자율주행 RC카 개발 프로젝트 경력증명서

프로젝트 개요

배경

- 회사: 노마드랩
- 기간: 2022년 (6개월)
- 역할: IoT 교육용 자율주행 RC카 솔루션 개발 담당
- 목표: 교육용으로 판매 가능한 저비용 자율주행 RC카 시스템 개발

핵심 과제

AWS Rekognition과 같은 클라우드 비전 서비스는 실시간 카메라 스트리밍 비용이 높아, 초음파 센서만을 이용한 강화학습 기반 자율주행 시스템을 개발하여 비용 효율적인 교육용 솔루션 구현

1단계: 시뮬레이터 개발 (완료)

개발 내용

- **시뮬레이터 v1~v5**: 점진적 기능 개선
  - v1: 기본 격자 환경 및 Q-learning
  - v2: 초음파 센서 시뮬레이션 추가
  - v3: DQN(Deep Q-Network) 도입
  - v4: 리플레이 메모리 및 경험 재생
  - v5: 캐시 시스템 및 Policy > Cache 원칙 적용

기술 스택

- **언어**: Python 3.8+
- **프레임워크**: PyTorch (DQN 구현), Pygame (시각화)
- **알고리즘**: Deep Q-Network (DQN) 강화학습

주요 구현 기능

```
# 초음파 센서 시뮬레이션 (environment.py)
- 4방향 거리 측정 (전/후/좌/우)
- 격자 환경에서 장애물까지의 거리 계산
- 동적 장애물 지원

# DQN 에이전트 (agent.py)
- 신경망 기반 Q-value 예측
- 경험 재생 (Experience Replay)
- 타겟 네트워크를 통한 안정적 학습
- 실행 캐시 시스템 (ActionCache)

# RC카 제어 로직 (car.py)
```

```
- 3가지 행동: 직진, 우회전, 좌회전
- 충돌 감지 및 보상 시스템
- 점수 및 스텝 추적
```

성과

- 시뮬레이터 환경에서 90% 이상 목적지 도달 성공률 달성
- 다양한 맵에서 일반화 능력 검증 완료

2단계: 하드웨어 전환 계획 및 준비

하드웨어 구성 결정

선택한 하드웨어

1. **라즈베리파이 4 Model B (4GB RAM)**
   - 이유: Python 실행 환경, GPIO 핀 제공, PyTorch Lite 지원
   - 대안 고려: 아두이노 (메모리 부족으로 제외)
2. **초음파 센서: HC-SR04 x 4개**
   - 측정 범위: 2cm ~ 400cm
   - 각도: 15도 (좁은 빔으로 정확도 높음)
   - 배치: 전방, 후방, 좌측, 우측
3. **DC 모터 드라이버: L298N**
   - 2채널 모터 제어 (좌/우 바퀴)
   - PWM 속도 제어 지원
   - 과전류 보호 기능
4. **RC카 섀시: 2WD 로봇 카 키트**
   - DC 모터 2개 (좌/우 독립 제어)
   - 배터리: 18650 리튬이온 배터리 (2S 7.4V)
5. **기타 부품**
   - 브레드보드, 점퍼 와이어
   - 전원 분리 회로 (라즈베리파이 5V, 모터 7.4V)

비용 분석

항목	단가	비고
라즈베리파이 4	$55	교육용 할인 적용
초음파 센서 x4	$8	HC-SR04
모터 드라이버	$6	L298N

항목	단가	비고
RC카 섀시	$25	2WD 키트
기타 부품	$10	배선, 브레드보드 등
합계	**$104**	**AWS Rekognition 월 비용의 1/10**

3단계: 라즈베리파이 환경 설정

OS 및 기본 설정

```
# Raspberry Pi OS Lite 설치 (64-bit)
# 이유: GUI 불필요, 메모리 절약

# 기본 업데이트
sudo apt update && sudo apt upgrade -y

# Python 3.9 설치 확인
python3 --version  # Python 3.9.2

# 필수 패키지 설치
sudo apt install -y python3-pip python3-dev \
                    python3-numpy
```

추론 엔진 설치 (TensorFlow Lite)

```
# 중요: 라즈베리파이는 추론(inference)만 수행
# 학습(training)은 개발 PC에서 완료

# TensorFlow Lite Runtime 설치 (경량 추론 엔진)
pip3 install tflite-runtime

# 또는 ONNX Runtime (대안)
# pip3 install onnxruntime

# GPIO 라이브러리
pip3 install RPi.GPIO

# 메모리 사용량 확인
free -h  # 4GB 중 500MB 사용 (매우 여유 있음)
```

