Chapter 10
Visual DQN Navigation in Webots

Platform	Webots R2025a + Python
Learning framework	Gymnasium + Stable-Baselines3
Agent	DQN with CnnPolicy
Core observation	Forward RGB image, 3 × 84 × 84 uint8

This is a step-by-step implementation guide for completing Workshop 10. It explains what each part of the environment should do, why the code is structured, how to validate it, and how to train and evaluate the DQN.

This chapter follows the structure of the Workshop 10 lab: Part 1 defines the RL interface; Part 2 completes and validates the Webots Gymnasium environment; Part 3 trains and inspects the visual DQN; and Part 4 evaluates and diagnoses the learned policy. The official lab also specifies the required submission files and evidence.

Use this chapter together with the official Week 10 lab sheet.

Week 10 combines three components you have already seen separately: Webots robot control, camera-based representation, and reinforcement learning. Your main engineering task is not to implement DQN mathematics from scratch. Instead, you should build a correct interface between a Webots simulation and a Gymnasium learning algorithm, then train and evaluate the resulting policy.

The complete pipeline

```
Webots camera
     ↓
RGB image
     ↓
Gymnasium observation
     ↓
DQN policy
     ↓
action ∈ {left, forward, right}
     ↓
wheel velocities
     ↓
Webots simulation
     ↓
reward + termination information
     ↓
replay learning
```

The policy sees only the forward RGB image. Simulator information such as robot pose, goal coordinates and obstacle geometry is used by the environment for reset, reward, collision detection and evaluation, but it should not be supplied as an additional policy input.

> Environment information is allowed to make the simulation measurable and trainable. It is not automatically allowed to become part of the agent's observation. If the policy receives the goal coordinates directly, it is no longer a visual-only navigation policy.

What you will complete

- Validate the observation and action contract.
- Complete the Gymnasium reset and step logic.
- Make the environment pass Gymnasium's environment checker.
- Run a short scripted episode before training.
- Train a DQN with Stable-Baselines3.
- Save checkpoints and training logs.
- Evaluate the frozen policy on reproducible training and held-out scenarios.
- Diagnose one failure using evidence rather than guessing.

Working order

1. Make the environment correct before touching DQN training.
2. Test individual actions and observations.
3. Validate reset(), then validate step().
4. Run check_env() and fix every reported issue.
5. Run one manually scripted episode.
6. Only then start DQN training.
7. Inspect training behaviour before claiming success.
8. Run deterministic evaluation on fixed seeds.
9. Diagnose a failure from recorded evidence.

Part 1: Inspect the Task and Define the RL Interface

Start by understanding the interface before writing or changing code. The lab specifies an 84 × 84 RGB observation in channel-first format and a three-action discrete action space. The code we provided also defines the action duration and robot speed.

Prepare the Python environment

1. Activate the course Conda environment. Use the same environment you configured in the earlier Webots workshops.

2. Install the RL packages if needed. The workshop requires Gymnasium, Stable-Baselines3 and TensorBoard.

3. Verify the imports. Run a small Python command that imports gymnasium and stable_baselines3 before opening Webots.

4. Check Webots' Python interpreter. Make sure Webots is using the same Python environment in which these packages are installed.

✓ **Check:** The interpreter path printed by Python should match the interpreter configured in Webots.

If Gymnasium or Stable-Baselines3 cannot be imported, stop here and fix the Python environment first. Otherwise later errors can look like environment bugs when they are actually package or interpreter mismatches.

Understand the three actions

DQN is being used with a discrete action space, so the robot controller exposes three high-level choices rather than arbitrary continuous wheel velocities.

Action	Meaning	Wheel behaviour
0	Turn left	Left wheel reverses/slow relative to right wheel.
1	Move forward	Both wheels move forward at the configured speed.
2	Turn right	Right wheel reverses/slow relative to left wheel.

The important idea is that the DQN does not need to understand wheel speeds directly. It chooses a discrete action, and the environment translates that action into robot motion.

