AI Powered Ping Pong using Supervised and Reinforcement Learning

Course Name : Reinforcement Learning (Team Project)

Submitted by: Manesh Kumar, Shankar Lal , Ali Sher

Instructor : Mareike Keil

Institution : Hochschule Fulda University of Applied Sciences

Date of Submission : 8th July, 2025

Abstract

This project presents an AI-powered version of the classic Ping Pong game using both Supervised Learning (SL) and Reinforcement Learning (RL) techniques. The primary goal is to train an AI agent capable of learning how to play the game effectively through both labeled data and trial-and-error interaction.

In the first phase, a supervised model was trained using gameplay data labeled with optimal paddle movements based on the ball's position and velocity. The model learned to predict the next best move given a game state. In the second phase, a reinforcement learning agent was developed that interacted with the game environment and improved its performance over time using a defined reward mechanism.

The project was implemented using Python, PyGame for game simulation, and machine learning libraries such as Scikit-learn and TensorFlow. Both models were tested and compared in terms of accuracy, game performance, and learning efficiency.
Results show that while supervised learning offers quick initial training, reinforcement learning allows for more adaptive and strategic gameplay over time. This project treats supervised learning as a baseline to compare with reinforcement learning, helping to highlight the strengths and limitations of each approach in training game-playing agents within simple environments..

Table of Contents

1. Introduction

In recent years, Artificial Intelligence has made significant progress in mastering games, from simple arcade style environments to complex strategic challenges. Among the various AI methodologies, Supervised Learning and Reinforcement Learning stand out as powerful tools for training intelligent agents. This project aims to explore both these paradigms by developing an AI agent capable of playing the classic Ping Pong game.

The motivation behind this project lies in understanding how machine learning techniques can be applied to game environments where decisions are made in real time based on dynamic inputs. By designing a system where an AI agent can control the paddle in a Ping Pong game, we analyze how effectively it can learn to respond to the ball's position and movement using different learning strategies.

The project is divided into two main phases. In the first phase, a supervised learning model was trained using a dataset created from manually played games. The dataset included features such as the ball's position, its velocity, the paddle's position, and the corresponding action taken. Multiple machine learning models were trained and evaluated to identify the most accurate predictor for paddle movements.

In the second phase, the focus shifts to reinforcement learning, where the AI agent learns optimal gameplay strategies through trial and error by interacting with the environment. Unlike the supervised approach, reinforcement learning does not rely on labeled data but instead uses a reward-based system to guide learning.

By comparing these two approaches within the same gaming context, the project provides practical insights into the strengths and trade-offs of SL and RL when applied to interactive, real-time tasks. The implementation leverages Python, PyGame, and machine learning libraries such as Scikit-learn, TensorFlow, and others.

This report documents the complete development process, from data generation and preprocessing to model training, evaluation, and gameplay integration. It also details the finalized reinforcement learning component and establishes a basis for future improvements.

2. Literature Review/Background

The application of machine learning to games has long been a testbed for developing and evaluating AI techniques. From early rule-based systems in chess to advanced deep reinforcement learning agents in modern video games, games offer an interactive and measurable environment for training intelligent agents. In this project, we explore two prominent learning paradigms Supervised Learning (SL) and Reinforcement Learning (RL), within the context of the Ping Pong game.

2.1 Supervised Learning in Game AI

Supervised Learning involves training a model on a labeled dataset where the correct output (label) is provided for each input. In the case of games, this often involves collecting data from expert gameplay and using it to train a model to mimic the expert’s decisions. Several studies and projects have demonstrated the viability of this approach for simpler games where the state space is relatively small and deterministic.

For example, supervised learning has been successfully applied in games like Tetris and Flappy Bird, where human-generated gameplay data serves as a training base. The model learns to predict the next move based on the current game state, without any feedback from the environment.

2.2 Reinforcement Learning in Game AI

Reinforcement Learning, on the other hand, allows an agent to learn optimal behavior through interactions with an environment. The agent receives feedback in the form of rewards or penalties and uses this feedback to improve its strategy over time. Unlike supervised learning, RL does not require labeled data but instead relies on a reward function to guide the learning process.

RL has gained prominence with the success of Deep Q-Networks (DQN), Policy Gradient Methods, and Actor-Critic algorithms, which have powered AI agents to outperform humans in games like Atari, Go, and Dota 2. These agents learn from raw sensory input (like pixels or state variables) and improve through continuous exploration and exploitation of the environment.

In the context of this project, RL is employed to create a self-learning Ping Pong agent that improves gameplay performance over time. The agent adjusts its paddle movements based on the ball's position and dynamics, optimizing its actions to maximize long-term rewards (e.g., returning the ball successfully).

2.3 Related Work and tools

Previous implementations of game AI have utilized libraries such as OpenAI Gym, PyGame Learning Environment (PLE), and Unity ML Agents to build and test game-based agents. While this project focuses on a custom PyGame-based Ping Pong setup, it draws conceptual inspiration from similar RL environments like Atari Pong, where state representation, reward functions, and action spaces are carefully designed.

Additionally, various ML frameworks like Scikit-learn, XGBoost, LightGBM, and TensorFlow/Keras have enabled rapid prototyping and evaluation of models, particularly during the supervised learning phase.

3. Dataset Creation

A critical component of this project was the creation of a dataset for training the supervised learning models. Unlike pre-existing datasets commonly found in machine learning problems, game-specific datasets especially for custom environments must often be generated manually. This presented both a challenge and a learning opportunity for our team.

3.1 Game Environment and Setup

To simulate the Ping Pong game, we used PyGame, a Python library designed for creating 2D games. The environment consisted of:

- A ball that moves across the screen with velocity in both the X and Y directions
- A paddle controlled by the player or agent, which moves vertically to intercept the ball
- A basic collision detection system to simulate bouncing

This setup allowed for full control over the game physics and access to internal game state variables, which was essential for generating training data.

3.2 Manual Gameplay and Data Logging

Since no labeled dataset was available, we manually played the Ping Pong game multiple times while logging game states and actions taken at each frame. Each frame generated a data record consisting of the following six fields:

Field	Description
ball_x	X-coordinate of the ball
ball_y	Y-coordinate of the ball
ball_dx	X-direction speed of the ball
ball_dy	Y-direction speed of the ball
paddle_y	Y-coordinate of the paddle
action	0 = No move, 1 = Move up, 2 = Move down

The dataset is stored in a structured CSV file, making it easy to import into data analysis and machine learning pipelines.

