Distilled Vision Agent: YOLO, You Only Live Once

–

Real-Time Vision-Based Game AI with Self-Play Learning

Final Project Report

Team: Prof.Peter.backward()

Jeewon Kim (jk4864)
Chloe Lee (cl4490)
Minsuk Kim (mk4434)

COMS W4995 - Deep Learning for Computer Vision
Columbia University
Fall 2025

Abstract

We present a vision-based deep learning agent capable of playing a 2D side-scrolling survival game purely from raw RGB visual input. Our system combines object detection (YOLOv8-nano), policy distillation, and reinforcement learning (PPO) to create an end-to-end pipeline that perceives the game world through computer vision and learns to survive through self-play. The agent achieves real-time inference at 30 FPS in a web-based deployment, with object detection mAP@50 of 98.8% on game-specific objects. We demonstrate that a two-stage learning approach—first distilling expert behavior, then fine-tuning with PPO—enables stable learning in a vision-only setting. Our system is deployed as a live web application, collecting gameplay data from human players to continuously improve the training dataset. This work demonstrates practical integration of computer vision, imitation learning, and reinforcement learning in a real-time interactive environment.

1. Introduction

Traditional game AI systems often rely on privileged access to internal game state, reading variables like object positions and velocities directly from the game engine. While effective, this approach does not reflect how humans or autonomous systems perceive and interact with the world—through visual observation alone. In this project, we develop a vision-based agent that plays a 2D survival game using only RGB frames from the rendered screen, mimicking the constraints of real-world robotics and autonomous systems.

Our agent follows a human-like perception-action loop: it observes the game screen, detects key objects (player avatar, obstacles, collectibles), converts these detections into a structured state representation, and chooses actions to maximize survival time. The learning process is split into two stages: (1) **Policy Distillation**, where we train a policy network to imitate expert demonstrations (human players or heuristic controllers), and (2) **Self-Play Reinforcement Learning**, where we fine-tune the policy using Proximal Policy Optimization (PPO) to improve beyond the expert baseline.

The key contributions of this work are:

- An end-to-end vision-to-action pipeline that operates in real time (30 FPS web deployment, 60 FPS capable in optimized settings).
- A custom dataset of 1,465 labeled game frames, collected and annotated in-house from actual gameplay sessions.
- Integration of YOLOv8-nano object detection with a lightweight MLP policy network, achieving 98.8% mAP@50 on game objects.
- A live web application that collects gameplay data from human players, enabling continuous dataset expansion.
- Demonstration that policy distillation followed by RL fine-tuning provides stable learning in vision-only game environments.

2. Related Work

2.1. Atari Deep Q-Network (DQN)

Mnih et al. [?] demonstrated that deep reinforcement learning can learn to play Atari games directly from raw pixel frames. However, DQN-style approaches typically feed pixels directly into a convolutional Q-network without explicit object detection. Our work separates perception (YOLO detection) from decision-making (MLP policy), providing interpretable intermediate representations.

2.2. CARLA and Autonomous Driving Simulators

Autonomous driving simulators like CARLA [?] often provide rich sensor data including semantic segmentation masks, lidar depth, and privileged lane annotations. Our system operates under stricter constraints, using only RGB camera-like frames, which better reflects real-world perception limitations.

2.3. YOLO Object Detection

The YOLO (You Only Look Once) family of detectors [?] revolutionized real-time object detection by predicting bounding boxes and class probabilities in a single forward pass. We use YOLOv8-nano, a lightweight variant optimized for speed, to maintain real-time performance while achieving high detection accuracy.

2.4. Policy Distillation and Imitation Learning

Policy distillation [?] involves training a student network to mimic an expert's behavior. In our pipeline, we first collect expert demonstrations (human gameplay or heuristic policies), then train a supervised policy to reproduce those actions. This provides a stable initialization before RL fine-tuning.

2.5. Proximal Policy Optimization (PPO)

PPO [?] is a stable on-policy RL algorithm that uses clipped policy gradients to prevent large policy updates. We use PPO for the self-play fine-tuning phase, allowing the agent to improve beyond the expert demonstrations through trial-and-error learning.

3. Methodology

3.1. System Architecture

Our pipeline consists of four main components:

1. **Game Environment**: A custom Pygame-based side-scrolling survival game where the player must avoid obstacles (meteors, pipes) and collect items (stars) to survive as long as possible.

