The Doom Slayer's Handbook: A Comprehensive Research Report on Deep Reinforcement Learning in ViZDoom

1. Introduction: The Renaissance of First-Person Shooters in AI Research

The quest for Artificial General Intelligence (AGI) has historically used games as the primary crucible for testing cognitive architectures. While the discrete, fully observable worlds of Chess and Go drove the early triumphs of symbolic AI and later Monte Carlo Tree Search, the advent of Deep Reinforcement Learning (DRL) necessitated environments that could simulate the messy, high-dimensional reality of the physical world. In this context, the First-Person Shooter (FPS) genre—specifically the 1993 classic *DOOM*—emerged not merely as a nostalgic artifact, but as a critical benchmark for visual navigation, partial observability, and rapid decision-making.

This report serves as an exhaustive guide for the implementation, training, and theoretical understanding of autonomous agents within the ViZDoom environment. It addresses the user's requirement for a project that balances competent performance with deep pedagogical insight. The analysis draws upon the latest documentation from the Farama Foundation, empirical studies from the IEEE Transactions on Games, and the collective engineering wisdom of the open-source community.[1]

1.1. From Atari to Doom: The Evolution of Complexity

To understand the position of ViZDoom in the RL landscape, one must contrast it with its predecessor, the Arcade Learning Environment (ALE). ALE, which popularized the use of Atari

2600 games, presented agents with 2D, third-person perspectives where the state of the world was often fully observable on a single screen.[4] The agent knew exactly where the enemies were because they were rendered pixels on the same 2D plane.

ViZDoom represents a quantum leap in complexity. Based on the ZDoom engine, it offers a 3D (technically 2.5D ray-casted) perspective. This introduces the problem of **partial observability**: an enemy might be behind a pillar, around a corner, or behind the agent. The agent must therefore develop a "theory of mind" or at least a recurrent memory representation to track objects that are no longer visible.[5] Furthermore, the input space is a raw screen buffer, necessitating the use of Convolutional Neural Networks (CNNs) to extract features from texture-mapped surfaces, complex lighting, and variable geometry.[1]

1.2. The Modern Ecosystem: Gymnasium and the Farama Foundation

A significant portion of early DRL literature references OpenAI's gym library. However, it is critical for any new project to acknowledge the paradigm shift that occurred in 2022-2023. OpenAI ceased maintenance of Gym, and the standard interface is now **Gymnasium**, maintained by the Farama Foundation.[8] This is not a trivial semantic change; it involves fundamental alterations to the API contract, such as the splitting of the done signal into terminated (the agent died or won) and truncated (the time limit was reached).[9]

As of 2025, ViZDoom has officially integrated Gymnasium support, rendering legacy wrappers obsolete.[11] However, for projects utilizing older codebases or specific multi-agent configurations, the **Shimmy** compatibility layer remains a vital tool, allowing legacy Gym environments to function within modern Gymnasium pipelines.[9] Understanding this ecosystem is the first step in creating a project that is both reproducible and future-proof.

2. Strategic Decision: CleanRL vs. Stable-Baselines3

Given your constraints—limited time (one month), a need for high effectiveness, but a specific requirement to "learn and understand"—the choice of library is the most critical architectural decision you will make.

2.1. The Recommendation: CleanRL

For this specific project, **CleanRL is the superior choice.**

While Stable-Baselines3 (SB3) is excellent for production, it is designed as a "black box." You import PPO, call model.learn(), and it works. However, when it fails (and it *will* fail on "Deadly Corridor"), debugging SB3 requires digging into their source code to understand how they handle buffer normalization or advantage calculation.

Why CleanRL fits your constraints:

1. **Single-File Logic:** CleanRL puts the *entire* algorithm (PPO, network definition, training loop) into a single .py file. You can read the code from top to bottom. This maximizes "understanding."
2. **Hackability:** When you need to add "Reward Shaping" for the Deadly Corridor (see Section 6), you don't need to write a complex Callback class as in SB3. You simply find line 150 where rewards are processed and add + game.get_game_variable(KILLCOUNT).
3. **Speed:** CleanRL is highly optimized and often faster than SB3 because it strips away the abstraction overhead.

