Distilled Vision Agent: YOLO, You Only Live Once – Real-Time Vision-Based Game AI with Self-Play Learning

Project Proposal Report

Team: Prof.Peter.backward()

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

1. Description of the Objectives

The goal of this project is to develop a **vision-based deep learning agent** capable of playing a simple 2D mini-game (e.g., obstacle-avoidance, meteor-dodging, or Flappy Bird-style survival) **purely from raw visual input**. The agent will observe RGB frames from the game screen, detect key objects in real time, and choose actions using a lightweight control policy.

Unlike traditional rule-based control systems, where behaviors are hand-coded ("if obstacle ahead, then jump"), our agent learns to act using data-driven perception and self-improving decision policies. Our pipeline is intentionally designed to mimic how a human plays a game: *see what's happening, judge what's dangerous, and react fast enough not to die.*

The learning process is split into two main stages:

1. **Policy Distillation.** We first train a policy network to imitate an "expert" player. That expert can be:
   - A human player who survives for a long time,
   - A heuristic / scripted policy that encodes reasonable behavior,
   - Or an LLM-guided controller that decides actions based on interpreted game state.

   We record state-action pairs (s, a) and train a supervised model to reproduce those expert actions. This gives us an initial competent player without requiring unstable, purely-from-scratch RL exploration.

2. **Self-Play Reinforcement Learning (PPO / DQN).** After distillation, we fine-

tune the agent via reinforcement learning (RL), such as Proximal Policy Optimization (PPO) or Deep Q-Network-style updates. The agent repeatedly plays the game, collects rewards (mainly "stay alive longer"), and updates its policy to maximize long-term survival. This phase allows improvement *beyond* the expert demonstrations.

Ultimately, the project demonstrates an **end-to-end automated loop of vision → state extraction → decision policy → self-improvement**. We are not just doing vision or just doing RL. We are building a closed system that:

- Perceives the world through computer vision,
- Converts perception into a compact, human-interpretable state,
- Chooses actions in real time,
- And becomes better through iterative self-play.

This miniature setup is, on purpose, a scaled-down analog of robotics and autonomous driving (e.g., CARLA simulators), but in a much safer, faster-to-iterate game environment where we control **all** data collection.

2. Success Criteria

The project will be considered successful if the agent satisfies the following:

- **Object Detection Quality.** The perception module (YOLO-like object detector) must detect and track critical entities in the game — player avatar, obstacles (pipes / meteors / columns), and any collectible items — with at least **70% mAP** or comparable precision / recall. This ensures the downstream policy is not acting on garbage signals.
- **Imitation Accuracy.** In the policy distillation stage, the learned policy should reproduce expert / LLM-guided actions with at least **75% action agreement** on held-out frames. This indicates we successfully captured "expert reflexes" and basic survival heuristics.
- **Self-Play Performance Gain.** After RL fine-tuning, the agent's average survival time should **increase by at least 20%** relative to the distilled baseline *or* reach an absolute benchmark of **119 seconds** mean survival in our test environment.
- **Real-Time Inference.** The full closed-loop system must sustain $\geq$**59–60 FPS** at runtime, implying an end-to-end inference latency of ≤ 16.7 **ms/frame** (vision

inference + policy forward pass + environment step). This is a hard engineering constraint to make the agent feel "alive" rather than offline.

If we hit these criteria, then:

1. Vision is good enough to extract the state we care about,
2. The policy can learn meaningful behavior from demonstration,
3. Reinforcement learning *actually improves* robustness,
4. And the overall system is fast enough to be interactive.

3. Project Overview

3.1. High-Level Goal and Framing

We aim to build an agent that **sees the game like a human does** and **learns how to survive**.

