AI-Powered Shape and Color Sorting

1st Vidheya Bole
Master Of Science in Computer Science
Northeastern University
Boston, MA
bole.v@northeastern.edu

2nd Tanya Mistry
Master Of Science in Computer Science
Northeastern University
Boston, MA
mistry.t@northeastern.edu

3rd Rajvi Desai
Master Of Science in Computer Science
Northeastern University
Boston, MA
desai.r1@northeastern.edu

***Abstract*—Object sorting by shape and color is common in many industries, but doing it manually can be prone to errors. The project explores to automate such sorting through reinforcement learning by developing a two-stage environment. The environment was created using Pygame and wrapped with Gym to support reinforcement learning. A Double Q-Network (Double DQN) was used to train agent capable of learning of to sort objects by shape in the first stage and by color in the second. Over multiple training episodes, the agent demonstrated accuracy which suggested that reinforcement learning can model and automate basic sorting operations.**

I. Introduction

Sorting of objects based on their shape and color play an important role in many industries like manufacturing, logistics etc. These sorting mechanism prove to be a critical stage within a large company, since one mistake can cost a lot. So, in these cases automated workflow and a robot trained to do the sorting is necessary for better accuracy and faster sorting time to reduce any unnecessary delays.

Traditionally, object-sorting tasks are handled by humans. Manual sorting can be effective for some applications but it is still prone to human error, fatigue, and inconsistencies. Even when there are mechanical systems to sort the objects they can lack a certain kind of adaptability that AI machines can provide through trial and error.

To address these challenges, integrating Artificial Intelligence technologies, particularly reinforcement learning, gives better results. Reinforcement learning involves training agents to perform tasks through interactive experimentation with an environment. In contrast to supervised methods that rely on labeled datasets, RL independently discovers the environment and makes strategies through a trial-and-error learning process which is guided by reward feedback. The adaptability and self-improvement capabilities make RL suitable for dynamic, sequential tasks that require strategic decision making and ongoing environment interaction like sorting objects.

The current project explores the use of RL to automate a two-stage sorting task, where objects must first be sorted according to their shape and then by their color. A custom interactive environment was developed by Pygame library, providing visual interactivity and real-time feedback during sorting. This environment was integrated with Gym that facilitated structured training and evaluation procedures for the RL agent. The reinforcement learning algorithm employed for this study is the Double Deep Q-Network (DQN), chosen for its improved performance compared to the standard Q-learning methods.

Through this project, reinforcement learning has shown potential in automating multi-step sorting tasks, highlighting the clear advantages AI-based solutions can bring to industries of manufacturing and logistics. The results encourage further exploration into applying these methods to more complex real-world tasks, potentially involving visual recognition and robotic controls. Overall, this study confirms that reinforcement learning can address structures sorting challenges that open doors to smarter and adaptable automation in the future.

II. Related Work

Recent advancements in artificial intelligence have employed reinforcement learning for object sorting tasks. This project builds on several previous studies and approaches described in recent papers and sites. Zhang et al. (2024) demonstrated an integration of a Deep Q-Network with an improved YOLO algorithm for sorting objects. Their approach used deep RL which was coupled with real-time object detection and achieved high sorting accuracy in dynamic scenarios.

Similarly, Monfared and Ebadzadeh (2018) used a hierarchical Machine-based RL strategy to enhance sorting process. This method reduced the training time and provided adaptability without relying on large labeled datasets or manual tuning.

Vision-based intelligent sorting robots have also been explored by Orouji (2023), who proposed a system that employs deep reinforcement learning techniques to manage sorting tasks using visual inputs. This study highlighted the capability of reinforcement learning agents to dynamically adapt and improve their sorting strategies over successive interactions.

In robotics, object sorting tasks utilizing RL have garnered attention, particularly due to their adaptability to dynamic conditions and minimal need for human intervention. Robotic systems employing RL have demonstrated efficiency in tasks such as waste sorting, package classification, and manufacturing quality control.

Now, how do researchers actually build these systems? Many start with tools like Gym, a toolkit that acts as a playground for training RL agents, and Pygame, which lets developers create simple visual simulations (think: a basic video game to test a robot's "vision"). These tools make it easier to prototype ideas and literally see how a digital trainee—say, a

robot arm—is behaving. No more guessing—just tweak, test, and repeat.