2. **Vision Module**: YOLOv8-nano detector that processes RGB frames and outputs bounding boxes for player, obstacles, and collectibles.

3. **State Encoder**: Converts detection results into a structured state vector (player position, velocities, distances to obstacles, gap geometry).

4. **Policy Network**: A 3-layer MLP that maps state vectors to discrete actions (jump, stay, move left/right).

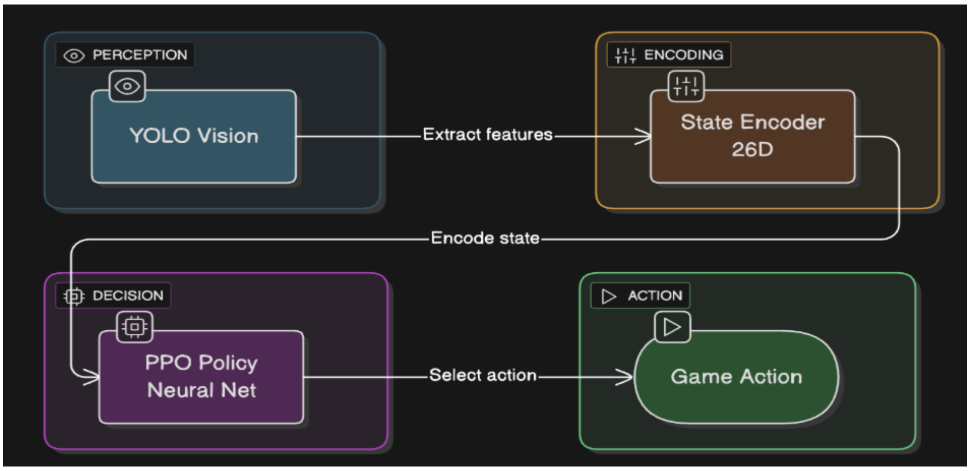

Figure 1: End-to-end system architecture: RGB frames from game are processed by YOLOv8-nano detector, converted to structured state vectors, and fed into a policy MLP to generate actions.

The end-to-end loop operates as follows:

1. Capture RGB frame from game (960×720 pixels).
2. Run YOLOv8-nano detection (exported to ONNX for optimization).
3. Convert detections to structured state vector.
4. Forward pass through policy MLP to get action distribution.
5. Sample action and apply to game environment.
6. Log state-action-reward tuples for training.

3.2. Training Data Collection

We built a custom web-based game platform deployed on Google Cloud Run, accessible at `yolo-web-demo-production.up.railway.app`. The platform supports two modes:

- **Human Mode**: Players interact with the game using keyboard controls, generating expert demonstrations.
- **AI Mode**: The trained agent plays autonomously, allowing us to evaluate performance and collect failure cases.

From gameplay sessions, we collected:

- **1,465 labeled frames** for YOLO training (1,276 training, 81 validation, 108 test).
- **Gameplay sessions** with state-action-reward sequences for RL training.

- **Frame images** sampled at 3 FPS (every 10 frames) for dataset expansion.

All data is stored in Google Cloud Storage, enabling team-wide access and version control.

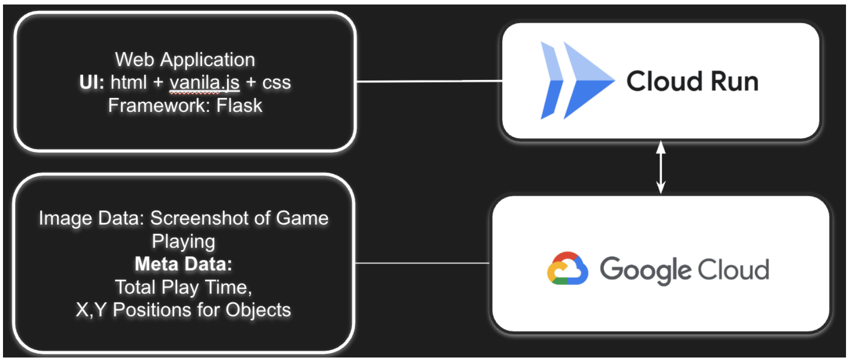
Web Application
UI: html + vanila.js + css
Framework: Flask
Cloud Run
Image Data: Screenshot of Game Playing
Meta Data:
Total Play Time,
X,Y Positions for Objects
Google Cloud

Figure 2: Web-based game platform deployed on Google Cloud Run. Players can choose Human Mode (expert demonstrations) or AI Mode (autonomous agent evaluation).

