

51.512 REINFORCEMENT LEARNING FOR EMBODIED AI

Paddle Duel RL Arena

Agent design, training, and competition evaluation

AUTHOR	Siddhant Shrivastava
IDENTIFIER	1011241_Siddhant
DATE	14 August 2026

This report documents the five-level solution, the tabular and neural experiments, residual and DAgger pixel learners, the observation-only competition controller, and held-out results.

Contents

1	Executive summary	4
2	Task definition	4
2.1	Game rules	4
2.2	Action and observation spaces	5
2.3	Project levels	5
2.4	Required agent interface	5
3	System design	6
3.1	Repository structure	6
3.2	Runtime flow	6
3.3	Reproducibility tools	6
4	Tabular control	7
4.1	Q-table representation	7
4.2	Action selection	7
4.3	Update rules	7
4.4	Tabular tracking policy	7
5	Pixel value learning	8
5.1	Feature extraction	8
5.2	Double DQN architecture	8
5.3	Replay and target updates	9
5.4	Level 5 self-play	9
6	Predictive competition policy	9
6.1	Design reason	9
6.2	Side handling	9
6.3	Velocity and arrival time	10
6.4	Wall reflection	10
6.5	Shot placement	10
6.6	Deadband and movement cost	10
7	Residual Double DQN	11
8	Imitation: DAgger and Double DQN	11
8.1	Expert and state	11
8.2	Behavioral cloning	11
8.3	DAgger	11
8.4	TD fine-tune	11
8.5	Level 5	12
8.6	Held-out gate	12
9	Training setup	12
9.1	Seed separation	12
9.2	Tabular schedule	12
9.3	Pixel schedule	12
9.4	Saved artifacts	13
10	Evaluation	13
10.1	Protocol	13
10.2	Competition results	13
10.3	Termination analysis	14
10.4	Learned-policy baselines	14
11	Analysis	14
11.1	Why tabular learning was difficult	14
11.2	Why the feature DQN was difficult	14

11.3	Why predictive control worked	15
11.4	Fairness and information use	15
12	Limitations	15
12.1	Opponent coverage	15
12.2	Feature extraction	15
12.3	Motion model	15
12.4	Learned policy development	15
13	Testing and reproducibility	15
13.1	Automated tests	15
13.2	Recreate the environment	16
13.3	Train and evaluate	16
13.4	Watch the games	16
14	Conclusion	16

Executive summary

The project asks for agents that play a one-rally Pong game across five levels. The first three levels expose a discrete state. The last two levels expose RGB frames. Every submitted agent must return one of three actions: `UP`, `DOWN`, or `STAY`.

The work began with course methods:

- Q-learning, SARSA, and Expected SARSA for tabular control.
- Double DQN with replay and a target network for pixel features.
- Two-agent self-play for Level 5.

The trained policies established useful baselines. Their competition results were limited by sparse terminal rewards, movement penalties, and incomplete state coverage. The submitted policy uses the permitted observations directly. It tracks incoming balls in tabular levels. In pixel levels, it extracts object positions, estimates velocity from consecutive frames, predicts wall bounces, and selects an impact point that aims away from the opponent.

Two extra pixel learners were trained on the same observations. Residual Double DQN keeps the bounce model and learns the final action. DAgger plus Double DQN clones the predictive expert on stacked centroids, then fine-tunes with TD. A held-out gate promotes a learned checkpoint only if it matches or beats predictive control on Level 4 zoo and Level 5 tracking.

Final result

The pixel competition policy achieved **95% win rate against the Level 4 opponent zoo** and **100% against the Level 5 tracking benchmark** across 20 held-out seeds per setting. The policy does not read diagnostic values from the `info` dictionary. DAgger + Double DQN reached 75% zoo and 95% Level 5 tracking on the same gate. Residual Double DQN reached 10% and 70%. Neither checkpoint was promoted.