Concretely:

- The agent ingests raw RGB frames from a Pygame-based mini-game (side-scrolling, Flappy Bird-style obstacle avoidance / meteor dodging).
- A lightweight YOLOv8-nano style detector identifies objects of interest each frame.
- We convert detections into a structured state vector (distances, gaps, approach speeds).
- A small MLP policy chooses an action (jump / flap / move / stay).
- Over time, via self-play PPO, the agent becomes harder to kill.

No privileged engine access (no direct pipe gap arrays, no "god mode" telemetry). We realistically simulate "vision $\rightarrow$ motor action" instead of reading internal variables.

3.2. Why This Is Interesting / Non-Trivial

Traditional RL projects in games often do *one* of the following:

- Just run PPO or DQN on Atari pixels.
- Just train YOLO on labeled frames.
- Just script Flappy Bird heuristics.

We are explicitly doing **all three layers together**:

1. **Perception**: fast object detection on live frames.
2. **Decision**: a distilled policy network that imitates an expert.
3. **Adaptation**: PPO-style self-play to surpass the expert.
4. **Deployment**: hard latency budget (target 60 FPS).

That puts us in a sweet spot between academic RL demos and practical autonomy systems.

3.3. Novelty and Originality of the Approach

(1) Hybrid Two-Stage Learning. We do **Policy Distillation first, RL second**. Stage 1 gives a competent baseline by imitating human / scripted / LLM-guided play. Stage 2 (self-play PPO / DQN) lets the agent *surpass* that baseline by trial-and-error survival optimization. This "distill then RL" pipeline is not standard in most simple game bots, which usually jump straight into RL from scratch.

(2) End-to-End Data Ownership. Instead of downloading Atari frames or using an existing benchmark dataset, we generate **everything in-house**:

- We build our own Pygame environment.
- We collect our own gameplay videos (human + scripted).
- We extract frames and hand-label them in Label Studio.
- We design our own augmentation (background randomization, motion blur, etc.).

This proves we understand the full vision pipeline from data collection to training.

(3) Real-Time Constraint as a First-Class Requirement. We impose a strict budget: ≤ 16.7 **ms/frame** for end-to-end inference. That pushes us toward:

- YOLOv8-nano (or similar tiny detector).
- A 3-layer MLP policy network instead of a huge transformer.
- ONNX Runtime export for accelerated inference.

A lot of RL papers don't care about FPS. We do. We are explicitly targeting "playable in real time," which is an engineering deliverable, not just a number in a table.

(4) Interpretability of the State. Rather than feed raw pixels to a black-box policy, we explicitly build a structured state vector:

[player_x, player_y, player_v_y, nextGap_x, nextGapTop, nextGapBottom, distanceToGap, timeToCollisi

That makes downstream behavior debuggable. We can literally log "the bot thought the pipe gap was too low" and understand why it flapped.

3.4. Comparison with Related Vision-Based Game AIs

3.4.1. Atari Deep Q-Network (DQN, Mnih et al. 2015)

DQN showed that an agent can learn Atari games directly from raw frames via convolutional Q-learning. But Atari agents typically read the emulator's framebuffer and directly regress Q-values from pixels. They don't explicitly *separate* perception and decision. We do: YOLO-like perception → interpretable state → MLP policy.

3.4.2. CARLA / Self-Driving RL

In autonomous driving simulators like CARLA, models often get high-level state like semantic segmentation masks, lidar depth, or privileged lane annotations. We *do not.* We force ourselves to work from RGB-only "camera" frames, which is closer to human-like visual perception (even though our world is much simpler).

3.4.3. OpenAI Gym / PyBullet Camera Agents

Gym / PyBullet pipelines can train RL from rendered cameras. But most open examples:

- Ignore real-time latency,
- Skip interpretability (policy is just a CNN),
- Don't combine imitation + PPO in stages.

We explicitly log latency and design for 60 FPS, and we use imitation to stabilize RL.

