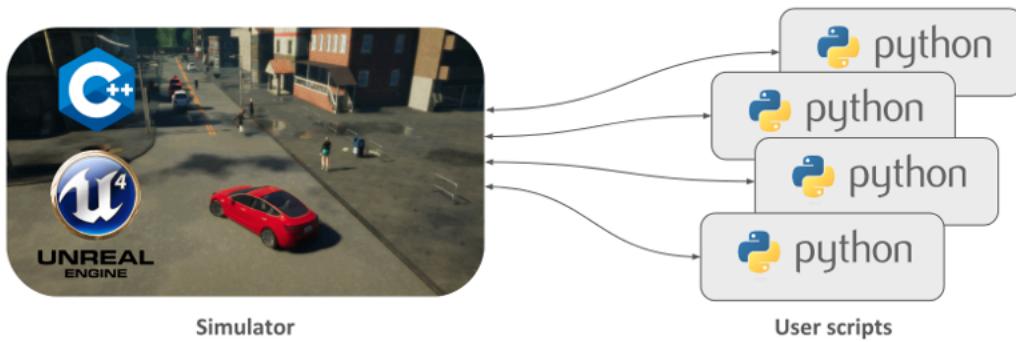


CARLA RL

CARLA Reinforcement Learning Framework

CARLA (Car Learning & Acting) Simulator is a popular open-source platform for developing and testing autonomous driving (AD) systems, built on the Unreal Engine for realistic 3D environments. It provides flexible tools, APIs (Python/C++), digital assets, and simulated sensors (like cameras, LiDAR) to generate data and evaluate AD algorithms safely, allowing researchers and engineers to train perception models and control policies without real-world risk, supporting everything from basic learning to complex traffic scenarios.



A modular and comprehensive Reinforcement Learning framework for training autonomous vehicles in the CARLA simulator using PyTorch.

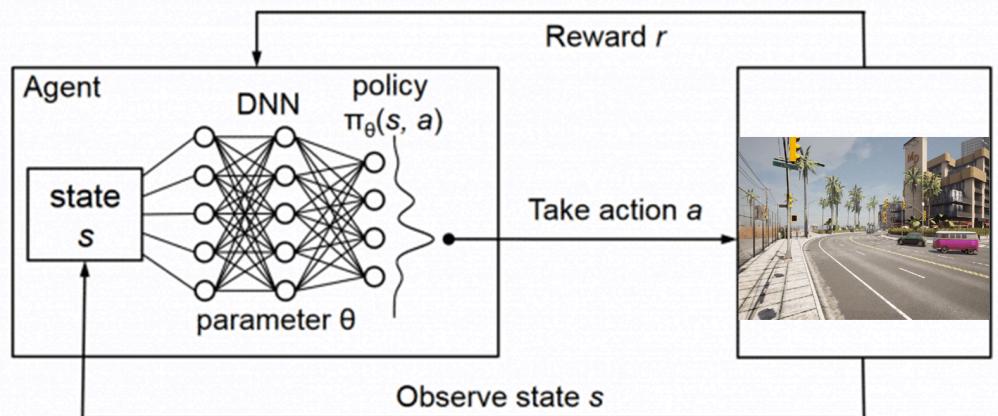


Table of Contents

- [Project Overview](#)
- [Features](#)
- [Project Structure](#)
- [Installation](#)
- [Quick Start](#)
- [Algorithms](#)

- [Environment Details](#)
 - [Usage](#)
 - [Configuration](#)
 - [Results and Evaluation](#)
 - [References](#)
-

Project Overview

This project implements three state-of-the-art Deep Reinforcement Learning algorithms for training autonomous vehicles to navigate in the CARLA simulator:

- **DQN** (Deep Q-Network) - Value-based method with experience replay
- **SAC** (Soft Actor-Critic) - Maximum entropy actor-critic with automated temperature tuning
- **PPO** (Proximal Policy Optimization) - Trust region policy optimization method

The framework supports both RGB camera input and Semantic Segmentation input, enabling the vehicle to learn visual-based navigation policies.

