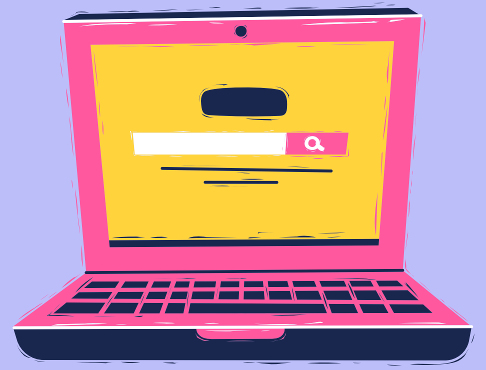
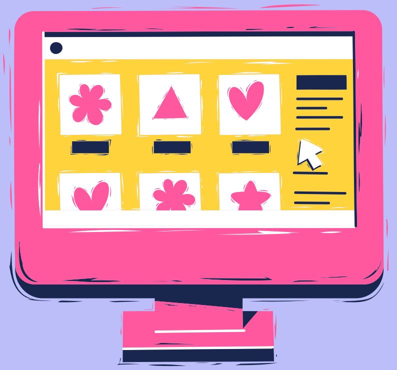
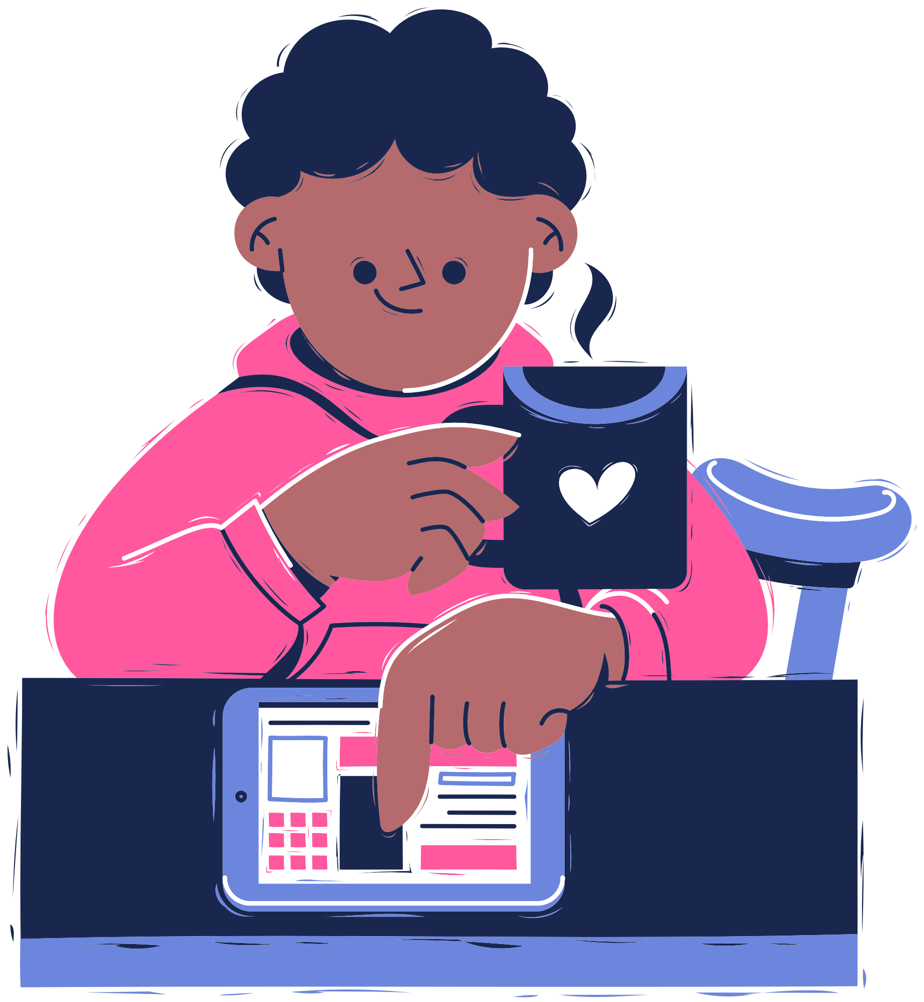




DESIGN OF INTELLIGENT CONTROLLERS USING REINFORCEMENT LEARNING FOR CONTROL APPLICATIONS

TABLE OF CONTENTS



INTRODUCTION

DC motors are widely used in automation due to their simplicity and reliability. While PID controllers are commonly used for speed control, they struggle with nonlinearities and disturbances. Reinforcement Learning (RL), such as Deep Q-Networks (DQN), offers an adaptive solution by learning through interaction. This project compares PID and DQN controllers in regulating a DC motor to a target speed of 50 rad/s.