개발 워크플로우

```
[개발 PC에서 학습]
1. PyTorch로 DQN 학습 (GPU 사용)
2. model_final.pth 저장
```

```
3. 모델 변환: PyTorch → ONNX → TFLite
   python convert_model.py
   → model_final.tflite 생성

[라즈베리파이 배포]
1. 라즈베리파이에 파일 복사
   scp model_final.tflite pi@192.168.1.100:~/
   scp rc_car_main.py pi@192.168.1.100:~/
2. 추론 코드 실행
   python3 rc_car_main.py
```

트러블슈팅 #1: 모델 파일 호환성 문제

문제: 개발 PC에서 만든 .pth 파일을 라즈베리파이에서 직접 로드 시도 → 실패 **원인**: PyTorch 모델은 같은 버전의 PyTorch가 필요하고, 라즈베리파이에는 PyTorch 없음 **해결**: 모델을 TFLite 형식으로 변환하여 TFLite Runtime으로 실행

```python
# 개발 PC에서 실행 (convert_model.py)
import torch
import onnx
import tensorflow as tf

# 1. PyTorch → ONNX
model = torch.load('model_final.pth')
dummy_input = torch.randn(1, 8)
torch.onnx.export(model, dummy_input, 'model.onnx')

# 2. ONNX → TensorFlow
import onnx_tf
onnx_model = onnx.load('model.onnx')
tf_rep = onnx_tf.backend.prepare(onnx_model)
tf_rep.export_graph('model_tf')

# 3. TensorFlow → TFLite
converter = tf.lite.TFLiteConverter.from_saved_model('model_tf')
converter.optimizations = [tf.lite.Optimize.DEFAULT]
tflite_model = converter.convert()
with open('model_final.tflite', 'wb') as f:
    f.write(tflite_model)

# 결과: model_final.tflite (12MB, 라즈베리파이용)
```

4단계: 초음파 센서 하드웨어 연결 및 테스트

회로 설계

```
[라즈베리파이 GPIO 핀맵]
초음파 센서 (전방):
  - VCC  → 5V (Pin 2)
```

```
  - TRIG → GPIO 23 (Pin 16)
  - ECHO → GPIO 24 (Pin 18)
  - GND  → GND (Pin 6)

초음파 센서 (후방):
  - TRIG → GPIO 27 (Pin 13)
  - ECHO → GPIO 22 (Pin 15)

초음파 센서 (좌측):
  - TRIG → GPIO 5 (Pin 29)
  - ECHO → GPIO 6 (Pin 31)

초음파 센서 (우측):
  - TRIG → GPIO 13 (Pin 33)
  - ECHO → GPIO 19 (Pin 35)

공통 VCC → 5V, GND → GND
```

초음파 센서 인터페이스 구현

```python
# ultrasonic_sensor.py
import RPi.GPIO as GPIO
import time

class UltrasonicSensor:
    def __init__(self, trig_pin, echo_pin):
        self.trig_pin = trig_pin
        self.echo_pin = echo_pin

        GPIO.setmode(GPIO.BCM)
        GPIO.setup(self.trig_pin, GPIO.OUT)
        GPIO.setup(self.echo_pin, GPIO.IN)

        # 초기화
        GPIO.output(self.trig_pin, False)
        time.sleep(0.1)

    def get_distance(self, timeout=0.05):
        """
        거리 측정 (cm 단위)
        timeout: 최대 대기 시간 (초)
        """
        # 트리거 신호 전송
        GPIO.output(self.trig_pin, True)
        time.sleep(0.00001)  # 10us 펄스
        GPIO.output(self.trig_pin, False)

        # 에코 신호 수신 대기
        pulse_start = time.time()
        pulse_end = time.time()
```

```
        # ECHO 핀이 HIGH가 될 때까지 대기
        start_wait = time.time()
        while GPIO.input(self.echo_pin) == 0:
            pulse_start = time.time()
            if pulse_start - start_wait > timeout:
                return 400  # 타임아웃: 최대 거리 반환

        # ECHO 핀이 LOW가 될 때까지 대기
        while GPIO.input(self.echo_pin) == 1:
            pulse_end = time.time()
            if pulse_end - pulse_start > timeout:
                return 400  # 타임아웃

        # 거리 계산
        pulse_duration = pulse_end - pulse_start
        distance = pulse_duration * 17150  # 음속 = 343m/s
        distance = round(distance, 2)

        # 유효 범위 제한
        if distance < 2 or distance > 400:
            return 400

        return distance

    def cleanup(self):
        GPIO.cleanup()


# 4방향 센서 관리 클래스
class FourWaySensor:
    def __init__(self):
        self.sensors = {
            'front': UltrasonicSensor(trig_pin=23, echo_pin=24),
            'back': UltrasonicSensor(trig_pin=27, echo_pin=22),
            'left': UltrasonicSensor(trig_pin=5, echo_pin=6),
            'right': UltrasonicSensor(trig_pin=13, echo_pin=19)
        }

    def get_all_distances(self):
        """
        4방향 거리 측정
        반환: [front, back, left, right] (cm)
        """
        distances = []
        for direction in ['front', 'back', 'left', 'right']:
            dist = self.sensors[direction].get_distance()
            distances.append(dist)
            time.sleep(0.01)  # 센서 간 간섭 방지

        return distances

    def cleanup(self):
        for sensor in self.sensors.values():
            sensor.cleanup()
```