Strategy: Use the ppo.py or ppo_atari.py from CleanRL as your base template. Copy it into your project, rename it agent.py, and modify the environment setup.

3. Step-by-Step Implementation Guide

This guide uses uv for high-speed dependency management. We assume you are using WSL2 (Windows Subsystem for Linux) or native Linux. Do not attempt this on raw Windows; dependency management for vizdoom on Windows is often painful.

Step 1: Environment & Dependencies

Instead of Conda, we will use uv to manage the project and virtual environment.

1. **Install System Dependencies (Linux/WSL2):** Even with uv, ViZDoom needs a few

C++ libraries (sound and graphics) that Python cannot install.

```shell
sudo apt update
sudo apt install build-essential zlib1g-dev libsdl2-dev
libjpeg-dev \
nasm tar libbz2-dev libgtk2.0-dev cmake libboost-all-dev \
libfluidsynth-dev libgme-dev libopenal-dev timidity
libwildmidi-dev unzip
```

2. **Initialize Project with uv**

```shell
# Install uv (if you haven't already)
curl -LsSf https://astral.sh/uv/install.sh | sh

# Create a new project
uv init doom_agent
cd doom_agent

# Create virtual environment and install dependencies
# We use gymnasium[accept-rom-license] to ensure compatibility
uv add "gymnasium[accept-rom-license]" vizdoom moviepy torch
torchvision numpy opencv-python
```

Step 2: The Wrapper Stack (The "Glue")

ViZDoom returns a dictionary (screen, health, ammo), but PPO expects a standard tensor. You must create a wrapper to bridge this gap.

Create wrappers.py:
This code adapts ViZDoom to the Gymnasium standard.

Python
```python
import gymnasium as gym
from gymnasium import spaces
import numpy as np
import cv2

class VizDoomGym(gym.Env):
    def __init__(self, game):
        super().__init__()
        self.game = game
        self.game.init()

        # Define Action Space (3 buttons: Left, Right, Shoot)
        self.action_space = spaces.Discrete(3)

        # Define Observation Space (C, H, W) for PyTorch
        # Resizing to 84x84 is standard for DQN/PPO
        self.observation_space = spaces.Box(
            low=0, high=255, shape=(1, 84, 84), dtype=np.uint8
        )

    def step(self, action):
        # Convert discrete action index to one-hot buttons
        buttons = [0] * 3
        buttons[action] = 1

        # Make action and skip 4 frames (standard RL practice)
        reward = self.game.make_action(buttons, 4)

        # Get state
        state = self.game.get_state()
        done = self.game.is_episode_finished()

        if state:
            # Grab screen buffer
            screen = state.screen_buffer
            # Resize and Grayscale
            screen = cv2.resize(screen, (84, 84))
            # Add channel dimension (1, 84, 84)
            obs = np.expand_dims(screen, axis=0)
        else:
            obs = np.zeros((1, 84, 84), dtype=np.uint8)

        # Handle "info" dict (useful for debugging)
        info = {}

        return obs, reward, done, False, info

    def reset(self, seed=None, options=None):
        self.game.new_episode()
        state = self.game.get_state()
        screen = state.screen_buffer
        screen = cv2.resize(screen, (84, 84))
        obs = np.expand_dims(screen, axis=0)
        return obs, {}
```

Step 3: The PPO Agent (CleanRL Adaptation)

Download the ppo.py script from the(https://github.com/vwxyzjn/cleanrl). You will modify the make_env function to use your wrapper.

Modify make_env in your script:

Python
```
import vizdoom as vzd
from wrappers import VizDoomGym

def make_env(scenario_path):
    def thunk():
        game = vzd.DoomGame()
        game.load_config(scenario_path)
        # Important: Set window to invisible for training speed
        game.set_window_visible(False)
        game.set_mode(vzd.Mode.PLAYER)
        env = VizDoomGym(game)
        # Standard wrappers for stability
        env = gym.wrappers.RecordEpisodeStatistics(env)
        # Stack 4 frames so agent sees motion
        env = gym.wrappers.FrameStack(env, 4)
        return env
    return thunk
```

Step 4: Training Loop & Curriculum

Do not start with the hardest map. Use the Curriculum strategy.