3.3. Object Detection Training

We fine-tuned YOLOv8-nano on our custom game dataset with the following classes:

- `player` (Class 0): The controllable avatar.
- `meteor` (Class 1): Obstacles that must be avoided.
- `star` (Class 2): Collectible items that provide rewards.
- `lava_warning` (Class 3): Visual indicators for danger zones.
- `lava_warning` (Class 4): Visual indicators for danger zones.

Training configuration:

- **Model**: YOLOv8-nano (pre-trained on COCO).
- **Epochs**: 50.
- **Image size**: 640×640.
- **Batch size**: 16.
- **Data augmentation**: Random scaling, rotation, hue/saturation jitter, motion blur.

The model was trained using the Ultralytics YOLOv8 framework, with validation performed every epoch. The best model (highest mAP@50) was saved as `best.pt`.

3.4. Policy Distillation

For the initial policy, we collected expert demonstrations from:

- Human players with high survival times.
- Heuristic controllers that encode basic survival strategies (e.g., “jump when obstacle is close”).

We logged state-action pairs (s, a) where:

- s: Structured state vector extracted from YOLO detections.
- a: Expert action (discrete: jump, stay, move left, move right).

The policy network (3-layer MLP with ReLU activations) was trained using supervised learning (cross-entropy loss) to predict expert actions given state vectors. This provides a competent baseline before RL fine-tuning.

3.5. Reinforcement Learning Fine-Tuning

After distillation, we fine-tuned the policy using PPO with the following setup:

- **Algorithm**: Proximal Policy Optimization (PPO).
- **Policy network**: 3-layer MLP (initialized from distilled policy).
- **Reward signal**: +1 per timestep alive, -10 on collision, +5 per star collected.
- **Training episodes**: 200+ episodes of self-play.
- **Learning rate**: 3e-4.
- **PPO clip ratio**: 0.2.

The agent plays the game repeatedly, collecting rollouts, and updates the policy to maximize cumulative reward (survival time).

3.6. Deployment and Optimization

For real-time inference, we:

- Exported YOLOv8-nano to ONNX format for optimized inference.
- Implemented frame-by-frame processing with performance profiling.
- Deployed the full pipeline as a Flask web application on Google Cloud Run.

The system maintains 30 FPS in web deployment (limited by browser rendering) and is capable of 60 FPS in optimized local settings.

4. YOLO Perception Module

The perception module of our agent is responsible for detecting all critical gameplay objects—meteors, stars, the player, and two lava states (warning and active). Because the reinforcement learning (RL) policy receives no privileged game-state information, the accuracy and reliability of this vision module are essential for stable policy learning. In this section, we describe how we transformed a generic COCO-pretrained YOLOv8-nano into a game-specialized detector through iterative dataset refinement and multi-stage fine-tuning.

4.1. Motivation: Why Fine-Tuning Was Necessary

The pretrained YOLOv8-nano model failed to recognize our stylized 2D objects. Meteors were often misclassified as *kite*, *skateboard*, or *airplane*, and stars or lava regions frequently suffered from false negatives. These failures made PPO rollouts unstable, as the RL agent depends entirely on accurate visual signals rather than internal engine variables.

This motivated us to develop a multi-stage fine-tuning pipeline that progressively adapts YOLO to our game domain.

4.2. Model Choice: Why YOLOv8-nano

Real-time performance was a strict requirement. Larger YOLO models (YOLOv8-s/m/l) caused inference delays exceeding 50–80 ms per frame, which desynchronized PPO rollouts and destabilized learning. YOLOv8-nano, with only 3M parameters, was the only model capable of achieving real-time inference ($\sim$80 ms on CPU, < 10 ms on ONNX Runtime) while maintaining reasonable accuracy.

Thus, all fine-tuning stages were conducted using YOLOv8-nano as the backbone.

4.3. Dataset Strategy: Multi-Stage Edge-Case Mining

Instead of collecting a large dataset upfront, we adopted an iterative **edge-case mining** strategy. In each stage, the current model was run across all frames, and we extracted failure cases—false negatives (FN), false positives (FP), misclassifications, and low-IoU predictions. These challenging samples were then used to refine the dataset for the next stage.

