Off-Policy Reinforcement Learning for Autonomous Driving

Hitesh Arora

CMU-RI-TR-20-34

July 30, 2020

The Robotics Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA

Thesis Committee:
Jeff Schneider, *Chair*
David Held
Ben Eysenbach

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Robotics.

Keywords: Reinforcement Learning, Autonomous Driving, Q-learning, Reinforcement Learning with Expert Demonstrations

To my friends, family and teachers.

Abstract

Modern autonomous driving systems continue to face the challenges of handling complex and variable multi-agent real-world scenarios. Some subsystems, such as perception, use deep learning-based approaches to leverage large amounts of data to generalize to novel scenes. Other subsystems, such as planning and control, still follow the classic cost-based trajectory optimization approaches, and require high efforts to handle the long tail of rare events. Deep Reinforcement Learning (RL) has shown encouraging evidence in learning complex decision-making tasks, spanning from strategic games to challenging robotics tasks. Further, the dense reward structure and modest time horizons make autonomous driving a favorable prospect for applying RL.

As there are practical challenges in running RL online on vehicles and most self-driving companies have millions of miles of collected data, it motivates the use of off-policy RL algorithms to learn policies that can eventually work in the real world. We explore the use of an off-policy RL algorithm, Deep Q-Learning, to learn goal-directed navigation in a simulated urban driving environment. Since Deep Q-Learning methods are susceptible to instability and sub-optimal convergence, we investigate different strategies to sample experiences from the replay buffer to mitigate these issues. We also explore combining expert agent's demonstration data with the RL agent's experiences to speed-up the learning process. We demonstrate promising results on the CoRL2017 and NoCrash benchmarks on CARLA.

Acknowledgments

I would foremost like to express my sincere gratitude to my advisor Prof. Jeff Schneider for his extraordinary guidance and constant support throughout my research work. It was under his supervision that I developed a profound interest in the domain of reinforcement learning and the field of academic research at large. I would also like to thank Prof. David Held and Ben Eysenbach for their valuable feedback as members of my thesis committee.

I am also grateful to my colleague and close friend, Tanmay Agarwal, for being a great collaborator, and particularly recognize his contributions in the work presented in Chapters 3 and 4. I would like to thank my amazing lab members Adam, Audrey, Christoph, Mayank, Shuby, Tanvir, Theophile and Vinay for their consistent and useful feedback on my research. I am also immensely grateful to my friends Aditya, Sarthak, Talha, Harjatin, Siddharth, Alex, Gautham, Tithi and Darshi for their help and support throughout these last couple of years. A special shoutout to Nidhi and Vishali for helping me in this journey despite being half way across the globe.

Lastly, I am forever indebted to my family for their sacrifices and commitment towards my education and growth. None of this would have been possible without their love and encouragement.

Funding

We thank the CMU Argo AI Center for Autonomous Vehicle Research[1] for supporting the work presented in this thesis.

[1]https://labs.ri.cmu.edu/argo-ai-center/

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

There has been substantive recent progress towards solving the long-standing goal of autonomous driving [4, 16, 25, 32, 44, 56]. The complexity of the problem ranges from learning to navigate in constrained industrial settings, to driving on highways, to navigation in dense urban environments. Simpler tasks within autonomous driving, such as lane following and cruise control on highways, naturally allow for simpler models. However, navigation in dense urban settings requires understanding complex multi-agent dynamics including tracking multiple actors across scenes, predicting their intended future trajectories, and adjusting agent behavior conditioned on history. Additionally, the agent must be able to generalize to novel scenes and learn to take 'sensible' actions for observations in the long tail of rare events such as a deer running onto a road, a woman in an electric wheelchair chasing a duck on road [38], and unanticipated behavior of vehicles violating traffic rules. These factors provide a strong impetus for the need of general learning paradigms that are sufficiently 'complex' to take these factors into account.

Current state-of-the-art systems generally use a variant of supervised learning over large datasets of collected logs to learn driving behavior [4, 16]. These systems typically consist of a modular pipeline with different components responsible for perception, mapping, localization, actor prediction, motion planning, and control [24, 31, 36, 50, 53]. The advantages offered by modular systems are ease of interpretation

and ability to optimize subsystem parameters in an understandable way. However, in practice, it is challenging to tune these subsystems and replicate the intended behavior leading to poor performance in new environments.

Another approach that has recently become popular is exploiting imitation learning, where we aim to learn a control policy for driving behavior by observing expert demonstrations [4, 7, 37, 39, 42, 43, 58]. The advantage of these methods is that the agent can be optimized using end to end deep learning to learn the desired control behavior which significantly reduces the effort of tuning each component that is common to more modular systems. The drawback, however, is that these systems are challenging to scale and generalize to novel situations, since they can never outperform the expert agent and it is impractical to obtain expert demonstrations for all the scenarios that we may come across.

Deep reinforcement learning (RL) has recently made large strides towards solving sequential decision-making problems, including learning to play Atari games and solving complex robotic manipulation tasks. The superhuman performance attained using this learning paradigm motivates the question of whether RL can be leveraged to solve the prevailing goal of creating autonomous vehicles. This has encouragingly inspired some recent works to use reinforcement learning in autonomous driving [12, 25, 26, 32]. It is challenging to use RL directly for autonomous driving in the real world, primarily due to safety considerations and poor sample complexity of the state-of-art RL algorithms leading to costly and arduous training. Hence, most of the current research in the domain is increasingly being carried out on simulators, such as TORCS [57] and CARLA [12], which can eventually be transferred to real world settings.

We believe autonomous driving could be among the first few domains where RL can successfully achieve impact in the real world. One of the obvious challenges, common to broader RL research, is designing more sample-efficient algorithms to learn from the limited samples, especially to handle the critical long tail of rare events. The upside is that the autonomous driving problem has better structure, viz. dense rewards, and shorter time horizons, which makes it an easier prospect for applying RL as compared to the hard-exploration problems. Many self-driving companies,

such as Argo AI[1] and Waymo[2], now have millions of miles [55] of collected data (logs) annotated with readings from a comprehensive sensor suite, high-definition maps and human drivers' optimal actions. This opens up a tremendous opportunity to design off-policy RL algorithms which can leverage from the already available data. There is an increasing interest in this direction with Offline RL [30] to utilize previously collected data without additional online data collection, which is in the early phases of research. Many self-driving car companies also have a *Log-Replay* system for their testing and evaluation, which replays the real-world collected logs, where they can update their agent with improved driving policy and test the new behavior in different situations. Although, this is not completely interactive as compared to the standard RL environments, since the other actors' behaviors remain the same, it is reasonable to utilize *Log-Replay* for short snippets of time, collect some on-policy data and combine with the existing data for improved training. This motivates us to design off-policy RL algorithms, to utilize both the limited amounts of on-policy data and large amounts of expert data, to extract policies with the maximum possible utility which can work in the real world [15, 21].

1.2 Contributions

The main contributions of this thesis are listed as follows:

- We formulate the problem of goal-directed navigation in the reinforcement learning (RL) framework and build the RL-environment on the CARLA simulator which is planned to be shared with the research community.
- We propose the use of waypoints as the navigation input and design an architecture to learn planning and control directly from semantically segmented images and waypoints using the model-free on-policy RL algorithm of Proximal policy optimization (PPO) [47].
- We explore the use of off-policy RL algorithm of Deep Q-Learning [33] for learning goal-directed navigation in the presence of other dynamic agents, present an analysis of the various issues in training instability and share techniques to

[1]https://www.argo.ai/
[2]https://waymo.com/

address them including n-step returns, dueling network architecture and clipped Double DQN [52].

- We propose *Backward-Sampling*, an intuitive sampling strategy to sample transitions close to episode termination more frequently from the replay buffer and augment it with uniformly sampled transitions and evaluate its effects on training.
- We present an analysis on how combining expert driving policy data along with RL driving agent's experiences can aid in faster learning.

