AgentPuzzler: Reinforcement Learning with MCTS for Efficient Jigsaw Puzzle Assembly

Renee Zbizika rzbizika@stanford.edu
Adrian Mendoza Perez mendy@stanford.edu
Allen Yuan ayuan06@stanford.edu

Abstract

Solving jigsaw puzzles computationally presents a significant challenge due to their inherent visual and spatial complexity. Traditional deep learning methods, such as convolutional neural networks (CNNs), struggle with this task as they primarily extract local features, failing to capture global spatial relationships necessary for successful puzzle assembly. Additionally, few works explore the non-square puzzle pieces. In this work, we propose AgentPuzzler, a reinforcement learning-based framework that integrates Monte Carlo Tree Search (MCTS) with a value network enhanced by visual information and an edge-matching function. Our approach allows the agent to iteratively refine its placements and make informed decisions by balancing exploration and exploitation. We evaluate AgentPuzzler against two baselines: a random action agent (AgentRandom) and an MCTS-based agent without visual information (AgentBlind). Our results demonstrate that AgentPuzzler significantly outperforms both baselines, achieving higher puzzle completion rates and improved learning efficiency. Our findings highlight the potential of reinforcement learning for structured visual problem-solving tasks, suggesting future directions in scalability, interpretability, and generalization to more complex puzzles.

1 Introduction

Jigsaw puzzles have long been a popular form of entertainment and cognitive exercise, combining visual perception, spatial reasoning, and problem-solving. Yet despite their simplicity for humans, the complexity inherent in puzzle-solving tasks, such as identifying puzzle pieces, clustering similar pieces, and determining correct adjacencies, poses a significant computational challenge.

A notable difficulty arises from combinatorial explosion–even modest puzzles can have a huge number of potential solutions, which makes it hard for brute-force or heuristic-based methods to scale. While deep learning methods, including convolutional neural networks (CNNs), have achieved impressive results across other visual tasks, they face limitations when applied to jigsaw puzzle assembly. CNNs typically focus on local feature extraction, reducing their effectiveness in tasks requiring comprehensive spatial reasoning and global understanding [4][8].

On the other hand, Reinforcement learning (RL) presents a promising alternative by allowing an agent to learn optimal assembly strategies through trial-and-error exploration, leveraging reward signals to refine decision-making over time.

We propose AgentPuzzler, an RL-based approach that integrates Monte Carlo Tree Search (MCTS) with an edge-matching function and visual information in a value network. This combination enables the model to approximate puzzle-solving strategies by leveraging both local features (color, edges) and global spatial context. We evaluate AgentPuzzler on complex pictorial jigsaw puzzles, comparing it against simpler baselines to assess the impact of visual and edge information on performance and robustness.

2 Related works

Jigsaw puzzle assembly has been explored through three primary computational approaches: (1) Deep learning-based methods, (2) Reinforcement learning (RL)-based approaches, and (3) Mathematical or heuristic-driven techniques. [4][5][6][7][10]; only one mathematical approach that we discuss is extended to apictorial pieces [3].

Noroozi et al. [4] and Paumard et al. [6] develop solutions utilizing CNNs to obtain information on puzzle pieces and then running classification algorithms after. As mentioned in our Introduction, while CNNs can extract local features of the puzzle pieces, they often strug-

gle to capture global spatial coherence, a crucial requirement for accurately assembling complex jigsaw puzzles [4][8].

Paumard et al. [5] also propose an RL-based approach inspired by AlphaZero, framing puzzle assembly as a sequential game. Song et al. [7] utilizes deep RL with a model-free RL approach, using a Siamese network and Q-learning instead. RL-based methods provide adaptability and can train without explicit heuristics, though often require more computational demand and can converge unpredictably [5][7].

On the other hand, mathematical formulations provide an alternative, more structured approach to puzzle solving. Hoff et al. [3] use differential invariant signatures to compare edges. An iterative approach was also developed by Wei et al. [10], which incrementally refines puzzle layouts by applying unary and binary constraints. These methods excel in deterministic settings, but can be sensitive to noise in boundary extraction [3].

3 Methods

We formulate jigsaw puzzle assembly as a sequential decision-making problem, where an agent must iteratively place pieces in selected locations to reconstruct an image. Our simulation environment is implemented in PyGame, where the agent interacts with puzzle pieces and receives feedback based on placement accuracy. Further details on problem formulation, environment setup, and dataset processing can be found in the Appendix.