3.4.4. Flappy Bird-Style Survival Agents

Flappy Bird clones are classic RL demos: survive by timing "flaps" to pass through pipe gaps. Most public solutions either (a) read internal game variables directly (cheating from our perspective), or (b) map pixels → action with a tiny CNN. Our agent is stricter: it treats the rendered frame like a camera feed, detects objects, and converts them into distances/gap geometry explicitly. That makes our pipeline feel more like “vision-driven control,” not “cheat by reading hidden game state.”

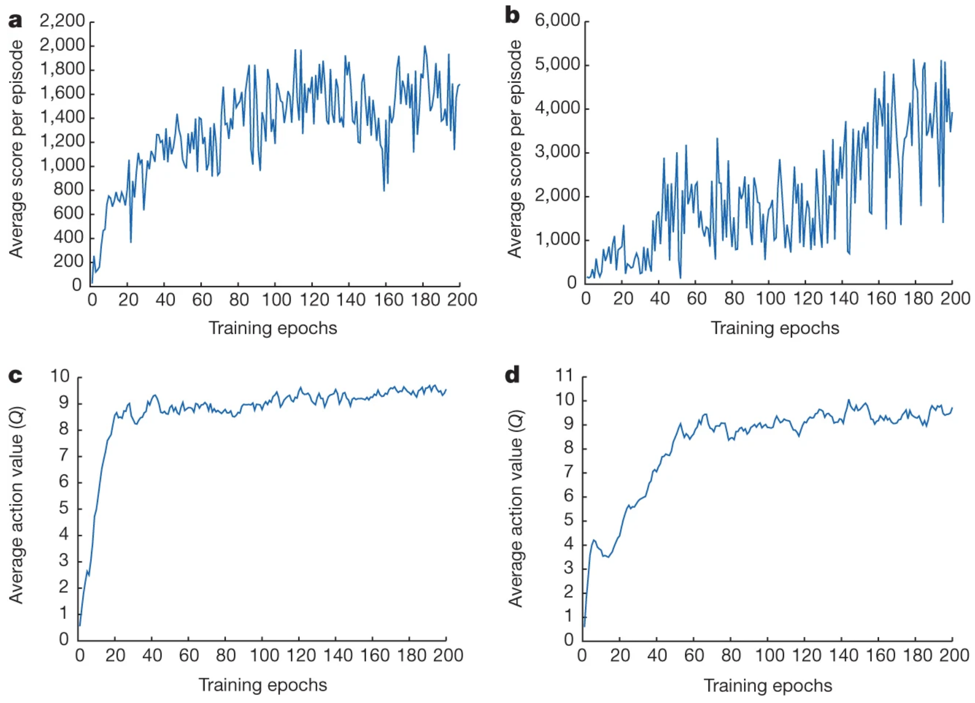


Figure 1: High-level pipeline comparison. Left: end-to-end pixel-to-action RL (e.g., Atari DQN). Middle: autonomy stacks (e.g., CARLA) using rich or privileged sensors. Right: our proposed Distilled Vision Agent: lightweight YOLO-style perception, distilled policy, then PPO fine-tuning, all under a strict 60 FPS runtime budget.

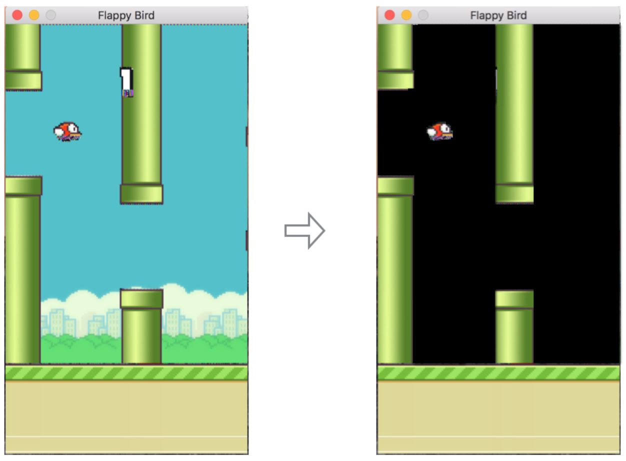


Figure 2: Example of a Flappy Bird-style side-scrolling survival game. The player-controlled avatar must avoid collisions with incoming pipes / obstacles. Our agent will detect player and upcoming gap regions directly from the rendered frame, estimate relative distances/velocities, and act in real time.