1.3 Organization

The thesis is organized as follows:

- Chapter 2 discusses the related work to this thesis, focusing on recent research using imitation and reinforcement learning for autonomous driving.
- Chapter 3 contains the relevant details about the CARLA simulator used in our work including waypoints, sensors, detectors for dynamic obstacles, and traffic lights. We also describe the official CARLA benchmarks used to evaluate our agent's performance.
- Chapter 4 explains the architecture design to learn planning and control directly from semantically segmented images and waypoints using an on-policy RL algorithm (PPO).
- Chapter 5 contains our work on using the off-policy RL algorithm of Deep Q-Learning for learning goal-directed navigation in the presence of other dynamic agents. We present an analysis of the various issues in training instability and share techniques to address them including clipped Double DQN, *Backward-Sampling*, and combining expert driving policy data along with RL driving agent's experiences.
- Chapter 6 lists the summary of our work and possible directions for future work.

Chapter 2

Related Work

Autonomous driving is a well-studied problem with a long history of different paradigms and approaches that have been attempted to solve the problem. Broadly, these approaches can be divided into three categories, namely Modular Systems, Imitation Learning and Reinforcement Learning. In this chapter, we discuss existing research in these categories in the following sections. Additionally, we discuss research related to the instability issues observed in Q-learning and the techniques to address them.

2.1 Modular Systems

These techniques are characterized by a modular pipelines which have a well-marked distinction between the various phases associated with autonomous navigation [24, 31, 36, 53]. These include perception, mapping & localization, prediction, motion planning, routing, and vehicle control. They are the most widely used approach for building state of the art autonomous vehicles. The advantage they offer is the ease of interpretation and ability to optimize subsystem parameters in an understandable way.

However, this ease of subsystem optimization is also a drawback, since in practice, these systems require a massive engineering effort to tune, which results in issues during scaling to more complex driving situations. Though these systems are the current focus of most real world self-driving car deployments, they present an enormous

engineering challenge to understand tuning behavior and system interdependencies.

2.2 Imitation Learning

Imitation learning provides a paradigm to produce a policy that will mimic the expert's actions given corresponding input [3, 8, 37, 39, 42]. These demonstrations are obtained offline (either in the real world or through simulation), and consist of a sequence of state observations and corresponding expert actions. Most of the baselines on the CARLA benchmark use this approach and hence, we describe them below.

In Learning by Cheating [6], authors provide an architecture for vision-based navigation via imitation learning. They have a two-step training protocol with two agents, Privileged agent and Sensorimotor agent. The privileged agent is trained with human expert's trajectories and has full state access. This agent is then used to generate data to train a Sensorimotor agent, which just uses images as observations.

In Conditional Affordance Learning [45], authors propose an architecture where they generalize direct perception approach to the urban setting. From the convolution features of the image, they develop intermediate representations in the form of affordances, suitable for urban navigation. These affordance models are conditioned on high-level directional commands. They report an improvement of up to 68% in goal-directed navigation on the challenging CARLA simulation benchmark.

2.3 Reinforcement Learning

There are few works that use reinforcement learning from scratch for learning to drive. An initial work is presented in the original CARLA paper [12] where the authors use asynchronous advantage actor-critic (A3C) algorithm [34], and train it for 10 million simulation steps, which corresponds to roughly 12 days of driving at 10 frames per second (fps). While they achieve decent performance in the straight task, they perform poorly in all other navigation tasks for the RL agent. A follow-up work, Controllable Imitative Reinforcement Learning (CIRL) [32], first trains the agent by imitation learning with the ground-truth actions from recorded human driving

videos, and then fine-tunes it further using reinforcement learning optimized via Deep Deterministic Policy Gradient (DDPG). However, most of the performance gains seem attributable to the pre-training stage using imitation learning and it is unclear about the performance rate that can be achieved using RL-based policy learning from scratch.

Kendall et al. [25] show the first application of RL for autonomous driving on a real car, where they are able to learn the simple task of lane-following using DDPG RL algorithm and then, a VAE to encode input camera images. They use a sparse reward structure that can make it challenging to learn, and focus on a constrained environment.

IA [51] is a very recent work that learns a ResNet encoder to predict relevant features, referred as implicit affordances and uses the encoding as the state-input to learn a driving policy using recent advancements in the DQN algorithm, namely Rainbow IQN Ape-X [20].

2.3.1 Stability in Deep Q-learning

We also discuss related research to the issues we see in off-policy Q-learning and our proposed *Backward-Sampling* approach. In DisCor [28], authors demonstrate a similar issue in bootstrapping-based Q-learning algorithms due to close interactions between the distribution of experience collected by the agent and the policy induced by training on that experience, referring to it as the absence of "corrective feedback". The authors share that these methods regress onto bootstrapped estimates of the current Q-value function, rather than the true optimal Q-value function (which is unknown). Thus, naively visiting states with high TD-error and updating the Q-value function at those states does not necessarily correct those errors, since high TD-target errors might be due to upstream errors in Q-value estimates of other states that are visited less often. They define an upper bound on the estimated error between the bootstrapped target Q-value estimate and optimal Q^*-value (this error is referred as the target value error), and use it to propose a sampling strategy to down-weight the transitions with high target value error. This results in sampling transitions with more accurate target values, that are expected to maximally improve the accuracy of the Q function.

Additionally, we add recent works which provide extensions to DQN such as prioritized experience replay (PER) [46], Dueling networks [54], Noisy DQN [13] and Multi-step DQN [35]. Each of these us aimed to improve different parts of DQN. PER tries to use a heuristic while sampling, rather than the traditional uniform sampling in DQN. It assigns high importance to samples which have high TD error and the ones which are pushed recently to the replay buffer hypothesizing that these samples will facilitate faster learning. Noisy DQN tries to address better exploration in DQN. Dueling networks factorize the Q function into value and advantage functions. They argue that in some states, it is sufficient to learn only the state value function rather than state action value function. Multi-step DQN points out that using the bias variance trade-off in N-step returns is worth considering, as TD learning (1-step return) might not be optimal for all the tasks. All these extensions are proposed independent of one another. Rainbow [20] does a comprehensive study on which modifications and extensions are complementary and lead to a significant improvement in performance by combining all these approaches.

Chapter 3

CARLA Environment

In this chapter, we describe the CARLA simulator [12], its relevant components, and the additions required to use it in our reinforcement learning framework. Next, we also describe the official CARLA benchmarks, namely CoRL2017 and NoCrash, used in the experimental evaluation of the agent's performance.

3.1 Introduction

CARLA (Car Learning to Act) is an open-source simulator based on Unreal Engine 4 for Autonomous Driving research [12]. It is designed as a Server-Client system, where the server runs the simulation based on commands from client and physics engine, and renders the scene. The python-API based client sends control commands (of steer, throttle, brake) to the server and receives sensor readings. The steer control ranges between [-1.0, 1.0] whereas the throttle and brake commands range between [0.0, 1.0].

Since most of the prior works carried out using CARLA do not share their codebase, we build a CARLA-RL environment on top of the CARLA API's to have *reset()* and *step()* methods to design an OpenAI gym-like RL environment in CARLA which is not natively supported. An important thing to be noted is that we run the simulator at fixed time-step in the synchronous mode, which means the simulator halts each frame until a control message is received. This ensures that the generated data of every sensor is received in every frame by the client irrespective of client speed.

3.2 Sensors

CARLA provides a diverse set of sensors that can be attached to the driving agent and the sensor readings are received at each timestep.

We use the Semantic Segmentation Camera sensor, which provides 12 semantic classes for each pixel: road, lane-marking, traffic sign, sidewalk, fence, pole, wall, building, vegetation, vehicle, pedestrian, and other. We remap the semantic class labels to 5 relevant classes: Vehicle, Pedestrian, Road, Lane-marking, and Everything-Else.

We use the Lane-Invasion sensor to detect if the driving agent crosses any lane marking or enters side-walk. It uses road data provided by the OpenDRIVE[1] description of the map, and not the visible lane markings, to determine whether the parent vehicle is invading another lane by considering the space between wheels. We use the Collision sensor to detect if the driving agent collides with any static or dynamic objects in the world. Both these sensors register an event each time their triggering condition is fulfilled, and that event is further handled appropriately.

3.3 Waypoints

Waypoints in CARLA can be described as 3D-directed points, which contain the location on the map and the orientation of the lane containing them, as shown in 3.1. These are useful for navigation as they help to mediate between the world in the simulator and the OpenDRIVE definition of the road.

Given a source and a destination location, we compute intermediate waypoints at a fixed spatial resolution (2m) along the optimal path using A^* search algorithm [18] with distance heuristic. We note that this does not take into account any potential dynamic actors (other vehicles/pedestrians) along the optimal path.

