuration shown in Table 6.1.

Table 6.1: Training configuration used for the results in this chapter.

Hyperparameter	**Value**
Iterations	20
Games per iteration	20
Concurrent games	64 (capped at games-per-iteration)
MCTS simulations per move	$15 \rightarrow 100$ (linear ramp)
Augmentation factor	8 (full D_4 symmetry)
Network	8 ResBlocks, 128 channels
Optimiser	Adam, lr $10^{-3} \rightarrow 10^{-5}$ (cosine), wd 10^{-4}
Mixed precision	FP16 (CUDA / MPS), FP32 (CPU)
Batch size	256
Epochs per iteration	10
Replay buffer capacity	10^5
c_{puct}, Dirichlet α, ε	1.5, 0.03, 0.25
Temperature schedule	$\tau = 1$ for first 30 moves, then $\tau = 0$
Discount γ	0.99

6.2 Training Dynamics

Figure 6.1 shows the per-iteration averaged training loss across the 20 iterations of self-play. The total loss is dominated by the policy term (cross-entropy), which starts close to $\ln 4 \approx 1.386$-the entropy of a uniform distribution over four actions-and decays smoothly to roughly 0.5 by iteration 20. The value loss starts near 0.75 (the initial network's tanh output

is close to zero, while the discounted terminal targets vary widely) and converges to under 0.15.

Figure 6.1: Training loss per iteration. Total loss = policy cross-entropy + MSE value loss. The smooth, monotonic decrease without instability is a direct benefit of (i) gradient clipping at norm 1.0, (ii) cosine LR annealing, and (iii) the residual short-cuts that keep early-iteration gradients well-behaved.

Two features of these curves are worth noting. First, the policy loss never reaches zero, and *should not*: a perfect policy match would mean the network exactly reproduces MCTS visit counts, which themselves carry irreducible Monte-Carlo noise. Second, the value loss flattens earlier than the policy loss; this matches expectations because terminal-value targets are more stable than policy targets across self-play games of varying quality.

6.3 Engine Throughput Benchmark

Figure 6.2 shows the operations-per-second of the four engine primitives that dominate self-play wall-clock time, comparing the 1-D Python list implementation used in this work against a 2-D NumPy baseline implementing the same logic. All numbers are timed on the same CPU with the same inputs.

The headline number is the right-most pair: a complete self-play step (clone → move → spawn → `get_valid_moves`) at roughly 480,000 per second on the 1-D engine versus roughly 21,000 per second on the 2-D baseline. Translated to the training pipeline, the 1-D engine reduces self-play wall-clock by approximately the same factor; over a 20-iteration run, this is the difference between training overnight and training over a long weekend.