95%

Level 4 zoo win rate

100%

Level 5 tracking win rate

20

Held-out seeds per setting

The solution is packaged with `uv` for Python dependencies and `bun` for common commands. Training, evaluation, saved checkpoints, a metrics dashboard, a browser arena, a method explainer, and this report are included in the project directory.

Task definition

Game rules

Paddle Duel is a two-player rally. The ball starts near the center and travels toward the left player. An episode ends when a player scores or when the time limit is reached. The default limit is 60 simulated seconds at 30 frames per second, which gives 1,800 environment steps.

The environment uses a 640 by 360 pixel board. Each paddle moves 7 pixels per frame. The ball starts near 9.45 pixels per frame and accelerates during long rallies. The paddle is slower than the ball, so useful policies must begin moving before the ball reaches the paddle.

Event	Reward	Purpose
Move up or down	-0.001	Limits unnecessary motion
Stay	0	Allows low-cost positioning
Paddle hit, first 10	+0.06	Provides dense feedback
Paddle hit, later	+0.02	Rewards sustained defense
Score a point	+10	Defines the main objective

Event	Reward	Purpose
Timeout	No terminal bonus	Cumulative reward selects the winner

The movement penalty has a meaningful effect at timeout. A player can defend successfully and lose because it moved more than the opponent. The final controller stays still when the ball travels away from its paddle. This reduces the cost without weakening the next interception.

Action and observation spaces

The action set is:

$$\mathcal{A} = \{0 : \text{UP}, 1 : \text{DOWN}, 2 : \text{STAY}\}$$

Levels 1 to 3 use the state tuple:

```
(my_paddle_y_bin,
 other_paddle_y_bin,
 ball_x_bin,
 ball_y_bin,
 ball_vx_sign,
 ball_vy_sign,
 speed_bin)
```

Levels 4 and 5 use an RGB image of the play area. The score bar is excluded. The right-side image is mirrored by the environment. This lets one observation-only policy control either side.

Project levels

Level	Setting	Observation	Implemented method
1	Practice against a wall	Tabular	Q-learning, SARSA, tracking
2	Scripted opponent	Tabular	Expected SARSA, tracking
3	Opponent zoo	Tabular	Expected SARSA, tracking
4	Scripted pixel opponent	RGB image	Double DQN, DAgger, residual DQN, predictive control
5	Two controlled agents	RGB image	Self-play, DAgger, predictive control

Required agent interface

The submitted class follows the course interface. Training code and action selection are separate.

```
class Agent(BaseAgent):
    def reset(self) -> None:
        self._tabular.reset()
        self._pixel.reset()

    def select_action(self, observation, info=None, training=False) -> int:
        if isinstance(observation, tuple):
            return self._tabular.select_action(observation, training=False)
        return self._pixel.select_action(observation, training=False)
```

The competition path does not update a Q-table, replay buffer, optimizer, or network. Every call returns an integer in the valid action range.

System design

Repository structure

```
1011241_Siddhant/  
  student_agent.py  
  paddle_duel_env.py  
  paddle_duel/  
    agents/  
      controllers.py  
      tabular.py  
      dqn.py  
      residual_dqn.py  
      imitation.py  
    train/  
      tabular.py  
      dqn.py  
      imitation.py  
    cli.py  
    eval.py  
  play/  
  explainer/  
  artifacts/  
    checkpoints/  
    metrics/  
    dashboard/  
  notebooks/  
  report/  
  tests/  
  tools/  
    dashboard.ts  
    game_server.py
```

The environment is kept in the project root to match the starter notebooks. Reusable agent code lives under `paddle_duel/agents`. Training loops live under `paddle_duel/train`. The `student_agent.py` file is the competition entry point.

Runtime flow

Observation → **State or features** → **Policy** → **Action 0, 1, or 2**

For tabular observations, the state tuple goes to the tracking policy. For pixel observations, the frame goes through color-based object localization, velocity estimation, bounce prediction, impact planning, and action selection.