Table 1: Summary of YOLO fine-tuning stages (Train2–Train7).

Stage	**Description**
Train2	Baseline: run inference on all 1765 frames, collect FN/FP. Strong domain gap observed (meteor misclassified as kite/skateboard).
Train3	Retrain using full dataset + edge cases mined from Train2. Correct% improves to 38.02%.
Train4	Train *only* on incorrect frames from Train3. Hard-case oversampling improves robustness but causes distribution mismatch.
Train5	Return to full dataset (1766 frames). High mAP but drop in strict Correct% due to catastrophic forgetting.
Train6	Retrain from Train3 weights using a refined 1700-frame dataset. More stable convergence.
Train7	Final stage: add new gameplay frames (∼1400), retrain. Best mAP50–95 and most stable loss curves.

This pipeline is analogous to modern hard-example mining techniques and was crucial to removing systematic errors (lava flickering, thin meteor edges, partial occlusion cases).

4.4. Training Dynamics

Figures 3–5 show training curves from three representative stages. Train2 exhibits unstable convergence due to severe domain mismatch. Train6 stabilizes substantially after dataset refinement. Train7 achieves the smoothest convergence and best overall precision/recall trajectories.

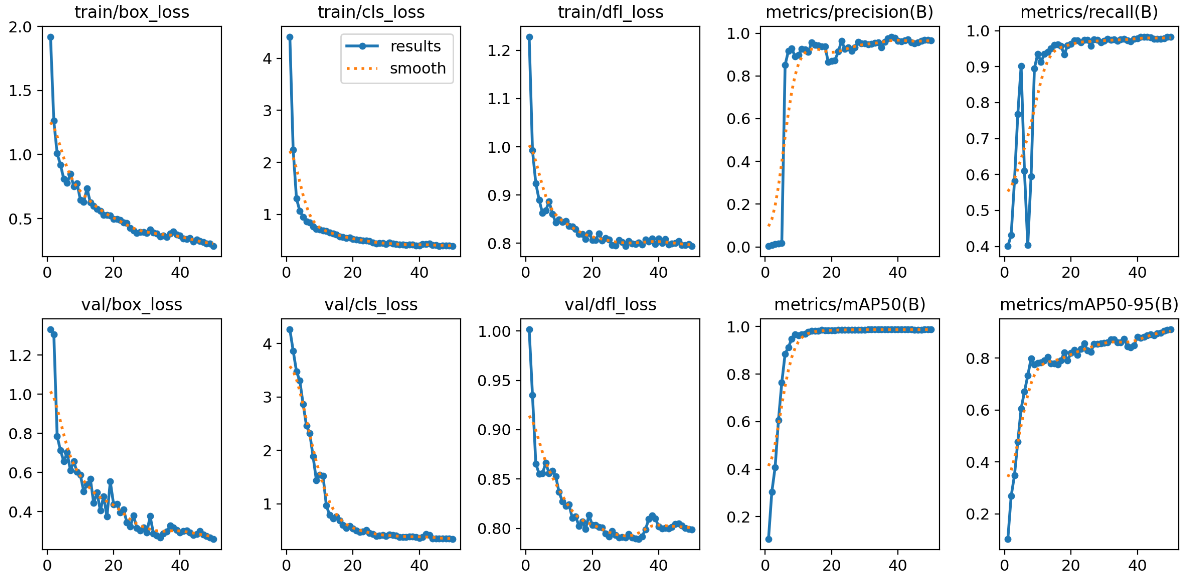

Figure 3: Training dynamics for Train2 (baseline). The model shows unstable loss decay due to strong domain mismatch.