In the goal-directed navigation tasks, we compute the trajectory waypoints at the beginning of the navigation from source to destination, and as the driving agent moves, we use the next n waypoints ahead of the agent as waypoints feature input, further detailed in Section 4.1.1. We compute the signed perpendicular distance of the driving agent from the line connecting the next 2 trajectory waypoints, and

[1]OpenDRIVE® is an open file format for the logical description of road networks. Refer http://www.opendrive.org/ for details.

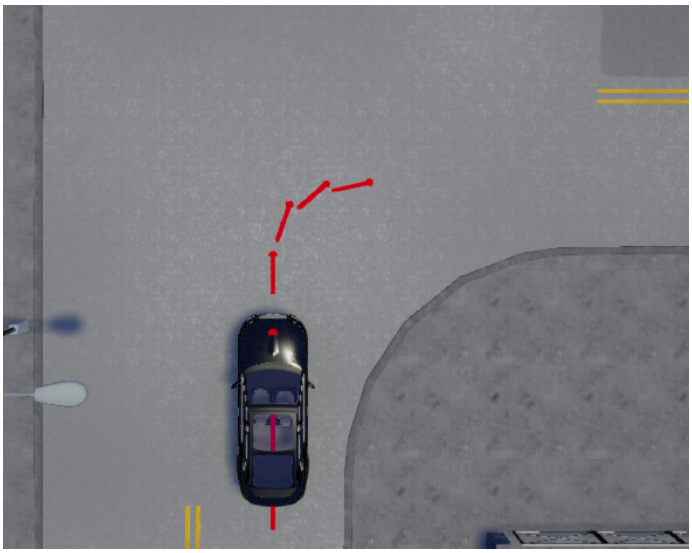

Figure 3.1: Waypoints (shown in red) are intermediate 3D-directed points containing location and orientation between a source and a destination location.

use it as *Distance-From-Trajectory* input. We also use these waypoints to compute the remaining distance to reach the goal along the trajectory, and refer to it as *Distance-To-Goal* input.

3.4 Dynamic Obstacle and Traffic Light Detection

To compute low dimensional inputs containing information about other dynamic obstacles (vehicles/pedestrians) and traffic light, we add a rule-based detector for vehicle and traffic light state using information from the CARLA simulator. We find the distance and speed of the vehicle right in front of the driving agent in the same lane within the vehicle-proximity-threshold (15m) distance. Since the CARLA benchmark towns contain only single-lane roads, detecting only the vehicle right in front of the driving agent is sufficient in the current setup. We query the state of the nearest traffic light in front of the driving agent within traffic-light-proximity-threshold (10m) distance from the simulator.

3.5 Benchmarks

In this section, we describe the two published autonomous driving benchmarks on the CARLA simulator, viz. CoRL2019 and NoCrash benchmarks.

3.5.1 CoRL2017 Benchmark

The CARLA CoRL2017 Benchmark [12], also referred as the Original CARLA Benchmark, consists of evaluating a driving agent on four goal-directed navigation tasks. In each task, the agent is initialized at a source location and needs to reach a destination location. Town 01 is used for training and Town 02 for testing. Each of the four tasks has 25 scenarios in each town. The tasks are described below.

1. **Straight**: Destination is straight ahead of the source location with no dynamic obstacles. Average goal distance is 200 m (Town 1) and 100 m (Town 2).
2. **One Turn**: Destination is one turn away from the source location with no dynamic obstacles. Average goal distance is 400 m (Town 1) and 170 m (Town 2).
3. **Navigation**: Destination can be anywhere relative to the source location with no dynamic obstacles. Average goal distance is 770 m (Town 1) and 360 m (Town 2).
4. **Navigation with dynamic obstacles**: Same as the Navigation task, but with dynamic obstacles (vehicles, pedestrians).

3.5.2 NoCrash Benchmark

As the original CARLA CoRL2017 benchmark does not consider collisions as failures, it is insufficient to realistically evaluate agent's driving behavior especially in scenarios with other dynamic actors. To address this, NoCrash benchmark [9] was proposed that considers collisions with both static and dynamic actors as failures. Note, it does not consider other infractions such as traffic light infractions, lane-invasion and off-road infraction as failures. The benchmark consists of goal-directed navigation tasks in three increasing levels of difficulty in terms of dynamic actors. The tasks are

Task	Description	Number of vehicles in Training Town (Town 01)	Number of vehicles in Testing Town (Town 02)
Empty Town	No dynamic objects	Cars: 0	Cars: 0
Regular Traffic	Moderate number of cars and pedestrians	Cars: 20	Cars: 15
Dense Traffic	Large number of cars and pedestrians	Cars: 100	Cars: 70

Table 3.1: Description of CARLA NoCrash Benchmark.

Empty Town, *Regular Traffic* and *Dense Traffic*, which are described in Table 3.1. *Town 01* is used for training and *Town 02* is used for testing.

3.6 Benchmark Differences across CARLA Versions

The above benchmark scenarios are defined on the older CARLA version 0.8.4 and we port them in the recent version CARLA 0.9.6 for evaluation in. Also, the CARLA 0.9.6 version has support for pedestrians to move only on the side-walks, and not across the road, in contrast to the older version which had support for pedestrian crossing. Hence, we are not able to evaluate with pedestrians crossing the road in the recent version CARLA 0.9.6.

Chapter 4

On-Policy Learning

In this chapter, we discuss our first work on formulating the problem of goal-directed navigation using reinforcement learning on the CARLA simulator [12]. We use Proximal policy optimization (PPO) [22, 47], an on-policy model-free RL algorithm to train the driving agents in this setup. First we describe the reinforcement learning setup, followed by the training methodology and proposed architecture. Then we describe the experimental setup and present results on the CARLA benchmarks.

4.1 Reinforcement Learning setup

In this section, we describe the reinforcement learning formulation for our problem of goal-directed navigation. This includes the definition of the state space, action space, and reward function.

4.1.1 State Space

We use the top-down semantically segmented image encoding and waypoint features as the state inputs for the RL agent. These are described in detail below.

Semantically Segmented Image

We choose the encoding of a top-down Semantically Segmented (SS) image as one of our state inputs. The SS image is easily obtained from the CARLA's

Semantic Segmentation Camera Sensor described in Section 3.2. Given the current state-of-the-art architectures in perception, we believe segmentation as a task can be trained in isolation and hence we focus on learning control agents using DRL directly from SS images. We use convolutional neural network based Auto-Encoder (AE) to reduce dimensionality of our SS image and use the bottleneck embedding as one of the inputs to the agent policy network (Figure 4.2). We refer to the AE bottleneck embedding as $\tilde{\mathbf{h}}$, and define it in Equation (4.1) with g being the encoder function of our AE.

$$\tilde{\mathbf{h}} = g(\mathbf{SS}_{\text{image}}) \tag{4.1}$$

Trajectory Waypoints

Apart from the SS image input, the agent also requires an input to guide its navigation. Previous approaches [12, 32] use high level commands for navigation viz. straight, turn-right, turn-left, lane-follow. Instead, we propose to use trajectory waypoints to guide navigation, as they are readily available in real world autonomous vehicles with GPS and map information, and can provide a more precise signal to the learning agent for navigation. We describe the trajectory waypoints generation logic in Section 3.3. We note that the waypoints are statically generated at the beginning of the navigation, and do not take into account any potential dynamic actor along the optimal path.