Reproducibility tools

`uv` creates the Python environment from `pyproject.toml` and `uv.lock`. `bun` runs the common tasks listed in `package.json`.

```
uv sync  
bun run train:smoke  
bun run train  
bun run eval  
bun run train:imitation  
bun run eval:imitation  
bun run dashboard  
bun run games  
bun run report:pdf
```

The dashboard is written to `artifacts/dashboard/index.html`. The live game viewer runs at `http://127.0.0.1:8765`. The browser arena lives at `play/` and is also the Play tab on the project site.

Tabular control

Q-table representation

The tabular agent stores one three-element action-value vector for each visited state. New states use optimistic initialization:

```
def _ensure(self, state):
    if state not in self.Q:
        self.Q[state] = np.full(
            self.n_actions,
            self.optimistic_init,
            dtype=np.float32,
        )
```

The default parameters are:

Parameter	Value	Role
α	0.20	Step size
γ	0.99	Future reward discount
ϵ_{start}	0.90	Initial exploration
ϵ_{end}	0.05	Final exploration
Optimistic value	0.05	Encourages action trials

Action selection

Training uses an epsilon-greedy behavior policy. Evaluation uses greedy action selection with random tie breaking.

$$\pi(a|s) = \begin{cases} \frac{\epsilon}{|\mathcal{A}|} + \frac{1-\epsilon}{|\mathcal{A}^*|} & \text{if } a \in \mathcal{A}^* \\ \frac{\epsilon}{|\mathcal{A}|} & \text{otherwise} \end{cases}$$

Here, $\mathcal{A}^*$ is the set of actions tied for the highest Q-value.

Update rules

The implementation supports three one-step TD targets.

SARSA

$$y_t = r_{t+1} + \gamma Q(s_{t+1}, a_{t+1})$$

Q-learning

$$y_t = r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a')$$

Expected SARSA

$$y_t = r_{t+1} + \gamma \sum_{a'} \pi(a'|s_{t+1}) Q(s_{t+1}, a')$$

Each method applies:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[y_t - Q(s_t, a_t)]$$

Terminal transitions set the target to the immediate reward. This removes bootstrapping beyond the end of the rally.

Tabular tracking policy

The learned tables reached 20% win rate in Level 2 and Level 3 evaluation. A small observation-only tracking policy gave more reliable competition behavior. It reads the first, fourth, and fifth tuple entries.

```

my_y, _, _, ball_y, ball_vx, _, _ = map(float, observation)

if int(ball_vx) == 1:
    return ACTION_STAY

paddle_center = my_y + 1.5
if ball_y < paddle_center - 0.4:
    return ACTION_UP
if ball_y > paddle_center + 0.4:
    return ACTION_DOWN
return ACTION_STAY

```

The `ball_vx` test suppresses movement when the ball travels toward the other side. The center offset accounts for the paddle height within the discretized vertical bins.

Pixel value learning

Feature extraction

The supplied `extract_simple_pixel_features` helper locates the cyan paddle, red paddle, and yellow ball. It returns six normalized coordinates:

$$\varphi(o_t) = [x_{\text{left}}, y_{\text{left}}, x_{\text{right}}, y_{\text{right}}, x_{\text{ball}}, y_{\text{ball}}]$$

Four consecutive feature vectors form the network input. The resulting input dimension is 24. Frame history gives the network enough information to infer motion.

Double DQN architecture

The network is a two-layer multilayer perceptron:

```

self.net = nn.Sequential(
    nn.Linear(input_dim, 128),
    nn.ReLU(),
    nn.Linear(128, 128),
    nn.ReLU(),
    nn.Linear(128, 3),
)

```

The output has one Q-value for each action. Experience replay breaks short-term sample correlations. A target network supplies a slower bootstrap target.