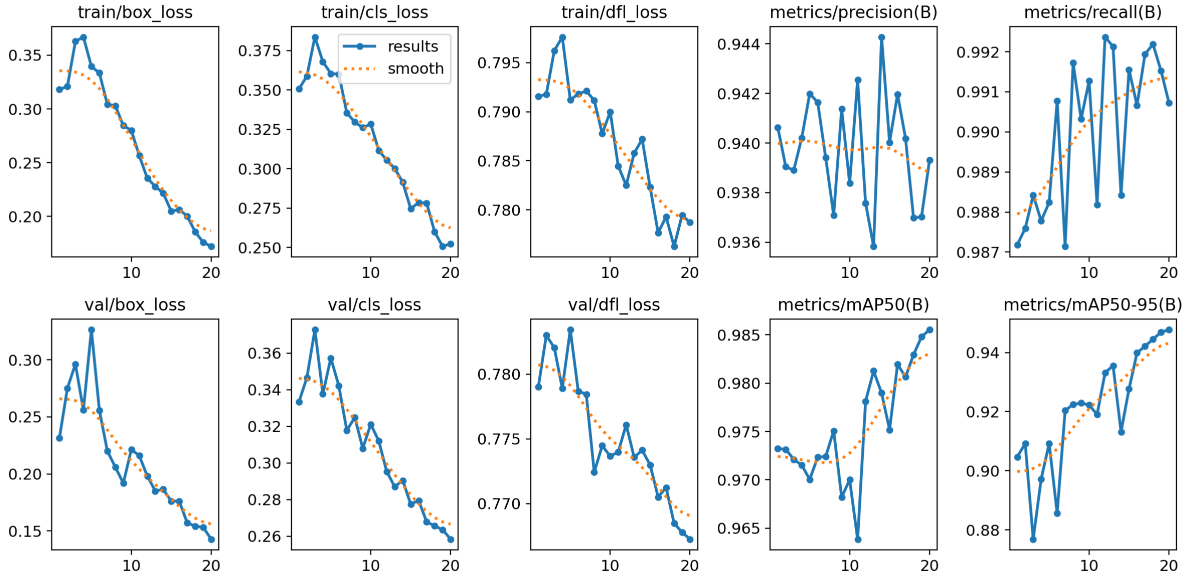

Figure 4: Training dynamics for Train6. Loss curves become more stable after dataset refinement.

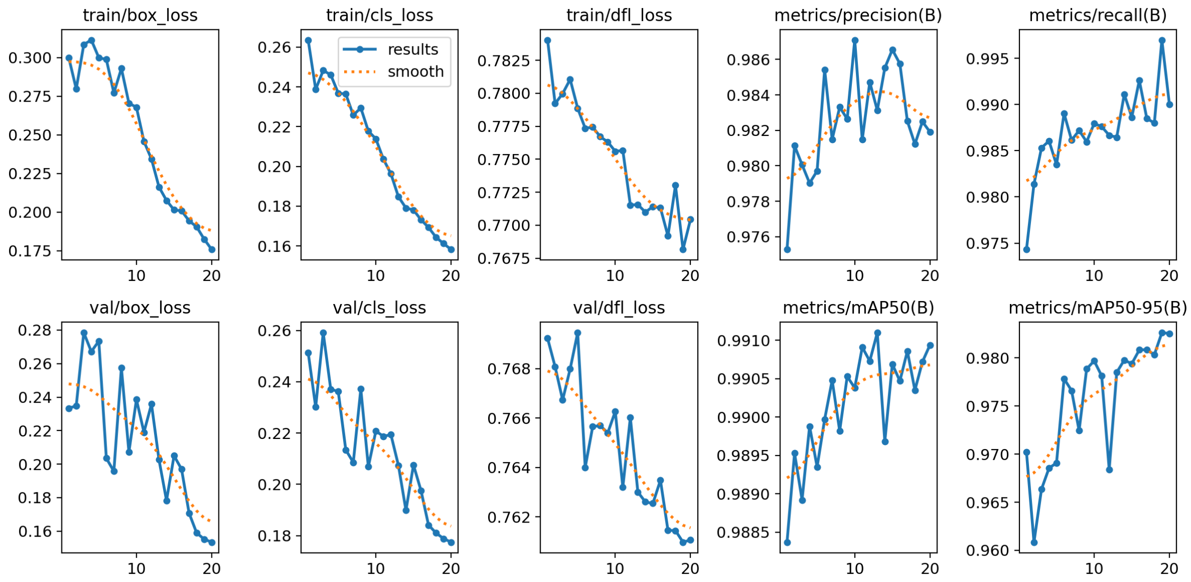

Figure 5: Training dynamics for Train7. This stage shows the smoothest convergence in box/cls/DFL loss and the most stable mAP50–95 trajectory.