Of course, not all RL algorithms are created equal. While the classic DQN is a workhorse, researchers often spice things up with upgrades like Double DQN (which reduces overconfidence in AI decisions) or techniques that prioritize past experiences—like a student focusing harder on the problems they got wrong. For this project, the team chose Double DQN because it's like giving the AI a built-in reality check, helping it learn faster and avoid costly mistakes.

III. Problem Statement and Methods

A. Problem Statement

Manual sorting methods present significant challenges, including being labor-intensive, error-prone, and poorly scalable. These limitations negatively impact productivity, efficiency, and accuracy in industries where sorting operations are critical, thus necessitating reliable automated solutions.

B. Environment Design

To address this, a custom, dual-stage sorting environment was developed using Pygame for visualization and integrated with Gym to standardize reinforcement learning interactions. The environment consists of two sequential sorting tasks:

- Stage 1: Sorting a falling object based on its shape into one of four predefined bins.
- Stage 2: Sorting the same object based on its color, again into one of four bins.

This structured setup effectively simulates real-world sorting tasks, providing clear and immediate feedback on sorting performance.

C. Agent Design

The AI agent interacts with the environment by observing a structured state and selecting from a discrete set of actions:

- State Space: A 10-dimensional vector comprising the current sorting stage (shape/color), a one-hot encoding for the object's shape (circle, triangle, square, cross), a one-hot encoding for the object's color (red, green, blue, yellow), and the object's normalized horizontal position.
- Action Space: The agent can choose from three possible actions:
  - Move left
  - Move Right
  - Drop the object into the bin

The reward structure reinforces correct sorting behavior with +10 points per correct sorting stage, penalizes incorrect sorting with -10 points, and provides an additional +10 bonus reward if both stages are correctly completed in a single trial.

IV. Environment Design

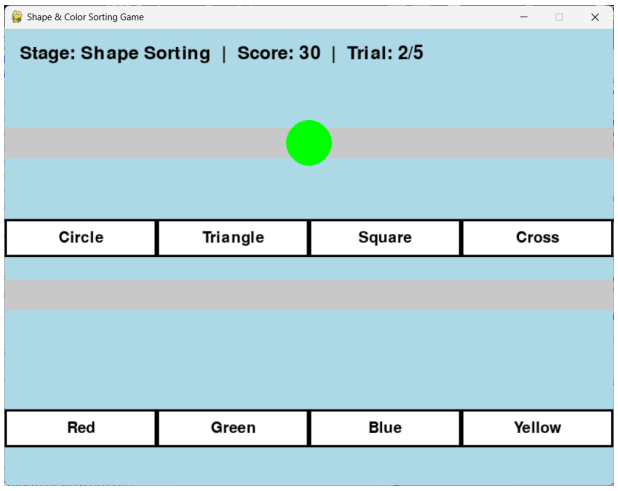

Fig 1. Environment Design

The environment is implemented in `shape_sort_game.py` using the `pygame` library, and wrapped as an OpenAI Gym environment for reinforcement learning compatibility. It simulates a two-stage sorting task:

Stages

1) **Stage 0 – Shape Sorting**: An object with a randomly assigned shape and color appears on a conveyor. The agent must move and drop the object into the correct shape bin.
   - Reward: +10 for correct drop, -10 for incorrect.
2) **Stage 1 – Color Sorting**: The same object is transferred to a second conveyor. Now the agent must sort it into the correct color bin.
   - Reward: +10 for correct drop, -10 for incorrect.
   - Bonus: Additional +10 if both stages are correct.

Observation Space

A 10-dimensional vector is used to represent the current state:

- Element 0: Sorting stage (0 = shape, 1 = color)
- Elements 1–4: One-hot encoding for shape (circle, triangle, square, cross)
- Elements 5–8: One-hot encoding for color (red, green, blue, yellow)
- Element 9: Normalized x-position of the object on the conveyor

Action Space

A discrete action space of size 3:

1) Move Left
2) Move Right
3) Drop the object

Rendering Modes

- `human`: Enables GUI display using pygame
- `rgb_array`: Disables GUI, used for faster training

Episodes

Each episode consists of a fixed number of trials (default is 5). After shape and color sorting in each trial, the environment either starts a new trial or terminates the episode.

V. Agent Design and Implementation

The reinforcement learning agent is implemented in `ai_agent.py` using PyTorch and the DQN (Deep Q-Network) algorithm with Double DQN enhancements.