3.1 State and Action Representation

Each state s consists of:

1. A collection of `Piece` objects, each with a position (x, y) and image data.
2. An `assembly` matrix that represents the puzzle's current layout.
3. A set of unplaced puzzle pieces indicating which puzzle pieces remain to be placed.
4. Time elapsed, used as a reference for progress evaluation

At each step, the action a is defined by

$$(\texttt{piece\_id}, \texttt{dx}, \texttt{dy})$$

where a selects a piece and applies a displacement. The state transitions from s to s' by updating the piece's coordinates and removing it from the unplaced set if correctly positioned. The puzzle is solved when all pieces are placed correctly and non-overlapping.

Initialization

At the start of each episode, the environment is initialized as follows. For puzzle pieces:

- Each puzzle piece is randomly placed within the board area but outside the solution box.
- A unique $piece_id$ is assigned to each piece for tracking.

For the initial state s_0:

- s_0.assembly = A$x_n \times y_n$ zero matrix representing the empty puzzle layout, where x_n and y_n define the number of puzzle pieces in rows and columns.
- s_0.unplaced pieces = The set of all puzzle pieces, since no placements have been made.
- s_0.time elapsed = 0, representing the initial time step.

Terminal State

The puzzle-solving process ends when all pieces are placed in the solution box. Formally, we define the terminal state by whether the centroid of every puzzle piece lies within the grey solution box. Once the terminal condition is met, the environment returns a completion signal, marking the puzzle as solved. See appendix for note on terminal state.

Rewards

Our reward function consists of intermediate rewards during assembly and a final accuracy metric at the end of the episode. At each step, the agent receives feedback based on its actions:

$$r_t = \begin{cases} R_{\text{place}}, & \text{if piece placed correctly} \\ -R_{\text{move}}, & \text{otherwise} \end{cases}$$

The agent gains positive reward R_{place} for correctly placing a puzzle piece in the solution box, and a small negative penalty R_{move} for additional movements. Since rewards are relatively sparse, the value network is crucial role for guiding the search process by predicting long-term success.

3.2 MCTS Overview

Our methodology uses Monte Carlo Tree Search (MCTS), which systematically explores potential actions from a given state s_t through a sequence of four phases: Selection, Expansion, Simulation, and Backpropagation, iterated over visits N_{visits} times to determine the most promising action a_t [2] (See Appendix for more). Starting from the root node, MCTS selects nodes based on a balance between exploration (trying unexplored moves) and exploitation (choosing moves with known high rewards) until it reaches a leaf node. The tree expands by adding new nodes, then simulations predict outcomes using a simplified model or random play. Finally, simulation results are backpropagated to update node statistics, improving future decision-making. MCTS is particularly effective when exhaustive search is infeasible [2].

3.3 Policy Network (P)

The policy network $P(s)$ is a neural network that predicts a probability distribution over possible actions a_t, where an action consists of selecting a puzzle piece and determining its placement. It takes as input the current puzzle state representation, including assembly matrix, unplaced pieces, time elapsed.

The network is trained via categorical cross-entropy loss, which minimizes the negative log-likelihood of the chosen action:

$$L_{policy} = -\sum_{t} logP(a_s|s_t)$$

where $P(a_s|s_t)$ represents the predicted probability of the chosen action at state s_t. By minimizing L_{policy}, the network learns to predict high-reward puzzle-placement strategies.

3.4 Value Network (V)

The value network $V(s)$ is another neural network that estimates the expected reward from a given puzzle state s_t. It predicts how close the current state is to the correct solution, helping MCTS prioritize promising paths. We explicitly define the value function as:

$$v(s_t) = V(s_t, \mathbf{f}(s_t))$$

, where we embed both the raw state s_t and feature vector $\mathbf{f}(s_t)$ and combine them for prediction. $\mathbf{f}(s_t)$ contains the key elements used to evaluate the puzzle state:

$$\mathbf{f}(s_t) = SSIM(s_t, s_{\text{goal}}, C)$$

where:

- $SSIM(s_t, s_{\text{goal}})$ uses Structural Similarity Index Measure [9]
- C is the overall edge compatibility score, which quantifies how well the piece edges align

3.5 Edge Matching and Compatibility

To evaluate the geometric compatibility of puzzle pieces, we use differential invariant signatures [3]. For edge e_i in set E of all puzzle-piece edges, the curvature of e_i is computed using finite-difference approximations, computing both curvature and its derivative with respect to arc length.