The waypoint features $\tilde{\mathbf{w}}$ can be computed using a generic function f defined by the next n waypoints $(\mathbf{w_1}, \mathbf{w_2}, ..., \mathbf{w_n})$ and agent's current pose $\mathbf{p}$. These features $\tilde{\mathbf{w}}$ form the second input to our agent policy network as defined in Equation (4.2).

$$\tilde{\mathbf{w}} = f(\mathbf{p}, \mathbf{w_1}, \mathbf{w_2}, ..., \mathbf{w_n}) \tag{4.2}$$

In our setup, we define the the waypoint feature function f as the mean waypoint orientation between the agent's current pose and the next $n = 5$ waypoints for simplicity, but it can extended to work with other functions as well. It is mathematically

defined in Equation 4.3 and shown in Figure 4.1.

$$\tilde{\mathbf{w}}_\theta = \frac{1}{n}\sum_{i=1}^{n}\left(\theta_{\mathbf{p}} - \theta_{\mathbf{w_i}}\right) \tag{4.3}$$

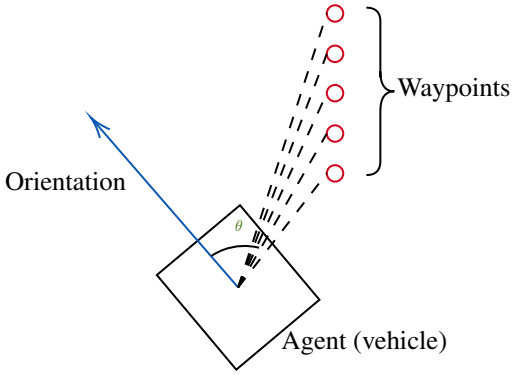

Figure 4.1: Average Waypoint orientation

4.1.2 Action Space

In CARLA, the control actions for the driving agent are defined by (s, t, b) where s is the steer, t is the throttle and b is the brake action. The s action ranges between [-0.5, 0.5], and the t and b actions range between [0.0, 1.0].

The input representation is then fed into our policy network $\pi(\hat{s}, \hat{v}|[\tilde{\mathbf{h}}\ \tilde{\mathbf{w}}])$ (Figure 4.2) which consists of a multi-layer perceptron and outputs $(\hat{s}, \hat{v})$, where $\hat{s}$ is the predicted steer action and $\hat{v}$ is the predicted target velocity for that timestep. To ensure better stability, we utilize a PID controller that computes the predicted throttle $\hat{t}$ and brake $\hat{b}$ actions.

4.1.3 Reward Function

We design a simple and dense reward function R that incentivizes our agent R_s based on its current speed u, penalizes R_d based on the perpendicular distance d from the nominal trajectory and incurs R_c if it collides with other actors or goes outside the road, denoted by indicator function $\mathbf{I}(c)$. Mathematically, our reward formulation

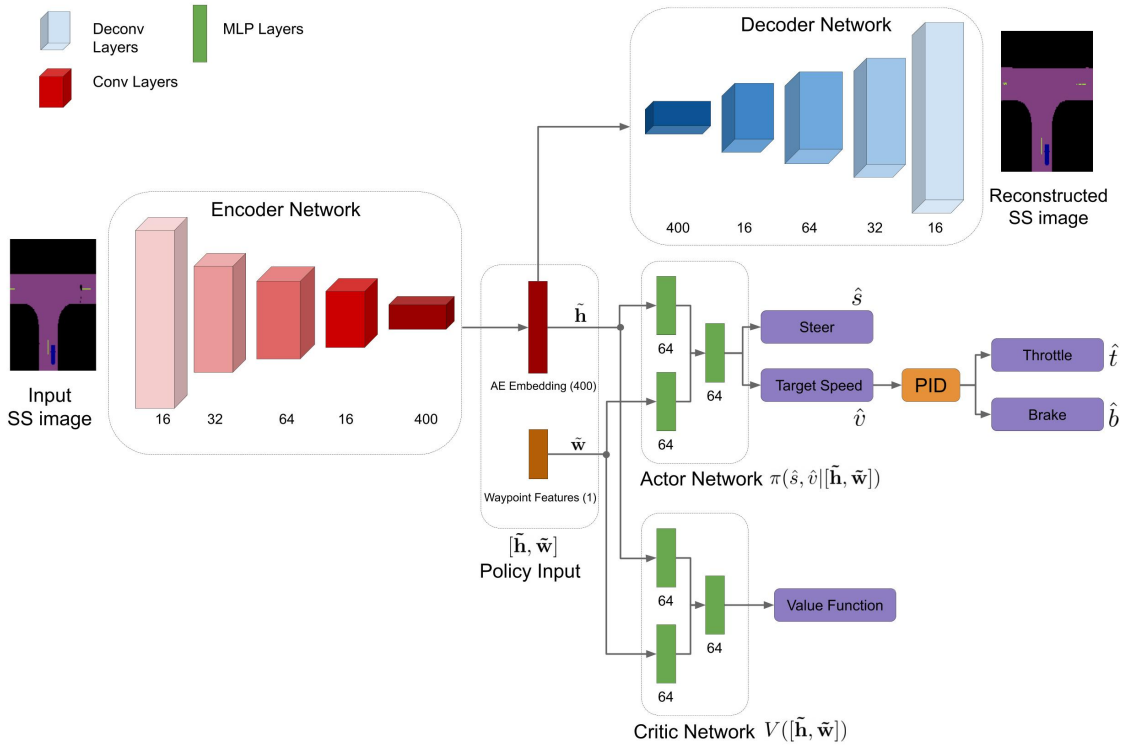

Figure 4.2: Proposed Architecture: The inputs to the above architecture are semantically segmented (SS) images and intermediate waypoints that are obtained from the CARLA simulator. The SS images are encoded using a pretrained auto-encoder whose bottleneck encoding alongwith waypoint features forms input to the policy network. The policy network outputs the control actions $(\hat{s}, \hat{v})$ where $\hat{s}$ is the predicted steer, $\hat{v}$ is the predicted target speed which is then mapped to predicted throttle and brake $(\hat{t}, \hat{b})$ using a PID controller.

can be described by Equation 4.4.

$$\begin{aligned} R &= R_s + R_d + \mathbf{I}(c) * R_c \\ R_s = \alpha * \mathrm{u}; R_d &= -\beta * \mathrm{d}; R_c = -\gamma * \mathrm{u} - \delta \end{aligned} \tag{4.4}$$

4.2 Methodology

We use state-of-the-art on-policy reinforcement learning algorithm of Proximal policy optimization (PPO) [47] to train our RL agent. This is done for each of the four

driving tasks - (a) Straight, (b) One Turn, (c) Navigation and (d) Navigation with dynamic obstacles, which are part of the CARLA[1] benchmark [12]. For each of the defined driving tasks, we set up each training episode as goal-directed navigational scenario, where an agent is initialized at a source location in town and has to reach to a destination location. The episode is terminated as a success case if the agent reaches within 10 m of the destination, while it is terminated as failure case if the agent faces a collision, or does not reach near the destination within the maximum number of timesteps (10,000).

In our setup, Town 1 is used for training and Town 2 for testing. Since the semantically segmented (SS) images contain a class label per pixel, the convolutional auto-encoder (AE) is trained to predict class label per pixel using reconstruction loss as the multi-class cross-entropy loss. The AE is pre-trained on SS images collected using an autonomous oracle agent in the training town to speed up agent policy training. The AE's bottleneck embedding ($\tilde{\mathbf{h}}$) and waypoint features ($\tilde{\mathbf{w}}$) are then fed into the agent policy network.

We found that fine-tuning AE on the diverse SS images seen during training helps in learning a better input representation, enabling the agent to learn a better policy. The agent policy network and AE are trained simultaneously and independently. For a fixed number of timesteps (n_{steps}), the AE is kept fixed and the agent policy is trained on transitions collected during those timesteps. Then, the AE is fine-tuned by optimizing on the most recent n_{steps} SS images collected in a buffer, and the training continues.

Figure 4.3 shows the mean cumulative reward and success rate achieved by our agent when validated in fixed intervals as the training progresses. We observe from this plot that our agent is able to learn the task in around 1.5M timesteps of training as it converges to the optimal policy.

4.3 Experimental Evaluation

For the experimental evaluation of our agent on the CARLA benchmarks, we use the following baselines available on the CARLA benchmarks. We directly report the

[1]The existing benchmark suite is on CARLA version 0.8.4 and we ported the same benchmark scenarios for evaluation in CARLA 0.9.6.