센서 테스트 코드

```
# test_sensors.py
from ultrasonic_sensor import FourWaySensor
import time

sensor_system = FourWaySensor()

print("초음파 센서 테스트 시작...")
print("Ctrl+C로 종료")

try:
    while True:
        distances = sensor_system.get_all_distances()
        print(f"전방: {distances[0]:6.1f}cm | "
              f"후방: {distances[1]:6.1f}cm | "
              f"좌측: {distances[2]:6.1f}cm | "
              f"우측: {distances[3]:6.1f}cm")
        time.sleep(0.2)
except KeyboardInterrupt:
    print("\n테스트 종료")
finally:
    sensor_system.cleanup()
```

트러블슈팅 #2: 센서 간섭 문제

문제: 4개 센서 동시 측정 시 거리 값 불안정, 상호 간섭 발생 **증상**: 전방 센서 측정 시 좌/우 센서도 반응, 부정확한 값 **원인**: 초음파 신호가 다른 센서의 ECHO 핀에도 감지됨 **해결**:

1. 센서 간 측정 시간 간격 추가 (10ms 대기)
2. 센서 물리적 배치 각도 조정 (서로 반대 방향 향하도록)
3. 타임아웃 설정으로 무한 대기 방지

트러블슈팅 #3: 전원 노이즈

문제: 센서 측정 중 라즈베리파이 재부팅 발생 **원인**: 모터와 센서가 동일 전원 사용, 모터 구동 시 전압 강하 **해결**: 전원 분리

- 라즈베리파이: 5V 2.5A 전용 어댑터
- 모터 시스템: 7.4V 배터리 (독립 전원)
- 공통 GND 연결 (전압 기준 통일)

5단계: 모터 제어 시스템 구현

L298N 모터 드라이버 연결

```
[라즈베리파이 → L298N]
- ENA (좌측 모터 PWM) → GPIO 12 (Pin 32)
- IN1 (좌측 방향1) → GPIO 17 (Pin 11)
- IN2 (좌측 방향2) → GPIO 18 (Pin 12)
- ENB (우측 모터 PWM) → GPIO 16 (Pin 36)
- IN3 (우측 방향1) → GPIO 20 (Pin 38)
- IN4 (우측 방향2) → GPIO 21 (Pin 40)

[L298N → DC 모터]
- OUT1, OUT2 → 좌측 바퀴 모터
- OUT3, OUT4 → 우측 바퀴 모터

[전원]
- L298N 12V 입력 → 7.4V 배터리
- GND → 라즈베리파이 GND와 공통
```