We define a compatibility matrix C, where each entry C_{ij} quantifies the geometric fit between edges e_i and e_j. To quantify the geometric compatibility between two puzzle piece edges, we compute the mean Euclidean distance between their curvature signatures. We then define C_{ij} as:

$$C_{ij} = \exp\left(-\alpha \cdot d\left(e_i, e_j\right)\right)$$

where α is a sensitivity parameter controlling the decay rate. A lower mean Euclidean distance $d\left(e_i, e_j\right)$ results in a higher compatibility score, meaning the edges are more likely to belong together.

To incorporate geometric constraints into decision-making, we extend the Euclidean Signature Method to generate compatibility scores for each state s_t, which are fed into the value network as input features in $\mathbf{f}(s_t)$.

4 Experiments

4.1 Architecture and Hyperparams

Neural Networks

In Tables 1 and 2, we summarize the Policy and Value Network architectures. We opted for simpler models to reduce overfitting and improve interpretability and control over input features. [1]

Table 1: Policy Network Architecture & Hyperparameters

	Component	Details
1	Number of Layers	3
2	Layer 1	Linear (`state_dim` $\rightarrow$ 30)
3	Activation 1	ReLU
4	Layer 2	Linear (30 $\rightarrow$ `action_dim`)
5	Activation 2	ReLU
6	Output Layer	Linear (`action_dim`)
7	Activation (Output)	Softmax

Table 2: Value Network Architecture & Hyperparameters

	Component	Details
1	Number of Layers	4
2	State Input Layer	Linear (`state_dim` $\rightarrow$ 25)
3	State Activation	ReLU
4	Visual Input Layer	Linear (`visual_dim` $\rightarrow$ 128)
5	Visual Activation	ReLU
6	Combined Layer	Linear (153 $\rightarrow$ 128)
7	Combined Activation	ReLU
8	Output Layer	Linear (128 $\rightarrow$ 1)

Training Hyperparameters and MCTS Architecture

In Table 3, we summarize our hyperparameter choices for training and the Monte Carlo Tree Search (MCTS) architecture.

Hyperparameter Optimization: To determine optimal hyperparameters, we conducted grid search experiments over learning rate (0.001–0.01), discount factor (0.95–0.99), and MCTS simulation depth (2–10). Given

[1] The 153-dimensional combined layer in the Value Network results from adding the visual input and state dimensions.

the sparse reward space, we incorporated high intermediate rewards for correct placements and tested exploration constants (0.1–1.0) to balance exploration and exploitation. (See Appendix for details.)

Standard Parameters: We followed standard RL practices, using Adam as the optimizer (consistent with Alphazzle) and setting $\gamma = 0.99$ to prioritize long-term rewards while maintaining responsiveness.

LazyPuzzler Due to compute constraints, we tested exploration constants, rewards, and optimizers using a simplified "lazy" trainer with 3 epochs, 3 steps per epoch, 3 MCTS iterations, and a depth of 2, enabling rapid evaluation before full training.

Table 3: Training and MCTS Hyperparameters

Hyperparameter	AgentPuzzler	LazyPuzzler
Epochs	10	5
Max Steps per Epoch	10	3
Max MCTS Iterations	5	3
Max Simulation Depth	5	2
Exploration Constant	0.25	0-1
Discount Factor Gamma	0.99	0-1
Optimizer	Adam	Adam, RMSProp, SGDM
R_place	50.0	0-75
R_time	-0.01	(-1, -0.001)
Smoothing	No	Yes

4.2 Models and Baselines

In addition to our full model presented in our Methodology and Experiments, we present our baselines below.

- AgentPuzzler - Our complete model, incorporating MCTS with edge matching and visual information.
- AgentBlind - A simplified MCTS-based agent where the value network lacks access to visual information, relying only on state-based features.
- AgentRandom - A baseline agent that selects actions randomly, with no learning or decision-making strategy.

These baselines help evaluate the impact of visual information, structured decision-making, and learning-based policies on puzzle-solving performance.

5 Results and Discussion

Metrics

To compare performance across experiments, we use the following quantitative metrics:

- Puzzle Completion Rate - percentage of pieces correctly placed in their target positions at the end of an episode.
- Cumulative Reward - total reward accumulated by an agent over an episode
- Loss Overtime - sum of policy loss and value loss over training iterations, measuring how well the agent is learning and converging.