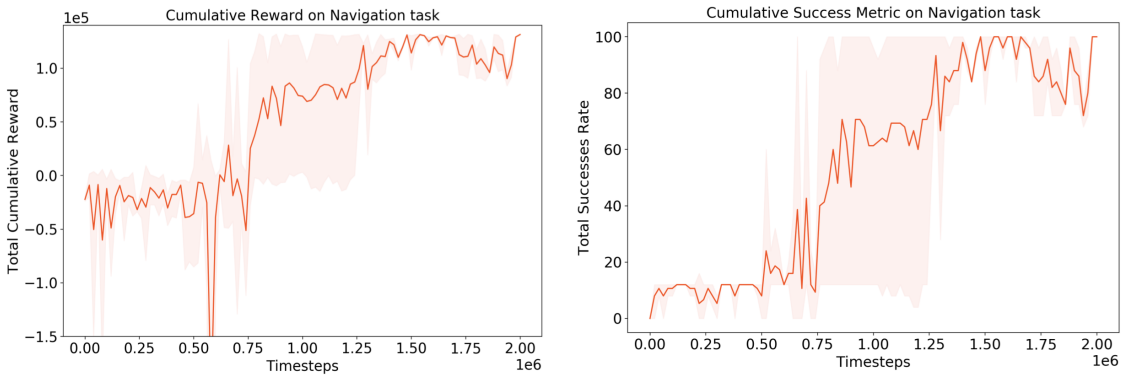

Figure 4.3: Mean reward and success metric reported on training our proposed setup on the Navigation task of the CoRL 2017 benchmark.

baseline numbers from their papers as most of the works do not open-source their implementations.

We acknowledge that the baseline methods use RGB images and high-level navigation features as input, in contrast to the semantically segmented images and richer low-level waypoint features used as input in our approach. The use of waypoint information is motivated by the fact that real-time waypoint and GPS information is readily available in real-world autonomous vehicles and hence, can be combined with other visual features for training agents using DRL. We use state-of-the-art baselines as a reference to evaluate our work.

- Modular Pipeline Approaches: CARLA *MP* [12] uses a vision-based module, a rule-based planner, and a classical controller. Automatic Control (*AT*) baselines refers to the CARLA built-in autopilot control that has access to simulator states and uses a classic rule-based approach to determine optimal control. *CAL* [45] uses a separate visual encoder to predicts low-dimensional representations, referred as Affordances, that are the relevant features for driving.
- Imitation Learning Approaches: CARLA *IL* [12] is a standard IL approach using a deep network to map sensor inputs to driving commands. *CIL* [8] and *CILRS* [9] use a conditional imitation learning pipeline to learn a driving policy from expert demonstrations of low-level control inputs, conditioned on the high-level navigation commands. *LBC* [6] uses a privileged agent to learn driving from privileged information (simulator states). The sensorimotor agent then learns

CARLA CoRL2017 Benchmark (% Success Episodes)										
Task	**Training Conditions (*Town 01*)**									
	MP	*IL*	*RL*	*CIL*	*CIRL*	*CAL*	*CILRS*	*LBC*	*IA*	*Ours*
Straight	98	95	89	98	98	**100**	96	**100**	**100**	**100**
One Turn	82	89	34	89	97	97	92	**100**	**100**	**100**
Navigation	80	86	14	86	93	92	95	**100**	**100**	**100**
Dyn. Navigation	77	83	7	83	82	83	92	**100**	**100**	**100**
Task	**Testing Conditions (*Town 02*)**									
	MP	*IL*	*RL*	*CIL*	*CIRL*	*CAL*	*CILRS*	*LBC*	*IA*	*Ours*
Straight	92	97	74	97	**100**	93	96	**100**	**100**	**100**
One Turn	61	59	12	59	71	82	84	**100**	**100**	**100**
Navigation	24	40	3	40	53	70	69	98	**100**	**100**
Dyn. Navigation	24	38	2	38	41	64	66	99	98	**100**

Table 4.1: Quantitative comparison with our chosen baselines that solve the four goal-directed navigation tasks using modular, imitation learning or reinforcement learning approaches on the CARLA CoRL2017 Benchmark [12]. The table reports the percentage (%) of successfully completed episodes for each task and for the training (*Town 01*) and testing town (*Town 02*). **Higher** is better. The baselines include *MP* [12], *IL* [12], *RL* [12], *CIL* [8], *CIRL* [32], *CAL* [45], *CILRS* [9], *LBC* [6] and *IA* [51] compared with our PPO + AE method. The results reported are the average over 3 seeds that are evaluated on 5 different runs of the benchmark.

to drive from sensor inputs using supervision by the privileged agent.

- Reinforcement Learning Approaches: CARLA *RL* [12] is the first RL approach on the CARLA benchmark which uses the A3C algorithm to do end-to-end RL on sensor inputs. *CIRL* [32] uses a pre-trained imitation learned policy and then improves it further using the DDPG algorithm. *IA* [51] is very recent work that learns a ResNet encoder to predict relevant features, referred as implicit affordances and uses encoding to learn a driving policy using an advanced variant of DQN algorithm, namely Rainbow IQN Ape-X.

4.4 Results

We observe from our results (Table 4.1) on the CARLA CoRL2017 Benchmark that the RL agent is able to learn all the tasks on both the towns, Town 01 and Town 02. Further, we note from the table that the success metric achieved by our agent is

CARLA NoCrash Benchmark (% Success Episodes)							
Task	**Training Conditions (*Town 01*)**						
	CIL	*CAL*	*CILRS*	*LBC*	*IA*	*AT*	*Ours*
Empty	79 ± 1	81 ± 1	87 ± 1	97 ± 1	**100**	$\mathbf{100 \pm 0}$	$\mathbf{100} \pm 0$
Regular	60 ± 1	73 ± 2	83 ± 0	93 ± 1	96	$\mathbf{99 \pm 1}$	52 ± 6
Dense	21 ± 2	42 ± 1	42 ± 2	71 ± 5	70	$\mathbf{86 \pm 3}$	19 ± 2
Task	**Testing Conditions (*Town 02*)**						
	CIL	*CAL*	*CILRS*	*LBC*	*IA*	*AT*	*Ours*
Empty	48 ± 3	36 ± 6	51 ± 1	$\mathbf{100} \pm 0$	99	$\mathbf{100 \pm 0}$	$\mathbf{100} \pm 0$
Regular	27 ± 1	26 ± 2	44 ± 5	94 ± 3	87	$\mathbf{99 \pm 1}$	45 ± 5
Dense	10 ± 2	9 ± 1	38 ± 2	51 ± 3	42	$\mathbf{60 \pm 3}$	12 ± 1

Table 4.2: Quantitative comparison with the chosen baselines that solve the three goal-directed navigation tasks using modular, imitation learning or reinforcement learning approaches on the CARLA NoCrash Benchmark [9]. The table reports the percentage (%) of successfully completed episodes for each task and for the training (*Town 01*) and testing town (*Town 02*). **Higher** is better. The baselines include *CIL* [8], *CAL* [45], *CILRS* [9], *LBC* [6], *IA* [51] and CARLA built-in autopilot control (*AT*) compared with our PPO + AE method. The reported results are the average over 3 seeds that are evaluated on 5 different runs of the benchmark.

higher than all baseline methods that we use for comparison. This increase in success on the benchmark task can be attributed to our simplified input representation.

Next, we want to also add that the CARLA CoRL 2017 benchmark does not terminate the episode on collision, using which our agent achieves almost 100% success rate even on the Task (d) (Navigation with dynamic obstacles) where it keeps on colliding with other vehicles, but finally reaches the destination. Hence, we chose to evaluate on a more realistic setting of CARLA NoCrash Benchmark in which we terminate the episode on collision and count it as a failure case.

As we see from our results on the CARLA NoCrash Benchmark (Table 4.2), we observe that we get perfect performance on the *Empty* town task whereas on the *Regular* and *Dense* traffic tasks we observe a drop in performance due to the effect of not learning the brake. We find that the agent does not learn to stop in presence of dynamic actors, and the marginal success rate achieved in *Regular* and *Dense* traffic tasks are in cases where any dynamic actor does not come in front of our agent.

4.5 Discussion

In this chapter, we present an approach to use waypoints as the navigation input and design an architecture to learn planning and control directly from semantically segmented images and waypoints using an on-policy RL algorithm (Proximal policy optimization). The agent successfully learns to navigate, except in the presence of dynamic actors. It is unclear whether the issue lies in learning a good representation from the semantically segmented image or in learning a good policy despite a good representation. To decouple these issues in representation and policy learning, we focus on learning from a simple low-dimensional input space to understand the challenges in learning to drive with dynamic actors. The follow-up to this work on using on-policy reinforcement learning with low-dimensional state space is presented in [1]. We focus on using off-policy reinforcement learning with low-dimensional state space in the next chapter.