Neural Network Architecture

The agent uses a fully connected neural network:

- Input layer: 10 neurons (same as state space dimension)
- Hidden layers: Two hidden layers with 256 neurons and ReLU activations
- Output layer: 3 neurons (corresponding to the 3 possible actions)

Double DQN

Two separate networks are used:

- **Policy Network**: Selects actions based on current policy.
- **Target Network**: Evaluates Q-values for stability.

The target network is periodically updated to reflect the policy network's weights.

Experience Replay

A replay buffer stores tuples of experiences:

$$(state, action, reward, next\_state, done)$$

Random samples from the buffer are used for training, breaking correlation between consecutive transitions.

Loss Function

The loss is computed using Mean Squared Error (MSE) between the predicted Q-values and the target Q-values:

$$L = \mathbb{E}\left[(Q(s,a) - (r + \gamma \cdot \max_a Q'(s',a)))^2\right]$$

Training Parameters

- Learning Rate: 1×10^{-3}
- Discount Factor (γ): 0.99
- Batch Size: 64
- Epsilon Decay: 0.98 (faster convergence)

Training Loop

Training is done over multiple episodes. The agent:

1) Selects actions using an ϵ-greedy policy
2) Stores transitions in replay memory
3) Optimizes the policy network using sampled batches
4) Periodically updates the target network

Checkpointing

The agent saves checkpoints after every episode to a file. On restart, it resumes from the last saved episode, preserving:

- Policy and target network weights
- Optimizer state
- Epsilon value

Evaluation

A separate evaluation function runs the trained agent in GUI mode for a number of episodes to visualize performance. No learning is performed during evaluation.

VI. Deep Q-Learning Algorithm (DQN)

Deep Q-Learning (DQN) is a model-free, value-based reinforcement learning algorithm that approximates the Q-value function using a deep neural network. It allows an agent to learn optimal policies directly from high-dimensional state spaces, such as the 10-dimensional vector in our sorting environment.

Q-Learning Recap

The goal of Q-learning is to learn a function $Q(s,a)$ that estimates the expected cumulative reward when taking action a in state s, and following the optimal policy thereafter. The update rule is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a)\right]$$

where:

- s: current state
- a: action taken
- r: reward received
- s': next state
- γ: discount factor
- α: learning rate

Neural Network as Function Approximator

Instead of storing Q-values in a table (which is infeasible for large state spaces), DQN uses a deep neural network with parameters θ to approximate $Q(s,a;\theta)$.

In our project, the architecture is:

- Input: 10-dimensional observation vector
- Hidden layers: 2 fully connected layers with 256 neurons each, using ReLU activation
- Output: 3 Q-values corresponding to each action (Move Left, Move Right, Drop)

Training the Network

The Q-network is trained by minimizing the Mean Squared Error (MSE) between the predicted Q-values and the target Q-values. The target for each transition (s,a,r,s') is computed as:

$$y = r + \gamma \cdot \max_{a'} Q(s',a';\theta^-)$$

where θ^- are the parameters of a separate target network, which is updated periodically to match the policy network.

The loss function becomes:

$$L(\theta) = \mathbb{E}_{(s,a,r,s')\sim D}\left[\left(Q(s,a;\theta) - y\right)^2\right]$$

Experience Replay

To stabilize learning, the agent stores transitions (s,a,r,s') in a replay buffer. During training, it samples random mini-batches from this buffer to break the correlation between sequential experiences and improve data efficiency.

Double DQN

Standard DQN can overestimate Q-values. Double DQN addresses this by decoupling action selection from evaluation:

$$y = r + \gamma \cdot Q(s', \arg\max_{a'} Q(s', a'; \theta); \theta^-)$$

Here, the next action is selected using the policy network (θ), but its value is evaluated using the target network (θ^-). This reduces overestimation and improves stability.