5.1 AgentPuzzler vs. AgentRandom vs. AgentBlind

We evaluate AgentPuzzler against the aforementioned baseline agents, AgentRandom and AgentBlind. After training, we analyze loss trends, puzzle completion rates, and cumulative rewards across epochs in Figure 1.

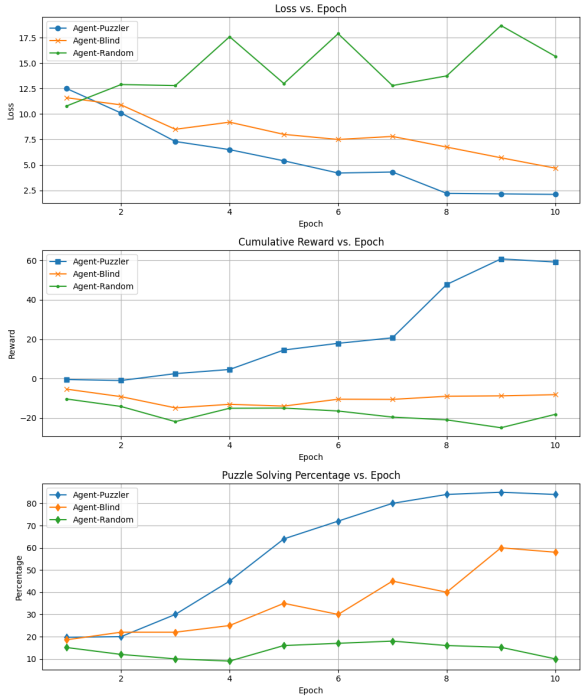


Figure 1: Training progression of AgentPuzzler, AgentBlind, and AgentRandom, showing loss reduction (top), cumulative reward accumulation (center), and puzzle completion rate (bottom) across epochs.

We summarize the key findings as follows:

Loss Trends: As expected, AgentRandom fails to converge, maintaining a high loss throughout training. Both AgentPuzzler and AgentBlind exhibit loss convergence, with AgentPuzzler achieving slightly lower final loss, suggesting improved policy learning.

Puzzle Completion Rate: While AgentBlind initially improves, its final performance lags behind AgentPuzzler, confirming our hypothesis that visual information in the value network enhances learning.

Cumulative Reward: Despite similar loss trends, AgentPuzzler achieves significantly higher cumulative rewards, which demonstrates that visual information enables more effective long-term planning.

Test Results After training, we run the models on two test images. Below we show a summarized table of the highest puzzle accuracy that each of the models scored on each of the images from the test set, and the average MSE of the piece placements.

Table 4: Highest Puzzle Completion Rate for Each Model on Test Images

Model	Img 1 Rate (%)	Img 2 Rate (%)
AgentRandom	0.05	0.00
AgentBlind	41.9	37.163
AgentPuzzler	61.5	72.163

The results in Table 4 demonstrate a clear performance hierarchy among the models:

AgentPuzzler outperforms both baselines, achieving 61.5% completion on Image 1 and 72.16% on Image 2. AgentBlind learns some structure but still struggles with complex visual differentiation, while AgentRandom fails entirely. The performance gap between AgentBlind and AgentPuzzler highlights the importance of visual features in the value network.

5.2 AgentPuzzler Self-Comparison

To further analyze AgentPuzzler's learning progression, we compare its performance against previous versions of itself during training.

To further confirm this, we visualize the policy and value networks using a heatmap. The heatmaps illustrate how AgentPuzzler's value function evolves over training.

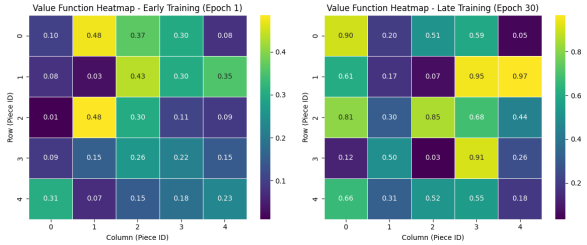


Figure 2: Evolution of AgentPuzzler's Value function: Early training (Epoch 1) vs. Late training (Epoch 10).

We analyze the differences below:

- Epoch 1: The agent assigns low and widely distributed values, reflecting random exploration with no clear strategy.
- Epoch 10: The value function becomes more structured, with higher values concentrated in key placements (e.g., corners and central positions).