Chapter 5

Off-Policy Learning

In this chapter, we describe our work on using an off-policy reinforcement learning algorithm, Deep Q-Learning (DQN), to learn goal-directed navigation in the presence of dynamic actors using low-dimensional state representation. We first present the reinforcement learning (RL) formulation for our problem setup. Then we discuss preliminary experiments and analysis on simplified tasks, which highlights the issues in learning with dynamic actors and motivates our proposed approaches of *Backward-Sampling* and Clipped Double DQN which are further explained in the following section. Next, we describe our preliminary work to combine expert agent demonstrations with the RL agent experience to help in faster learning. This is an initial step towards our long-term goal to utilize large amounts of existing experts' driving data and limited amounts of on-policy data to learn policies with the maximum possible utility. Thereafter, we introduce our experimental setup and present results on the CARLA benchmarks.

5.1 Reinforcement Learning Formulation

In this section, we describe the reinforcement learning formulation for our problem of goal-directed navigation. This includes the definition of the state space, action space, reward function and episode termination conditions.

5.1.1 State Space

Here, we describe the state space of the RL agent. We use a low-dimensional state space to focus on policy learning in the presence of dynamic actors. We use the waypoint features as the navigation input, and add additional input to capture the agent's current state, viz. current speed, previous steer action and distance from the trajectory. To handle dynamic actors, we add inputs of front dynamic actor distance and speed, and traffic light state. These are described in detail below.

1. **Waypoints Orientation**
   We continue to use the trajectory waypoints as the navigation input. Specifically, we use the average waypoint orientation (in radians) of the next $n = 5$ waypoints as explained earlier in Chapter 4. It ranges from [-4.0, -4.0].
2. **Agent's Current Speed**
   The current speed of the agent in m/s normalized to be in the interval [0, 1] by dividing by 10.
3. **Agent's Previous Steer Action**
   The steer action taken by the agent in the previous timestep. Since the CARLA simulator does not provide the current steer value of the agent, we use the last steer action as a proxy to provide current state information related to steering. It ranges from [-0.5, 0.5].
4. **Signed Distance From Waypoints Trajectory**
   We use the signed perpendicular distance of the driving agent from waypoints trajectory, described as *Distance-From-Trajectory* input in Section 3.3. This provides information on the agent's position relative to the optimal waypoint trajectory, where the negative sign indicates the agent is on the left side of trajectory and the positive sign indicates that it is on the right side. Its value ranges from [-1.0, 1.0] when the agent is moving on the straight road, and gets larger on turns ranging from [-2.0, 2.0]. If the agent moves far away from the trajectory, its value can get further large, and it can be used to identify that the agent has drifted far from the trajectory.
5. **Front Obstacle Distance**
   We detect the obstacle in front of the driving agent within the obstacle-proximity-threshold

distance (15m) and use the normalized distance as the input such that its range is [0.0, 1.0]. If there is no obstacle within the obstacle-proximity-threshold, we set the value as 1.0 to ensure monotonic input space. The computation logic is described in Section 3.4.

6. **Front Obstacle Speed**
   This is similar to the Front Obstacle Distance input. If there is an obstacle within the obstacle-proximity-threshold, we take its normalized speed in m/s, otherwise set the value as 1.0. It ranges from [0.0, 1.0].

7. **Traffic Light State/Distance**
   We encode the nearest traffic light distance and state in this state input. If the traffic light state is RED and within the traffic-light-proximity threshold (10m), we set the input's value as the normalized distance to traffic light with value range as [0.0, 1.0]. If there is no traffic light within the proximity threshold, or the traffic light state is GREEN or YELLOW, we set the input's value as 1.0.

   The CARLA benchmark tasks do not require the driving agent to follow traffic light rules as per the success definition in the benchmark. However, we find that it becomes difficult for the driving agent to cross road-intersections without collision, if the agent does not have the knowledge of the traffic light state. This is because the other automatic agents in CARLA cross the intersections following traffic light rules.

5.1.2 Discrete Action Space

Similar to the RL setup in Chapter 4, the action space consists of steer and target speed. Since DQN works with discrete action space, we discretize the continuous control space into 12 discrete actions. We use 7 steering values [-0.5, -0.3, -0.1, 0.0, 0.1, 0.3, 0.5], which we find are sufficient to take turns in the CARLA towns. We use two values of target speed, 0 and 20 kmph for all the steer values, except for [-0.5, 0.5] for which we use only 20 kmph. All the actions are listed in Table 5.1. The actions could be discretized further using more number of steering and target speed values to allow finer control.

Action	0	1	2	3	4	5	6	7	8	9	10	11
Target-Speed	0	0	0	0	0	20	20	20	20	20	20	20
Steer	-0.3	-0.1	0.0	0.1	0.3	-0.3	-0.1	0.0	0.1	0.3	-0.5	0.5

Table 5.1: Discrete Action Space: The continuous control action space is discretized into the above discrete actions to work with DQN. The Target-Speed (in kmph) and Steer values (in normalized units in the range [-1.0, 1.0]) corresponding to each action are listed.

5.1.3 Reward Function

We design a simple and dense reward function extending the reward function defined previously in Section 4.1.3, defined in equation 5.1.

$$R = R_{\text{Speed}} + R_{\text{Steer}} + R_{\text{TrajectoryDist}} + R_{\text{Infraction}} + R_{\text{Success}} \tag{5.1}$$

We define each of the reward components below:

- R_{Speed} incentivizes the agent to move at high speed by giving a reward proportional to its current speed.
- R_{Steer} penalizes the agent to high steer values by giving a penalty proportional to its current steer. This is added to address the observed shaky driving behavior caused alternating between high values of steer, specifically in case of discrete action space.
- $R_{\text{TrajectoryDist}}$ incentivizes the agent to follow the optimal trajectory from goal to the destination by giving a penalty proportional to perpendicular distance from the trajectory. A constant reward is added to keep the TrajectoryDist reward positive in most cases, such that the agent does not keep accumulating negative rewards over time when it is stopped very close to the trajectory.
- $R_{\text{Infraction}}$ refers to the penalty provided in case of different infractions, namely collisions, lane-invasions, off-road infractions, and traffic light infractions. In case of infractions, a constant penalty and a penalty proportional to the speed of the agent at the time of the infraction is provided to incentivize the agent to gradually learn to slow down and eventually stop before causing infractions.
- R_{Success} refers to the success reward provided to the agent on reaching the goal.

Since the agent's state space does not include any goal state information, adding a success reward based on the discount factor accounts for the correct expected reward for the state if it was not the goal state.

5.1.4 Episode Termination

Each episode is set up as a goal-directed navigation scenario, where the agent is initialized at a source location in town and has to reach to a destination location. The episode is terminated as a success case if the agent reaches within 10 m of the destination. It is terminated as a failure case if the agent faces any infraction (collision, lane-invasion, off-road infraction, traffic light infraction). It is also terminated as a *MaxSteps* failure case if it does not reach near destination within the maximum threshold (5000 in our setup) of total episode timesteps.

The recent work of time limits in RL [40] suggests bootstrapping the value of the state at the end of the episode terminated due to max steps, as time limits are not part of the environment and are only used to facilitate learning. We note that in our setup we rarely observe termination due to max steps in training due to the epsilon-greedy exploration policy, so we do not need to explicitly account for this. Nonetheless, we recommend use of this strategy for correct estimates of the terminal states.

5.2 Preliminary Analysis

In this section, we discuss preliminary experiments and analysis on simplified tasks, which highlights the issues in learning with dynamic actors and motivates our proposed approaches of *Backward-Sampling* and Clipped Double DQN, that are explained in the next section.

5.2.1 RL Algorithm

We use Double DQN [52], which applies the Double Q-learning technique [19] to the original DQN [33] to reduce the over-estimation bias in DQN. Additionally, we use dueling neural network architecture [54] which factorizes the Q function into

state-value function and state-dependent action advantage function. This can enable the RL agent to learn which states are valuable or not, without having to learn the effect of each action for each state, which has shown improved performance in various settings.

5.2.2 Experiment Setup for Preliminary Analysis

In the initial experiments using DQN, we observe that the agent learns to navigate in scenarios without the dynamic actors, where it learns the steer action and outputs the maximum target-speed at all times, which is optimal in that setting. However, it fails to learn to stop in scenarios with dynamic actors and collides with them. For understanding and analysis of this issue with fast learning iterations, we design simplified tasks described below.