These baselines show that our approach is in between pure black-box pixels-to-action RL and heavyweight autonomy stacks: we keep the “vision only” restriction, but we still demand interpretability and deployment-level runtime.

4. How the Project Will Be Completed

4.1. Training Data and How It Will Be Obtained

We will create our own dataset:

- We implement a custom Pygame mini-game with a continuous stream of hazards (pipes, meteors, debris). Survive as long as possible.
- We record ∼5–10 minutes of gameplay across multiple sessions: human play, scripted baseline policy, and possible LLM-guided heuristics.
- From recordings, we sample roughly 500–1,000 representative frames.
- We manually annotate these frames in Label Studio with bounding boxes for:
  - `player` (the controllable avatar),
  - `obstacle` / `pipe` / `meteor`,
  - `gap` / `safe corridor`,
  - `item` (if applicable).

To make the detector robust, we apply data augmentation to generate ∼5,000 synthetic samples:

- Random background swaps (sky textures, cave textures, etc.).
- Motion blur / Gaussian noise to mimic speed and uncertainty.
- Scaling, small rotation, hue and saturation jitter.
- Partial occlusion / fake lighting changes.

In parallel, for **policy distillation**, we will log (s, a) pairs:

- s: structured state extracted from detection (e.g., player position, next gap position, distance to collision).
- a: expert action taken at that moment (e.g., flap, no flap).

This labeled dataset directly trains the initial policy network using supervised learning.

4.2. Frameworks and Tooling

We will primarily use **PyTorch** because:

- It supports YOLOv8-style detectors (Ultralytics ecosystem).
- It interoperates with Stable-Baselines3 for PPO / DQN style reinforcement learning.
- It exports cleanly to **ONNX Runtime** for optimized inference.
- It integrates well with TensorBoard / Weights & Biases for experiment tracking.

The RL backbone will likely be PPO (following OpenAI Spinning Up style implementations), because PPO is relatively stable for on-policy updates and supports stochastic policies that can explore during self-play.

4.3. Planned Network Architecture

Vision Module (Perception). We will adapt a lightweight YOLOv8-nano style detector to run in real-time. Each frame $\rightarrow$ bounding boxes + classes + confidence scores. We can also track object motion over consecutive frames to estimate velocity or approach speed.

State Encoder. We will convert detector output into a compact numerical state vector, e.g.:

[player_x, player_y, player_v_y, nextGap_x, nextGapTop, nextGapBottom, distanceToGap, timeToCollisio

This vector is intentionally human-readable, so we can debug.

Policy Network (Decision-Making). A 3-layer MLP maps the state vector to a discrete action distribution. The candidate actions are things like:

- `flap / jump`
- `do nothing`
- (optionally) `horizontal adjust`, if our environment supports x-movement

During **Phase 1 (Distillation)**, this MLP is trained with supervised learning to imitate the expert. During **Phase 2 (RL)**, we fine-tune it with PPO or DQN using survival reward.

Real-Time Control Loop. Every frame:

1. Capture RGB frame from the game.
2. Run YOLOv8-nano (exported to ONNX Runtime) to detect objects.
3. Convert detections to structured state.
4. Run the policy network to choose an action.
5. Apply that action back to the Pygame environment.
6. Log survival time, reward, collisions, etc.

All of this must complete in ≤ 16.7 ms to achieve $\geq$60 FPS.

5. Project Roles and Responsibilities (can be altered later)

For grading, each person owns a technically distinct component. Each student will submit an individual report, so we are making contributions clearly attributable.