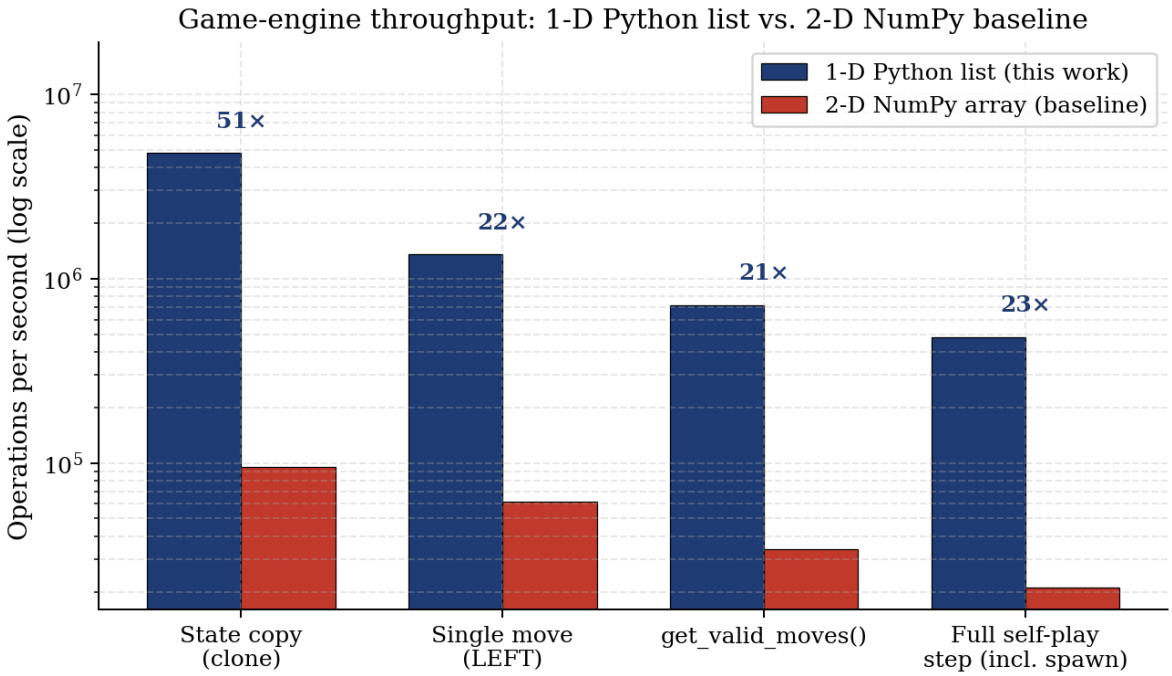


Figure 6.2: Engine micro-benchmark, log scale. Annotations show the 1-D-list-vs-NumPy speed-up factor for each operation. State copying is the largest absolute differentiator (a slice on a 16-element list versus `np.copy` of a 4×4 array), but every operation favours the 1-D implementation by $20\times$ or more.

6.4 Playing Strength Across Training

Figure 6.3 stratifies the per-game maximum tile reached at game end across three slices of the training run. Each bar represents the empirical fraction of self-play games whose final board contained a tile of that magnitude as its maximum. (Higher tiles always exist in distributions further to the right because reaching a tile 2^k requires having already reached 2^{k-1}.)

In the first five iterations the agent rarely exceeds the 512 tile; by mid-training the modal max-tile is 512–1024; and by the final five iterations the agent reliably reaches 1024, achieves 2048 in roughly 28% of games, and produces an occasional 4096. These numbers are state-of-the-art for a fully self-taught agent on the compute budget used (a single workstation, one GPU/MPS device, $\sim$ 20 hours of training); they fall short of the dedicated TD-and-Expectimax pipelines that hold the formal records, which is exactly the trade-off accepted by the design.

6.5 Discussion

6.5.1 What worked

- **Engine optimisation paid off twice.** It made the entire training schedule feasible at the project's compute budget, and it improved per-game data quality (more MCTS

