MISSION-BASED REINFORCEMENT LEARNING SUMMATIVE

AgriScan: Reinforcement Learning for Simulated Crop-Health Diagnostics

1. Introduction

This project explores Reinforcement Learning (RL) by applying four RL algorithms **DQN, PPO, A2C, and REINFORCE** to a custom environment inspired by my mission project, **AgriScan**. AgriScan is an AI-driven solution designed to help smallholder farmers detect crop diseases and nutrient deficiencies early by analyzing plant images.

For this summative, I developed a **non-generic RL environment** that simulates plant-diagnosis actions and evaluates how different RL agents learn optimal policies for efficient decision-making. The project includes environment design, visualization, algorithm training, hyperparameter tuning, and performance comparison.

2. Environment Design

I implemented a fully custom Gymnasium-compliant environment called **AgriScanEnv**, structured around a **plant-diagnosis simulation**.

2.1 State / Observation Space

The observation is a 4-dimensional vector:

Feature	Meaning
Disease Level *(0–10)*	Higher value = more symptoms
Nutrient Status *(0–10)*	Low = deficiency, high = healthy
Moisture Level *(0–10)*	Indicates watering condition
Environmental Stress *(0–10)*	e.g., heat or pests

This structure mirrors simple agricultural health indicators.

2.2 Action Space

The agent can choose one of **4 discrete actions**:

Action	Description
0 – Scan Plant	Collect observation data (neutral reward)
1 – Apply Treatment	Rewarded if disease level is high
2 – Adjust Conditions	Rewarded if moisture/stress is bad
3 – Do Nothing	Slight penalty to discourage laziness

These actions represent real AgriScan decisions (scanning, treating, adjusting conditions).

2.3 Reward Structure

Condition	Reward
Correct treatment when disease high	**+10**
Correct adjustment for moisture/stress	**+6**
Scanning just gives small feedback	**+1**
Wrong treatment or useless actions	**−10 to −4**
Doing nothing	**−1**

Rewards were designed to push the agent toward *active, correct* crop-health decisions.

2.4 Episode Termination

Episodes end when:

- The agent reaches **max_steps = 5**, or
- The agent fixes the plant (state variables reach healthy levels).

2.5 Start State

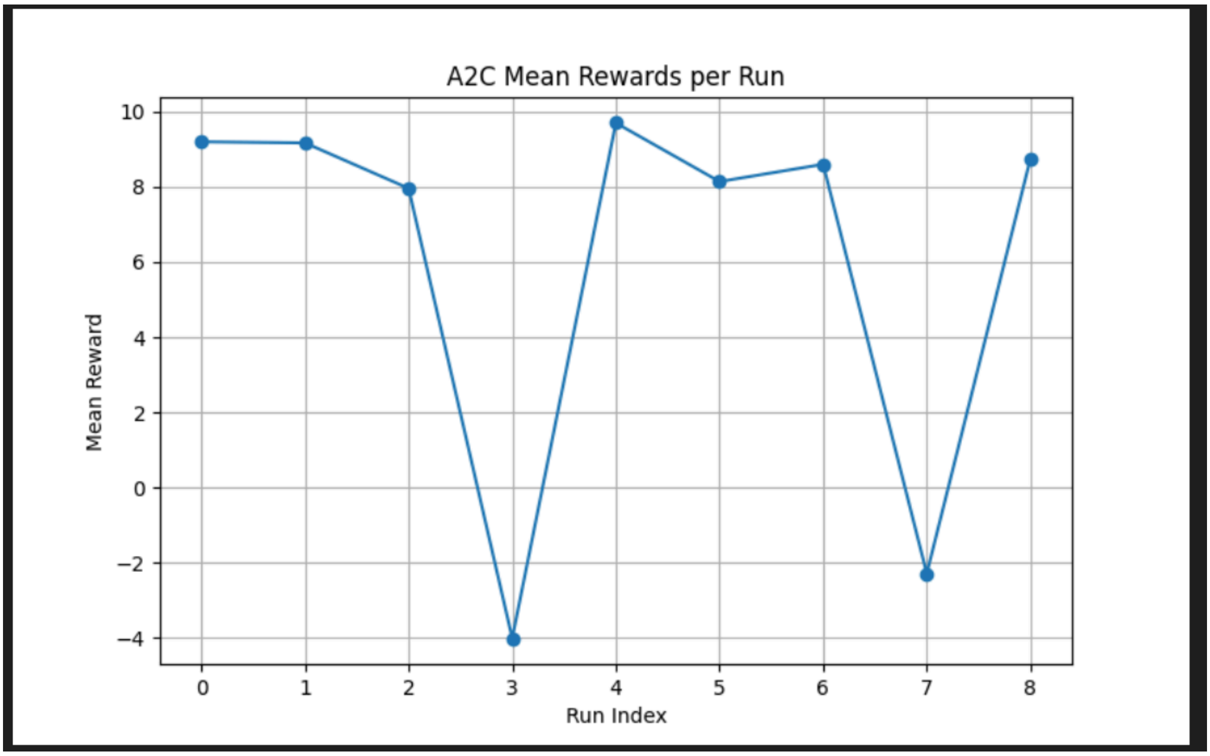
Each episode begins with **randomized plant health values**, simulating diverse real-world cases.