5.1. Jeewon Kim (jk4864)

- **System Architecture and Module Integration.** Designs the overall pipeline: how the detector output becomes a state vector, how that vector is consumed by the policy network, and how decisions are fed back into the game in real time.
- **YOLOv8 Fine-Tuning for Detection.** Leads training of the object detector on our labeled frames. This includes setting up Label Studio exports, writing training configs, monitoring mAP / precision / recall, and iterating on augmentation to improve robustness.
- **Policy Distillation Pipeline.** Trains the supervised imitation policy. Defines the loss (cross-entropy over actions), the train/validation split of state-action pairs, and the metric for imitation accuracy ($\geq$ 75% target).
- **Quantitative Evaluation.** Evaluates detection accuracy and initial (pre-RL) survival performance to establish the baseline that PPO must beat.
- **Data Acquisition and Preprocessing (Shared).** Participates in recording gameplay, extracting frames, and maintaining clean annotations.

5.2. Chloe Lee (cl4490)

- **Environment and Self-Play Simulation.** Builds the Pygame environment and automates rollout generation for both supervised data collection and RL self-play.

Implements obstacle spawning, collision detection, reward assignment, and difficulty ramping.

- **Reinforcement Learning (PPO / DQN).** Implements and tunes PPO / DQN for our policy network in Phase 2 (post-distillation). Handles reward shaping, curriculum tweaks, entropy regularization, PPO clip ratio, and other stability heuristics.

- **Reward + Survival Metric Design.** Defines the scalar reward signal that the agent maximizes (e.g., +1 per timestep alive, penalties on collision). Designs what counts as "good" survival and when an episode terminates.

- **Experiment Tracking and Visualization.** Owns TensorBoard / Weights & Biases logging. Produces learning curves such as reward vs. training steps, survival time vs. epoch, and PPO value loss vs. update iteration. These plots will go in the final report.

- **Latency / Real-Time Evaluation.** Profiles FPS with the full loop running (detection + policy). Confirms whether we satisfy the target of ≤ 16.7 ms/frame end-to-end.

- **Data Acquisition and Preprocessing (Shared).**

5.3. Minsuk Kim (mk4434)

- **Augmentation and Robustness Engineering.** Implements and documents the automated data augmentation pipeline: background randomization, motion blur, scaling, hue jitter, partial occlusions. Uses this to scale ~1k labeled frames to ~5k effective samples and studies how that affects YOLOv8 generalization.

- **Model Debugging and Visualization Tools.** Builds visualization scripts that render bounding boxes, class confidences, and (if applicable) attention / saliency maps on top of game frames. Also logs per-frame state vectors so we can interpret *why* the policy chose a certain action (example: "agent thought gap was below it, so it flapped").

- **Deployment / Runtime Optimization.** Exports both the detector and the policy MLP to ONNX Runtime. Profiles inference cost per frame and helps ensure we hit the $\geq$59–60 FPS real-time requirement during live play.

- **RL Loop Instrumentation.** Works with the RL pipeline to capture rollout statistics: chosen actions, rewards over time, failure modes (what situations kill the agent), and how PPO / DQN improves them.

- **Repository, Version Control, and Documentation.** Maintains the GitHub repo layout, training scripts, dependency setup, and config files. Ensures that each stage (dataset labeling, YOLO training, distillation, PPO fine-tune) is reproducible and attributable for final grading.
- **Data Acquisition and Preprocessing (Shared).**

This breakdown intentionally maps each person to a technically deep piece of the pipeline:

- Vision + distillation and baseline evaluation (Jeewon),
- RL / reward shaping / self-play (Chloe),
- Augmentation + deployment + infra / reproducibility (Minsuk).

So when individual reports are graded separately, each student can point to a clear, defensible area of ownership.

Appendix A. Lecture References and Visuals

Below are key lecture materials from Prof. Peter Belhumeur's *Deep Learning for Computer Vision* course that directly informed our project design.