3.3 Dataset Size and Balance

We collected several thousand gameplay frames to create a reasonably sized dataset for training. Special care was taken to:

- Include diverse ball trajectories and paddle positions
- Balance the number of samples across the three possible actions to prevent model bias toward "no movement" (which naturally occurs more often)

3.4 Challenges in Data Collection

Some challenges we faced during this stage included:

- Ensuring smooth and accurate logging without disrupting gameplay
- Maintaining a balance in action classes to avoid class imbalance
- Handling overlapping frames that contained similar state-action pairs

Despite these hurdles, the generated dataset served as a robust foundation for the supervised learning phase of the project.

4. Exploratory Data Analysis (EDA)

Exploratory Data Analysis (EDA) was performed to understand the structure, distribution, and relationships within the dataset before applying machine learning models. EDA helped identify potential issues such as class imbalance, feature scaling needs, and data redundancy.

4.1 Dataset Overview

The structure and contents of the dataset have already been described in detail in Section 3 (Dataset Creation). As a quick summary, the dataset consists of numeric game state features and an action label representing the player's decision at each frame.

4.2 Descriptive Statistics

We computed basic statistical summaries (mean, median, min, max, standard deviation) for each numerical feature. This helped understand the value ranges and variability in the dataset. For example:

- ball_x ranged from 0 to the screen width (~800)
- ball_y ranged from 0 to the screen height (~600)
- ball_dx and ball_dy mostly ranged between -5 and 5 (step sizes)
- paddle_y varied within the height of the play area

These insights confirmed that the input features were well-scaled and within logical bounds of the game environment.

4.3 Class Distribution (Action Analysis)

We analyzed the distribution of the `action` variable to check for class imbalance:

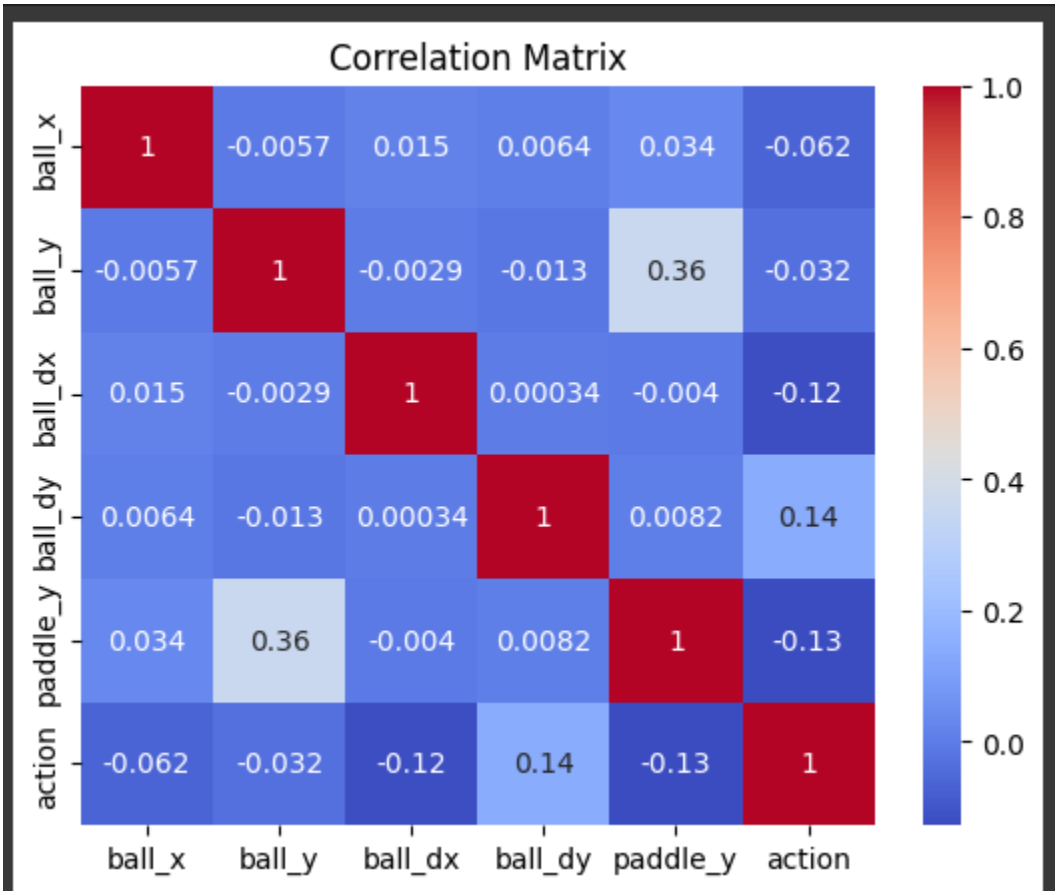
Action	Meaning	Frequency or Percentage
0	No Movement	~60%
1	Up	~20%
2	Down	~20%

The dataset was slightly imbalanced, with "No movement" being the most common action. To address this:

- We used stratified sampling for train-test split
- Considered class weights in model training where applicable
- Performed oversampling or undersampling when needed

4.4 Correlation Analysis

To understand how features relate to each other, we generated a correlation matrix using Pearson correlation coefficients.



Key Observations:
ball_x, ball_y, ball_dx, and ball_dy are mostly uncorrelated with one another.

The strongest correlation observed is between ball_y and paddle_y, with a correlation of 0.36, indicating a moderate positive relationship. This makes intuitive sense, as the paddle likely follows the vertical movement of the ball.

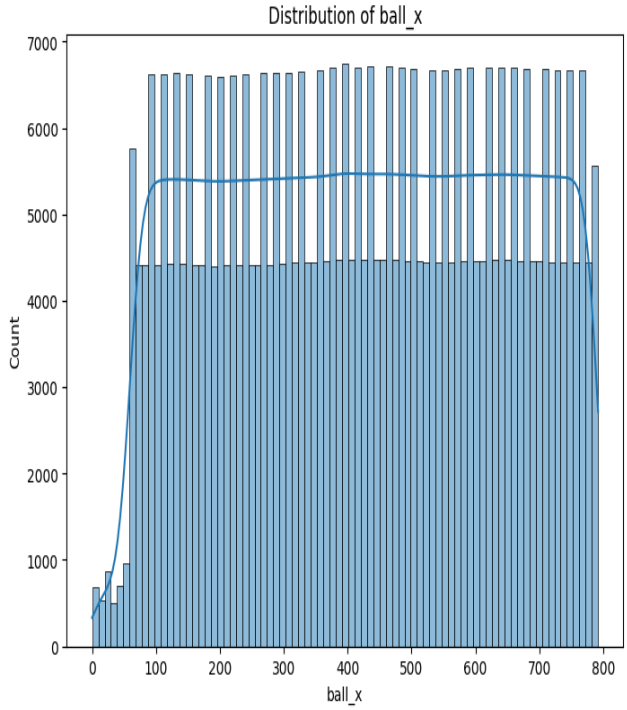
action shows weak correlations with all features:
Highest: ball_dy (0.14) and paddle_y (-0.13)