Key Objectives

1. Train vehicles to navigate safely using only visual input
 2. Demonstrate the effectiveness of different RL algorithms
 3. Provide a modular, extensible framework for CARLA-based RL research
-

Features

Core Capabilities

- **Multiple RL Algorithms:** DQN, SAC, and PPO implementations
 - **Dual Input Modes:** RGB camera or Semantic Segmentation
 - **Custom Gym Environment:** Fully compatible with OpenAI Gym interface
 - **CNN Architectures:** Specialized convolutional networks for visual input processing
 - **Checkpoint Management:** Save and load trained models
 - **Inference Mode:** Evaluate trained agents with visualization
 - **Video Recording:** Save agent performance as video files
-

Project Structure

```
carla_project/
├── carla_env.py      # Custom Gym Environment for CARLA
├── networks.py       # Neural network architectures (CNN-based)
├── dqn_agent.py     # DQN algorithm implementation
├── sac_agent.py      # SAC algorithm implementation
├── ppo_agent.py     # PPO algorithm implementation
├── train.py          # Main training script
├── inference.py      # Inference/evaluation script
└── requirements.txt  # Python dependencies
```

```
|── checkpoints/      # Saved model checkpoints (created during training)
└── README.md        # Technical Report
```

Installation

Prerequisites

- Python 3.12
- CARLA Simulator 0.9.x
- CUDA-capable GPU (recommended) or CPU

Step 1: Install CARLA

1. Download CARLA from carla.org
2. Extract to a directory (e.g., `C:\CARLA`)
3. Run CARLA server:

```
# Windows
cd C:\CARLA
CarlaUE4.sh.exe

# Linux
cd /opt/carla
./CarlaUE4.sh

# To Disable the rendering during the training...
./CarlaUE4.sh -RenderOffScreen
```

Step 2: Create Python Environment

```
# Create conda environment
conda create -n carla python=3.12
conda activate carla

# Or use venv
python -m venv carla_env
source carla_env/bin/activate # Linux/Mac
carla_env\Scripts\activate # Windows
```

Step 3: Install Dependencies

```
cd carla_project
pip install -r requirements.txt
```

Step 4: Install CARLA Python API

```
# Find the Python API in your CARLA installation
# For CARLA 0.9.16 with Python 3.12:
```

```
pip install path/to/carla/PythonAPI/carla/dist/carla-0.9.16-py3.12-win-amd64.egg
```

Quick Start

1. Start CARLA Server

```
# Navigate to CARLA directory and run  
.CarlaUE4.sh # Linux  
CarlaUE4.sh.exe # Windows
```

2. Train a Model

```
# Train PPO with semantic segmentation (default)  
python train.py --algo ppo --camera seg --episodes 1000  
  
# Train DQN with RGB camera  
python train.py --algo dqn --camera rgb --episodes 1000  
  
# Train SAC with custom settings  
python train.py --algo sac --camera seg --episodes 500 --lr 1e-3
```

3. Run Inference

```
# Evaluate trained model  
python inference.py --checkpoint checkpoints/ppo_final.pth --episodes 10  
  
# Save video of inference  
python inference.py --checkpoint checkpoints/ppo_final.pth --episodes 10 --save_video
```

Algorithms

DQN (Deep Q-Network)

Architecture:

- CNN feature extractor with 3 convolutional layers
- Dueling architecture (separate value and advantage streams)
- Target network for stable training
- Double DQN to reduce overestimation bias

Hyperparameters:

Parameter	Value	Description
Learning Rate	1e-4	Optimizer learning rate
Gamma	0.99	Discount factor
Batch Size	32	Training batch size
Buffer Size	100,000	Replay buffer capacity
Target Update	1000	Steps between target network updates

Parameter	Value	Description
Epsilon Start	1.0	Initial exploration rate
Epsilon End	0.01	Final exploration rate
Epsilon Decay Episodes	80,000	Episodes over which epsilon decays linearly

Epsilon Decay Schedule:

- Uses linear decay instead of multiplicative decay for more predictable exploration
- Decays from 1.0 to 0.01 over 80,000 episodes (80% of default 100k training)
- After episode 1,000: $\epsilon \approx 0.988$
- After episode 10,000: $\epsilon \approx 0.876$
- After episode 40,000: $\epsilon \approx 0.505$
- After episode 80,000: $\epsilon = 0.01$ (minimum)

SAC (Soft Actor-Critic)

Architecture:

- Actor network (policy) with CNN backbone
- Twin critic networks (Q1, Q2) to reduce overestimation
- Automated entropy coefficient tuning
- Soft target updates with $\tau=0.005$

Hyperparameters:

Parameter	Value	Description
Learning Rate	3e-4	Optimizer learning rate
Gamma	0.99	Discount factor
Batch Size	64	Training batch size
Buffer Size	100,000	Replay buffer capacity
Tau	0.005	Soft update coefficient
Alpha	0.2	Initial entropy coefficient
Auto Alpha	True	Automatic entropy tuning

PPO (Proximal Policy Optimization)

Architecture:

- Actor network with categorical distribution
- Critic network for value estimation
- GAE for advantage computation
- Clipped surrogate objective

Parameter	Value	Description
Learning Rate	3e-4	Optimizer learning rate
Gamma	0.99	Discount factor
GAE Lambda	0.95	GAE smoothing parameter
Clip Epsilon	0.2	PPO clipping parameter
Value Coef	0.5	Value loss coefficient

Parameter	Value	Description
Entropy Coef	0.01	Entropy bonus coefficient
Rollout Size	2048	Steps per rollout
PPO Epochs	10	Optimization epochs per rollout

Environment Details

State Space

The observation space consists of images from the vehicle's camera:

Input Type	Shape	Description
RGB	(84, 84, 3)	Color camera images
Segmentation	(84, 84, 1)	Semantic class labels

Action Space

Discrete action space with 4 actions:

Action Index	Name	Description
0	Straight	Accelerate forward
1	Left	Turn left
2	Right	Turn right
3	Brake	Apply brakes

Reward Function

The reward is composed of several components:

Reward = velocity_bonus - lane_deviations_penalty - collision_penalty - lane_invasion_penalty - brake_penalty + survival_bonus

Component	Formula	Purpose
Velocity Bonus	+0.1 × velocity (km/h)	Encourage forward movement
Lane Deviation	-0.5 × deviation (m)	Stay in lane center
Collision	-100.0	Heavy penalty for collisions
Lane Invasion	-10.0	Penalty for leaving lane
Brake	-1.0	Small penalty for braking
Survival	+0.1	Reward each step survived

Episode Termination

Episodes terminate when:

- Collision occurs
- Vehicle is stuck (velocity < 0.1 km/h for 50+ steps)
- Maximum episode length (1000 steps) reached

Usage

Training Commands

```
# Basic training
python train.py --algo [dqn|sac|ppo] --camera [rgb|seg] --episodes [number]

# Full example with all options
python train.py \
    --algo ppo \
    --camera seg \
    --episodes 1000 \
    --dim 84 \
    --town Town01 \
    --host 127.0.0.1 \
    --port 2000 \
    --render \
    --save_freq 50 \
    --save_dir checkpoints \
    --lr 3e-4 \
    --gamma 0.99 \
    --batch_size 64
```

Training Arguments

Argument	Type	Default	Description
--algo	str	ppo	Algorithm: dqn, sac, or ppo
--camera	str	seg	Camera type: rgb or seg (segmentation recommended for faster learning)
--episodes	int	1000	Number of training episodes
--dim	int	84	Image dimension (84x84 recommended for speed)
--town	str	Town01	CARLA town name (Town01 recommended for training)
--host	str	127.0.0.1	CARLA server IP
--port	int	2000	CARLA server port
--render	flag	True	Show cv2 window
--no-render	flag	-	Disable rendering
--save_freq	int	50	Checkpoint save frequency (in episodes)
--save_dir	str	checkpoints	Directory for saving checkpoints
--load	str	-	Path to load checkpoint from
--device	str	auto	Device: auto (uses CUDA if available), cpu, or cuda
--lr	float	Algorithm-specific*	Learning rate for optimizer
--gamma	float	0.99	Discount factor (reward decay)
--batch_size	int	Algorithm-specific*	Training batch size

Algorithm-Specific Defaults:

- **DQN:** lr=1e-4 , batch_size=32
- **SAC:** lr=3e-4 , batch_size=64

- **PPO:** `lr=3e-4`, `batch_size=64`

Recommended Settings for Fast Training:

```
python train.py --algo dqn --camera seg --dim 84 --town Town01 --episodes 100000
```