PROBLEM STATEMENT

Classical PID controllers are widely used but have limitations:

- Require manual tuning
- Poor adaptability to disturbances or nonlinear dynamics

The effectiveness of RL-based controllers compared to PID in real-time control needs to be evaluated.

This project focuses on controlling the speed of a DC motor to 50 rad/s using both PID and DQN controllers and comparing their performance.

OBJECTIVES

Model and simulate a DC motor using differential equations

To understand the motor's behavior and accurately represent its dynamics in simulation.

Develop a DQN based reinforcement learning controller

To implement an intelligent control strategy that learns optimal actions through trial and error.

Analyze performance based on settling time, overshoot and MSE

To measure how fast, accurate and stable each controller is in reaching and maintaining the target speed.

METHODOLOGY

System Selection and Modeling	PID Controller Design	Performance Evaluation
Control Objective	Reinforcement Learning Controller (DQN)	

PID CONTROLLER DESIGN

```python
# ------------------------------
# 2. PID Controller
# ------------------------------
class PIDController:
    def __init__(self, Kp, Ki, Kd):
        self.Kp = Kp
        self.Ki = Ki
        self.Kd = Kd
        self.prev_error = 0
        self.integral = 0

    def compute(self, setpoint, measurement, dt):
        error = setpoint - measurement
        self.integral += error * dt
        derivative = (error - self.prev_error) / dt
        output = self.Kp * error + self.Ki * self.integral + self.Kd * derivative
        self.prev_error = error
        return output
```

```python
def run_pid(target_speed):
    J, b, K, R, L = 0.01, 0.01, 0.05, 0.5, 0.05
    dt, T = 0.02, 10.0
    steps = int(T / dt)
    omega, i, t = 0.0, 0.0, 0.0
    pid = PIDController(1, 2, 0.05)
    time_pid, speed_pid = [], []
    for _ in range(steps):
        V = pid.compute(target_speed, omega, dt)
        domega = (K * i - b * omega) / J
        di = (V - R * i - K * omega) / L
        omega += domega * dt
        i += di * dt
        t += dt
        time_pid.append(t)
        speed_pid.append(omega)
    return time_pid, speed_pid
```

- The controller is not trained, but is manually tuned by the user.

Kp (proportional gain) - 1.0
It decides how strongly the controller reacts to the current error

Ki (integral gain) - 2.0
It decides how much it considers past error.

Kd (derivative gain) - 0.05
It decides how it predicts future trends.