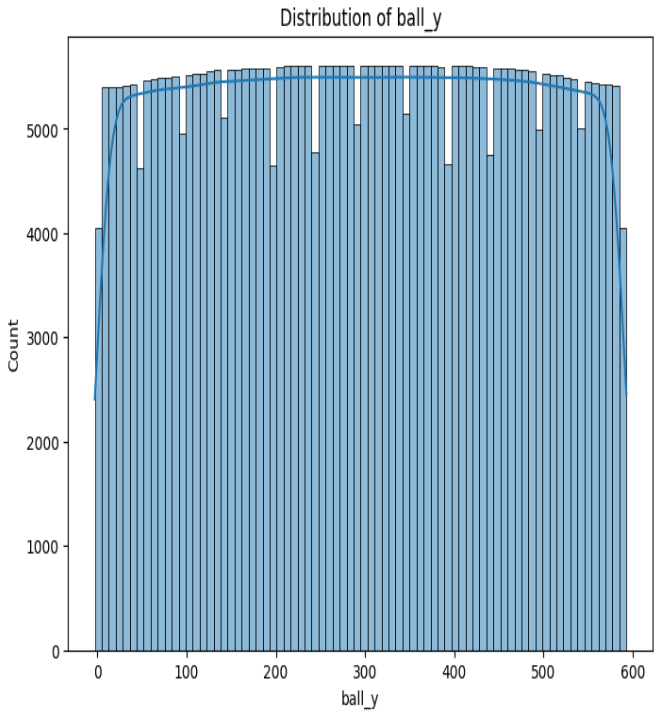
Lowest: ball_x (-0.062)

This suggests that no single feature dominates the decision-making, which is ideal for training machine learning models without biasing them towards one variable.
So there is no multicollinearity issue in the dataset, and all features were retained for model training. While some relationships exist, particularly between ball_y and paddle_y these do not pose any problem for modeling and may even improve prediction accuracy.

4.5 Data Visualization :

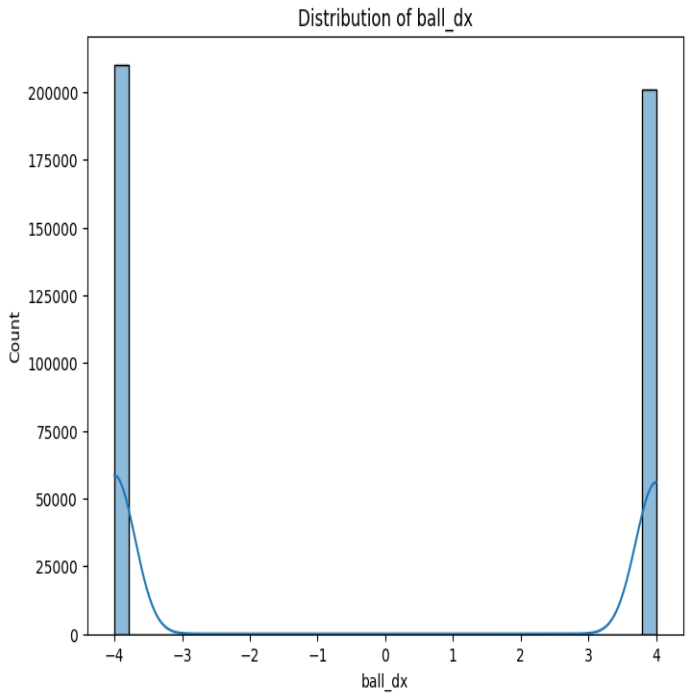
To better understand the distribution of individual features, histograms were plotted for all columns in the dataset. These visualizations provided insights into the spread, skewness, and general behavior of the variables.

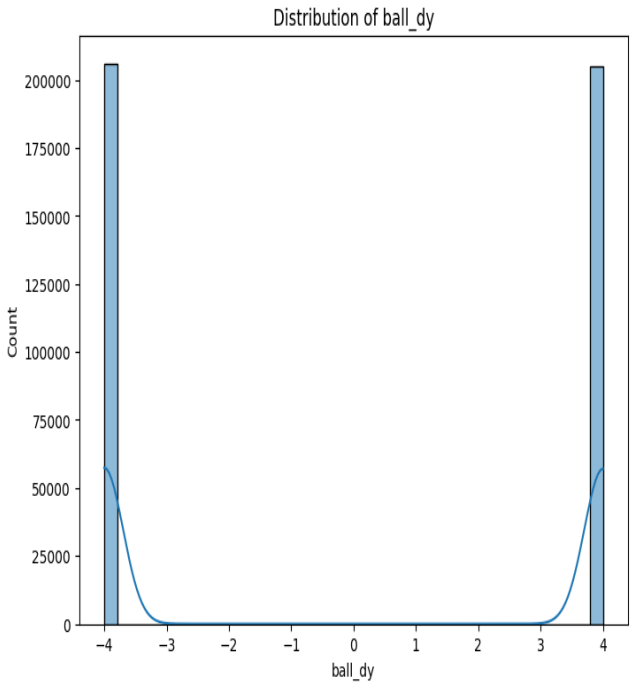




ball_x Distribution:
The histogram of ball_x shows a nearly uniform distribution across the screen width (0 to ~800 pixels), with slightly lower counts at the edges.
This suggests that the ball travels quite evenly across the horizontal axis during gameplay, which is expected behavior in a balanced ping pong environment.