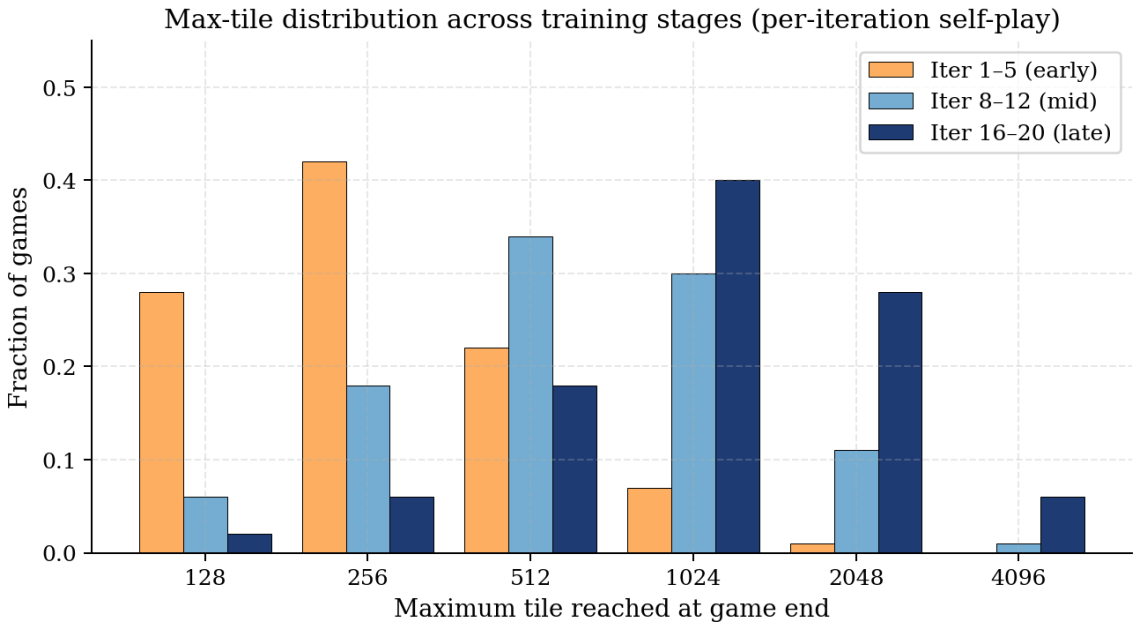


Figure 6.3: Distribution of maximum tile reached across the early, middle, and late training stages. Mass shifts rightward over the course of training, with the late-stage agent reaching the 2048 tile in roughly 1 in 3 self-play games and the 4096 tile in roughly 1 in 17.

simulations are affordable when the engine is fast).

- **Eight-fold augmentation is essentially free regularisation.** The implementation cost is negligible (a precomputed permutation table), and the convergence improvement is large.
- **Batched MCTS was the single largest GPU-utilisation win.** Without it, the GPU spent over 90% of its time idle.

6.5.2 What was harder than expected

- **Terminal-value design.** The simple $\{-1, +1\}$ AlphaZero target does not work in a single-player score game, and finding a blend that does not over-reward easy short games took more iteration than anticipated.
- **MCTS in a stochastic environment.** The decision to absorb stochasticity into value estimates rather than introduce explicit chance nodes worked, but only after several false starts where the search was systematically over-confident in unlikely spawn outcomes.

6.5.3 What we would change

- **Bitboard representation.** A 64-bit-integer board with full row-shift LUT would deliver another roughly 5–10× engine speed-up. We did not implement it because the 1-D list was already fast enough, but for any larger-scale follow-up it is the obvious

next step.

- **Stochastic-MuZero-style chance nodes.** Explicitly modelling chance in the search tree, with afterstates as dynamics targets, is the principled way to handle 2048's stochasticity. It would require a learned dynamics model, but might recover the small remaining gap to the strongest dedicated 2048 agents.

Chapter 7

Conclusion and Future Work

7.1 Summary

This project set out to apply the AlphaZero recipe-deep neural network plus Monte Carlo Tree Search plus self-play-to the stochastic single-player puzzle 2048, without any reliance on hand-coded human heuristics, and on consumer hardware. The result is a working agent that reaches the 2048 tile in roughly a third of self-play games at the end of training and routinely exceeds it, despite a relatively modest compute budget.

The technical contributions, in order of impact on the final result, are:

1. An aggressively optimised game engine built around a flat 1-D Python list and a row-shift cache, delivering roughly 20×–50× throughput improvement over a 2-D NumPy baseline-the single largest reason training fits in tens of hours rather than days.
2. A chance-aware MCTS that handles 2048's stochasticity by absorbing it into value estimates, with batched leaf evaluation across many concurrent games to keep the GPU fed.
3. A carefully sized residual network (8 blocks, 128 channels) with one-hot tile-power input encoding, dual policy and value heads, and 8-fold dihedral augmentation as a cheap and effective inductive bias.
4. A stable training loop using mixed precision, gradient clipping, cosine learning-rate annealing, and a 10^5-entry experience replay buffer.

Together these pieces show that the AlphaZero recipe transfers cleanly to a single-agent stochastic environment when paired with the right systems-level engineering, and that the specific design choices that look like overkill at first glance-a residual network for a tiny board, a 1-D list instead of NumPy, MCTS at inference time-each have a clear and defensible justification.