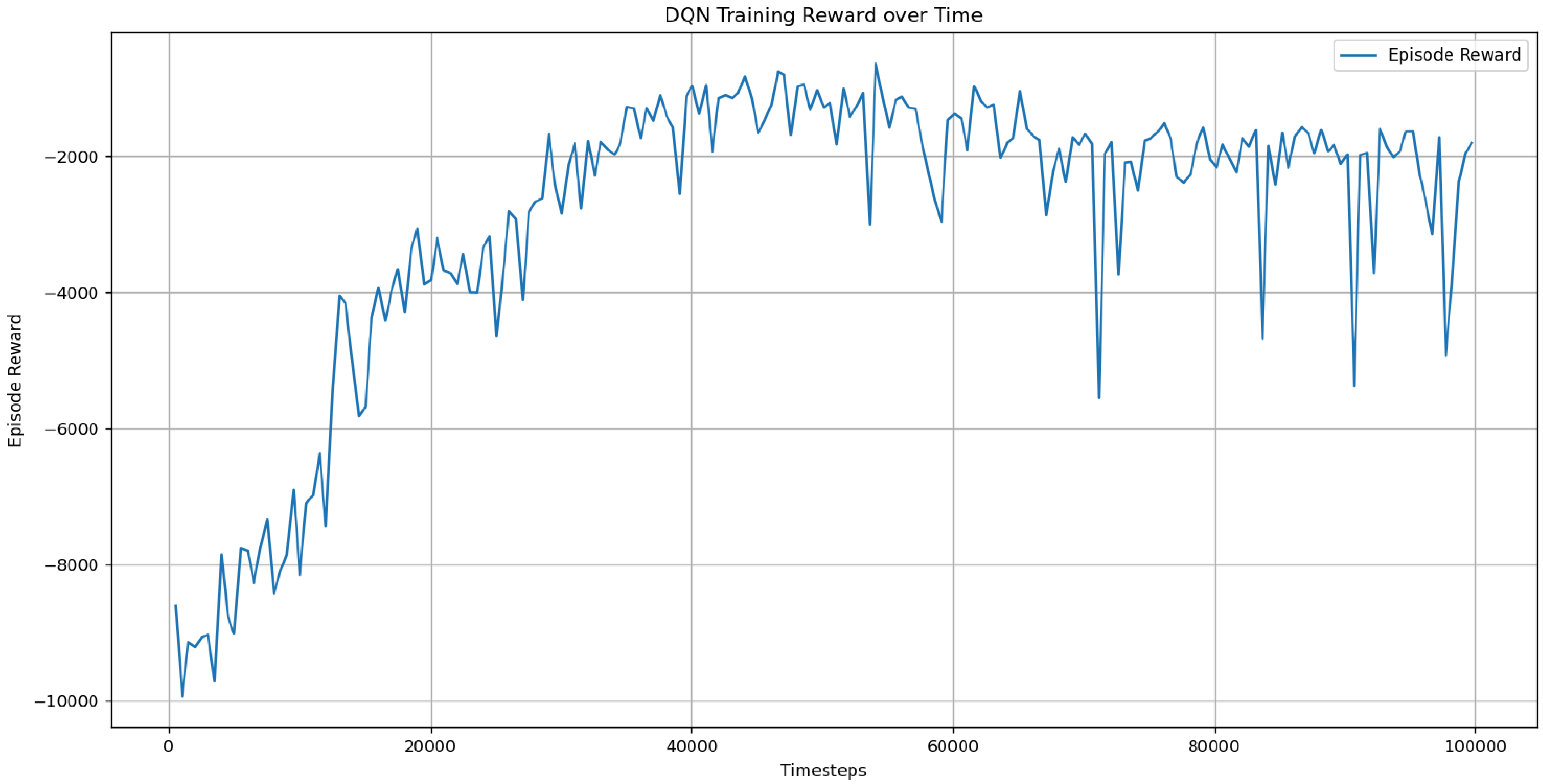
DQN CONTROLLER DESIGN

```python
# Create and train the DQN model
model = DQN(
    policy='MlpPolicy',
    env=env,
    verbose=1,
    learning_rate=1e-3,
    buffer_size=10000,
    learning_starts=1000,
    batch_size=64,
    gamma=0.99,
    train_freq=1,
    target_update_interval=250,
    exploration_fraction=0.3,
    exploration_final_eps=0.05,
)

# Train longer
model.learn(total_timesteps=100000, progress_bar=True)

# Save trained model
model.save("dqn_dc_motor")
```