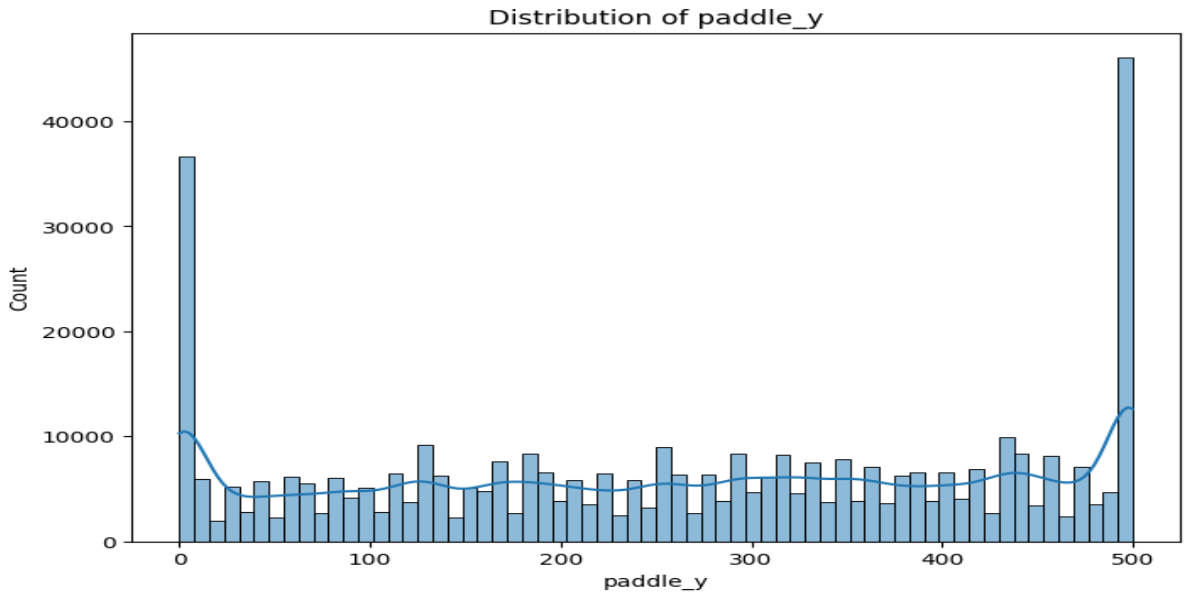
ball_y Distribution:
The ball_y distribution is also approximately uniform but slightly more concentrated in the middle Y-range.
The slight bell shape indicates that the ball tends to spend more time around the center vertically, possibly because of paddle interactions or bounce logic keeping it away from extreme top and bottom edges.





ball_dx & ball_dy Distribution Analysis (Directional Velocities):
The values are concentrated at two extremes: +4 and -4.
This indicates that the ball mostly moves at a fixed speed, either left or right (ball_dx) and either up or down (ball_dy). The sharp bimodal distribution (with peaks at both ends and almost no values in between) suggests that The ball does not accelerate gradually. Its direction switches suddenly. This observation is important for modeling because:

- It implies velocity isn't a complex continuous feature rather, a binary directional indicator (left/right or up/down).
- For ML models, these can almost behave like categorical features in behavior, even if they're technically numeric.



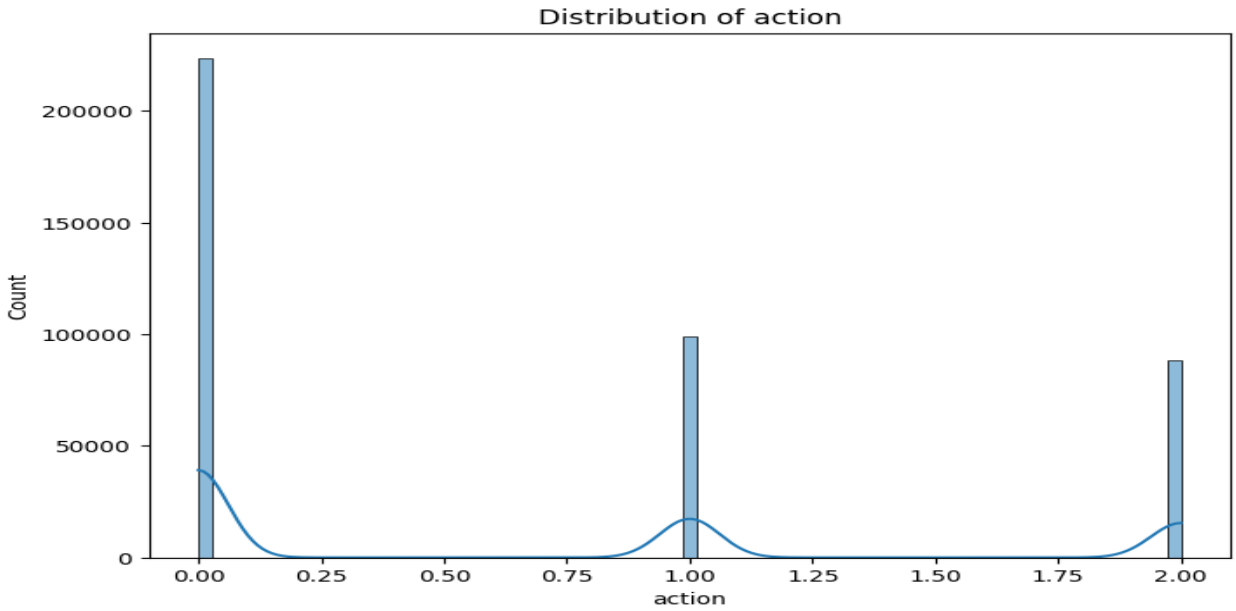
Paddle Position Distribution Issue

During data collection, the paddle_y feature representing the vertical position of the paddle was found to have a skewed distribution. The values range from 0 to 500, where 0 indicates the paddle is stuck at the bottom and 500 indicates the paddle is stuck at the top while values between 0 and 500 represent normal paddle movement.

However, due to incorrect key presses during data collection (e.g pressing the "up" key when the paddle was already at the top, or "down" when it was at the bottom), the dataset contains a disproportionately high number of records with values exactly at 0 and 500. This has resulted in a skewed distribution with heavy concentration at the extremes and fewer samples in the middle range. Such imbalance can negatively affect model performance by biasing it towards predicting extreme positions more often than necessary. However, we found the solution which you can find below.

Solution for Paddle Position Imbalance

Instead of removing the over-represented data points where the paddle was stuck at the top (paddle_y = 500) or bottom (paddle_y = 0), we applied a correction strategy to retain all samples while improving data quality. Specifically, we adjusted the action labels based on the relative position of the ball. If the paddle was at the bottom but the ball was still above, we reassigned the action to "move up." Similarly, if the paddle was at the top but the ball was still below, we reassigned the action to "move down." This approach helped preserve the dataset's size and diversity while mitigating the effect of incorrect labels caused by invalid key presses during data collection.