1. **Level 1: Basic.cfg** (10-20 minutes)
   - **Goal:** Sanity check. Can the agent learn to simply press "Shoot" when it sees pixels?
   - **Success Metric:** Average return > 0 consistently.
2. **Level 2: Defend the Center** (2-4 hours)
   - **Challenge:** The agent must learn to rotate.
   - **Tip:** If it spins in circles forever, check your living_reward in the .cfg file. It should not be positive!
3. **Level 3: Deadly Corridor** (Overnight)
   - **The Trap:** As mentioned in Section 6, the default reward is distance-based. The agent will learn to commit suicide to avoid the penalty of being shot.
   - **The Fix:** You *must* implement Reward Shaping.

Implementing Reward Shaping in Code:
Modify your step function in VizDoomGym to add custom rewards.

```python
    def step(self, action):
        #... existing action logic...

        # EXTRACT VARIABLES
        # You need to add these variables to the.cfg file first!
        kill_count =
self.game.get_game_variable(vzd.GameVariable.KILLCOUNT)
        ammo =
self.game.get_game_variable(vzd.GameVariable.AMMO2)

        # SHAPED REWARD CALCULATION
        reward = game_reward

        # Reward for kills (Encourage aggression)
        if kill_count > self.last_kill_count:
            reward += 100

        # Penalty for wasting ammo (Encourage aiming)
        if ammo < self.last_ammo:
            reward -= 5

        self.last_kill_count = kill_count
        self.last_ammo = ammo

        return obs, reward, done, False, info
```

4. Theoretical Foundations: Algorithms for FPS Agents

The choice of algorithm dictates the agent's learning capability. In the context of FPS games, the literature is dominated by two families: **Value-Based** methods (DQN) and **Policy-Based** methods (PPO).

4.1. Deep Q-Networks (DQN): The Value Approximator

DQN was the algorithm that started the DRL revolution. It works by learning a Q-function $Q(s, a)$, which estimates the total expected future reward of taking action a in state s.[5]

In Doom, vanilla DQN faces significant challenges:

- **Overestimation Bias:** The max operator in Q-learning tends to overestimate the value of noisy states. In a complex 3D corridor, this can lead the agent to believe that getting stuck in a corner is a "good" state. **Double DQN (DDQN)** mitigates this by decoupling action selection from value estimation.[13]
- **Sparse Rewards:** DQN struggles to propagate reward signals back through long trajectories. If the agent only gets a reward after 500 steps (finding the vest), the gradient signal may vanish before it teaches the agent the first step (opening the door).[15]

Despite these issues, DQN remains a strong baseline, particularly when enhanced with **Prioritized Experience Replay (PER)** and **Dueling Architectures** (which separate state value from action advantage).[16]

4.2. Proximal Policy Optimization (PPO): The Stable Standard

For a project focused on "learning and understanding," **PPO (Proximal Policy Optimization)** is the recommended algorithm. PPO belongs to the Actor-Critic family. It maintains two networks:

1. **Actor:** Outputs a probability distribution over actions $\pi(a|s)$.
2. **Critic:** Estimates the value of the current state $V(s)$.

The key innovation of PPO is the **clipped surrogate objective**. In standard policy gradients, a single bad update (too large a step size) can destroy the policy, causing the agent's performance to collapse irrecoverably. PPO constrains the update so that the new policy does not deviate too wildly from the old policy.[3]

Why PPO for Doom?

- **Sample Efficiency:** While typically less sample-efficient than DQN, PPO's stability allows it to utilize massive parallelization. We can run 8 or 16 instances of Doom simultaneously, collecting data in parallel, which maximizes the throughput of the ViZDoom engine.[18]
- **Hyperparameter Robustness:** PPO is notoriously more forgiving with hyperparameters than DQN. A learning rate of 2.5×10^{-4} works for almost everything.[19]
- **Performance:** Empirical studies on the "Deadly Corridor" scenario show that PPO significantly outperforms both DQN and A2C in terms of final accumulated reward and

convergence speed.[20]

4.3. Recurrent Architectures (DRQN and LSTM)

Standard CNNs assume the current frame stack contains all necessary info. In a deathmatch, if an enemy walks behind a wall, they disappear from the frame stack. A standard CNN "forgets" they exist.