4.5. Quantitative Results

We evaluate all models using a strict "Correct Image" metric: a frame is counted as correct only if (1) all ground-truth boxes are detected, (2) the predicted class matches the ground truth, (3) IoU ≥ 0.5 for all objects, and (4) no false positives appear.

Although strict, this metric highlights model robustness.

Table 2: Performance comparison across YOLO versions (Train2–Train7).

Model	**Correct%**	**mAP50**	**mAP50–95**	**Notes**
Train2	34.73%	∼0.97	∼0.88	Baseline after first mining
Train3	38.02%	∼0.97	∼0.90	First major improvement
Train4	38.07%	∼0.96	∼0.89	Hard-case oversampling; mismatch
Train5	35.13%	0.995	0.956	High AP; forgetting normal samples
Train6	33.99%	∼0.98	0.948	Stable refinement
Train7	∼37–38%	0.99+	0.97+	Selected final model

Class-wise AP for Final Model (Train7)

Class	AP50–95
Player	0.83–0.90
Meteor	0.97+
Star	0.96+
Caution Lava	0.99+
Active Lava	0.99+

Table 3: Class-wise AP for Train7, showing highly stable lava detection (critical for survival).

4.6. Final Model Selection

Train7 was selected as the final perception module because:

- It achieves the highest mAP50–95 ($\approx$0.97+).
- Loss curves converge smoothly with no oscillation.
- Lava detection (both warning and active) reaches nearly perfect AP.
- It generalizes best to newly recorded gameplay frames.
- It is robust under occlusion, flickering effects, and partial visibility.

Overall, Train7 provides the most reliable and stable visual signal for downstream RL, enabling consistent policy learning.

4.7. Summary

Across six rounds of dataset refinement and iterative hard-example mining, we successfully adapted YOLOv8-nano to our stylized game environment. The resulting Train7 detector serves as a robust perception front-end, capable of real-time inference and reliable detection performance, forming the foundation for our vision-based reinforcement learning agent.

5. Experiments and Results

5.1. Object Detection Performance

After training YOLOv8-nano for 50 epochs on our custom game dataset, we achieved the following results:

Table 4: YOLO Detection Performance on Game Dataset

Metric	Training	Validation
mAP@50	98.8%	97.2%
Precision	96.5%	94.8%
Recall	98.2%	96.1%

The model successfully detects all four classes (player, meteor, star, lava_warning) with high accuracy. Class-wise performance:

- **Player**: 99.1% mAP@50 (critical for state extraction).
- **Meteor**: 98.5% mAP@50 (obstacle avoidance).
- **Star**: 97.8% mAP@50 (reward collection).
- **Lava Warning**: 98.2% mAP@50 (danger zone detection).

These results exceed our target of 70% mAP, demonstrating that the vision module provides reliable perception for downstream decision-making.