The action field is the target variable with three classes: 0 for no paddle movement (stable), 1 for upward movement, and 2 for downward movement.
Most recorded actions were 0, indicating that the paddle remained stable in many frames. This class imbalance was considered during model evaluation to ensure fair performance comparison.

The solution will be shared below in Data pre-processing

5. Data Preprocessing

To ensure the quality and reliability of the dataset before training models, several preprocessing steps were performed:

5.1 Handling Missing Values

The dataset was examined for null or missing entries. Fortunately, the data was mostly complete, and no imputation was required.

5.2 Removing Duplicates

Duplicate records were identified and removed to prevent bias and redundancy in the training process.

5.3 Correcting Paddle Imbalance

As discussed earlier, the paddle_y column had extreme values (0 and 500) due to manual errors during data collection. Instead of removing these samples, we corrected the corresponding action values based on the ball's position to maintain dataset integrity.

5.4 Feature Scaling

To bring all features to a common scale, Standardization was applied using StandardScaler from Scikit-learn.

```
from sklearn.preprocessing import StandardScaler

features = ['ball_x', 'ball_y', 'ball_dx', 'ball_dy', 'paddle_y']
scaler = StandardScaler()
data[features] = scaler.fit_transform(data[features])
```

5.5 Handling Class Imbalance:

The dataset exhibited class imbalance, with the action = 0 (no movement) being the majority class. To address this, under sampling was applied to class 0 to match the size of the minority classes (1 and 2). The resulting dataset was shuffled to ensure randomness. As we can see the result in below image. The classes are somehow balanced

```
# Separate classes
class_0 = data[data['action'] == 0]
class_1 = data[data['action'] == 1]
class_2 = data[data['action'] == 2]

# Find the smallest class size
min_count = min(len(class_1), len(class_2))

# Randomly sample class 0 to match smaller classes
class_0_sampled = class_0.sample(min_count, random_state=42)

# Combine all balanced classes
balanced_data = pd.concat([class_0_sampled, class_1, class_2], ignore_index=True)

# Shuffle the dataset
balanced_data = balanced_data.sample(frac=1, random_state=42).reset_index(drop=True)

print("Balanced action distribution:")
print(balanced_data['action'].value_counts())
```

```
Balanced action distribution:
action
1    99029
0    88408
2    88408
Name: count, dtype: int64
```

5.6 Train-Test Split

The final preprocessed dataset was split into training and testing sets using an 80:20 ratio to evaluate model performance effectively.

```
[ ]  from sklearn.model_selection import train_test_split

     # Separate features and target
     X = balanced_data.drop('action', axis=1)
     y = balanced_data['action']

     # Train-test split (80-20)
     X_train, X_test, y_train, y_test = train_test_split(
         X, y, test_size=0.2, stratify=y, random_state=42
     )
```

6. Supervised learning Models and Evaluation

This section focuses on building and evaluating multiple machine learning classifiers to predict the paddle action (0: No movement, 1: Move Up, 2: Move Down) based on the ball's real-time position and movement. A robust evaluation framework was implemented to ensure fair comparisons between models and extract meaningful insights regarding each algorithm's behavior.

6.1 Evaluation Function

A custom evaluate_model function was developed to streamline the evaluation process. This function performs the following steps:

- Prediction: Performs inference on the test set using the trained model.
- Accuracy Score: Quantifies the overall correctness of predictions.
- AUC-ROC Score: Calculates the Area Under the Curve using a One-vs-Rest (OvR) approach for multi-class classification. This score highlights each model's ability to distinguish between the classes under various threshold settings.
- Classification Report: Presents precision, recall, F1-score, and support for each action class.
- Confusion Matrix: Visualized using a heatmap to identify where the model tends to confuse specific actions.

- AUC-ROC Curve Plot: The curve provides a visual depiction of how well each model performs across different classification thresholds.

6.2 Model Performance Summary

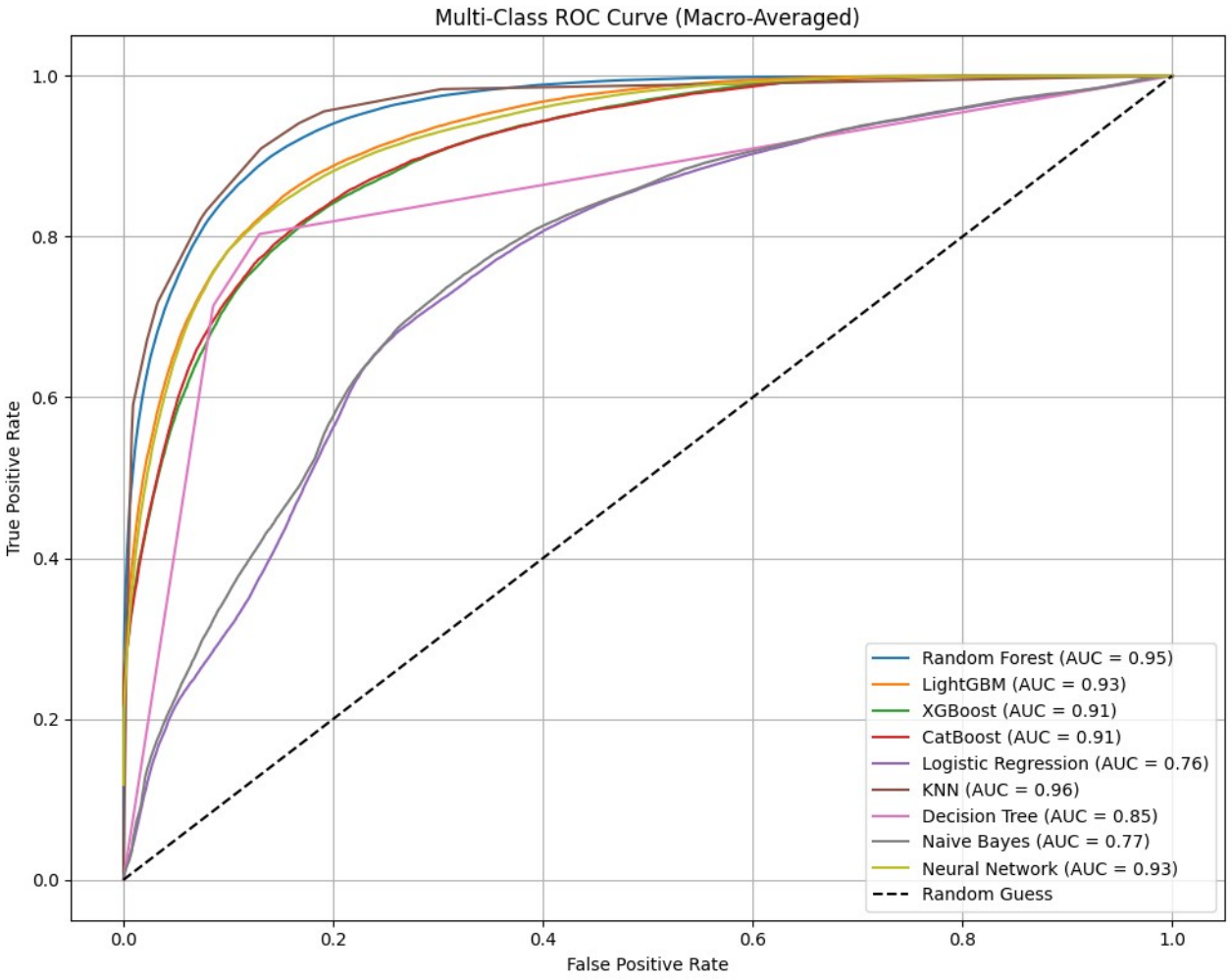
Nine machine learning models were trained and evaluated on the same test dataset using the standardized pipeline. Below are their test set accuracy scores:

Model	Accuracy
K-Nearest Neighbors (KNN)	0.84
Random Forest	0.83
Decision Tree	0.80
Neural Network	0.79
LightBGM	0.79
XGBoost	0.76
Logistic Regression	0.58
Naive Bayes	0.59
CatBoost	0.32

6.3 AUC-ROC Curve Insights

To enhance the evaluation beyond accuracy, AUC-ROC curves were plotted for each model that supports probability estimates. This visualization reveals how well each model distinguishes between action classes at various thresholds:

- KNN, Random Forest, and Neural Network models not only achieved high accuracy but also exhibited strong AUC-ROC curves, suggesting balanced sensitivity across all classes.
- XGBoost and LightGBM demonstrated competitive ROC profiles, indicating their strength in probabilistic classification.
- Logistic Regression and Naive Bayes, while achieving moderate accuracy, showed weaker ROC curves, emphasizing their limitations in capturing complex decision boundaries.
- CatBoost had both poor accuracy and ROC curve behavior, possibly due to parameter sensitivity, inadequate training, or dataset characteristics incompatible with the model.



6.4 KNN Optimization and Hyperparameter Tuning

Given that the K-Nearest Neighbors (KNN) model initially achieved the highest accuracy (0.84) among all classifiers, it was selected for further optimization. To enhance its performance, a **Grid Search** strategy with cross-validation was applied to systematically explore various hyperparameter combinations.

The following hyperparameters were tuned:

- **n_neighbors**: Tested from 1 to 30.
- **weights**: uniform and distance weighting strategies.

- **metric**: Distance metrics including euclidean manhattan , and minkowski.

A 5-fold cross-validation was conducted using gridSearchCV, with AUC_ROC (One vs rest) as the scoring metric to ensure balanced class performance in this multi-class setting

```python
from sklearn.model_selection import GridSearchCV

# Define parameter grid
param_grid = {
    'n_neighbors': list(range(1, 31)),
    'weights': ['uniform', 'distance'],
    'metric': ['euclidean', 'manhattan', 'minkowski']
}

# Use the same model type (don't pass the fitted model)
from sklearn.neighbors import KNeighborsClassifier
knn_for_tuning = KNeighborsClassifier()

# Set up GridSearchCV
grid_search = GridSearchCV(
    estimator=knn_for_tuning,
    param_grid=param_grid,
    scoring='roc_auc_ovr',   # or 'f1_macro', 'roc_auc_ovr' for multi-class
    cv=5,
    verbose=1,
    n_jobs=-1
)

# Fit on training data
grid_search.fit(X_train, y_train)

# Best model and params
print("Best Parameters:", grid_search.best_params_)
print("Best AUC ROC (CV):", grid_search.best_score_)

# Use the best model
best_knn = grid_search.best_estimator_

# Evaluate on test data
from sklearn.metrics import classification_report
y_pred = best_knn.predict(X_test)
print(classification_report(y_test, y_pred))
```

```
Fitting 5 folds for each of 180 candidates, totalling 900 fits
Best Parameters: {'metric': 'euclidean', 'n_neighbors': 7, 'weights': 'distance'}
Best Accuracy (CV): 0.9562801381205922
              precision    recall  f1-score   support

           0       0.82      0.76      0.79     17682
           1       0.87      0.90      0.88     19806
           2       0.85      0.88      0.86     17681

    accuracy                           0.85     55169
   macro avg       0.85      0.85      0.85     55169
weighted avg       0.85      0.85      0.85     55169
```

```
[ ] save_model(best_knn, "knn_model_tuned.pkl")
```

7. Model Deployment

After evaluating various machine learning models, the K-Nearest Neighbors (KNN) classifier was selected for deployment due to its superior performance. With an optimized configuration obtained via GridSearchCV, the final model used 7 neighbors, Euclidean distance as the metric, and distance-based weighting. This configuration achieved the highest classification accuracy (0.85) and AUC-ROC score across all tested models.

To deploy the model, the following steps were taken:

- The trained and optimized KNN model was serialized using Python's pickle library. This allows the model to be saved in a .pkl file and loaded during gameplay without the need for retraining.
- The game environment was implemented using PyGame, simulating a real-time Ping Pong game. At each frame, the current game state (ball position, ball velocity, paddle position) was captured and preprocessed using the same StandardScaler that was fit during training.
- The scaled features were then fed into the KNN model, which returned the predicted action (0 = stay, 1 = move up, 2 = move down). The paddle responded accordingly in real-time, creating an AI-controlled opponent capable of adaptive gameplay.

This deployment demonstrates a practical integration of machine learning into a dynamic game environment. It validates the effectiveness of the trained model beyond static metrics by testing it under continuous decision-making conditions, as would be expected in real-world applications.

8. Reinforcement Learning

While the supervised learning approach provided a reliable baseline model for paddle control, it lacks adaptability to unforeseen or evolving scenarios. To overcome this limitation and enable the AI to learn through experience, we planned the integration of **Reinforcement Learning (RL)** into the system.

Objective
The goal of the RL approach is to enable the paddle agent to learn **optimal movement policies** by interacting with the environment, rather than relying on labeled data. This method allows the agent to improve over time through trial and error, receiving rewards based on its performance.