모터 제어 클래스 구현

```python
# motor_controller.py
import RPi.GPIO as GPIO
from time import sleep

class MotorController:
    def __init__(self):
        # GPIO 핀 설정
        self.LEFT_PWM = 12
        self.LEFT_IN1 = 17
        self.LEFT_IN2 = 18

        self.RIGHT_PWM = 16
        self.RIGHT_IN3 = 20
        self.RIGHT_IN4 = 21

        # GPIO 초기화
        GPIO.setmode(GPIO.BCM)
        GPIO.setup(self.LEFT_PWM, GPIO.OUT)
        GPIO.setup(self.LEFT_IN1, GPIO.OUT)
        GPIO.setup(self.LEFT_IN2, GPIO.OUT)
        GPIO.setup(self.RIGHT_PWM, GPIO.OUT)
        GPIO.setup(self.RIGHT_IN3, GPIO.OUT)
        GPIO.setup(self.RIGHT_IN4, GPIO.OUT)

        # PWM 설정 (1000Hz)
        self.left_pwm = GPIO.PWM(self.LEFT_PWM, 1000)
        self.right_pwm = GPIO.PWM(self.RIGHT_PWM, 1000)

        self.left_pwm.start(0)
        self.right_pwm.start(0)

        self.base_speed = 50  # 기본 속도 (0-100)
```

```
    def forward(self, speed=None):
        """직진"""
        if speed is None:
            speed = self.base_speed

        # 좌측 모터 전진
        GPIO.output(self.LEFT_IN1, GPIO.HIGH)
        GPIO.output(self.LEFT_IN2, GPIO.LOW)
        self.left_pwm.ChangeDutyCycle(speed)

        # 우측 모터 전진
        GPIO.output(self.RIGHT_IN3, GPIO.HIGH)
        GPIO.output(self.RIGHT_IN4, GPIO.LOW)
        self.right_pwm.ChangeDutyCycle(speed)

    def backward(self, speed=None):
        """후진"""
        if speed is None:
            speed = self.base_speed

        GPIO.output(self.LEFT_IN1, GPIO.LOW)
        GPIO.output(self.LEFT_IN2, GPIO.HIGH)
        self.left_pwm.ChangeDutyCycle(speed)

        GPIO.output(self.RIGHT_IN3, GPIO.LOW)
        GPIO.output(self.RIGHT_IN4, GPIO.HIGH)
        self.right_pwm.ChangeDutyCycle(speed)

    def turn_left(self, speed=None):
        """좌회전 (좌측 정지, 우측 전진)"""
        if speed is None:
            speed = self.base_speed

        # 좌측 모터 정지
        GPIO.output(self.LEFT_IN1, GPIO.LOW)
        GPIO.output(self.LEFT_IN2, GPIO.LOW)
        self.left_pwm.ChangeDutyCycle(0)

        # 우측 모터 전진
        GPIO.output(self.RIGHT_IN3, GPIO.HIGH)
        GPIO.output(self.RIGHT_IN4, GPIO.LOW)
        self.right_pwm.ChangeDutyCycle(speed)

    def turn_right(self, speed=None):
        """우회전 (우측 정지, 좌측 전진)"""
        if speed is None:
            speed = self.base_speed

        # 좌측 모터 전진
        GPIO.output(self.LEFT_IN1, GPIO.HIGH)
        GPIO.output(self.LEFT_IN2, GPIO.LOW)
        self.left_pwm.ChangeDutyCycle(speed)
```

```
        # 우측 모터 정지
        GPIO.output(self.RIGHT_IN3, GPIO.LOW)
        GPIO.output(self.RIGHT_IN4, GPIO.LOW)
        self.right_pwm.ChangeDutyCycle(0)

    def stop(self):
        """정지"""
        self.left_pwm.ChangeDutyCycle(0)
        self.right_pwm.ChangeDutyCycle(0)
        GPIO.output(self.LEFT_IN1, GPIO.LOW)
        GPIO.output(self.LEFT_IN2, GPIO.LOW)
        GPIO.output(self.RIGHT_IN3, GPIO.LOW)
        GPIO.output(self.RIGHT_IN4, GPIO.LOW)

    def cleanup(self):
        """리소스 정리"""
        self.stop()
        self.left_pwm.stop()
        self.right_pwm.stop()
        GPIO.cleanup()
```

모터 테스트

```
# test_motors.py
from motor_controller import MotorController
import time

motor = MotorController()

try:
    print("전진 3초")
    motor.forward(speed=60)
    time.sleep(3)

    print("정지 1초")
    motor.stop()
    time.sleep(1)

    print("좌회전 2초")
    motor.turn_left(speed=60)
    time.sleep(2)

    print("정지 1초")
    motor.stop()
    time.sleep(1)

    print("우회전 2초")
    motor.turn_right(speed=60)
    time.sleep(2)

    print("정지")
```

```
    motor.stop()

except KeyboardInterrupt:
    print("\n중단")
finally:
    motor.cleanup()
```

트러블슈팅 #4: 좌우 바퀴 속도 차이

문제: 직진 명령 시 RC카가 한쪽으로 치우침 **원인**: 좌/우 모터의 특성 차이 (제조 공차) **해결**: 모터별 속도 보정 계수 추가

```
# 캘리브레이션 코드
self.left_correction = 1.0   # 좌측 모터 보정 계수
self.right_correction = 0.95  # 우측 모터 5% 느림

def forward(self, speed=None):
    left_speed = speed * self.left_correction
    right_speed = speed * self.right_correction
    # ... PWM 적용
```

캘리브레이션 방법:

1. 직진 명령으로 3m 주행
2. 치우친 방향의 반대편 모터 보정 계수 증가
3. 직진성 확인 반복

6단계: 시뮬레이터 코드를 실제 RC카로 이식

아키텍처 변경

```
[기존 시뮬레이터]
environment.py (가상 격자) → car.py (가상 이동) → agent.py (DQN)

[실제 RC카]
real_environment.py (실제 센서) → real_car.py (실제 모터) → agent.py (동일)
```

실제 환경 클래스 구현

```
# real_environment.py
from ultrasonic_sensor import FourWaySensor
import numpy as np

class RealEnvironment:
    def __init__(self):
        self.sensors = FourWaySensor()
```

```
        self.collision_threshold = 10  # 10cm 이하는 충돌로 간주

    def get_state(self):
        """
        현재 상태 벡터 생성
        시뮬레이터의 get_state()와 동일한 형식
        """
        distances = self.sensors.get_all_distances()

        # 시뮬레이터와 동일한 정규화
        # 거리 범위: 0 ~ 400cm → 0.0 ~ 1.0
        normalized = [min(d / 400.0, 1.0) for d in distances]

        # 충돌 위험도 추가 (10cm 이하 = 위험)
        danger_flags = [1.0 if d < self.collision_threshold else 0.0
                        for d in distances]

        # 상태 벡터: [정규화된 거리 4개 + 위험 플래그 4개] = 8차원
        state = np.array(normalized + danger_flags, dtype=np.float32)

        return state

    def is_collision(self):
        """충돌 감지"""
        distances = self.sensors.get_all_distances()
        return any(d < self.collision_threshold for d in distances)

    def cleanup(self):
        self.sensors.cleanup()
```

실제 RC카 클래스 구현

```
# real_car.py
from motor_controller import MotorController
import time

class RealCar:
    def __init__(self):
        self.motor = MotorController()
        self.action_duration = 0.5  # 각 행동 지속 시간 (초)
        self.steps = 0
        self.score = 0

    def move(self, action, environment):
        """
        행동 실행
        action: 0=직진, 1=우회전, 2=좌회전
        """
        # 충돌 전 체크
        if environment.is_collision():
            self.motor.stop()
```

```
            reward = -100  # 충돌 페널티
            done = True
            return reward, done

        # 행동 실행
        if action == 0:  # 직진
            self.motor.forward(speed=60)
        elif action == 1:  # 우회전
            self.motor.turn_right(speed=60)
        elif action == 2:  # 좌회전
            self.motor.turn_left(speed=60)

        # 행동 지속
        time.sleep(self.action_duration)

        # 정지
        self.motor.stop()
        time.sleep(0.1)  # 안정화

        # 보상 계산
        state_after = environment.get_state()
        distances = state_after[:4] * 400  # 정규화 해제

        # 보상: 전방 거리 유지 (너무 가깝지도, 멀지도 않게)
        front_distance = distances[0]
        if front_distance > 50:
            reward = 1  # 안전 거리 유지
        elif front_distance > 20:
            reward = 0.5  # 조금 가까움
        else:
            reward = -10  # 너무 가까움 (위험)

        # 충돌 체크
        done = environment.is_collision()
        if done:
            reward = -100

        self.steps += 1
        self.score += reward

        return reward, done

    def reset(self):
        self.motor.stop()
        self.steps = 0
        self.score = 0

    def cleanup(self):
        self.motor.cleanup()
```