Understand the camera observation

```
# Conceptual stages, not a complete implementation
Webots BGRA buffer
        ↓
OpenCV colour conversion
        ↓
RGB image
        ↓
resize to 84 × 84
        ↓
channel-first layout
(C, H, W)
        ↓
uint8 observation
```

- Webots provides four-channel image data in the camera buffer.
- OpenCV is used to convert BGRA to RGB.
- The image is resized to 84 × 84.
- The axes are transposed from height × width × channels to channels × height × width.
- The final array remains uint8 in [0, 255].

Why keep uint8?
The lab keeps the observation as uint8 because Stable-Baselines3's CnnPolicy performs its standard image normalisation. Do not casually add another normalisation step unless you understand the resulting input range.

Your Part 1 checks

- Open the Week 10 Webots world and identify the e-puck, camera, green goal and obstacles.
- Run each of the three actions separately and observe the wheel behaviour.
- Print the observation shape, dtype, minimum and maximum.
- Write down what information is visible to the policy and what information remains privileged environment state.
- Record the Webots basic timestep and the action-repeat value.

Part 2: Build the Gymnasium Environment Correctly

This is the most important programming part of the workshop. Before an RL algorithm can learn, the environment should obey Gymnasium's interface exactly. Think of reset() and step() as the contract between your simulator and Stable-Baselines3.

Declare the spaces

```
self.observation_space = spaces.Box(
    low=0, high=255,
    shape=(3, 84, 84),
    dtype=np.uint8
)

self.action_space = spaces.Discrete(3)
```

The declared spaces are promises. Every observation returned by reset() and step() should satisfy the observation space, and every action accepted by step() should satisfy the action space.

Do not declare one shape and return another. For example, (84, 84, 3) is not the same contract as (3, 84, 84). A mismatch here can stop training before learning even begins.

Understand the environment state

Variable / component	Purpose	Policy input?
Camera image	Visual observation	YES
Robot x/y pose	Distance, collision and evaluation	NO

Variable / component	Purpose	Policy input?
Goal x/y position	Distance, reward and evaluation	NO
Obstacle geometry	Collision detection	NO
Previous distance	Progress reward	NO
Step count	Timeout logic	NO

Implement reset() in the right order

A reliable reset should establish a clean episode, choose a reproducible scenario, restore the robot and goal, advance Webots enough for the state to settle, initialise episode bookkeeping, and return the first camera observation.

1. Call the Gymnasium reset. Use super().reset(seed=seed). This establishes the environment's reproducible random-number generator.

2. Stop the motors. A new episode should not inherit motion from the previous episode.

3. Select the training or held-out layout. The supplied environment supports separate scenario banks so evaluation can test generalisation.

4. Sample the start and goal. Use the environment's seeded generator rather than Python's unrelated global random state.

5. Reset robot and goal pose. Write the sampled values into the Webots fields used by the Supervisor.

6. Reset physics and advance Webots. Allow the simulator to process the reset before requesting the first useful observation.

7. Initialise bookkeeping. Set step_count to zero and store the initial goal distance.

8. Return observation and info. The observation should satisfy observation_space immediately after reset.

```
# Skeleton only: fill in the project-specific operations
def reset(self, *, seed=None, options=None):
    super().reset(seed=seed)
    self.stop_motors()

    # choose layout
    # sample scenario with self.np_random
    # reset robot and goal
    # reset physics
    # advance Webots

    self.step_count = 0
    self.previous_distance = self.goal_distance()

    observation = self.read_observation()
    info = {...}
    return observation, info
```

Why seeded randomness matters

If the same seed does not reproduce the same scenario, debugging and fair evaluation become much harder. Reproducibility is an engineering requirement, not just a research nicety.

- Implement step() Without Breaking the Contract

The step() method is the heart of the simulation-learning interface. One RL action should not necessarily correspond to one Webots timestep. The workshop uses action repeat so that each DQN decision controls the robot for several simulator steps.

Think in two time scales

```
DQN decision
     ↓
apply action
     ↓
Webots step × ACTION_REPEAT
     ↓
check collision / goal after each simulator step
     ↓
read next image
     ↓
calculate reward
     ↓
return transition
```

This distinction is important. The DQN operates in environment steps, while Webots advances in smaller simulator timesteps. The action-repeat value determines how long each learned action is held.