Lecture 7: Convolutional Networks

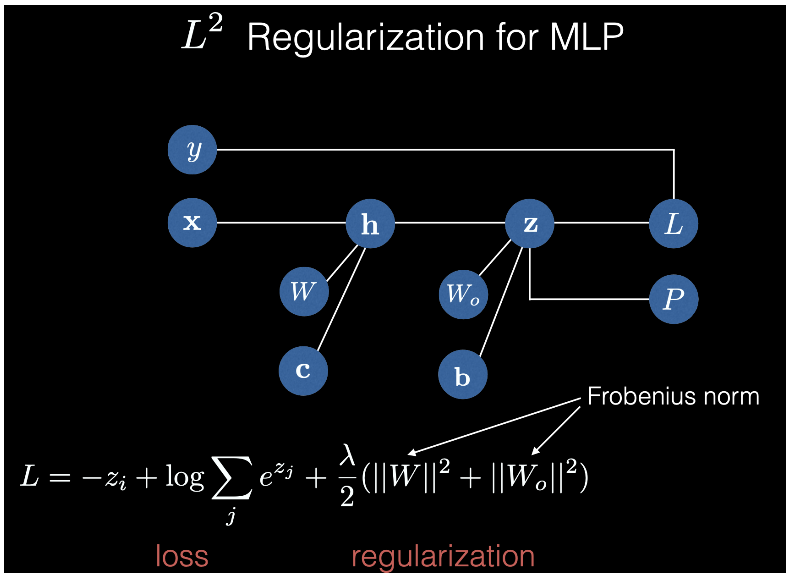
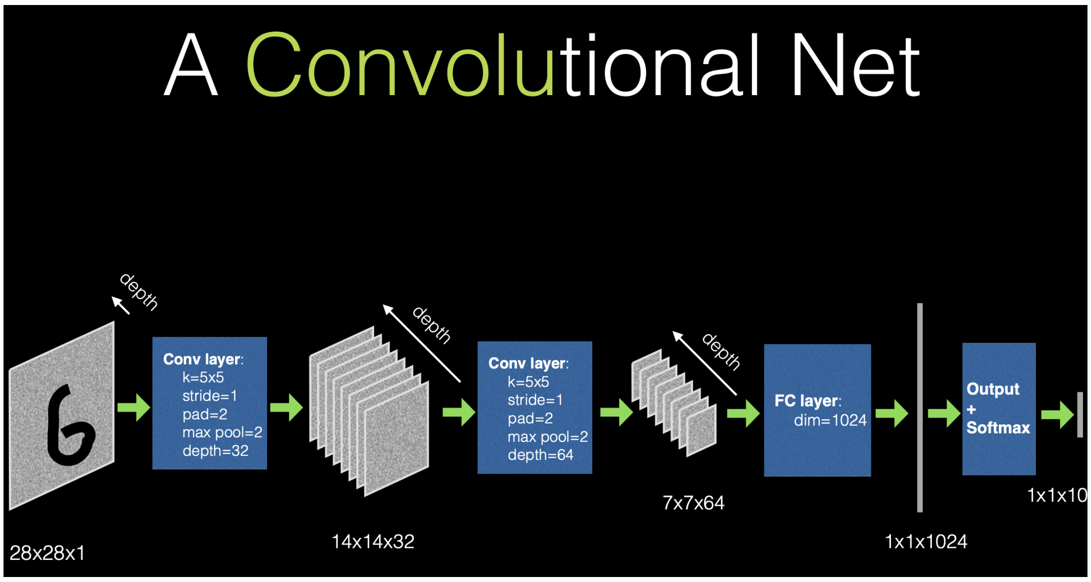


Figure 3: Lecture 7 visual summary of convolutional feature extraction. Our YOLOv8-nano backbone inherits these convolutional principles for efficient spatial feature encoding.