This shift suggests that AgentPuzzler progressively learns to prioritize optimal placements, reinforcing its ability to solve puzzles more efficiently as training progresses.

Failure Analysis

To better understand AgentPuzzler's limitations, we conducted a qualitative failure analysis, identifying scenarios where the model consistently underperforms. We observed that AgentPuzzler performed the worst on the puzzle shown in Figure 3.

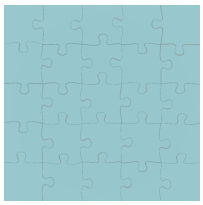


Figure 3: Image Associated with the Lowest Puzzle Completion Rate

With visually identical pieces, the value network relies almost entirely on edge-matching, forcing the agent into trial-and-error moves. In contrast, puzzles with diverse visual features help the model learn spatial relationships more effectively. This suggests that AgentPuzzler's dependence on visual cues may limit performance in low-texture puzzles, highlighting the need for stronger structural or spatial heuristics.

6 Conclusion and Future Work

In this project, we introduced a novel method combining MCTS inspired by [2][5], with an edge-matching function—originally proposed by [3]—and visual information integrated into the value network to improve jigsaw puzzle-solving. Our results demonstrate that AgentPuzzler significantly outperforms a random baseline and an MCTS-based agent without visual input, confirming that integrating learned representations with search-based methods improves puzzle-solving performance.

Our current scope was constrained primarily by limited resources. With increased compute capabilities, we see several encouraging directions which can be explored:

- **Enhanced Baseline Comparisons**: Implementing and comparing additional sophisticated baselines, such as an interactive game with human players or alternative RL algorithms using Q-learning.
- **Deeper Model Interpretability Analysis**: Conducting detailed investigations into model decision-making processes, policy evolution, and value network transformations to provide deeper insights into model behavior and learning dynamics.

By addressing these areas, we can further validate and refine reinforcement learning-driven puzzle assembly, contributing to broader advancements in structured visual problem-solving.

7 Appendix

7.1 Game and Problem Formulation

We define a jigsaw puzzle as an arbitrary image that is randomly divided into $N = H \times W$ puzzle pieces, where

H and W represent the number of pieces per column and row, respectively. Each piece contains interjambs (tabs) and indentations (holes), which serve as structural features for determining correct placements. To solve the puzzle, we use PyGame as our simulation environment, where the agent interacts with and manipulates puzzle pieces. The environment provides feedback based on the agent's actions, guiding the learning process.

7.2 Pipeline Overview

Our approach begins with a training dataset of unlabeled images, which are first divided into randomized jigsaw-like pieces. Each piece is saved as a separate .svg file and later compiled into a PyGame-rendered simulation, where our algorithm attempts to reassemble the puzzle. The agent's objective is to optimize piece placements based on learned spatial and visual relationships. The final output is the fully reconstructed puzzle, assembled by our model to the best of its ability.

7.3 Assumptions

During our game and problem formulation, we make several assumptions about the nature of our problem: 1. The board is proportional to the size of the puzzle.

2. The pieces in the source image comprise one entire puzzle solution (you can not mix and match pieces from other puzzles).

3. The final puzzle is rectangular in shape and all pieces fit neatly into a grid.

4. Pieces in game may overlap or touch, and removing the top one will reveal more information about the one below

5. The pieces of the puzzle are standard or "canonical" in shape - square with 4 distinct, sharp corners and four resulting sides.

6. All intersections of pieces in the puzzle will be at the corners of the puzzle pieces and all internal intersections will be at the corners of four pieces.

7. Each side is has up to one interjamb (colloquially known as "tabs"), hole, or is a flat edge.

8. The environment knows there are N puzzle pieces and the final puzzle is rectangular in shape where its grid dimensions are $H \times W$

9. Only one puzzle is handled at a time, and all pieces on the current board pertain to that puzzle (no missing or extra pieces)