Double DQN selects the next action with the online network and evaluates that action with the target network:

$$a^* = \arg \max_{a'} Q_{\text{online}}(s', a')$$

$$y = r + \gamma(1 - d)Q_{\text{target}}(s', a^*)$$

The done mask d removes future value at termination. The implementation uses Huber loss and clips gradient norm to 10.

```

next_actions = self.online(next_states).argmax(dim=1)
next_q = self.target(next_states).gather(
    1, next_actions.unsqueeze(1)
).squeeze(1)
targets = rewards + gamma * (1.0 - dones) * next_q

loss = F.smooth_l1_loss(q_sa, targets)
loss.backward()
nn.utils.clip_grad_norm_(self.online.parameters(), 10.0)

```

Replay and target updates

The replay buffer stores (state, action, reward, next_state, done) tuples. Training begins after 500 transitions. Mini-batches contain 64 transitions. The target network receives an online-network copy every 250 environment steps.

DQN setting	Value	Reason
Replay capacity	50,000	Retains varied rallies
Batch size	64	Stable CPU training
Warmup	500	Collects initial experience
Learning rate	10^{-3}	Adam optimizer
Target interval	250	Controls target drift
Frame stack	4	Supplies motion history
Hidden width	128	Fits the compact feature state

Level 5 self-play

Level 5 creates separate left and right agents. Each agent has its own replay buffer, optimizer, online network, and target network. Both agents learn from their side-specific reward.

```

a_l = left.select_action(obs_l, training=True)
a_r = right.select_action(obs_r, training=True)

next_l, next_r, r_l, r_r, terminated, truncated, info = \
    env.step_duel(a_l, a_r)

left.observe(a_l, r_l, next_l, terminated or truncated)
right.observe(a_r, r_r, next_r, terminated or truncated)

```

The Level 4 checkpoint can initialize both players. The self-play run used 400 episodes in the measured training pass. Its 25% win rate against the tracking benchmark showed that the learned policy needed a stronger motion model and more opponent diversity.

Predictive competition policy

Design reason

The pixel features contain precise object coordinates. Consecutive frames provide ball velocity. The court geometry is fixed. These properties support a short-horizon model that predicts where the ball will meet the controlled paddle.

The controller has four stages:

1. Locate both paddles and the ball.
2. Estimate velocity from the current and previous ball positions.
3. Predict the vertical intercept after wall reflections.
4. Move toward an impact point that sends the ball into open space.

This design uses only the RGB observation. The info argument is accepted to match the interface and is not read.

Side handling

The right-side observation is mirrored. The controlled paddle is therefore the paddle with the smaller normalized x-coordinate in either side's observation.

```

features = extract_simple_pixel_features(observation)
paddles = (features[:2], features[2:4])
own_paddle = min(paddles, key=lambda p: float(p[0]))
other_paddle = max(paddles, key=lambda p: float(p[0]))
ball = features[4:6]

```

This rule removes side-specific geometry from the policy.

Velocity and arrival time

Let the current ball position be $b_t = (x_t, y_t)$. The frame difference gives:

$$v_t = b_t - b_{t-1}$$

The policy acts only when $v_x < 0$ in its mirrored coordinate system. The estimated number of frames to the paddle is:

$$\tau = \max\left(0, \frac{x_t - x_p}{-v_x}\right)$$

The unreflected vertical prediction is:

$$\hat{y} = y_t + \tau v_y$$

The controller stores one previous ball position. The first action after reset is **STAY** because velocity is not available yet.