통합 실행 코드

```
# rc_car_main.py
import numpy as np
import tflite_runtime.interpreter as tflite
from real_environment import RealEnvironment
from real_car import RealCar
import time

class TFLiteAgent:
    """TensorFlow Lite 기반 추론 에이전트"""
    def __init__(self, model_path="model_final.tflite", use_cache=True):
        # TFLite 모델 로드
        self.interpreter = tflite.Interpreter(model_path=model_path)
        self.interpreter.allocate_tensors()

        # 입출력 텐서 정보
        self.input_details = self.interpreter.get_input_details()
        self.output_details = self.interpreter.get_output_details()

        # 캐시 시스템 (선택)
        self.use_cache = use_cache
        self.cache = {}

    def select_action(self, state):
        """행동 선택"""
        # 캐시 확인
        if self.use_cache:
            state_key = tuple(np.round(state, 2))
            if state_key in self.cache:
                return self.cache[state_key]

        # TFLite 추론
        state_tensor = np.array([state], dtype=np.float32)
        self.interpreter.set_tensor(self.input_details[0]['index'],
state_tensor)
        self.interpreter.invoke()
        q_values = self.interpreter.get_tensor(self.output_details[0]
['index'])

        action = np.argmax(q_values[0])

        # 캐시 저장
        if self.use_cache:
            self.cache[state_key] = action

        return action

def main():
    # 초기화
    env = RealEnvironment()
    car = RealCar()

    # TFLite 에이전트 로드
    try:
```

```
        agent = TFLiteAgent(model_path="model_final.tflite",
use_cache=True)
        print("✅ TFLite 모델 로드 성공!")
        print("📦 모델 크기: 12MB (경량화됨)")
    except Exception as e:
        print(f"❌ 모델 파일이 없습니다: {e}")
        return

    print("🚗 자율주행 RC카 시작!")
    print("Ctrl+C로 종료")

    try:
        state = env.get_state()

        while True:
            # AI 행동 선택 (TFLite 추론)
            start_time = time.time()
            action = agent.select_action(state)
            inference_time = (time.time() - start_time) * 1000  # ms

            action_names = ["직진", "우회전", "좌회전"]
            print(f"[Step {car.steps}] 행동: {action_names[action]} (추론:
{inference_time:.1f}ms)")

            # 행동 실행
            reward, done = car.move(action, env)
            next_state = env.get_state()

            if done:
                print(f"❌ 충돌 감지! 점수: {car.score:.1f}, 스텝:
{car.steps}")
                print("5초 후 재시작...")
                time.sleep(5)
                car.reset()
                state = env.get_state()
            else:
                state = next_state
                print(f"  점수: {car.score:.1f}, 전방거리:
{state[0]*400:.1f}cm")

            time.sleep(0.1)  # 센서 안정화

    except KeyboardInterrupt:
        print("\n종료")
    finally:
        car.cleanup()
        env.cleanup()

if __name__ == "__main__":
    main()
```

실행 결과

```
✅ TFLite 모델 로드 성공!
📦 모델 크기: 12MB (경량화됨)
🚗 자율주행 RC카 시작!
Ctrl+C로 종료

[Step 0] 행동: 직진 (추론: 43.2ms)
  점수: 1.0, 전방거리: 85.3cm
[Step 1] 행동: 직진 (추론: 41.8ms)
  점수: 2.0, 전방거리: 72.1cm
[Step 2] 행동: 우회전 (추론: 12.5ms)  ← 캐시 히트!
  점수: 2.5, 전방거리: 95.4cm
...
```

트러블슈팅 #5: 시뮬레이터와 실제 환경의 차이

문제: 시뮬레이터에서 잘 작동하던 모델이 실제 환경에서 충돌 빈번 **원인**:

1. 시뮬레이터는 격자 단위 이동, 실제는 연속 공간
2. 센서 노이즈 (시뮬레이터는 완벽한 측정값)
3. 모터 반응 지연 시간

해결:

1. **센서 노이즈 필터링**: 이동평균 필터 적용

```
from collections import deque

class SensorFilter:
    def __init__(self, window_size=3):
        self.buffer = {
            'front': deque(maxlen=window_size),
            'back': deque(maxlen=window_size),
            'left': deque(maxlen=window_size),
            'right': deque(maxlen=window_size)
        }

    def filter(self, distances):
        """이동평균 필터"""
        filtered = []
        for i, direction in enumerate(['front', 'back', 'left', 'right']):
            self.buffer[direction].append(distances[i])
            avg = sum(self.buffer[direction]) /
len(self.buffer[direction])
            filtered.append(avg)
        return filtered
```

2. **안전 마진 증가**: 충돌 임계값 10cm → 15cm로 상향
3. **속도 감소**: 60% → 50%로 속도 조정 (반응 시간 확보)

트러블슈팅 #6: 라즈베리파이 추론 속도 문제

문제: 초기 접근에서 PyTorch 모델(.pth)을 라즈베리파이에서 직접 실행하려 했으나 불가능

- PyTorch 설치 자체가 1.2GB (라즈베리파이에 과부하)
- 설치해도 추론 시간 200ms 이상 (목표: 50ms 이내) **목표**: 라즈베리파이에서 실시간 추론 (50ms 이내)

해결:

1. **개발 워크플로우 변경**

```
[기존 잘못된 접근]
개발 PC에서 학습 → .pth 파일 → 라즈베리파이에 PyTorch 설치 → 추론 (실패)

[올바른 접근]
개발 PC에서 학습 → .pth → 개발 PC에서 TFLite 변환 → 라즈베리파이에 TFLite Runtime만 설치 → 추론 (성공)
```