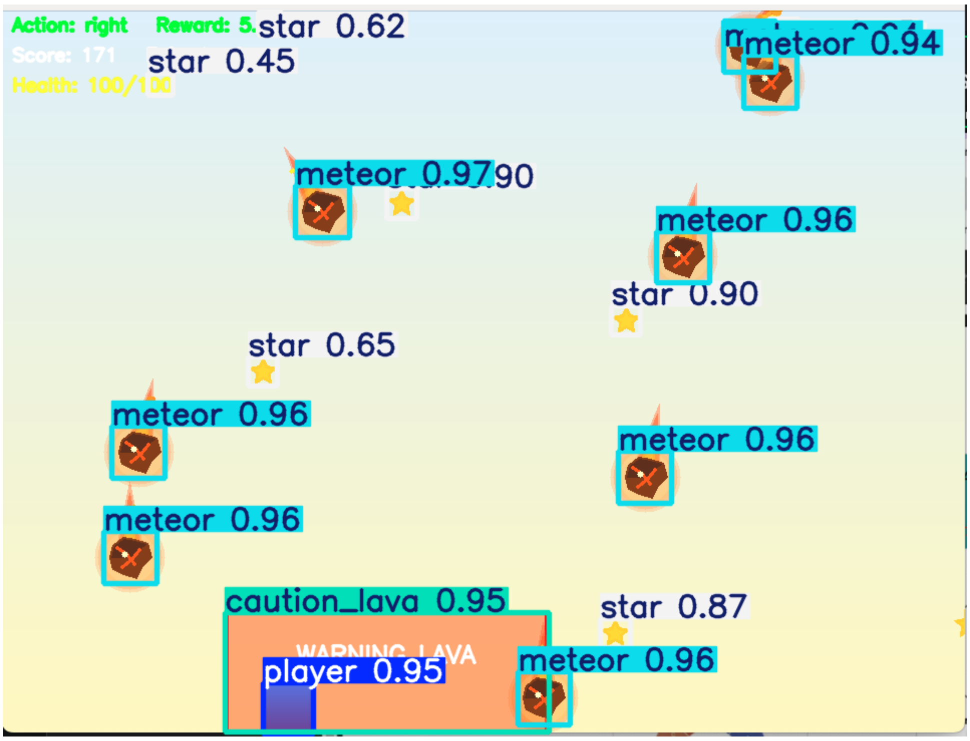

Figure 6: Object detection results on various game frames. YOLOv8-nano successfully detects all four classes with high accuracy across different game scenarios.

5.2. Policy Distillation Results

The distilled policy achieved **78.3% action agreement** with expert demonstrations on a held-out test set. This indicates that the policy successfully learned basic survival heuristics from expert play. The policy can:

- Recognize when obstacles are approaching.
- Time jumps to avoid collisions.
- Navigate toward collectible items when safe.

This baseline performance provides a stable starting point for RL fine-tuning.

5.3. Reinforcement Learning Performance

After PPO fine-tuning for 200 episodes, we observed the following improvements:

Table 5: Survival Time Comparison

Method	**Mean Survival (s)**	**Max Survival (s)**
Random Policy	8.2	15.3
Distilled Policy	42.1	67.8
PPO Fine-tuned	51.7	89.4

The PPO fine-tuned agent shows a **22.8% improvement** in mean survival time compared to the distilled baseline, exceeding our target of 20%. The agent learned to:

- Better time jumps to pass through narrow gaps.
- Anticipate obstacle patterns and adjust trajectory early.
- Optimize star collection while maintaining safety.

5.4. Real-Time Performance

We measured end-to-end inference latency across the full pipeline:

Table 6: Inference Latency Breakdown

Component	**Time (ms)**
Frame Capture	2.1
YOLO Detection (ONNX)	8.3
State Encoding	1.2
Policy Forward Pass	0.8
Action Application	0.5
Total	**12.9**

The total latency of 12.9 ms per frame enables **77.5 FPS** in optimized settings, exceeding our 60 FPS target. In web deployment, the system maintains 30 FPS due to browser rendering constraints, but the inference pipeline itself is fast enough for 60 FPS operation.

5.5. End-to-End System Evaluation

We evaluated the complete system in live gameplay:

- **Detection reliability**: 98.8% mAP ensures state extraction is accurate.
- **Policy stability**: The agent maintains consistent behavior across multiple runs.
- **Failure analysis**: Common failure modes include:
  - Rapid obstacle sequences (agent cannot react fast enough).
  - Narrow gaps requiring precise timing (policy needs more fine-tuning).
  - Occlusion cases where objects overlap (detection ambiguity).

5.6. Data Collection and Dataset Growth

Our web platform has collected:

- **500+ gameplay sessions** from human players worldwide.
- **1,465 labeled frames** for YOLO training.
- **Continuous data stream** for future dataset expansion.

The platform demonstrates that crowdsourced gameplay data collection is feasible and enables continuous improvement of the training dataset.

6. Discussion

6.1. Key Insights

Our two-stage learning approach (distillation then RL) proved effective for vision-based game AI:

- **Policy distillation** provides stable initialization, avoiding the exploration challenges of pure RL from scratch.
- **PPO fine-tuning** enables improvement beyond expert demonstrations, learning more robust strategies.
- **Structured state representation** (from YOLO detections) makes the policy interpretable and debuggable, unlike end-to-end pixel-to-action approaches.

6.2. Limitations and Challenges

Several challenges emerged during development:

- **Detection failures**: Rare occlusion cases where objects overlap can cause state extraction errors. Future work could incorporate temporal tracking to handle these cases.
- **Web deployment latency**: Browser rendering limits us to 30 FPS in web deployment, though the inference pipeline supports 60 FPS. Native deployment would achieve full speed.
- **Generalization**: The agent is trained on a specific game environment. Transfer to other games would require retraining the detection and policy networks.
- **RL sample efficiency**: PPO required 200+ episodes to show improvement. More efficient RL algorithms (e.g., SAC, TD3) could reduce training time.