Deep Recurrent Q-Networks (DRQN) introduce an LSTM (Long Short-Term Memory) layer after the CNN feature extractor. This hidden state persists across time steps, allowing the agent to "remember" the enemy's last known position.[6] While theoretically superior for Deathmatch, LSTMs add significant implementation complexity (managing hidden states during training batches) and are computationally heavier. For a student project, they are an advanced extension, not a requirement for the initial scenarios.[21]

5. Anatomy of the ViZDoom Platform

Before delving into neural architectures, one must master the environment itself. ViZDoom is a research platform that wraps the ZDoom engine, exposing its internal state to C++, Python, and Julia.[1]

5.1. The Engine Mechanics: Tics, Buffers, and Variables

At its core, the Doom engine operates in discrete time steps called **tics**. In the original game, the logic runs at 35 tics per second. ViZDoom decouples this from real-time, allowing the engine to run as fast as the hardware permits—up to 7,000 fps in synchronous mode.[1] This speed is a decisive advantage over more photorealistic simulators like Unity ML-Agents, allowing for the massive sample throughput required by algorithms like PPO.[3]

The environment provides several data streams:

- **Screen Buffer:** The primary input, typically a byte array of pixel data (RGB or Grayscale).
- **Depth Buffer:** A rendering of the Z-depth of each pixel. While typically used for "cheat"

agents or debugging, it allows for the integration of simultaneous localization and mapping (SLAM) techniques.[1]

- **GameVariables:** Direct access to internal memory addresses holding values like Health, Ammo, Position (X, Y, Z), and Kill Count. These are essential for constructing reward functions but should generally *not* be fed into the agent's policy network if the goal is to simulate human-like visual play.[1]

5.2. The Configuration Architecture (.cfg and.wad)

A ViZDoom "scenario" is defined by two files: a .wad file (Where's All the Data) containing the map geometry and textures, and a .cfg file controlling the RL parameters.[25]

For a student project, the .cfg file is the control center. It defines:

- **Available Actions:** Which buttons can the agent press? For a basic movement task, this might just be MOVE_LEFT, MOVE_RIGHT, and ATTACK. For a deathmatch, it expands to include TURN_LEFT, TURN_RIGHT, MOVE_FORWARD, etc..[25]
- **Living Reward:** A constant reward given at every step. Setting this to -1 creates "pressure" to finish the level quickly. Setting it to +1 (as in survival tasks) encourages the agent to hide and do nothing.
- **Episode Timeout:** The maximum number of tics before the episode is truncated.[27]

6. Deep Dive: The "Deadly Corridor" Challenge

This scenario is the crucible where the project transitions from "following a tutorial" to "doing research." It is defined by the **Sparse Reward Problem**.

6.1. The Geometry of Failure

The corridor is narrow. Enemies are positioned in a way that creates a "crossfire." A simple reactive agent (one that maps pixels directly to actions without temporal context) will fail because it cannot dodge projectiles effectively while moving forward. It requires **strafing** (moving sideways while facing forward).

6.2. Intrinsic Motivation and Curiosity

When extrinsic rewards (the vest) are too far away, the agent needs **Intrinsic Motivation**. One successful technique referenced in the literature is the **Rarity of Events (RoE)**.[28]

- **Concept:** The agent maintains a buffer of states it has visited. If it enters a state (or sees a visual pattern) it has rarely seen before, it gives itself an internal reward.
- **Implementation:** In a student project, this can be simplified to "Exploration Bonuses." Reward the agent for visiting new X-coordinates in the corridor.
- **Outcome:** RoE has been shown to enable agents to solve VizDoom scenarios without *any* extrinsic reward, purely by driving them to explore the novelty of the end of the corridor.[28]

6.3. Solving the Suicide Loop

As mentioned, the suicide loop is a mathematical certainty under specific reward configurations.
To diagnose this, monitor the Average Episode Length.