Exploration Strategy

An ϵ-greedy strategy is used:

- With probability ϵ, select a random action (exploration)
- With probability $1 - \epsilon$, select the best action based on current Q-values (exploitation)

ϵ is decayed exponentially over time:

$$\epsilon \leftarrow \max(\epsilon_{\text{min}}, \epsilon \cdot \epsilon_{\text{decay}})$$

Optimization Details

- Optimizer: Adam
- Learning Rate: 1×10^{-3}
- Discount Factor γ: 0.99
- Batch Size: 64
- Replay Buffer Size: 50,000 transitions
- Target Network Update Frequency: every 10 episodes

Benefits of DQN in This Project

- Learns from high-dimensional, non-tabular state space
- Improves over time through exploration and experience replay
- Can generalize to unseen states, improving real-world applicability
- Modular architecture allows easy integration with Gym environments

VII. Discussion and Conclusion

Discussion

The project presents a successful application of RL and Double DQN in a sorting environment. The environment which is designed using `pygame` and wrapped in a Gym interface which then offers a two-stage task that mimics real-word scenario: first by shape and then by color. This multi-stage structure introduces complexity and tests the agent's ability to retain state awareness across transitions.

Throughout training, the agent progressively learned to associate the state features (stage, shape, color, and position) with the correct actions (left, right, drop). The use of Double DQN reduced the overestimation of Q-values, leading to more stable and accurate learning. Additionally, techniques such as experience replay and target network separation were instrumental in achieving convergence.

Several experiments showed that the agent's reward per episode increased over time, reflecting its learning progress. The agent performed poorly in early episodes due to random exploration but gradually improved as the ϵ value decayed and the agent exploited its learned policy more frequently.

However, the training performance was sensitive to hyper-parameters such as the learning rate, batch size, and epsilon decay. Faster epsilon decay improved convergence speed but sometimes led to suboptimal policy learning. Also, running the agent with GUI rendering enabled during training significantly slowed down performance, which is why a non-GUI render mode (`rgb_array`) was preferred for training.

Conclusion

This project validates that reinforcement learning can be effectively applied to dynamic, multi-stage decision-making problems like object sorting. By using a custom-built environment and implementing a Double DQN-based agent, the system was able to learn to sort objects accurately by both shape and color. The agent's ability to generalize sorting strategies highlights the strength of DRL in solving structured but non-trivial tasks.

The key takeaways from this project include:

- Double DQN provides improved stability over standard DQN by reducing overestimation.
- Experience replay and target network separation are essential components for successful training.
- A well-designed environment with structured observation and reward signals significantly contributes to effective learning.

Future Work

Several enhancements can be explored in future iterations:

- Add object attributes (e.g. size, texture) for multi-dimensional sorting.
- Adapt to multi-object or continuous action environments.
- Test complex RL algorithms including Dueling DQN, Prioritized Experience Replay, and Actor-Critic techniques.
- Gradually increase the complexity of sorting tasks to support curriculum learning.

This experiment demonstrates how reinforcement learning can automate real-world categorization and sorting tasks.

VIII. Team Contribution

Tanya Developed the interactive sorting environment using Pygame, integrating it with Gym for visualization and agent interaction. Vidheya and Rajvi implemented the reinforcement learning agent using the Double DQN algorithm and agent training. All team members collaboratively analyzed experimental outcomes, discussed results, prepared the project presentation and the final report.

Code Availability

GitHub Code Link: GitHub

References

[1] F. Zhang, L. Zhang, W. Li, and J. Zhang, "Object Classification and Sorting Based on Deep Q-Network and Improved YOLO Algorithm," *Mathematics*, vol. 12, no. 13, p. 2001, 2024. [Online]. Available: https://www.mdpi.com/2227-7390/12/13/2001

[2] M. T. Monfared and A. M. Ebadzadeh, "Utilizing Hierarchical Extreme Learning Machine Based Reinforcement Learning for Object Sorting," *ResearchGate*, 2018. [Online]. Available: https://www.researchgate.net/publication/329990491

[3] A. Orouji, "Vision-Based Intelligent Sorting Robot Using Deep Reinforcement Learning," Bishop's University Technical Report, Jan. 2023. [Online]. Available: https://www.ubishops.ca/wp-content/uploads/orouji20230108.pdf

[4] N. Tagliapietra, A. Croce, S. Graziani, and M. Matteucci, "Robotic Object Sorting via Deep Reinforcement Learning: A Comparison Study," *Semantic Scholar*, 2019. [Online]. Available: https://www.semanticscholar.org/paper/86950ffcf39ef0d145f13365390c9504cdabd6b7

[5] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[6] OpenAI, "Gym: A toolkit for developing and comparing reinforcement learning algorithms," [Online]. Available: https://www.gymlibrary.dev

[7] PyTorch, "PyTorch Documentation," [Online]. Available: https://pytorch.org

[8] Pygame, "Pygame Documentation," [Online]. Available: https://www.pygame.org/docs/