Wall reflection

The play area is normalized to $[0, 1]$. The ball radius creates a margin $m = 0.0235$. The valid vertical span is $h = 1 - 2m$. Repeated bounces can be folded into this span with a triangle-wave mapping:

$$z = (\hat{y} - m) \bmod(2h)$$

$$y_{\text{hit}} = \begin{cases} m + z & \text{if } z \leq h \\ m + 2h - z & \text{if } z > h. \end{cases}$$

The implementation is compact:

```
@staticmethod
def _reflect_y(y: float) -> float:
    ball_margin = 0.0235
    span = 1.0 - 2.0 * ball_margin
    folded = (y - ball_margin) % (2.0 * span)
    if folded > span:
        folded = 2.0 * span - folded
    return ball_margin + folded
```

This formula handles any number of top and bottom wall contacts before the ball reaches the paddle.

Shot placement

The game changes the outgoing ball angle according to the impact point on the paddle. The controller uses `aim_offset = 0.60`. It chooses the sign from the opponent's vertical position:

```
impact_offset = (
    -self.aim_offset
    if float(other_paddle[1]) > 0.5
    else self.aim_offset
)
target_y = intercept_y - impact_offset * 0.1275
```

An opponent in the lower half receives an upward shot. An opponent in the upper half receives a downward shot. The value `0.1275` is the normalized paddle half height.

Deadband and movement cost

The action controller uses a deadband of `0.012` around the target. The deadband prevents rapid action changes near the desired position.

```
error = target_y - float(own_paddle[1])
if error < -0.012:
    return ACTION_UP
if error > 0.012:
    return ACTION_DOWN
return ACTION_STAY
```

When the ball travels away, the policy returns **STAY**. This change improved the Level 4 weak-opponent win rate from 65% to 70% on the evaluation seed set. It also increased the score margin against tracking opponents because the agent paid fewer movement penalties.

Residual Double DQN

The bounce model already produces intercept error, time-to-arrival, velocity, and a predictive prior action. Residual Double DQN keeps that geometry and learns only the discrete action. The state is ten numbers:

$$s = \left[y_{\text{target}} - y_{\text{own}}, \frac{\tau}{60}, v_x, v_y, y_{\text{own}}, y_{\text{other}}, 1_{\text{approaching}}, \text{one-hot}(a_{\text{predictive}}) \right]$$

A 128-wide MLP outputs Q-values over {UP, DOWN, STAY}. Training uses Double DQN, replay, a target network, and a weak-to-tracking-to-zoo curriculum. Early episodes copy the predictive action with probability 0.55.

At evaluation the network is greedy. If the checkpoint is missing, the submission falls back to **PredictivePixelAgent**. Promotion requires matching or beating predictive win rates on Level 4 zoo and Level 5 tracking.

The measured residual gate (after a 120-episode curriculum pass) was 10% on Level 4 zoo and 70% on Level 5 tracking, against 90% and 100% for the predictive baseline on that seed protocol. The residual checkpoint was not promoted.

Imitation: DAgger and Double DQN

Feature Double DQN from random exploration reached 6.7% on the Level 4 zoo. The imitation stack uses the same 24-dimensional stacked centroids at test time. It does not run the bounce model when acting.

Expert and state

PredictivePixelAgent is the expert. Each RGB frame is converted to six normalized coordinates. Four frames are concatenated, so the learner sees positions and an implicit velocity. The expert action is stored beside each state.

Behavioral cloning

The Q-network is treated as a three-way classifier. Mini-batches from the expert buffer minimize cross-entropy:

$$L_{\text{BC}} = -\log \text{softmax}\left(Q_{\theta(s)}\right)_{a_E}$$

Level 4 collected 50 expert episodes (15,383 labeled states) and ran 1,200 Adam updates. Cross-entropy fell from about 0.85 to 0.50. A short greedy eval after cloning scored 20% on the zoo.

DAgger

Dataset aggregation follows Ross, Gordon, and Bagnell. On episode e the learner takes the expert action with probability β_e , otherwise an epsilon-greedy Q action. The expert still labels every visited state.

$$\beta_e = 0.80 + (0.10 - 0.80) \frac{e}{N_{\text{DAgger}} - 1}$$

Level 4 used 220 DAgger episodes. Zoo eval win rate moved from 0% at episode 80 to 70% at 160 and 80% at 220.