The action-repeat loop

```
self.apply_action(int(action))

for _ in range(self.action_repeat):
    # advance Webots
    # check collision
    # check goal
    # stop early if an episode-ending event occurs
```

1. Convert the action to an integer. This makes the action compatible with the discrete action mapping.

2. Apply the corresponding wheel velocities. The environment, not the DQN, translates the discrete choice into differential-drive motion.

3. Advance Webots repeatedly. Use the configured action-repeat count.

4. Check collision and goal inside the loop. An event can occur before the full action duration finishes.

5. Break early on collision or success. Do not continue driving after the episode has already ended.

Suppose the robot reaches the goal on the second simulator step but the action-repeat value is six. Waiting until all six steps finish would let the robot continue moving after success. The same issue occurs for a collision.

Calculate the reward from the post-action state

The design combines four simple components: progress toward the goal, a small per-step cost, a goal bonus, and a collision penalty.

```
progress = previous_distance - distance

reward_progress = 2.0 * progress
reward_step = -0.01
```

```
reward_goal = 5.0 if reached_goal else 0.0
reward_collision = -1.0 if collision else 0.0

reward = (
    reward_progress
    + reward_step
    + reward_goal
    + reward_collision
)
```

The progress term is positive when the robot reduces its distance to the goal and negative when it moves farther away. The step penalty discourages unnecessarily long episodes. The goal and collision terms encode important task outcomes.

Understand terminated versus truncated

Flag	Meaning here	Typical cause
terminated	The episode ended because the task/environment produced a terminal event.	Goal reached or collision
truncated	The external episode limit ended the episode without a terminal event.	Maximum step count

A timeout is not the same thing as reaching the goal or colliding. Gymnasium distinguishes these cases so learning algorithms can treat the transition correctly.

Stop the robot at episode boundaries

When an episode ends, the motors should be stopped. This prevents residual motion from carrying into the next environment operation and makes the simulation behaviour easier to inspect.

- Validate Before Training

Do not start a 50,000-step DQN run until the environment has passed basic validation. Training is too slow and too opaque to be used as your first debugging tool.

Use Gymnasium's environment checker

```
from stable_baselines3.common.env_checker import check_env

env = WebotsNavigationEnv()
check_env(env, warn=True)
```

check_env() checks the interface assumptions expected by Gymnasium and Stable-Baselines3. Treat every error as something to resolve before training.

Validate one reset manually

```
observation, info = env.reset(seed=9)

print("shape:", observation.shape)
```

```
print("dtype:", observation.dtype)
print("min:", observation.min())
print("max:", observation.max())
print("action space:", env.action_space)
```

For the workshop contract, you should see a three-channel 84 × 84 uint8 image, values in the expected byte range, and a three-choice discrete action space.

Run a scripted episode

Before asking DQN to learn, drive the robot with a simple hand-written action sequence. The exact sequence is less important than the evidence you collect.

- Reset with a fixed seed.
- Print the selected scenario and initial distance.
- Apply a few forward actions.
- Apply a few turning actions.
- Print action, reward, distance and boundary flags after each step.
- Confirm that the robot stops when the episode ends.

If the robot moves incorrectly, the problem is probably in the action mapping or Webots control. If the image has the wrong shape, the problem is in observation processing. If rewards behave strangely, inspect distance and boundary logic. Fix the earliest broken layer before moving on.

A practical validation checklist

Check	Expected	Status
Observation shape	(3, 84, 84)	☐
Observation dtype	uint8	☐
Observation range	0–255	☐
Action space	Discrete(3)	☐
Reset reproducibility	same seed → same scenario	☐
Goal detection	episode ends on success	☐
Collision detection	episode ends on collision	☐
Timeout	truncated=True only	☐
Motor stop	zero velocity at boundary	☐
check_env()	no errors	☐

Part 3: Train the Visual DQN

Once the environment is validated, Stable-Baselines3 can provide the DQN implementation. You do not need to implement the neural network, replay buffer, target network, epsilon-greedy schedule or gradient optimisation yourself for this workshop.