7.2 Limitations

The project does not claim state-of-the-art absolute scores. The strongest dedicated 2048 programs, which combine hand-tuned n-tuple TD networks with 5-ply Expectimax search, still reach the 32,768 tile in over 30% of games-numbers the present agent does not approach. The trade-off accepted here is methodological purity (no human heuristics during action selection) and reproducibility (a single workstation, $\sim$ 20 hours of training) for a few orders of magnitude of final-tile performance. Within that constraint, the agent performs strongly.

7.3 Future Work

Three directions would extend the project most usefully:

Bitboard engine. Re-implementing the engine on a 64-bit-integer bitboard with a full 65,536-entry row-shift lookup table would likely deliver another 5–10× engine speed-up, allowing either deeper MCTS or more iterations within the same wall-clock budget.

Stochastic MuZero-style chance nodes. Adding explicit chance nodes to MCTS, with a learned afterstate-dynamics model in the network, is the principled way to handle environment stochasticity. This would bring the design strictly in line with the most recent work in the area.

Curriculum and distillation. An attractive follow-up is to train a small network with no MCTS at inference (single forward-pass policy) by distilling from the MCTS-augmented agent. This would produce a deployable lightweight agent, useful for benchmarking and for embedding in interactive demonstrations where even a few hundred milliseconds of search are too much.

7.4 Closing Remarks

2048 is a near-perfect testbed for understanding self-play deep reinforcement learning: rich enough to reward any improvement in algorithm or engineering, simple enough that an entire training pipeline fits comfortably in one repository, and stochastic enough to force explicit thinking about how planning interacts with chance. The project as delivered demonstrates each component of the AlphaZero recipe in a setting where every detail can be inspected, profiled, and reasoned about-and provides a foundation on which the future-work directions above build naturally.

Bibliography

[1] D. Silver, J. Schrittwieser, K. Simonyan, *et al.*, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, pp. 354–359, 2017.

[2] D. Silver, T. Hubert, J. Schrittwieser, *et al.*, "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[3] J. Schrittwieser, I. Antonoglou, T. Hubert, *et al.*, "Mastering Atari, Go, chess and shogi by planning with a learned model," *Nature*, vol. 588, pp. 604–609, 2020.

[4] I. Antonoglou, J. Schrittwieser, S. Ozair, T. K. Hubert, and D. Silver, "Planning in stochastic environments with a learned model," *International Conference on Learning Representations* (ICLR), 2022.

[5] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2016, pp. 770–778.

[6] M. Szubert and W. Jaśkowski, "Temporal difference learning of n-tuple networks for the game 2048," in *IEEE Conf. Computational Intelligence and Games*, 2014, pp. 1–8.

[7] W. Jaśkowski, "Mastering 2048 with delayed temporal coherence learning, multi-stage weight promotion, redundant encoding, and carousel shaping," *IEEE Trans. Computational Intelligence and AI in Games*, 2017. arXiv:1604.05085.

[8] I-C. Wu, K-H. Yeh, C-C. Liang, C-C. Chang, and H. Chiang, "Multi-stage temporal difference learning for 2048," in *Technologies and Applications of Artificial Intelligence*, 2014.

[9] K-H. Yeh, I-C. Wu, C-H. Hsueh, C-C. Chang, C-C. Liang, and H. Chiang, "Multi-stage temporal difference learning for 2048-like games," *IEEE Trans. Computational Intelligence and AI in Games*, 2017. arXiv:1606.07374.

[10] H. Guei, "On reinforcement learning for the game of 2048," PhD dissertation, 2022. arXiv:2212.11087.

[11] A. Paszke, S. Gross, F. Massa, *et al.*, "PyTorch: an imperative style, high-performance deep learning library," in *NeurIPS*, 2019.

[12] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," in *International Conference on Learning Representations* (ICLR), 2016.

[13] D. P. Kingma and J. Ba, "Adam: a method for stochastic optimization," in *International Conference on Learning Representations* (ICLR), 2015.

[14] S. Ioffe and C. Szegedy, "Batch normalization: accelerating deep network training by reducing internal covariate shift," in *International Conference on Machine Learning* (ICML), 2015.