TD fine-tune

After DAgger, $\beta = 0$. The learner acts from Q-values. Each update mixes Huber Double DQN loss with a smaller cloning term:

$$L = L_{\text{TD}} + \lambda L_{\text{BC}}$$

λ is 0.75 during DAgger and 0.25 during the 350-episode RL stage. The Level 4 zoo eval peaked at 100% (episode 80 of RL) and finished at 75% on a 20-episode zoo check.

Level 5

The Level 4 weights initialize Level 5. The expert still labels the left paddle. The right paddle is a tracking agent with noise 0.00, then 0.05, then 0.10. The schedule was 30 expert duels, 600 cloning updates, 150 DAgger episodes, and 200 RL episodes. In-training eval vs tracking noise 0.05 stayed near 100% and finished at 95% on 20 held-out duels.

Held-out gate

The same 20-seed, 15-second protocol used for competition compared PredictivePixelAgent to ImitationDQNAgent loaded from level5_imitation_dqn.pt.

Setting	Predictive	DAgger + DQN
L4 zoo	95%	75%
L4 tracking	100%	95%
L4 strong	100%	100%
L5 tracking, noise 0.05	100%	95%

Imitation mean return on the zoo was 5.53 against 4.56 for predictive, with 11 scoring finishes and 2 losses. Predictive won 19 of 20 zoo rallies (9 scores, 11 timeouts). Promotion requires both zoo and Level 5 win rates to match or exceed the baseline. The gate wrote `promote: false` to `artifacts/metrics/imitation_promoted.json`. The submission therefore still dispatches image observations to PredictivePixelAgent.

Training setup

Seed separation

Training and evaluation use separate seed ranges. Training begins at the command-line seed, normally zero. Periodic evaluation uses offsets at 10,000, 20,000, or 30,000. Final evaluation uses offsets at 50,000 or above. Competition evaluation uses a separate held-out range.

This separation reduces the chance that a policy is selected from favorable training trajectories.

Tabular schedule

Level	Method	Episodes	Train limit	Opponent
1	Q-learning	2,000	12 s	Wall
1	SARSA baseline	1,500	12 s	Wall
2	Expected SARSA	2,500	12 s	Weak
3	Expected SARSA	3,000	12 s	Zoo

Epsilon follows exponential decay:

$$\varepsilon_e = \varepsilon_{\text{end}} + (\varepsilon_{\text{start}} - \varepsilon_{\text{end}}) \exp\left(-\frac{e}{k}\right)$$

The run used decay scale $k = 900$ for tabular agents.

Pixel schedule

Level	Method	Episodes	Train limit	Opponent
4	Feature Double DQN	800	8 s	Weak
4	Residual Double DQN	120	12 s	Weak / tracking / zoo
4	DAgger + Double DQN	50 demo + 220 + 350	12 s	Weak / tracking / zoo

Level	Method	Episodes	Train limit	Opponent
5	Two-agent self-play	400	8 s	Learned agent
5	DAgger + Double DQN	30 demo + 150 + 200	10 s	Tracking

The Level 4 run began with epsilon 1.0 and decayed toward 0.05. The Level 5 run began at 0.80 and used the same final value.

Saved artifacts

Tabular agents use compressed Python pickle files. Neural agents use PyTorch state dictionaries. The project stores:

- Level-specific Q-tables for Q-learning, SARSA, and Expected SARSA.
- Level 4 Double DQN weights.
- Level 4 residual Double DQN weights.
- Level 4 and Level 5 DAgger + Double DQN weights.
- Separate left and right Level 5 self-play weights.
- JSON histories, gate files, and evaluation summaries.
- The final observation-only policy in source form.

Evaluation

Protocol

The final competition test used:

- 20 held-out seeds per setting.
- A 15-second episode limit.
- Pixel observations at `render_scale=2`.
- Greedy deterministic actions.
- No parameter updates.
- No use of diagnostic `info` values.