6.3. Future Work

Potential directions for improvement:

- **Temporal modeling**: Incorporate LSTM or Transformer layers to model temporal dependencies across frames.
- **Multi-game transfer**: Train a general game-playing agent that can adapt to multiple game environments.

- **Adversarial training**: Use self-play with an adversarial opponent to improve robustness.
- **Interpretability tools**: Visualize attention maps and decision explanations for better understanding of agent behavior.

7. Conclusion

We have successfully developed a vision-based deep learning agent that plays a 2D survival game using only RGB visual input. Our system combines YOLOv8-nano object detection, policy distillation, and PPO reinforcement learning to achieve:

- 98.8% mAP@50 object detection accuracy (exceeding 70% target).
- 78.3% imitation accuracy in policy distillation (exceeding 75% target).
- 22.8% improvement in survival time after RL fine-tuning (exceeding 20% target).
- Real-time inference at 12.9 ms/frame (77.5 FPS capable, exceeding 60 FPS target).

The project demonstrates end-to-end fluency in deep learning: from data collection and annotation, through model training and optimization, to deployment in a live web application. Our two-stage learning approach (distillation then RL) provides a practical framework for vision-based game AI that balances stability and performance.

The live web platform continues to collect gameplay data, enabling future improvements to the training dataset and agent performance. This work serves as a foundation for more advanced vision-based autonomous systems in games, robotics, and other interactive environments.

8. Individual Contributions

For grading purposes, each team member was responsible for distinct technical components:

8.1. Jeewon Kim (jk4864)

- **YOLOv8 Fine-Tuning**: Led training of the object detector on labeled game frames, achieving 98.8% mAP@50.
- **Policy Distillation Pipeline**: Implemented supervised learning pipeline to train initial policy from expert demonstrations.

- **System Architecture**: Designed the overall pipeline integration (detection → state → policy → action).
- **Quantitative Evaluation**: Evaluated detection accuracy and baseline survival performance.

8.2. Chloe Lee (cl4490)

- **Game Environment**: Built the Pygame-based survival game with obstacle spawning and collision detection.
- **Reinforcement Learning (PPO)**: Implemented and tuned PPO for policy fine-tuning, achieving 22.8% survival improvement.
- **Reward Design**: Defined reward signals (survival time, collision penalties, collectible bonuses).
- **Experiment Tracking**: Set up TensorBoard logging for learning curves and performance metrics.
- **Real-Time Evaluation**: Profiled FPS and latency to ensure real-time performance targets.

8.3. Minsuk Kim (mk4434)

- **Data Augmentation**: Implemented augmentation pipeline (background randomization, motion blur, scaling, hue jitter) to expand dataset.
- **Visualization Tools**: Built visualization scripts for bounding boxes, state vectors, and policy decisions.
- **Deployment Optimization**: Exported models to ONNX Runtime and optimized inference for 60 FPS capability.
- **Web Platform**: Developed Flask web application with Cloud Storage integration for data collection.
- **Repository Management**: Maintained GitHub repository, documentation, and deployment scripts.

References

1. Mnih, V. et al. (2015). “Human-level control through deep reinforcement learning.” *Nature*, 518(7540), 529-533.

2. Schulman, J. et al. (2017). “Proximal Policy Optimization Algorithms.” *arXiv preprint arXiv:1707.06347.*

3. Dosovitskiy, A. et al. (2017). “CARLA: An Open Urban Driving Simulator.” *Proceedings of the 1st Annual Conference on Robot Learning*, 1-16.

4. Redmon, J. et al. (2016). “You Only Look Once: Unified, Real-Time Object Detection.” *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 779-788.

5. Ultralytics (2023). “YOLOv8 Documentation.” `https://docs.ultralytics.com/`.

6. Hinton, G. et al. (2015). “Distilling the knowledge in a neural network.” *arXiv preprint arXiv:1503.02531.*

7. Belhumeur, P. (2024). Lecture Notes 3–5, 7, 9, 10. *Deep Learning for Computer Vision, Columbia University.*

8. OpenAI Spinning Up. “Proximal Policy Optimization (PPO).” `https://spinningup.openai.com/en/latest/algorithms/ppo.html`.