Understand what Stable-Baselines3 is doing

Component	Role	Why it matters
CnnPolicy	Processes image observations	Learns visual features
DQN	Learns action values	Chooses among three actions
Replay buffer	Stores past transitions	Reuses expensive simulator experience
Target network	Provides a more stable target	Reduces rapid feedback instability
Exploration schedule	Chooses exploratory actions	Allows the agent to discover behaviour
Monitor	Records episode statistics	Supports plots and diagnosis
CheckpointCallback	Saves intermediate models	Allows early/final comparison

Build the training setup

```
from stable_baselines3 import DQN
from stable_baselines3.common.callbacks import CheckpointCallback
from stable_baselines3.common.monitor import Monitor

env = Monitor(WebotsNavigationEnv())

checkpoint_callback = CheckpointCallback(
    save_freq=10_000,
    save_path="models/checkpoints/",
    name_prefix="dqn_week10"
)

# Configure DQN using the workshop settings.
# Then call model.learn(...) and save the final model.
```

Use the lab's parameter values as your workshop baseline. Treat them as a starting configuration, not as universal optimal settings. The important learning exercise is understanding what the settings control and interpreting the resulting behaviour.

Parameters worth understanding

Parameter	Workshop role	Interpretation
learning_rate	1e-4	Step size for optimisation
buffer_size	50,000	Maximum stored transitions

Parameter	Workshop role	Interpretation
learning_starts	2,000	Delay before gradient learning begins
batch_size	32	Transitions used per optimisation step
gamma	0.99	Discount applied to future value
target_update_interval	1,000	Frequency of target-network updates
exploration_fraction	0.25	Fraction of training used to decay exploration
exploration_final_eps	0.05	Final exploration probability
seed	9	Reproducibility

The workshop notes that 50,000 interactions may show improvement without producing stable convergence. Training return can also improve while task success remains poor because exploration and reward shaping affect the return.

- Inspect Training Instead of Watching Only the Reward

A common beginner mistake is to start training, see the reward curve rise, and conclude that the robot has learned navigation. The lab explicitly asks you to inspect trajectories and task-level outcomes.

Save two checkpoints

- Keep an early checkpoint from the training run.
- Keep the final checkpoint.
- Use the same evaluation scenarios later so their behaviour can be compared fairly.

Plot useful training information

The workshop asks you to inspect episode return and episode length against environment interactions. These plots answer different questions.

Metric	What it can tell you
Episode return	Whether the reward signal is changing during learning.
Episode length	Whether episodes tend to finish sooner or remain long.
Success rate	Whether the robot actually reaches the task goal.
Collision rate	Whether the policy is becoming safer.
Timeout rate	Whether the policy often fails to finish.

Inspect actual trajectories

Watch at least two runs. Record whether the robot spins, zig-zags, stalls, collides, reaches the goal, or repeatedly performs a visually similar but ineffective action sequence.

A policy can obtain positive progress reward while still failing to navigate reliably. The final question is not 'Did the reward increase?' but 'Does the robot reliably complete the navigation task?'

Understanding replay learning

The simulator produces transitions such as observation → action → reward → next observation. The replay buffer stores these experiences so DQN can sample batches from previously observed transitions instead of learning only from the newest interaction.

This is especially useful here because Webots interactions have a real computational cost. Reusing experience improves the amount of learning obtained from each simulator interaction.

Understanding the target network

DQN uses a separate target network to provide a more stable learning target. The target parameters are updated periodically rather than changing at every optimisation step. This reduces one source of rapid feedback between the values being predicted and the values used to train them.

Part 4: Evaluate the Frozen Policy Properly

Evaluation is where you determine whether the learned policy actually works. The workshop deliberately uses fixed seeds and separates training-layout scenarios from held-out scenarios.

Why fixed seeds?

If the random policy and DQN see different scenarios, their results are difficult to compare. Using identical seeds makes the comparison controlled and reproducible.