Level 4 was evaluated against `random`, `weak`, `tracking`, `strong`, and the weighted opponent zoo. Level 5 was evaluated against tracking agents with noise levels 0.00, 0.05, and 0.10.

The reported metrics are win rate, mean return, mean score difference, mean episode length, and termination reason.

Competition results

Setting	Win rate	Mean return	Score difference	Mean steps
L4 random	100%	9.040	9.112	220.1
L4 weak	70%	6.034	5.994	316.9
L4 tracking	100%	0.077	0.074	450.0
L4 strong	100%	0.105	0.119	450.0
L4 zoo	95%	2.554	2.556	411.8
L5 tracking 0.00	100%	9.525	9.598	236.7
L5 tracking 0.05	100%	8.533	8.612	267.6
L5 tracking 0.10	100%	9.037	9.106	228.6



L5 tracking

100%

Termination analysis

Against the random opponent, the agent scored in 18 of 20 episodes. Two episodes reached timeout. Against the weak opponent, it scored in 12 episodes and reached timeout in eight.

All tracking and strong Level 4 episodes reached the 15-second limit. The agent won every timeout through a higher cumulative reward. Its efficient movement and hit rewards produced a positive score difference. In the opponent zoo, the agent won 19 of 20 episodes. Five wins came from scoring and the remaining wins came from timeout reward.

Level 5 produced larger score margins. The final agent won all 60 rallies across the three tracking noise settings.

Learned-policy baselines

Policy	Win rate	Mean return	Setting
Expected SARSA	20%	2.026	Level 2 weak
Expected SARSA	20%	1.996	Level 3 zoo
Feature Double DQN	6.7%	0.676	Level 4 zoo
Self-play Double DQN	25%	2.481	Level 5 tracking
Residual Double DQN	10%	0.840	Level 4 zoo
Residual Double DQN	70%	6.893	Level 5 tracking
Dagger + Double DQN	75%	5.527	Level 4 zoo
Dagger + Double DQN	95%	9.014	Level 5 tracking
Predictive controller	95%	2.554	Level 4 zoo
Predictive controller	100%	9.525	Level 5 tracking

The baseline results show the main development path. The Q-tables and neural agents implemented the course algorithms and produced reproducible saved models. Residual DQN kept the intercept features and still trailed the geometry controller. DAgger copied that controller onto a 24-D network and closed most of the gap. The predictive source policy remains the submitted pixel agent.

Analysis

Why tabular learning was difficult

The raw state space contains:

$$12 \times 12 \times 16 \times 12 \times 2 \times 3 \times 5 = 829,440$$

possible tuples. The Level 3 run visited roughly 84,100 states. Sparse coverage limits generalization between nearby paddle and ball positions. A movement penalty also makes STAY attractive before the table learns the value of early interception.

Relative state features, tile coding, eligibility traces, or a curriculum that loads Level 1 values into Level 2 and Level 3 could improve sample efficiency.

Why the feature DQN was difficult

The Level 4 DQN learned from 800 short episodes. Its terminal score reward was rare during early training. The six centroid features contained enough information for control, yet the network had to learn velocity, wall reflection, and paddle geometry from sampled transitions.

The 6.7% zoo win rate shows that the training budget and curriculum were insufficient. Training against the weak opponent also gave limited exposure to the full zoo distribution.

Later runs addressed those gaps. Residual Double DQN used a weak, tracking, and zoo curriculum on intercept features. DAgger cloned PredictivePixelAgent on the 24-D stack, then fine-tuned with Double DQN. Imitation reached 75% zoo and 95% Level 5 tracking. Residual reached 10% and 70%. Remaining

gaps versus predictive control include expert-student mismatch after β decays, the movement penalty under greedy Q, and the 20-seed gate requiring a match on both settings.