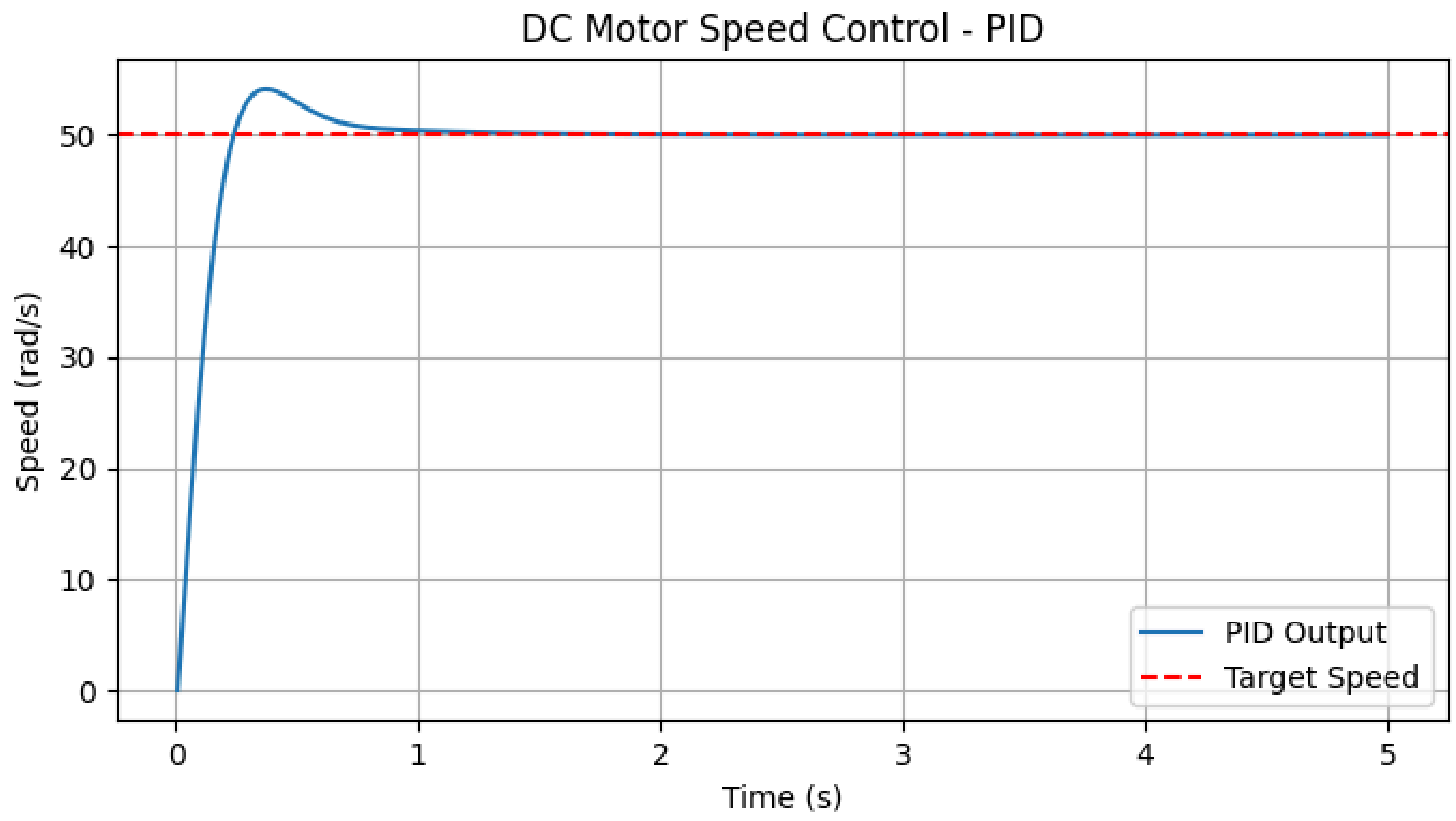
```python
# 3. Load trained DQN
# -----------------------------
model = DQN.load("dqn_dc_motor")

# -----------------------------
# 4. Simulation functions
# -----------------------------
def run_dqn(target_speed):
    env = DCMotorEnv()
    env.omega_target = target_speed
    obs, _ = env.reset()
    time_dqn, speed_dqn = [], []
    steps = int(env.max_time / env.dt)  # ✅ Fixed steps
    for _ in range(steps):
        action, _ = model.predict(obs, deterministic=True)
        obs, reward, terminated, truncated, _ = env.step(action)
        time_dqn.append(env.t)
        speed_dqn.append(obs[0])
    return time_dqn, speed_dqn
```

- It is a machine learning controller - learns by interacting with the environment.
- Uses experience replay: stores past (state, action, reward, next state) pairs.
- Uses a neural network to estimate Q-values (expected future rewards for each action).
- Updates its network by minimising the difference between predicted and actual rewards.