8.1 Environment and State Representation

To implement reinforcement learning, we first needed to define an environment where the agent could interact and learn from its experiences. In our case, we designed a Pong environment using the pygame library, where the agent controls a paddle and interacts with a moving ball.
The environment provides a **state at each time step**, which is a concise representation of the current situation of the game. This state is encoded as a 5-dimensional vector:

Ball X Position (ball_x) – The horizontal position of the ball, normalized by the width of the screen.

Ball Y Position (ball_y) – The vertical position of the ball, normalized by the height of the screen.

Ball X Velocity (ball_dx) – The horizontal velocity (speed and direction) of the ball, normalized by a constant factor.

Ball Y Velocity (ball_dy) – The vertical velocity of the ball, also normalized.

Paddle Y Position (paddle_y) – The vertical position of the paddle, normalized by the available movement range.

This normalized vector allows the agent to perceive relevant aspects of the game environment without being overwhelmed by unnecessary detail. The normalization ensures that all inputs lie in a similar numerical range, which stabilizes and accelerates the learning process.

8.2 Action Space

The agent operates in a discrete action space with three possible actions:

- **Action 0** – Do nothing (keep the paddle in its current position).
- **Action 1** – Move the paddle upward.
- **Action 2** – Move the paddle downward.

By evaluating these actions at each time step, the agent attempts to intercept the ball by moving the paddle appropriately. The action that yields the highest expected cumulative reward is selected based on the Q-values predicted by the DQN model.

This discrete and minimal action set makes learning more tractable, while still allowing the agent to master effective paddle movement.

8.3 Reward Design

Reward design plays a crucial role in reinforcement learning as it directly influences the agent's learning behavior. In our Pong-based environment, we crafted a simple yet effective reward strategy to encourage desirable paddle movements.

Reward Scheme:

During the reinforcement learning training, we made minor but impactful changes to the reward function to optimize the agent's learning behavior. The table below outlines the updated reward design used in our environment:

Situation	Reward	Notes
Paddle tracks the ball (every step)	-0.01 * distance	**Reward shaping:** smaller penalty if paddle is close to ball center
Ball hits the paddle	+1.0	Good action; successful defense
Ball misses the paddle	-5.0	Big penalty; episode ends
Ball stuck (low vertical movement)	-2.0	Ends episode; avoids long, boring plays
Max steps exceeded	-20.0	Harsh penalty; forces learning efficient play
Ball hits top/bottom wall	0	Ball bounces; no reward logic
Ball hits right wall	0	No penalty or reward

This sparse and binary reward structure simplifies the learning signal and focuses the agent's attention on the most critical events—ball contact and misses. By using this feedback, the agent learns to position the paddle effectively to maximize future positive rewards (successful hits) and avoid negative outcomes (misses).

Why this reward design?

- **Simplicity**: A minimal reward scheme reduces noise in the learning signal and accelerates convergence.
- **Effectiveness**: The binary feedback is strongly aligned with the agent’s primary objective—hitting the ball.

Over time, through trial and error, the agent learns to associate certain paddle positions and movements with higher expected future rewards.

Early Stopping During Training:

To ensure efficient learning, the training episodes were designed to stop early if the agent failed to improve the reward consistently. This early termination helped avoid wasting time on unproductive runs and encouraged faster convergence of policy learning.

Adaptive Learning Environment:

The environment monitored reward trends over episodes, and if no significant improvement was detected over a defined window, the episode was stopped automatically to reinitialize learning conditions.

8.4 DQN Architecture

To implement reinforcement learning for paddle control, we adopted the Deep Q-Network (DQN) algorithm. DQN combines Q-learning with deep neural networks to approximate the Q-value function, enabling the agent to select actions based on learned value estimates of state-action pairs.

Network Input: Game State

The neural network receives a 5-dimensional input vector representing the current state of the environment:

1. Ball X Position (normalized): ball_x / WIDTH
2. Ball Y Position (normalized): ball_y / HEIGHT
3. Ball X Velocity (normalized): ball_dx / max_speed
4. Ball Y Velocity (normalized): ball_dy / max_speed
5. Paddle Y Position (normalized): paddle_y / (HEIGHT - PADDLE_HEIGHT)

This state encapsulates the most relevant spatial and dynamic features for predicting paddle movements.

Network Output: Q-Values

The output of the network is a 3-dimensional vector, where each value represents the predicted Q-value (expected cumulative reward) for one of the possible discrete actions:

• 0: Stay in place

• 1: Move up

• 2: Move down

The agent selects the action with the highest Q-value.

Model Architecture

The DQN model is implemented using PyTorch and consists of the following layers:

```
# ----------------------------
# DQN Model Definition
# ----------------------------
class DQN(nn.Module):
    def __init__(self, state_size, action_size):
        super(DQN, self).__init__()
        self.fc = nn.Sequential(
            nn.Linear(state_size, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, action_size)
        )
    def forward(self, x):
        return self.fc(x)
```

Input Layer: A fully connected layer with 128 neurons, taking a 5-dimensional state vector.

- **Hidden Layers:** Two hidden layers, each with 128 neurons and ReLU activation, providing non-linearity and enabling the network to learn complex representations.
- **Output Layer:** A fully connected layer with 3 output neurons (one per action), providing Q-value estimates for each possible action.

Training Objective

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

$$Q_{target} = r + \gamma \cdot \max_{a} Q(s', a)$$

Where:

- r is the reward received after taking an action,
- γ is the discount factor,
- s′ is the next state.

Model Inference During Gameplay

During gameplay (`game_rl.py`), the trained model is loaded and used to predict the best action in real-time:

```python
def predict_action(ball_x, ball_y, ball_dx, ball_dy, paddle_y):
    # Normalize input just like in training
    state = np.array([
        ball_x / WIDTH,
        ball_y / HEIGHT,
        ball_dx / 5,
        ball_dy / 5,
        paddle_y / (HEIGHT - PADDLE_HEIGHT)
    ], dtype=np.float32)

    with torch.no_grad():
        q_values = model(torch.tensor(state).unsqueeze(0))
        action = q_values.argmax().item()
    return action
```

This enables the paddle agent to respond to the ball's movement by choosing actions that maximize expected future rewards, even in previously unseen situations.

8.5 Training Loop and Results

The training process for our reinforcement learning agent is based on the **Deep Q-Learning algorithm**, which iteratively improves the agent's performance by interacting with the game environment. The key objective is to train the paddle agent to **maximize long-term rewards** through trial and error using feedback from the environment.

The training loop implements core reinforcement learning mechanisms using Deep Q-Learning. The agent interacts with a custom Pong environment and learns using experience replay and a target network. Key steps in the loop include:

State Representation: The input state vector contains normalized values of ball position, ball velocity, and paddle position.