Simplified Task 1 - Straight-Dynamic: In this task, the agent's goal is to drive within-lane on a fixed long straight road and reach the end of the road without colliding with the other dynamic actors and causing traffic light infractions. The 300 m straight road consists of 3 traffic-light intersections and dense traffic to ensure the agent can learn to stop before other actors and the RED traffic light. We refer to this as the *Straight-Dynamic* task as shown in Figure 5.1. For this task, we could have the agent only learn the target-speed action to move or stop, and fix steer action to zero as it needs to drive straight. However, there is some inherent noisy drift in the CARLA simulator, where the steer of zero would sometimes end up drifting to the other lane or the curb. Hence, we choose a reduced set of steering values of [-0.1, 0.0, 1.0] within the action space to drive straight successfully. The reduced action space consists of six actions listed in table 5.2.

Simplified Task 2 - Straight-Dynamic-TJunction: After the agent learns to drive on a straight road in the *Straight-Dynamic* task, we extend the task by adding two more scenarios at a T-junction, where the agent needs to learn to take left and right turns while respecting the traffic lights at the junction. We believe that these 3 scenarios (straight-dynamic, left turn and right turn) are representative of the broader navigation with dynamic actors task, where the agent needs to learn to

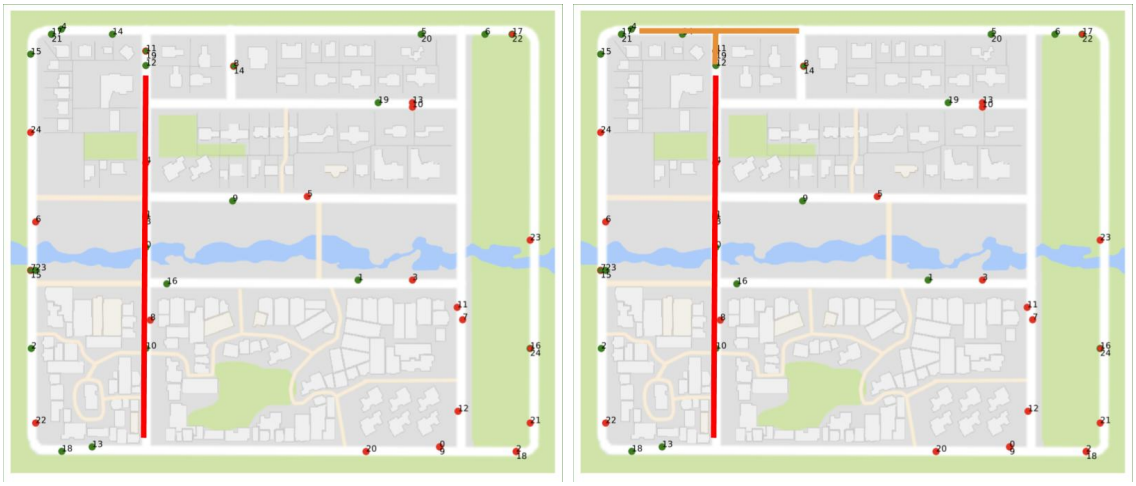

Figure 5.1: The simplified tasks, *Straight-Dynamic* (left) and *Straight-Dynamic-TJunction* (right) shown on the map of Town01 in CARLA.

Action	0	1	2	3	4	5
Target-Speed	0	0	0	20	20	20
Steer	-0.1	0.0	0.1	-0.1	0.0	0.1

Table 5.2: Reduced Discrete Action Space: The continuous control action space is discretized into the above reduced discrete actions to work with DQN in the *Straight-Dynamic* task. The Target-Speed (in kmph) and Steer values (in normalized units in the range [-1.0, 1.0]) corresponding to each action are listed.

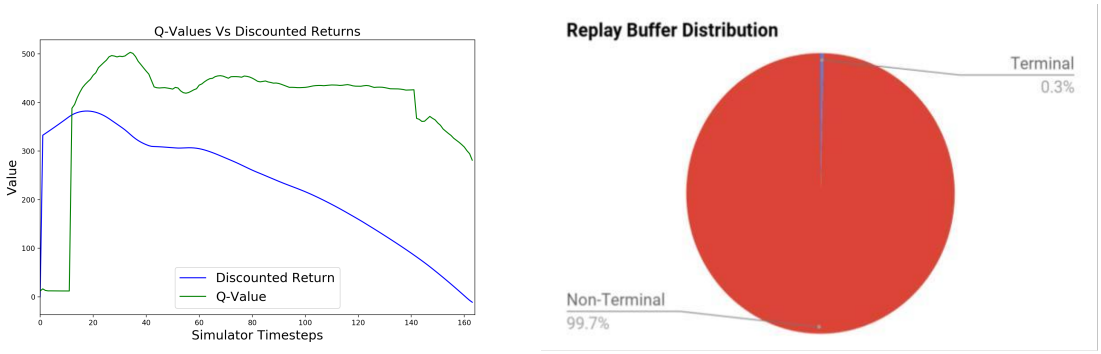

Figure 5.2: The left figure shows the Q-values and Discounted Returns for the states observed and actions taken by the agent in an episode. This shows that there is a wide difference in the Q-values and discounted returns, specifically for the terminal state-action pair. The right figure shows the replay buffer distribution of the DDQN agent showing the that terminal states constitute only 0.3% of all the transitions.

control both steer and target-speed correctly. Hence, the agent uses the complete action space in this task as described in Section 5.1.2. The motivation for this task is to iterate faster on different design choices and hyperparameters to find an optimal setting, which can be then used in the benchmark tasks. We refer to this task as the *Straight-Dynamic-TJunction* task, and it is shown in Figure 5.1

5.2.3 Observations and Analysis

Here, we discuss the issues seen on the simplified tasks, and our approaches to address them.

***Straight-Dynamic* Task**: In this task, we observe that the agent does not learn to stop and collides with other dynamic actors despite a collision penalty proportional to the agent's speed, which we would expect to provide a signal to the agent to slow down and eventually stop before other actors to get higher rewards. To analyze this, we look at the agent's Q-values for the state-action pairs (states observed and actions taken by the agent) and the cumulative discounted returns during an episode, as shown in Figure 5.2. We observe that there is a wide gap in the Q-values and discounted returns, specifically for the terminal state-action pair. This is concerning since the TD-target for the terminal transition depends only on the observed reward and is free from any potential errors due to bootstrapping. In bootstrapping-based

Q-learning algorithms, it is imperative for the terminal states to have correct Q-values to propagate the corrected TD-target estimations to the upstream states, which would lead to eventual convergence to optimal Q-values. To understand this further, we look at the distribution of transitions in the replay buffer, shown in Figure 5.2, where we see that there are only 0.3% terminal transitions in 1M sized replay buffer, which although reasonable for the average episode length of 300, leads to limited sampling of these terminal transitions. We also observe that all the terminal states have a fair distribution of different actions, which suggests that exploration may not be the issue in this setup.

We explore the following approaches to address the issue:

- We experiment with Prioritized Experience Replay (PER) [46], which samples transitions proportional to their TD-error with importance sampling correction for introduced bias. In general, PER has shown faster learning and improved performance on various benchmarks, however, it does not seem to help significantly in our setup as shown in Figure 5.3. PER prioritizes sampling the state-action pairs with high TD-error, but the high error values might be due to upstream errors in bootstrapped Q-value estimates of other states, in which case, it may not help to correct those errors.
- As PER does not seem to mitigate the issue, we try to address this issue by proposing an intuitive sampling strategy to sample terminal transitions and those close to episode termination more frequently and augment it with uniformly sampled transitions. We refer to this as **Backward Sampling**, and explain it algorithmically in the next section. It is encouraging to see that recent concurrent works of DisCor [28] and EBU [29] also discuss similar issues and propose sampling strategies to address them as explained in the related works section. We observe that Backward-Sampling seems to help in this simplified *Straight-Dynamic* task as shown in Figure 5.3, as the agent learns to stop and avoid collisions with other dynamic agents. We evaluate and compare it with uniform and PER sampling strategies on the comprehensive benchmarks tasks and share findings in section 5.5.