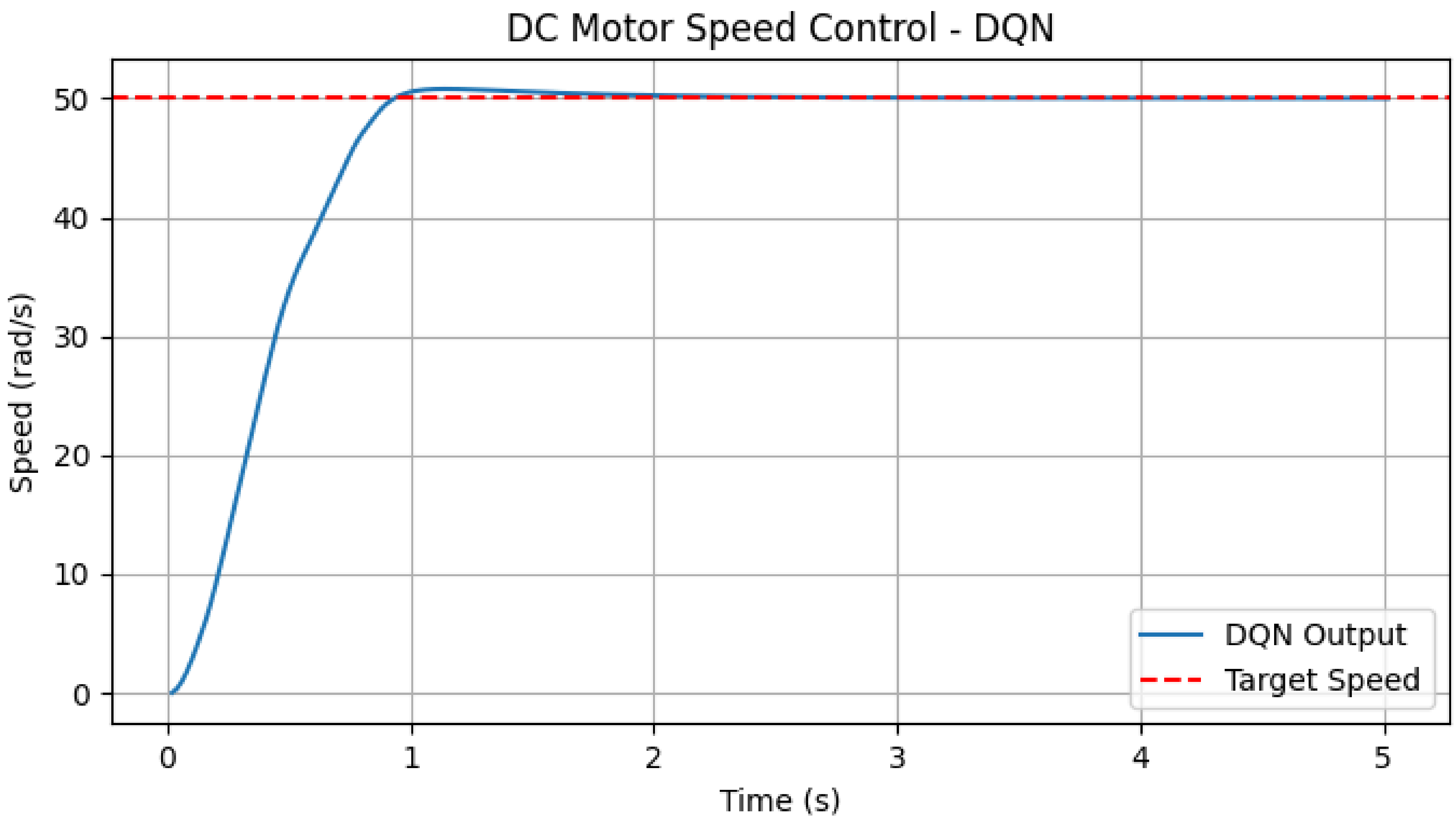
DQN TRAINING REWARD



SIMULATION OF PID CONTROLLER



SIMULATION OF DQN CONTROLLER



ANALYSIS

Performance Metrics

PERFORMANCE METRICS (PID)

Rise Time: 0.140 s
Overshoot: 9.07 %
MSE: 14.9980
Settling Time (approx.): 0.480 s

OK

Performance Metrics

PERFORMANCE METRICS (DQN)

Rise Time: 0.600 s
Overshoot: 1.60 %
MSE: 69.6488
Settling Time (approx.): 0.800 s

OK

COMPARISON

ASPECT	PID CONTROLLER	DQN CONTROLLER
Rise Time	Fast, reaches target before 1s	Slower, reaches target before 2s
Overshoot	High overshoot percentage	Low overshoot percentage
Settling Time	Quickly to settle	Takes longer to settle
Smoothness	Less smooth, aggressive rise and correction	Smoother, gradual rise without sharp corrections
Adaptability	Fixed	Adaptive, learns and improves over time
Computational Demand	Simple, only tune Kp,Ki,Kd	Complex, requires training episodes and reward design

CONCLUSION

In designing controller for DC motor speed, the PID controller outperformed the DQN controller by achieving a much faster rise time and quicker settling at the target speed of 50 rad/s. Although it exhibited some overshoot, the PID controller corrected the error rapidly and maintained accurate tracking with a lower overall MSE compared to the DQN. This shows that for this application, where fast and precise speed regulation is important, the PID controller is the more effective choice.