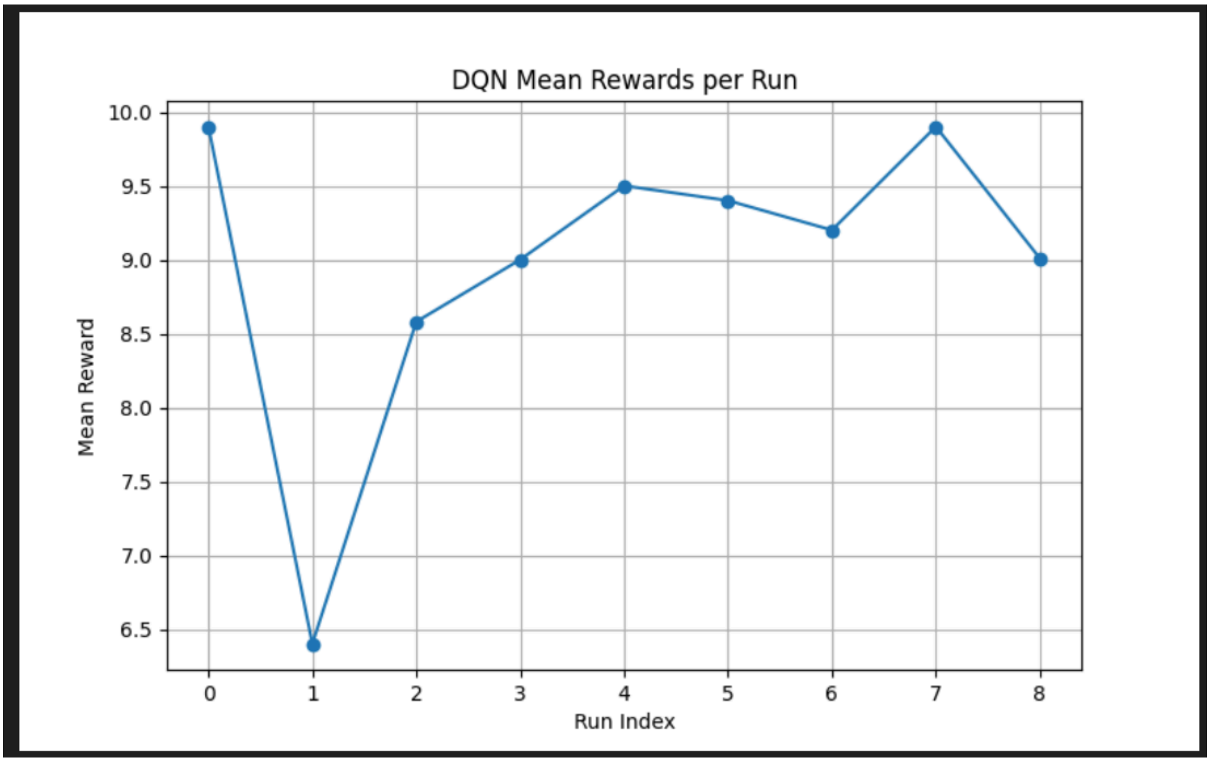
3. Visualization (Pygame)