- **Hypothesis:** If Average Episode Length drops to near zero, the agent is suiciding.
- **Fix:** Reduce the living penalty. Or, paradoxically, *remove* the death penalty entirely during the early phases of training. Let the agent learn that death is just a "reset" without a massive point deduction, encouraging it to take risks.[24]

7. Evaluation and Analysis

A competent research project is defined by how it evaluates success. "Getting a high score" is insufficient.

7.1. Statistical Significance

RL is noisy. A single training run might succeed by luck (random initialization of weights). To prove the agent is competent, one must run the training multiple times (e.g., 5 seeds) and average the results.
The literature recommends using the Mann-Whitney U test to compare the performance of different algorithms (e.g., PPO vs DQN). This non-parametric test determines if one distribution of rewards is stochastically greater than another, providing a P-value to support claims of superiority.20

7.2. Metrics Beyond Reward

- **Entropy:** Monitor the policy entropy. High entropy = random behavior. Low entropy = deterministic behavior. A sudden drop in entropy usually signals that the agent has converged. If it converges too early (while reward is low), it has collapsed into a local optimum.[29]
- **Value Loss:** If the Critic network's loss explodes, the agent acts randomly because it cannot predict the value of its actions. This often happens if the reward scale is too large (e.g., +1000). **Reward Normalization** (dividing all rewards by 100 or using a ClipReward wrapper) is the standard fix.[24]

7.3. Visual Debugging

One cannot understand an agent solely through graphs. You must watch it play.

- **Video Recording:** Use gymnasium.wrappers.RecordVideo. Set the episode_trigger to record every 50th episode.[30]
- **Analysis:** Watch the videos.
  - Does the agent "jitter" (shake back and forth)? This suggests the frame stack is too shallow or the action repetition is too low.
  - Does it spin in circles? This suggests the living_reward is positive (farming points).
  - Does it hug the wall? This is a common strategy to minimize the visible cross-section to enemies.

8. Conclusion and Future Horizons

Successfully training an agent to navigate the "Deadly Corridor" is a significant achievement. It demonstrates mastery over the complexities of high-dimensional visual processing, partial observability, and sparse reward dynamics.

This investigation highlights that while the tools have evolved—from Gym to Gymnasium, from monolithic scripts to CleanRL—the core challenges remain theoretical. The "suicide trap" in Deadly Corridor is not a bug; it is a correct mathematical solution to a poorly posed reward function. Understanding this distinction is the hallmark of a true RL practitioner.

8.1. Future Work: Generalization

The current frontier in Doom research, as highlighted by the ViZDoom competitions, is Generalization. An agent trained on "Deadly Corridor" will likely fail instantly on "Defend the Center." It has memorized the texture of the corridor walls, not the concept of "corridor." Future iterations of this project could explore:

- **Domain Randomization:** Randomizing the textures and lighting during training to force the agent to learn geometric features rather than overfitting to specific pixel patterns.[31]
- **Multi-Task Learning:** Training a single network on multiple scenarios simultaneously, using a "task ID" as an additional input, to develop a robust "Doom Generalist".[23]

For the student, the path is clear: Start with the Basic scenario, adopt CleanRL for transparency, embrace the Gymnasium ecosystem, and respect the lethal mathematics of the Deadly Corridor.

Citations:

Obras citadas

1. Farama-Foundation/ViZDoom: Reinforcement Learning ... - GitHub, fecha de acceso: noviembre 24, 2025, https://github.com/Farama-Foundation/ViZDoom
2. [1809.03470] ViZDoom Competitions: Playing Doom from Pixels - arXiv, fecha de acceso: noviembre 24, 2025, https://arxiv.org/abs/1809.03470
3. vwxyzjn/cleanrl: High-quality single file implementation of Deep Reinforcement Learning algorithms with research-friendly features (PPO, DQN, C51, DDPG, TD3, SAC, PPG) - GitHub, fecha de acceso: noviembre 24, 2025, https://github.com/vwxyzjn/cleanrl
4. Playing FPS Games with Deep Reinforcement Learning - The Association for the Advancement of Artificial Intelligence, fecha de acceso: noviembre 24, 2025,