Why held-out scenarios?

The training scenarios and held-out scenarios are different. Held-out evaluation tests whether the policy learned a useful visual-navigation behaviour rather than simply memorising the exact training layouts.

Evaluation	Seeds	Purpose	Learning?
Random policy	same training seeds	Baseline	No
DQN training set	101–110	Performance on training-layout distribution	No
DQN held-out set	201–210	Generalisation	No

Deterministic prediction

```
action, _ = model.predict(
    observation,
    deterministic=True
)
```

During evaluation, deterministic=True removes the training exploration behaviour. You are measuring the frozen policy itself, not a mixture of policy and exploration.

Record task-level metrics

Metric	Report?	Meaning	Better direction
Success rate	YES	Episodes reaching the goal	Higher
Collision rate	YES	Episodes ending in collision	Lower
Timeout rate	YES	Episodes reaching step limit	Lower
Mean episode length	YES	Average environment steps	Interpret with success

For the final robot task, success, collision and timeout rates are more directly meaningful than cumulative training reward. Reward is the learning signal; task success is the outcome you ultimately care about.

Appendix

Diagnose Failure Like an Engineer

The workshop asks you to choose one failed DQN episode and identify the earliest layer where the system stopped behaving as intended. This is an engineering diagnosis exercise, not an invitation to guess a hyperparameter.

Use the failure-layer model

```
Visual observation
      ↓
Action selection / timing
      ↓
Reward design
      ↓
Exploration / training
      ↓
Evaluation logic
```

Start from the earliest plausible failure. For example, if the camera observation is malformed, changing the learning rate will not solve the problem.

Layer	Evidence to inspect	Possible symptom
Visual observation	shape, dtype, image sequence	Policy cannot use expected visual information
Action timing	actions and motion after each step	spins, overshoots, sluggish control
Reward	reward components and distance	reward rises without useful behaviour
Training	checkpoint comparison, exploration	learning unstable or insufficient
Evaluation	determinism, seeds, preprocessing	training works but evaluation is inconsistent

Build an evidence chain

- State exactly what failed: for example, the robot repeatedly timed out.
- Identify what the logs and trajectory show.
- Locate the earliest stage that could explain the evidence.
- Make one targeted change.
- Explain why that change addresses the observed evidence.
- Re-run the same evaluation scenario to see whether the diagnosis is supported.

Changing the reward, network settings, action timing and evaluation code simultaneously destroys your ability to identify what actually fixed the problem.

Example of good reasoning

Suppose the robot repeatedly turns left and never approaches the goal. First inspect the action mapping and confirm that action 0 really produces a left turn. Then inspect whether the camera observation is valid. Only if those layers are correct should you investigate whether the DQN has learned a poor policy.

Common Problems and Debugging Guide

Symptom	What to check	Layer
ImportError for gymnasium / stable_baselines3	Check the active Conda environment and the Python executable used by Webots. Install packages into that exact environment.	Python environment mismatch
check_env() reports observation mismatch	Print shape, dtype, min and max immediately after reset() and step(). Compare them with observation_space.	Observation contract
DQN starts but robot does not move correctly	Test actions 0, 1 and 2 manually before training. Inspect left/right motor velocities.	Action mapping
Robot keeps moving after success/collision	Check the action-repeat loop and ensure the episode breaks immediately after the event. Also stop motors at boundaries.	Boundary handling
Timeout is reported as termination	Separate terminated from truncated. Goal/collision should terminate; non-terminal step limit should truncate.	Gymnasium episode semantics
Same seed gives different reset behaviour	Use super().reset(seed=seed) and sample scenarios from self.np_random.	Reproducibility
Reward increases but navigation is poor	Inspect progress reward, trajectory behaviour and task success. Do not judge the agent from return alone.	Reward interpretation
Training is very slow	Run Webots in the fastest practical mode, reduce unnecessary visualisation, and record wall-clock time. Do not silently change the workshop protocol when comparing results.	Simulation cost
DQN evaluation is inconsistent	Use deterministic=True and the same image preprocessing and action timing as training.	Evaluation mismatch
Held-out performance is much worse	Compare training-set and held-out trajectories. Consider whether the learned behaviour is tied to the training scenario distribution.	Generalisation

How to Use AI Effectively in This Workshop

AI can be useful for debugging and explanation, but the goal is to understand and verify the environment rather than outsource the entire implementation.