2. **모델 변환 스크립트 (개발 PC에서 실행)**

```python
# convert_to_tflite.py - 개발 PC에서 실행!
import torch
import onnx
from onnxsim import simplify
import onnx_tf
import tensorflow as tf

# 1. PyTorch 모델 로드
model = torch.load('model_final.pth')
model.eval()

# 2. PyTorch → ONNX
dummy_input = torch.randn(1, 8)
torch.onnx.export(model, dummy_input, "model.onnx",
                  input_names=['input'], output_names=['output'])

# 3. ONNX 최적화
onnx_model = onnx.load("model.onnx")
simplified_model, check = simplify(onnx_model)
onnx.save(simplified_model, "model_simplified.onnx")

# 4. ONNX → TensorFlow
tf_rep = onnx_tf.backend.prepare(simplified_model)
tf_rep.export_graph("model_tf")

# 5. TensorFlow → TFLite (양자화)
converter = tf.lite.TFLiteConverter.from_saved_model("model_tf")
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.target_spec.supported_types = [tf.float16]
tflite_model = converter.convert()
```

```
with open('model_final.tflite', 'wb') as f:
    f.write(tflite_model)

print("✅ 변환 완료: model_final.tflite")
```

3. **라즈베리파이 설정 (TFLite Runtime만 설치)**

```
# PyTorch 설치 안 함!
pip3 install tflite-runtime  # 10MB만 설치 (vs PyTorch 1.2GB)
```

결과:

- 모델 크기: 50MB → 12MB (76% 감소)
- 추론 시간: 불가능 → 45ms (목표 달성)
- 설치 크기: 1.2GB → 10MB (99% 감소)
- 메모리 사용: 850MB → 420MB

7단계: 최종 통합 테스트 및 결과

테스트 환경

- 실내 공간: 4m x 4m
- 장애물: 박스, 의자, 벽면
- 목표: 충돌 없이 60초 이상 주행

테스트 결과

테스트 #1: 기본 주행

- 일시: 2022년 5월 10일
- 환경: 빈 공간 (장애물 없음)
- 결과: **성공** - 120초 무충돌 주행
- 평균 속도: 0.3m/s
- 특이사항: 직진 안정성 우수

테스트 #2: 장애물 회피

- 일시: 2022년 5월 12일
- 환경: 박스 3개 배치
- 결과: **부분 성공** - 2회 충돌, 평균 45초 주행
- 문제점: 좁은 공간 통과 실패
- 개선: 좌우회전 각도 조정 (45도 → 60도)

테스트 #3: 복잡한 환경

- 일시: 2022년 5월 15일
- 환경: 박스 5개 + 의자 2개
- 결과: **성공** - 90초 무충돌 주행
- 평균 캐시 히트율: 67%
- 특이사항: 학습된 패턴 재사용 확인

테스트 #4: 동적 장애물

- 일시: 2022년 5월 18일
- 환경: 사람이 이동하면서 방해
- 결과: **성공** - 충돌 없이 장애물 감지 및 회피
- 반응 시간: 평균 0.8초

최종 성능 지표

지표	수치
평균 주행 시간	87초
충돌률	15%
캐시 히트율	67%
추론 시간	45ms
배터리 사용 시간	약 2시간
총 개발 비용	$104/대

트러블슈팅 #7: 배터리 방전 시 오작동

문제: 배터리 전압 6.5V 이하 시 모터 출력 불안정, 센서 오류 **해결**: 배터리 전압 모니터링 추가

```
import board
import busio
import adafruit_ina219

# 전압 센서 (INA219)
i2c = busio.I2C(board.SCL, board.SDA)
ina219 = adafruit_ina219.INA219(i2c)

def check_battery():
    voltage = ina219.bus_voltage + ina219.shunt_voltage
    if voltage < 6.5:
        print("⚠  배터리 부족! 충전 필요")
        return False
    return True
```

8단계: 교육용 패키징 및 문서화

제품 구성

1. **하드웨어 키트**
    - 조립된 RC카 (라즈베리파이 포함)
    - USB 충전기
    - 예비 배터리
    - 퀵 스타트 가이드
2. **소프트웨어**
    - SD 카드 (OS + 코드 사전 설치)
    - 학습된 모델 파일 포함
    - Jupyter 노트북 튜토리얼
3. **교육 자료**
    - 강화학습 개념 설명서
    - 코드 상세 주석
    - 실습 과제 (난이도별)

판매 가격 책정

- 원가: $104
- 교육 자료 개발 비용: $20
- 이윤: $50
- **판매가: $174**

고객 피드백 (베타 테스트)