Inference Commands

```
# Basic inference
python inference.py --checkpoint checkpoints/ppo_final.pth --episodes 10

# With video recording
python inference.py \
    --checkpoint checkpoints/ppo_final.pth \
    --episodes 10 \
    --save_video \
    --video_path results.mp4

# Without rendering (faster)
python inference.py \
    --checkpoint checkpoints/ppo_final.pth \
    --episodes 100 \
    --no-render
```

Configuration

Recommended Settings

Scenario	Algorithm	Camera	Episodes	Learning Rate
Quick Test	PPO	seg	100	3e-4
Baseline Training	PPO	seg	1000	3e-4
RGB Input	SAC	rgb	1500	3e-4
High Resolution	DQN	seg	2000	1e-4

Hardware Requirements

Setup	GPU	RAM	Training Time
Minimum	CPU	8GB	~10x slower
Recommended	GTX 1060+	16GB	Baseline
Optimal	RTX 5090	32GB	~10x faster

Results and Evaluation

Training Metrics

During training, the following metrics are tracked:

- **Episode Reward:** Total reward per episode
- **Average Reward:** Rolling mean over 100 episodes

- **Episode Length:** Number of steps per episode
- **Algorithm-Specific Metrics:**
 - DQN: Epsilon (exploration rate)
 - SAC: Alpha (entropy coefficient)
 - PPO: Policy loss, Value loss, Entropy

Inference Output

Inference Summary

Total episodes: 10
 Average reward: 45.23 +/- 12.45
 Average episode length: 234.50 +/- 45.30
 Best episode reward: 67.80
 Worst episode reward: 22.10

Algorithm Comparison

Feature	DQN	SAC	PPO
Type	Value-based	Actor-Critic	Actor-Critic
On-Policy	No	No	Yes
Stability	Medium	High	High
Sample Efficiency	Medium	High	Low
Exploration	Epsilon-greedy	Entropy	Entropy
Training Speed	Fast	Medium	Medium
Convergence	Slower	Faster	Fastest
Memory Usage	Low	Medium	Low

When to Use Each Algorithm

- **DQN:** Baseline comparison, simple tasks
- **SAC:** Complex environments, need exploration
- **PPO:** Stable learning, on-policy requirements

Neural Network Architecture

CNN Feature Extractor

All algorithms share a common CNN backbone:

```

Input (84×84×C)
  ↓
Conv2d(C, 32, k=8, s=4) + ReLU → 20×20×32
  ↓
Conv2d(32, 64, k=4, s=2) + ReLU → 9×9×64
  ↓

```

```

Conv2d(64, 64, k=3, s=2) + ReLU → 4×4×64
↓
Flatten → 1024
↓
Linear(1024, 512) → Feature Vector

```

Network Heads

Algorithm	Head	Output
DQN	Q-Head	Q-values per action
SAC Actor	Policy-Head	Action logits
SAC Critic	Q-Head	Q-values per action
PPO Actor	Policy-Head	Action logits
PPO Critic	Value-Head	State value V(s)

Future Improvements

Potential Enhancements

1. **Multi-Task Learning:** Add traffic light recognition, speed limit signs
2. **Hierarchical RL:** High-level route planning + low-level control
3. **Imitation Learning:** Pre-train with expert demonstrations
4. **Multi-Agent:** Multiple vehicles interacting
5. **Recurrent Networks:** LSTM/GRU for temporal memory
6. **Attention Mechanisms:** Focus on relevant road features
7. **Domain Randomization:** Improve generalization
8. **Integration with Stable-Baselines3:** The environment class follows the Gymnasium/Gym interface, making it compatible with Stable-Baselines3 (SB3) algorithms. You can use pre-trained models or train with SB3's optimized implementations of DQN, PPO, SAC, and other algorithms by simply passing the `CarlaGymEnv` instance to any SB3 agent.

Code Contributions

This framework is designed to be modular. To add a new algorithm:

1. Create `new_agent.py` following the pattern of existing agents
2. Implement required methods: `select_action()`, `store_transition()`, `train_step()`
3. Add `load_checkpoint()` and `save_checkpoint()` methods
4. Update `train.py` to include the new algorithm

References

- [CARLA Simulator](#)
- [CARLA GitHub](#)
- [CARLA Documentation](#)