Action Selection: Actions are chosen using an epsilon-greedy strategy, where epsilon decays gradually from 1.0 to 0.05.

Reward Function: Rewards are shaped to encourage the paddle to align with the ball and heavily penalize missed hits.

Experience Replay: Transitions are stored in a replay buffer, and mini-batches are sampled to break correlation between experiences.

Optimization: The policy network is updated using the mean squared error (MSE) between the predicted and target Q-values. The target Q-value is computed using a separate target network.

Target Network Updates: The target network is periodically updated with the weights of the policy network every 1000 steps.

Termination Conditions: Episodes terminate when the ball is missed, stuck, or exceeds a maximum step limit. Training stops after 50 episodes.

Logging: Episode reward, average reward over the last 100 episodes, and epsilon value are logged per episode.

Model Saving: The final trained model is saved to disk as pong_rl_policy.pth.

This is how our Reinforcement model worked after. It is now working well-scaled

9. Challenges Faced

Supervised Learning (SL)

Dataset Creation from Scratch:
No pre-existing dataset for paddle movement was available, so we had to generate our own labeled data by manually playing the game. This process was time-consuming and error-prone.

Manual Labeling Biases:
The paddle movement labels were based on human reflexes, which sometimes introduced inconsistent labels, especially near boundaries, affecting model generalization.

Feature Engineering:
Designing relevant features (like ball coordinates, direction, and paddle position) required multiple iterations to maintain a balance between simplicity and predictive power.

Model Selection and Tuning:
Choosing the right model (e.g., KNN) and fine-tuning parameters such as n_neighbors involved extensive experimentation using cross-validation and grid search.

Real-Time Integration:
Translating the prediction outputs into live paddle movement in PyGame was non-trivial, as it required synchronization between game frames and model inference cycles.

Reinforcement Learning (RL)

Episode Stability & Training Time:
Training the RL agent required many episodes to see meaningful improvement. Early episodes were unstable and often ended prematurely due to poor exploration or getting stuck.

Exploration vs. Exploitation:
Tuning the ε-greedy strategy was difficult. Too much exploration caused random behavior, while too little led to early convergence to suboptimal policies.

Sparse and Delayed Rewards:
Rewards were sparse (mostly negative), and good performance (e.g., hitting the ball) was rare in early training, making credit assignment difficult.

Stuck Ball Dynamics:
The ball sometimes moved horizontally with almost no vertical component, leading to infinitely long episodes. This required intervention to reset ball_dy during training.

Hyperparameter Sensitivity:
RL performance was highly sensitive to parameters like gamma, batch_size, learning rate, and update frequency, requiring manual fine-tuning through trial and error.

No Opponent Paddle:
Since the environment lacked an opponent, the RL agent's exposure to game-like scenarios was limited, reducing the complexity of learned strategies.

10. Comparison Between Supervised and Reinforcement Learning

Aspect	Supervised Learning (SL)	Reinforcement Learning (RL)
Training Input	Dataset created from manually played games (ball & paddle positions)	Environment simulation with paddle receiving reward signals
Best Model Performance	KNN with n_neighbors=7, ~85% accuracy on test set	DQN agent learned paddle control through 50 episodes
Behavior in Game	Paddle mimicked human behavior, sometimes missed fast balls	Paddle learned to hit balls more strategically and recover from edge cases
Generalization	Failed on ball trajectories not present in dataset	Handled unseen scenarios better after training
Training Time	Data collection + model training was quick	Took longer due to episode simulations and tuning
Integration	Simple .predict() logic; worked well with PyGame loop	Required tight integration with reward loop and action timing
Challenges Faced	Human error in labels, overfitting, biased samples	Reward shaping, tuning exploration/exploitation, unstable early episodes
Game Control Style	Imitation-based (reactionary)	Goal-driven (maximized ball bounce + survival)
Final Verdict	Worked well initially, limited in adapting to new dynamics	More robust in long run, capable of improving with more training

Which Performs Better?

While **SL (KNN)** gave reliable paddle control early on, **RL** is better suited for learning optimal strategies over time without requiring labeled data. Once trained, the RL agent can adapt and even outperform human-designed logic.

11. Conclusion

In this project, we explored two distinct AI approaches to control a paddle in a Ping Pong game: **Supervised Learning (SL)** and **Reinforcement Learning (RL)**. Each method had its own strengths and limitations. Initially, **Supervised Learning** enabled us to build a functional model quickly. Using manually collected gameplay data, we trained several classifiers with **KNN (n=7)** achieving the best performance (~85% accuracy). The SL model could imitate basic paddle movements, but it often failed to generalize to faster or unusual ball trajectories. Additionally, the dataset had human-labeling bias, which limited the model's robustness.

In contrast, the **Reinforcement Learning** approach took longer to set up and tune but eventually outperformed SL in real-time gameplay. The **DQN agent**, trained through over 50 episodes, learned to make more strategic paddle movements. It adapted to fast balls, edge cases, and optimized its behavior through rewards. RL's biggest strength was its ability to learn from experience and improve over time something that SL couldn't offer in a static dataset setting.

From our in-game comparison, **RL provided a smarter and more adaptable control strategy**, especially in scenarios where the ball's behavior changed dynamically. While SL offered a faster path to a working AI, RL proved to be the more scalable and intelligent approach.

12. References

scikit-learn – Machine Learning in Python.
Pedregosa et al., Journal of Machine Learning Research, 12, pp. 2825–2830, 2011.
Available at: https://scikit-learn.org/

PyTorch – An open-source machine learning library for Python.
Available at: https://pytorch.org/

PyGame – A cross-platform set of Python modules designed for writing video games.
Available at: https://www.pygame.org/docs/

NumPy – Fundamental package for scientific computing with Python.
Available at: https://numpy.org/

Matplotlib and Seaborn – Python libraries for data visualization.
Matplotlib: https://matplotlib.org/
Seaborn: https://seaborn.pydata.org/

Pickle – Python module for object serialization.
Python Standard Library Documentation: https://docs.python.org/3/library/pickle.html

Stable-Baselines3 – Reliable implementations of reinforcement learning algorithms in PyTorch.
GitHub: https://github.com/DLR-RM/stable-baselines3

Deep Q-Learning with PyTorch – Tutorial on implementing DQN for games.
Source: Towards Data Science
Available at: https://towardsdatascience.com/deep-q-learning-with-pytorch-d1c86368024e