10. Each piece has a correct placement, where minor overlapping of pixels is tolerated

11. All pieces are in the correct orientation and require no rotations

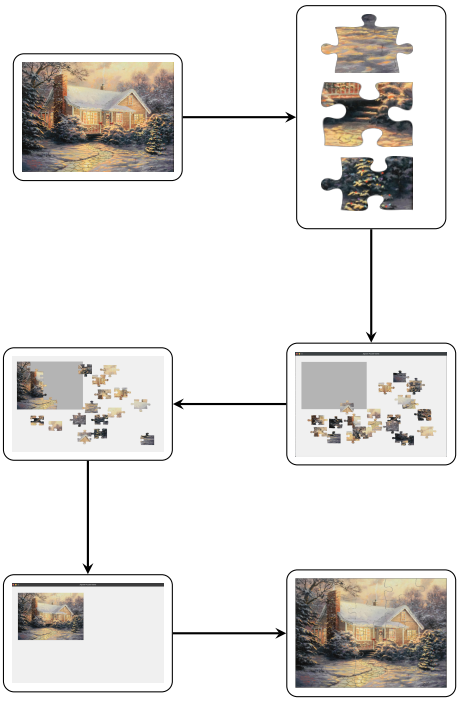
12. A grey solution box, where the puzzle pieces are placed, matches the assembled image size.

For a visual representation, see the Puzzle Workflow below.

7.4 Puzzle Workflow

In Figure 5, we present a visual workflow of AgentPuzzler (which hopefully also builds some intuition about the overall game and problem formulation).

Figure 4: Puzzle workflow. The original image is split into multiple puzzle pieces and then loaded into PyGame, which is then solved to form a completed puzzle.



7.5 Extended Euclidean Signature Method

$$d\left(e_i, e_j\right) = \frac{1}{n} \sum_{k=1}^{n} \sqrt{\left(\kappa_i(k) - \kappa_j(k)\right)^2 + \left(\kappa_i'(k) - \kappa_j'(k)\right)^2}$$

where:

- $\kappa_i(k)$ and $\kappa_j(k)$ are the curvature values at point k along edges e_i and e_j, respectively.
- $\kappa_i'(k)$ and $\kappa_j'(k)$ are the curvature derivatives at point k.
- n is the total number of sampled points along the edge contours.

Quick note on terminal State

Note that the solution returned may not be correct, for which we evaluate the accuracy of the final arrangement separately by comparing the placed pieces to the ground-truth solution. This ensures that while simply filling the solution box indicates a finished game, it does not falsely indicate success if pieces are misplaced.

7.6 Dataset and Features

Our dataset contains six unlabeled images from open source image websites [1]. We dedicate 4 to our training set and 2 to our test set. We preprocess unlabeled images through our pipeline by first generating an SVG puzzle template using a random seed for rows, columns, and tab size. Images are resized to match this template, which defines puzzle-piece boundaries. The SVG is converted into a binary PNG mask using intensity thresholding, clearly separating pieces from outlines. Puzzle pieces are then isolated using this mask, removing noise, and scaled to a uniform size for PyGame. Our features include each piece's RGBA values, edge signatures, and metadata such as piece ID.

7.7 MCTS

Selection

During selection, the algorithm traditionally selects a node to expand to by following the Upper Confidence Bound (UCB) derived vector $U(a|s_t)$[2], which assigns values to each available action from the state s_t. Then, the best action, according to $U(a|s_t)$, is applied recursively until a leaf state is reached. If the leaf has children already, the Selection algorithm can either stop or selects a child.

UCB was adapted for trees in Upper Confidence Trees (UCT) and further adapted for policy integration in Polynomial Upper Confidence Tree (PUCT) [2][5]. During selection, we implemented this PUCT, a variation of the Upper Confidence Bound (UCB) strategy adapted for trees, specifically tailored to integrate neural network-based policy outputs, which influence the selection strategy during tree exploration.

PUCT uses the following formula to select actions:

$$U(a|s_t) = Q(a|s_t) + C \cdot \pi_\theta(a|s_t) \cdot \frac{\sqrt{N(s_t)}}{1 + N(a|s_t)}$$

where $Q(a|s_t)$ is the expected value of taking action a from state s_t, derived from previous simulations. $\pi_\theta(a|s_t)$ is the policy prediction from the neural network, which provides a probability distribution over actions suggesting how promising each action is according to the learned policy. C is a constant determining the level of exploration; higher values encourage exploring less visited nodes. $N(s_t)$ is the total number of visits to the current node, and $N(a|s_t)$ is the number of times action a has been explored from state s_t. During the tree traversal, PUCT balances between exploiting actions that have historically led to high rewards and exploring actions that are less tried but might lead to higher rewards according to the neural network's policy output. The integration of $\pi_\theta(a|s_t)$ makes PUCT particularly powerful in systems where a neural network models the underlying policy. This model-based approach allows PUCT to leverage deep learning insights directly in the decision-making process, enhancing the traditional MCTS which relies solely on simulation data.