https://aaai.org/ocs/index.php/AAAI/AAAI17/paper/download/14456/14385%3E

5. A Survey of Deep Reinforcement Learning in Video Games - arXiv, fecha de acceso: noviembre 24, 2025, https://arxiv.org/pdf/1912.10944
6. Using VizDoom Research Platform Scenarios for Benchmarking Reinforcement Learning Algorithms in First-Person Shooter Games - IEEE Xplore, fecha de acceso: noviembre 24, 2025, https://ieeexplore.ieee.org/iel7/6287639/10380310/10413478.pdf
7. Vizdoom - Nolan Winsman's Porfolio, fecha de acceso: noviembre 24, 2025, https://nolanwinsman.com/pdfs/VizDoom-Paper.pdf
8. Gymnasium Documentation, fecha de acceso: noviembre 24, 2025, https://gymnasium.farama.org/
9. Gymnasium Release Notes, fecha de acceso: noviembre 24, 2025, https://gymnasium.farama.org/gymnasium_release_notes/index.html
10. Basic Usage - Gymnasium Documentation, fecha de acceso: noviembre 24, 2025, https://gymnasium.farama.org/v1.1.0/introduction/basic_usage/
11. Releases · Farama-Foundation/ViZDoom - GitHub, fecha de acceso: noviembre 24, 2025, https://github.com/mwydmuch/ViZDoom/releases
12. Shimmy Compatibility Wrappers - PettingZoo Documentation, fecha de acceso: noviembre 24, 2025, https://pettingzoo.farama.org/api/wrappers/shimmy_wrappers/
13. Building Improved AI Agents for Doom using Double Deep Q-learning - Medium, fecha de acceso: noviembre 24, 2025, https://medium.com/data-science/discovering-unconventional-strategies-for-doom-using-double-deep-q-learning-609b365781c4
14. Using VizDoom Research Platform Scenarios for Benchmarking Reinforcement Learning Algorithms in First-Person Shooter Games - ResearchGate, fecha de acceso: noviembre 24, 2025, https://www.researchgate.net/publication/377671572_Using_VizDoom_Research_Platform_Scenarios_for_Benchmarking_Reinforcement_Learning_Algorithms_in_First-Person_Shooter_Games
15. (PDF) ViZDoom Competitions: Playing Doom from Pixels - ResearchGate, fecha de acceso: noviembre 24, 2025, https://www.researchgate.net/publication/327570317_ViZDoom_Competitions_Playing_Doom_from_Pixels
16. Doom Deadly Corridor | Reinforcement-Learning - GitHub Pages, fecha de acceso: noviembre 24, 2025, https://deepanshut041.github.io/Reinforcement-Learning/cgames/04_doom_corridor/
17. Proximal Policy Gradient (PPO) - CleanRL, fecha de acceso: noviembre 24, 2025, https://docs.cleanrl.dev/rl-algorithms/ppo/
18. CleanRL: Advanced PPO - PettingZoo Documentation, fecha de acceso: noviembre 24, 2025, https://pettingzoo.farama.org/tutorials/cleanrl/advanced_PPO/
19. The 37 Implementation Details of Proximal Policy Optimization - The ICLR Blog Track, fecha de acceso: noviembre 24, 2025,