- 교육 기관 3곳 베타 테스트 진행
- 긍정 평가: "직관적인 강화학습 학습 도구"
- 개선 요청: "더 많은 센서 추가 옵션"

성과 및 기술적 기여

정량적 성과

1. **비용 절감**: AWS Rekognition 월 $1,000 → 하드웨어 일회성 $104
2. **실시간 처리**: 추론 시간 45ms (초당 22프레임)
3. **안정성**: 87초 평균 무충돌 주행
4. **교육 효과**: 베타 고객 만족도 4.2/5.0

기술적 기여

1. **강화학습 실용화**
    - DQN 알고리즘을 저비용 임베디드 환경에 성공적으로 이식
    - 시뮬레이터 → 실제 환경 전환 노하우 확립

2. **센서 퓨전**
    - 4방향 초음파 센서 데이터를 통합하여 공간 인식
    - 노이즈 필터링 및 캘리브레이션 자동화
3. **임베디드 최적화**
    - PyTorch 모델을 TFLite로 변환하여 4.4배 속도 개선
    - 메모리 사용량 50% 절감 (캐시 시스템 활용)
4. **교육 콘텐츠 개발**
    - IoT 집필 경험을 활용한 체계적인 교육 자료
    - 초보자도 이해할 수 있는 단계별 가이드

향후 개선 방향

단기 개선 (1-3개월)

1. **추가 센서 통합**
    - IMU(관성센서): 자세 제어, 경사면 주행
    - 라이다(LiDAR): 고정밀 거리 측정 (선택 사항)
2. **무선 모니터링**
    - 웹 대시보드: 실시간 센서 데이터 시각화
    - 원격 제어: 비상 정지 기능
3. **자동 충전 스테이션**
    - 배터리 부족 시 자동으로 충전소 복귀

장기 개선 (6개월+)

1. **멀티 에이전트 협업**
    - 여러 RC카가 협력하여 복잡한 미션 수행
    - 분산 강화학습 적용
2. **비전 센서 추가 (선택)**
    - 라즈베리파이 카메라 모듈 (저해상도)
    - 객체 감지: TensorFlow Lite 모델 (MobileNet)
    - 비용 증가 최소화 ($15 추가)
3. **클라우드 연동**
    - 학습 데이터 수집 → 클라우드 재학습 → 모델 업데이트
    - 교육 기관 간 모델 공유 플랫폼

결론

이 프로젝트는 **비용 효율적인 교육용 자율주행 시스템**을 성공적으로 구현했습니다. 시뮬레이터에서 검증된 강화학습 알고리즘을 실제 하드웨어로 이식하는 과정에서 다양한 기술적 도전을 극복했으며, 초음파 센서만으로도 실용적인 장애물 회피가 가능함을 입증했습니다.

특히, **AWS Rekognition의 1/10 비용**으로 교육용 솔루션을 개발함으로써 비용 효율성을 크게 개선했으며, 임베디드 환경에서 딥러닝 모델을 최적화하는 실무 경험을 축적했습니다.

사용 기술 스택 요약

소프트웨어

- **언어**: Python 3.9
- **프레임워크**: PyTorch, TensorFlow Lite
- **라이브러리**: NumPy, RPi.GPIO, Pygame
- **알고리즘**: Deep Q-Network (DQN), Experience Replay

하드웨어

- **메인보드**: Raspberry Pi 4 Model B (4GB)
- **센서**: HC-SR04 초음파 센서 x4
- **액추에이터**: DC 모터 x2, L298N 모터 드라이버
- **전원**: 18650 리튬이온 배터리 (2S 7.4V)

개발 도구

- **버전 관리**: Git
- **IDE**: VS Code (원격 SSH)
- **디버깅**: UART 시리얼 통신, 원격 Jupyter Notebook

프로젝트 타임라인

```
2022년 1월 – 3월: 시뮬레이터 개발 (v1 ~ v5)
2022년 4월 – 5월: 하드웨어 전환 및 통합
  – 4월 1주: 하드웨어 선정 및 구매
  – 4월 2주: 라즈베리파이 환경 설정
  – 4월 3주: 센서 연결 및 테스트
  – 4월 4주: 모터 제어 구현
  – 5월 1–2주: 코드 이식 및 디버깅
  – 5월 3–4주: 통합 테스트 및 최적화
2022년 6월: 교육 자료 개발 및 베타 테스트
```

작성자: [귀하의 이름] **직책**: IoT 교육용 자율주행 RC카 개발자 **회사**: 노마드랩 **작성일**: 2022년 6월 30일