Why predictive control worked

The environment has regular motion, visible objects, fixed wall geometry, and a small action set. Two frames identify direction and speed. The bounce model maps that estimate to an intercept. The action controller then solves a local positioning problem.

The policy also handles the reward function directly. It stays still during the away phase, earns hit rewards through accurate defense, and selects an impact offset that creates scoring chances.

Fairness and information use

The pixel controller calls `extract_simple_pixel_features(observation)` and stores the previous ball coordinate. It does not access `info`. The tabular controller reads only the supplied tuple. Tests verify that altered `info` values do not change the pixel action.

This satisfies the competition interface and supports right-side play through the environment's mirrored observation.

Limitations

Opponent coverage

The competition test includes all built-in scripted opponents and three tracking noise levels. Student agents may use policies outside this set. A policy that predicts the chosen impact offset could reduce the scoring rate. The defensive intercept remains useful in that case.

Feature extraction

The pixel locator depends on the environment's cyan, red, and yellow colors. Changes to rendering colors or severe image compression would require updated masks. The official environment keeps these colors fixed.

Motion model

Velocity comes from a one-frame difference. A missing ball detection can create a bad estimate for one step. A short median filter or confidence check could reduce this risk.

The bounce prediction assumes the observed velocity remains constant until the next paddle contact. The environment gradually changes speed. The direction remains stable during free flight, so the intercept error stays small in the tested horizon.

Learned policy development

The submitted pixel controller is source-based and needs no checkpoint. Residual and imitation checkpoints remain loadable. Promotion is a JSON gate on held-out zoo and Level 5 tracking win rates. Longer DAgger, checkpoint selection by that gate, and a lower TD weight late in training are the obvious next steps for the 24-D policy.

Testing and reproducibility

Automated tests

The project includes tests for:

- Valid actions under tabular and pixel observations.
- Pixel action independence from `info`.
- A minimum win rate on a fixed Level 4 zoo seed set.
- Imitation agent save/load and a finite behavioral-cloning loss.

Run them with:

```
uv run python -m unittest discover -s tests -v
```

The final test run completed all three tests successfully.

Recreate the environment

```
uv sync
```

This command installs NumPy, Matplotlib, Pillow, Pygame, IPython, and PyTorch from the lockfile.

Train and evaluate

```
# Short check
bun run train:smoke

# Full configured run
bun run train

# Competition evaluation
bun run eval

# DAgger + Double DQN
bun run train:imitation
bun run eval:imitation

# Metrics page
bun run dashboard
```

Raw competition results are stored in:

```
artifacts/metrics/competition_eval.json
artifacts/metrics/imitation_vs_predictive.json
artifacts/metrics/residual_vs_predictive.json
```

Watch the games

```
bun run games
```

Open <http://127.0.0.1:8765>. The viewer supports all five levels, opponent selection, playback speed, and starting seed.

The browser arena at `play/` uses the same physics and rewards as `paddle_duel_env.py`. On the project site it is the Play tab. Opponents include wall, random, weak, tracking, strong, tabular tracking, predictive control, and the opponent zoo. Points to win defaults to 5. Setting it to 1 matches the course first-point rule.

Conclusion

The project implements the full path from tabular TD control to neural value learning, two-agent self-play, residual Double DQN, and DAgger plus Double DQN. These experiments exposed the main control challenges: sparse score rewards, movement cost, partial observability, large state coverage, and opponent variation.

The submitted policy uses frame memory and a compact court model. It observes object positions, predicts the paddle intercept, chooses a shot direction, and limits movement during the away phase. This policy achieved 95% against the Level 4 zoo and 100% against the Level 5 tracking benchmark on the measured held-out seeds. DAgger + Double DQN reached 75% and 95% on the same gate.

The submission is reproducible, uses the required interface, avoids diagnostic information during action selection, and includes training code, checkpoints, metrics, tests, a dashboard, a browser arena, and a live game viewer.