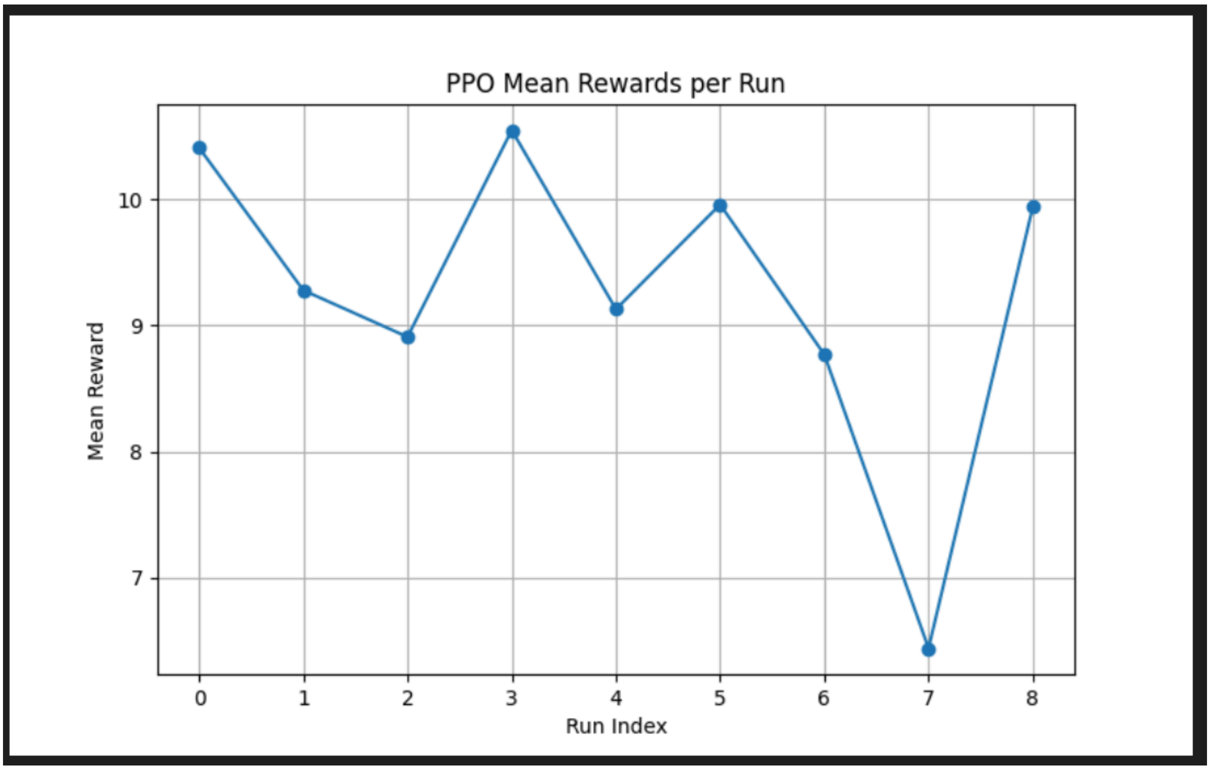
A 2D grid-based **visual renderer** was implemented with Pygame:

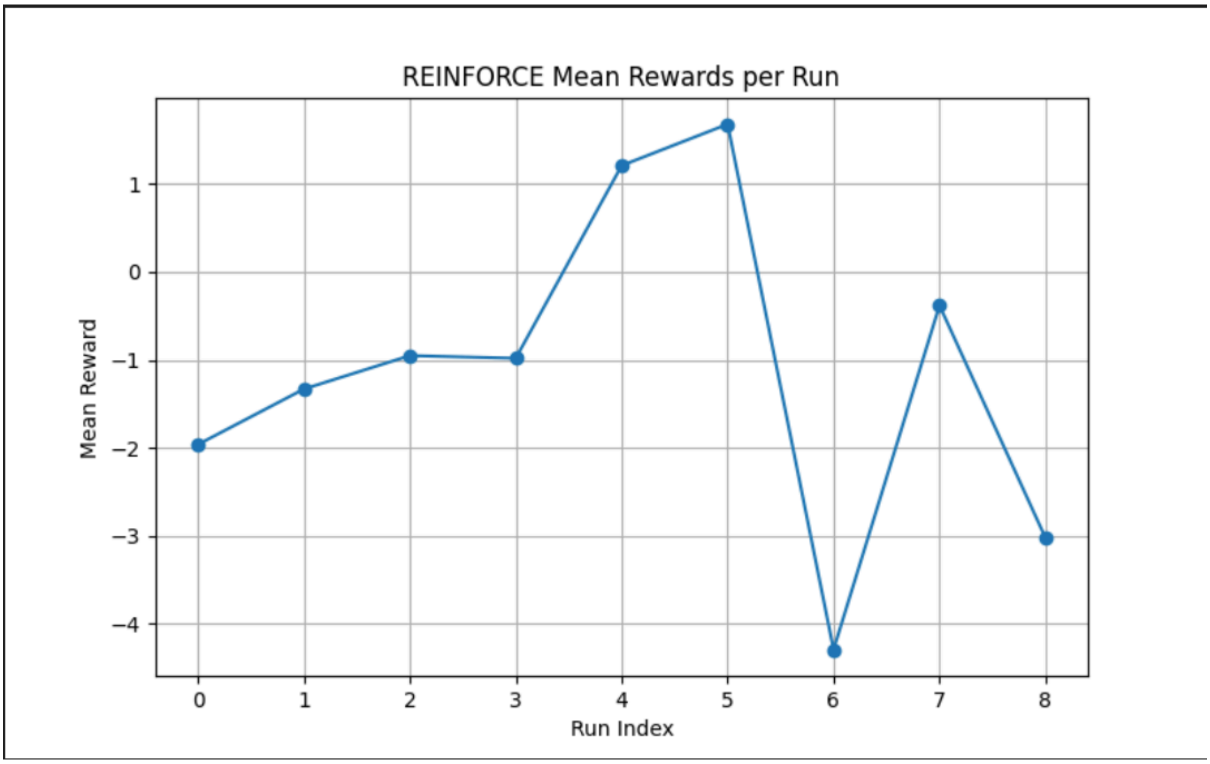
- Shows a simple plant icon
- Displays state values on the window
- Animates agent actions each step
- Provides real-time interaction