https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/
20. Evaluating the effects of hyperparameter optimization in VizDoom - DiVA portal, fecha de acceso: noviembre 24, 2025, https://www.diva-portal.org/smash/get/diva2:1679888/FULLTEXT01.pdf
21. ViZDoomBot/stable-baselines-agent - GitHub, fecha de acceso: noviembre 24, 2025, https://github.com/ViZDoomBot/stable-baselines-agent
22. ViZDoom - Julia Packages, fecha de acceso: noviembre 24, 2025, https://juliapackages.com/p/vizdoom
23. [PDF] ViZDoom Competitions: Playing Doom From Pixels | Semantic Scholar, fecha de acceso: noviembre 24, 2025, https://www.semanticscholar.org/paper/ViZDoom-Competitions%3A-Playing-Doom-From-Pixels-Wydmuch-Kempka/de0f96ebabb75700a9febcfe4ee0c5eb0adea7ba
24. Reward implementation for DeadlyCorridor · Issue #441 · Farama-Foundation/ViZDoom, fecha de acceso: noviembre 24, 2025, https://github.com/mwydmuch/ViZDoom/issues/441
25. Deep Reinforcement Learning for Playing Doom — Part 1: Getting Started - Leandro Kieliger, fecha de acceso: noviembre 24, 2025, https://lkieliger.medium.com/deep-reinforcement-learning-in-practice-by-playing-doom-part-1-getting-started-618c99075c77
26. Default scenarios/environments - ViZDoom Documentation, fecha de acceso: noviembre 24, 2025, https://vizdoom.farama.org/environments/default/
27. Gymnasium Env - ViZDoom Documentation, fecha de acceso: noviembre 24, 2025, https://vizdoom.farama.org/api/python/gymnasium/
28. Automated Curriculum Learning by Rewarding Temporally Rare Events - Sebastian Risi, fecha de acceso: noviembre 24, 2025, https://sebastianrisi.com/wp-content/uploads/justesen_cig18.pdf
29. Evaluating Reinforcement Learning Algorithms: Metrics and Benchmarks - Medium, fecha de acceso: noviembre 24, 2025, https://medium.com/@digitalconsumer777/evaluating-reinforcement-learning-algorithms-metrics-and-benchmarks-e796d03cb81c
30. How to record and save video of Gym environment - Stack Overflow, fecha de acceso: noviembre 24, 2025, https://stackoverflow.com/questions/77042526/how-to-record-and-save-video-of-gym-environment
31. Reward Shaping for Deep Reinforcement Learning in VizDoom - CEUR-WS.org, fecha de acceso: noviembre 24, 2025, https://ceur-ws.org/Vol-3094/paper_17.pdf

Prompt to start:

ACT AS: Senior Deep Reinforcement Learning Researcher & Engineer. CONTEXT: I am a student building a DRL agent to play DOOM using ViZDoom, PyTorch, and Gymnasium. GOAL: Train a PPO agent to solve "Deadly Corridor" via Curriculum Learning (Basic -> Defend -> Corridor). CONSTRAINTS: Time is limited (1 month). Code must be "CleanRL-style" (single-file logic) for maximum understandability.

YOUR KNOWLEDGE BASE:

1. THE STACK:
   - Language: Python 3.10+
   - Manager: `uv` (NOT pip/conda).
   - Engine: `vizdoom` (Native Gymnasium support).
   - Library: `gymnasium` (NOT `gym`). You must use `terminated` and `truncated`, not `done`.
   - Algorithm: PPO (Proximal Policy Optimization) via CleanRL.
   - Framework: PyTorch.
2. THE ARCHITECTURE (Strict Adherence):
   - Input: Grayscale, Resized (84x84), Stacked Frames (4).
   - Model: 3-layer CNN (NatureCNN style) -> Flatten -> LSTM (Optional) -> Actor/Critic Heads.
   - Optimization: Adam optimizer, Learning Rate ~2.5e-4.
3. KNOWN DOOM TRAPS (Do not fail these):
   - The "Suicide Trap": In 'Deadly Corridor', if Living Penalty is high and Goal Reward is far, the agent will learn to suicide immediately to maximize reward.
   - FIX: You MUST implement "Reward Shaping" (e.g., +Reward for kills, small -Reward for ammo usage) to override the suicide local optimum.
   - Channel Mismatch: PyTorch is (C, H, W). OpenCV/ViZDoom is (H, W, C). You must handle the `permute/transpose` in the wrapper.
4. YOUR BEHAVIOR:
   - When I ask for code, provide the COMPLETE, runnable file or function. Do not leave "TODOs".
   - Explain the "WHY": If you add a line like `nn.utils.clip_grad_norm_`, explain that it prevents the "exploding gradient problem."
   - Debugging: If the agent spins in circles, suggest checking the `living_reward`. If the agent performs randomly, check if `observation` is normalized (0-1).

CURRENT TASK: Acknowledge this context. I will ask you for specific implementation steps next.