Dynamic Adjustment: The term $\frac{\sqrt{N(s_t)}}{1+N(a|s_t)}$ dynamically adjusts the exploration term based on how many times the node and the action have been visited, ensuring that all potentially good actions are sufficiently explored over time.

Expansion and Simulation

Following selection, nodes are expanded based on unexplored actions, enhancing the tree structure with potentially beneficial new states derived from the best available actions determined by UCT, as described in AlphaZero [5].

The Simulation phase aims to accurately estimate the value of an expanded node and its predecessors for use in the Backpropagation phase. Instead of random playouts, we incorporate a neural network V that predicts the expected value of states, replacing traditional simulation strategies. This network evaluates the outcomes of actions more efficiently and accurately, informing subsequent phases of the MCTS.

Backpropagation

This phase updates the values $Q(a|s_t)$, and visit counts $N(a|s_t)$ and $N(s_t)$ for each node are updated based on the value $v(s_t)$ from simulations, using the formula:

$$Q(a \mid s_t) \leftarrow \frac{N(a \mid s_t) \cdot Q(a \mid s_t) + v(s_t)}{N(a \mid s_t) + 1},$$
$$N(a \mid s_t) \leftarrow N(a \mid s_t) + 1, \quad N(s_t) \leftarrow N(s_t) + 1. \tag{1}$$

The expected utility of actions is refined as more outcomes are observed.

7.8 Hyperparameter Choices (cont.)

MCTS Iterations: Higher iterations provide better policy rollouts by reducing variance in action selection but at increased compute cost. We selected 10 iterations to balance this.

Simulation Depth: A deeper simulation horizon allows the agent to evaluate future rewards more effectively. However, increased depth exponentially increases computational cost. We tested depths from 2 to 10 and found 5-8 to be a reasonable balance between foresight and efficiency.

7.9 Additional Results

Figure 5 shows that AgentPuzzler achieves faster convergence in later training stages, indicating improved learning efficiency and better policy refinement over time.

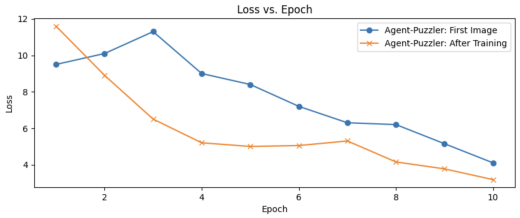


Figure 5: AgentPuzzler: First Train vs. After Train Loss

8 Contributions

Renee: Model architecture, training, results, and writeup.
Adrian: Dataset creation, data evaluation, pipeline creation, and writeup.
Allen: Preprocessing data, Pygame environment creation, and writeup.

References

[1] Libraries and resources: NumPy, SciPy, scikit-image, OpenCV, PyTorch, Matplotlib, cairosvg, svgwrite, Pygame, unsplash.com, and freepik.com.

[2] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[3] Daniel J. Hoff and Peter J. Olver. Automatic solution of jigsaw puzzles. *Journal of Mathematical Imaging and Vision*, 49(1):234–250, August 2013.

[4] Mehdi Noroozi and Paolo Favaro. Unsupervised learning of visual representations by solving jigsaw puzzles. In *European Conference on Computer Vision (ECCV)*, pages 69–84. Springer, 2016.

[5] Adrien Paumard, Bastien Dupont, and Claude Martin. Alphazzle: A reinforcement learning approach for jigsaw puzzle assembly. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*. IEEE, 2021.

[6] Adrien Paumard, Bastien Dupont, and Claude Martin. Deepzzle: A graph-based approach for jigsaw puzzle reassembly with convolutional classifiers. In *Proceedings of the IEEE Winter Conference on Applications of Computer Vision (WACV)*. IEEE, 2022.

[7] Wei Song, Qiang Liu, and Jin Zhao. Deep reinforcement learning for jigsaw puzzle solving via siamese networks. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. IJCAI, 2021.

[8] Xiaolong Wang, Ross Girshick, Abhinav Gupta, and Kaiming He. Non-local neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7794–7803. IEEE, IEEE, 2018.

[9] Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.

[10] Yu Wei, Lei Wang, and Lin Zhu. An iterative constraint-based approach to jigsaw puzzle reassembly. In *Proceedings of the British Machine Vision Conference (BMVC)*. BMVA, 2020.