Good uses of AI

- Ask what a Gymnasium error message means and what evidence to inspect.
- Ask for an explanation of terminated versus truncated.
- Ask why a tensor/image shape is incompatible with CnnPolicy.
- Ask for help interpreting a reward curve or evaluation table.
- Ask for a small debugging snippet that prints action, distance and reward components.
- Ask AI to review your own code and identify possible contract violations.

Poor uses of AI

- Copying a complete environment implementation without understanding reset() and step().
- Accepting a proposed reward function without checking whether it matches the workshop task.
- Changing several hyperparameters because an AI says they might improve training.
- Reporting training results that you did not actually run.
- Using simulator state as a hidden policy input because an AI suggested it.

Ask AI for an explanation → make one small change → run the code → inspect the evidence → decide whether the explanation was correct. The experiment, not the AI response, is the final authority.

Useful prompts

```
Explain why my Gymnasium environment is returning an
observation that does not match observation_space. Here is
the printed shape, dtype and range: ...

My robot reaches the goal but the episode is marked as
truncated. Here are step_count, distance, terminated and
truncated from the final step. Diagnose the boundary logic.

My DQN reward increases but success remains near zero.
Here are the training curve and evaluation results.
What evidence should I inspect before changing hyperparameters?
```

Submission and Evidence Checklist

The workshop submission asks you to provide both the implementation and evidence that the environment and learned policy were tested.

Code archive

- Completed Webots/Gymnasium environment for Parts 2–4.
- Training script.
- Evaluation script.
- Supporting files required to reproduce the run.
- No machine-specific absolute paths.

Result.pdf or Result.docx

- Observation shape, dtype, minimum and maximum checks.
- Action-space verification.
- check_env() result.
- Training curves for episode return and episode length.
- Training wall-clock time and relevant training information.
- Early and final checkpoint comparison.
- Evaluation table for random policy, DQN training set and DQN held-out set.
- Representative trajectory screenshots or other useful run evidence.
- One failed DQN episode with evidence-based diagnosis.
- One targeted improvement and justification.

Final quality check

Final question	Done
Does reset() return a valid observation every time?	☐
Does step() return the correct five outputs?	☐
Are collision and goal checked during action repeat?	☐
Are terminated and truncated distinguished correctly?	☐
Does the environment pass check_env()?	☐
Can I explain every reward component?	☐
Can I reproduce evaluation with fixed seeds?	☐
Did I evaluate both training and held-out scenarios?	☐
Did I inspect actual robot trajectories?	☐
Can I justify my failure diagnosis with evidence?	☐

Quick Reference

Environment contract

```
Observation: Box(0..255, shape=(3,84,84), dtype=uint8)
Action:      Discrete(3)

reset() → observation, info

step(action) →
    observation,
    reward,
    terminated,
    truncated,
    info
```

Episode logic

```
action
  ↓
wheel speeds
  ↓
Webots × action_repeat
  ↓
collision / goal checks
  ↓
new image
  ↓
distance + reward
  ↓
terminated / truncated
  ↓
return transition
```

Training logic

```
validated environment
        ↓
Monitor(...)
        ↓
DQN("CnnPolicy", ...)
        ↓
learn(...)
        ↓
checkpoints + logs
        ↓
frozen model
        ↓
deterministic evaluation
```

Evaluation logic

```
Random policy       → training seeds
DQN final model      → training seeds
DQN final model      → held-out seeds
                         ↓
             success / collision /
             timeout / mean steps
                         ↓
               failure diagnosis
```

A reinforcement-learning result is only as trustworthy as the interface around the learner. Correct observations, correct actions, correct episode boundaries, reproducible scenarios, logged training, and task-level evaluation are all part of the system.

End of Chapter 10