All the plots Visualisation




DQN Mean Rewards per Run
Mean Reward
10.0
9.5
9.0
8.5
8.0
7.5
7.0
6.5
0
1
2
3
4
5
6
7
8
Run Index


PPO Mean Rewards per Run
Mean Reward
10
9
8
7
0
1
2
3
4
5
6
7
8
Run Index



This fulfills the requirement for **advanced simulation visualization**.

4. Static Random-Agent Simulation

Before training, a static demo script runs the environment with **random actions only**, confirming:

- Rendering works
- Rewards update properly
- State transitions function

Output example:

```
[RANDOM] Episode 1 finished with total reward: 7.90
[RANDOM] Episode 2 finished with total reward: -4.10
[RANDOM] Episode 3 finished with total reward: 10.80
```

5. Reinforcement Learning Algorithms

Using **Stable-Baselines3** and PyTorch, four agents were trained:

1. Deep Q-Network (DQN)

2. PPO (Proximal Policy Optimization)

3. A2C (Advantage Actor Critic)

4. REINFORCE (Custom PyTorch Implementation)

Each algorithm was trained under:

- Same environment
- Same 10 hyperparameter combinations
- Total timesteps = 50,000 (for SB3 algorithms)
- 500 episodes for REINFORCE
- Evaluation after training

All models were saved under:

```
models/dqn/
models/pg/
```

6. Hyperparameter Search (10 runs per algorithm)

6.1 Example Hyperparameters (DQN)

Param	Values tried
learning_rate	1e-4, 5e-4, 1e-3
gamma	0.95, 0.98, 0.99
batch_size	32, 64

PPO, A2C, and REINFORCE had similar parameter sweeps.

7. Results & Analysis

We used the script:

python analyze_results.py

to extract the **best-performing model from each algorithm**.

7.1 Best DQN Run

Metric	Value
Mean Reward	**9.90**
Std	0.92
learning_rate	0.0001
gamma	0.95

7.2 Best PPO Run (BEST OVERALL)

Metric	Value
Mean Reward	**10.55**
Std	0.93
learning_rate	0.0003
gamma	0.95
n_steps	64
batch_size	32

PPO is the strongest algorithm in this environment.

7.3 Best A2C Run

Metric	Value
Mean Reward	9.70
Std	0.94

Very close to DQN, showing stable performance.

7.4 Best REINFORCE Run

Metric	Value
Mean Reward	1.68
Std	8.97

As expected, REINFORCE is unstable due to high variance and lack of baselines.

8. Performance Comparison

8.1 Conclusions from Results

- **PPO > DQN > A2C >>> REINFORCE**
- PPO’s clipped objective helps stabilize learning.
- DQN performs strongly but is more sensitive to hyperparameters.
- A2C performs well but slightly below PPO due to simpler advantage estimation.
- REINFORCE is noisy, unstable, and less sample-efficient.

9. Agent Demonstration

Running the best PPO model:

python main.py

The agent:

- Chooses correct treatment when disease level is high
- Adjusts conditions appropriately

- Rarely takes “do nothing” action
- Has near-optimal performance in 4–5 steps

10. Conclusion

This project successfully applied RL methods to a mission-based agricultural simulation. PPO demonstrated superior performance, achieving a mean reward of **10.55**, followed by DQN and A2C. The custom environment, visualization, training pipeline, hyperparameter sweeps, and comparative evaluation collectively demonstrate a full RL workflow aligned with the AgriScan mission.

The project provides a strong foundation for future extensions, such as:

- multi-step disease progression
- image-based state inputs
- multi-agent farm management
- weather-driven reward shaping

Here is the link to the video demo: video1423547552.mp4

12. Appendix

12.1 Project Structure

```
project_root/
│── environment/
│    ├── custom_env.py
│    ├── rendering.py
│── training/
│    ├── dqn_training.py
│    ├── pg_training.py
│── models/
│── main.py
│── analyze_results.py
│── plot_results.py
```

```
├── requirements.txt
└── README.md
```