Lecture 9: Object Detection I — YOLO and SSD



A Convolutional Net
depth
28x28x1
Conv layer: k=5x5 stride=1 pad=2 max pool=2 depth=32
depth
14x14x32
Conv layer: k=5x5 stride=1 pad=2 max pool=2 depth=64
depth
7x7x64
FC layer: dim=1024
1x1x1024
Output + Softmax
1x1x10

Figure 4: Lecture 9 introduction to single-shot object detectors (YOLO/SSD). Provided theoretical grounding for our real-time detection pipeline.

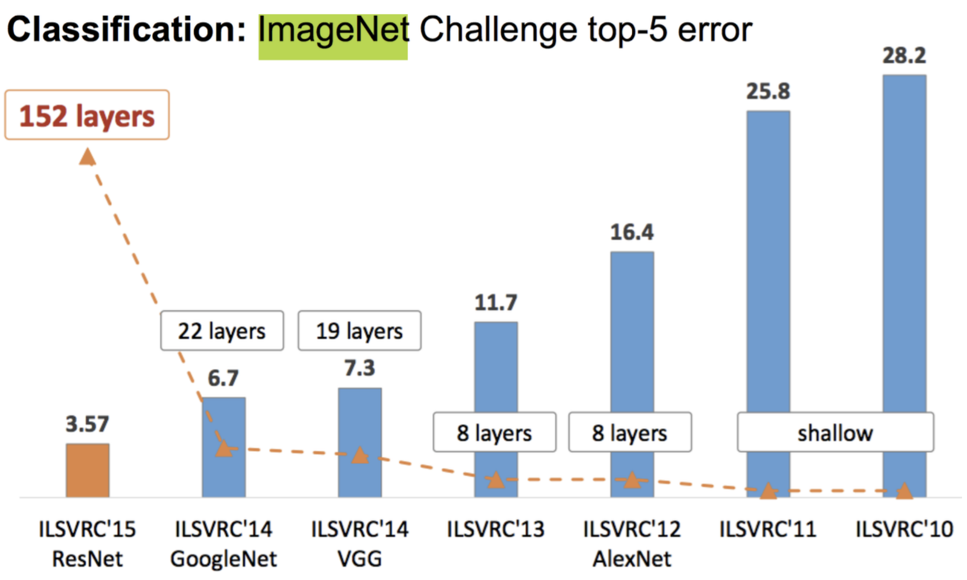


Figure 5: Lecture 10 comparison between region-based and one-stage detectors. Helped justify our design choice favoring YOLOv8 over heavier RCNN architectures.

References

1. Mnih, V. et al. (2015). “Human-level control through deep reinforcement learning.” *Nature.* (Introduced DQN and Atari pixel-based RL.)

2. Schulman, J. et al. (2017). “Proximal Policy Optimization Algorithms.” (PPO: clipped policy gradient with stability tricks; standard for on-policy RL.)

3. Dosovitskiy, A. et al. (2017). “CARLA: An Open Urban Driving Simulator.” *CoRL.* (Autonomous driving simulator; vision + control loop inspiration.)

4. Redmon, J. et al. (2016). “You Only Look Once: Unified, Real-Time Object Detection.” *CVPR.* (YOLO: real-time multi-object detection, foundational to our perception stack.)

5. Ultralytics (2023). “YOLOv8.” `https://github.com/ultralytics/ultralytics`. (Modern YOLO implementation; provides fast, small backbones like YOLOv8-nano.)

6. OpenAI Spinning Up. “Proximal Policy Optimization (PPO).” (Reference for PPO training loop, clipping objective, KL early-stopping, and logging best practices.)

7. Belhumeur, P. (2024). Lecture Notes 3–5, 7, 9, 10. *Deep Learning for Computer Vision, Columbia University.* (Covers logistic regression, perceptron, backpropagation, gradient descent, convolutional detection, and deployment constraints relevant to our detector and policy training.)