***Straight-Dynamic-TJunction* Task**: We observe that the agent learns to perform the *Straight-Dynamic* task, and the *T-Junction* task independently. However,

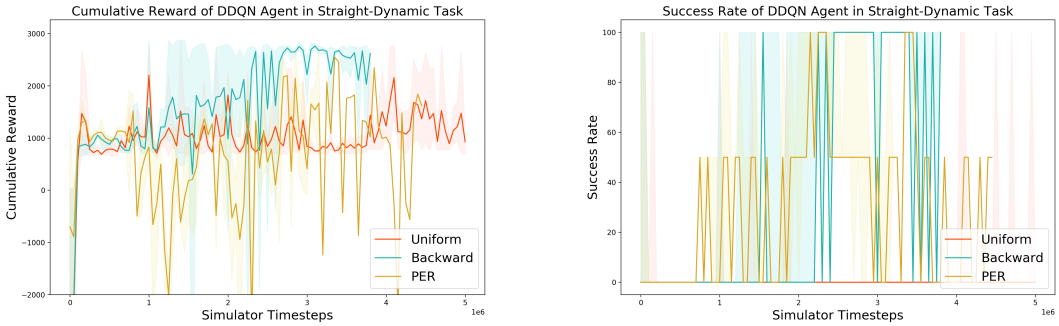

Figure 5.3: The figure shows the training performance for the DDQN agent on the Straight-Dynamic Task. We observe that the agent learns the task using Backward-Sampling, but does not learn using Uniform-Sampling and PER.

when we combine them in the *Straight-Dynamic-TJunction* task, surprisingly the agent does not learn all of them. In most cases, we observe that the agent learns to stop for dynamic actors and traffic light, and learns to take the right turn, but fails to take the left turn stably. In some cases, the agent learns all the tasks being stable for around 100-200K training steps, and thereafter, forgets to take the left turn pointing to some instability in the policy training, as shown in Figure 5.6. We experiment with deeper neural networks to increase the capability of the network, and slower training (slower target network update frequencies and lower learning rates), which seem to help marginally, but the instability issue remains.

We analyze this by looking at the agent's average episodic Q-value and discounted returns across training, depicted in Figure 5.4. We observe that Q-values for the states with dynamic actors diverge drastically as the training proceeds. Although the agent policy with diverged Q-values works optimally for the dynamic actor case as the agent learns to stop before them, the Q-values are orders of magnitude higher than the expected returns in those states. We suspect that Q-values divergence in these states could be affecting Q-values for correlated states and making it difficult to learn a stable policy. This is a known issue in Q-learning due to the inherent overestimation bias [49], and thus we use the Double DQN variant as the initial baseline, but that does not seem to solve this issue. To address it, we add the **Clipped Double Q-learning** (clipped-DDQN) formulation from TD3 [14] in our DQN setup as explained in Section 5.3. We find that clipped-DDQN does not seem

to help address the Q-value divergence issue as shown in Figure 5.5. The training performance comparison of DDQN and clipped-DDQN agents is shown in Figure 5.6, where both the agents report high performance for around 100-200K training steps, and thereafter become unstable. Although clipped-DDQN seems to marginally slow down the rate of Q-value divergence, it does not address the underlying issue causing the divergence. Nonetheless, to evaluate the agent's performance on the benchmarks, we experiment training the DDQN agent on the complete *Dynamic Navigation* task of the CARLA CoRL2017 benchmark. Interestingly, we observe that somehow the DDQN agent does not face the Q-value divergence issue in this case as shown in Figure 5.7. It is possible that training on a larger number of scenarios adds as a form of regularization which prevents or delays the Q-value divergence. The agent achieves high performance across all sampling strategies without the need for clipped formulation as shown in Figure 5.9, later in the results section. We also evaluate and compare clipped-DDQN with DDQN on the comprehensive benchmarks tasks and share findings in the results section.

5.3 Methodology

5.3.1 Double DQN with Backward-Sampling

As motivated in the previous section, we propose *Backward-Sampling*, an intuitive sampling strategy to sample terminal transitions and the transitions close to episode termination more frequently and augment it with uniformly sampled transitions. It is based on the idea that it is important for terminal states to have correct Q-value estimates to propagate corrected TD-target estimations to the upstream states, which would potentially lead to faster convergence to optimal Q-values. It is explained pictorially in Figure 5.8.

The pseudocode for the integration of Backward-Sampling in DQN is presented in Algorithm 1. For each transition, we compute the remaining timesteps to termination and refer them as *StepsToTerm*. For example, *StepsToTerm* is zero for all terminal transitions, 1 for the transitions just before the terminal and so on. *StepsToTerm* can be computed only after an episode terminates either inherently by the environment or terminated due to MaxSteps. Hence, we modify the DQN algorithm to store

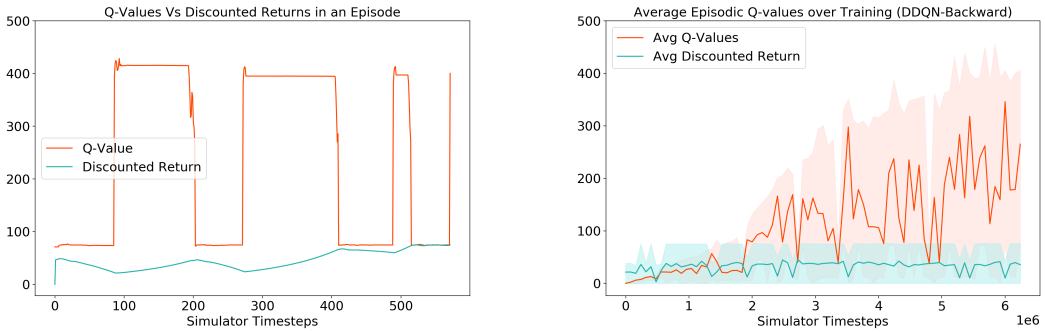

Figure 5.4: Q-value divergence in DDQN agent on *Straight-Dynamic-TJunction* task. The left figure shows the Q-values and Discounted Returns for the states observed and actions taken by the agent at each timestep in an episode. The high Q-values shown by peaks in between timesteps [90-200], [280-410] and [490-510] correspond to the states where a dynamic obstacle is present in front of the agent. The right figure shows the average episodic Q-values and Discounted Returns for the states observed and actions taken by the agent during training. The solid lines show the average values, while the shaded region shows the minimum and maximum values. It is important to consider the maximum Q-value as the Q-values are observed to diverge only for states where a front dynamic obstacle is visible for the agent as seen in the left figure. We observe the maximum Q-value keeps on increasing and diverging from the returns as the training proceeds which leads to instability.

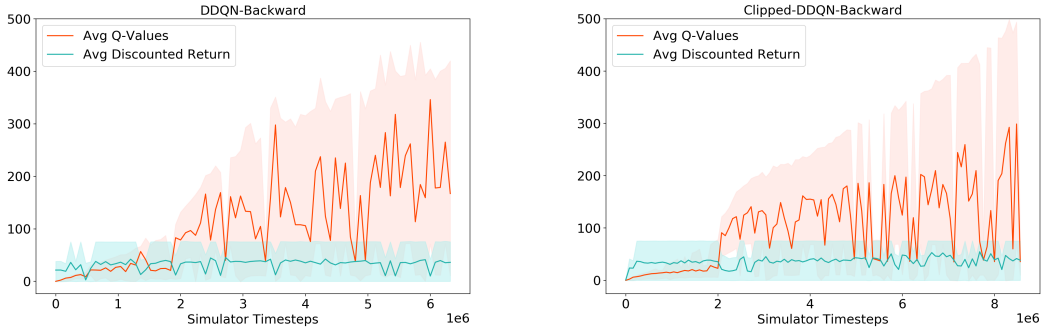

Figure 5.5: Q-value divergence in DDQN and Clipped-DDQN agents on *Straight-Dynamic-TJunction* task. The figure shows the average episodic Q-values and Discounted Returns for the states observed and actions taken by the agent during training. The solid lines show the average values, while the shaded region shows the minimum and maximum values. The left sub-figure is for the DDQN-Backward agent and the right sub-figure corresponds to the Clipped-DDQN-Backward agent. We observe that using Clipped-DDQN seems to be marginally slow down the rate of Q-value divergence but it does not address the issue as the Q-values continue to diverge as the training proceeds.

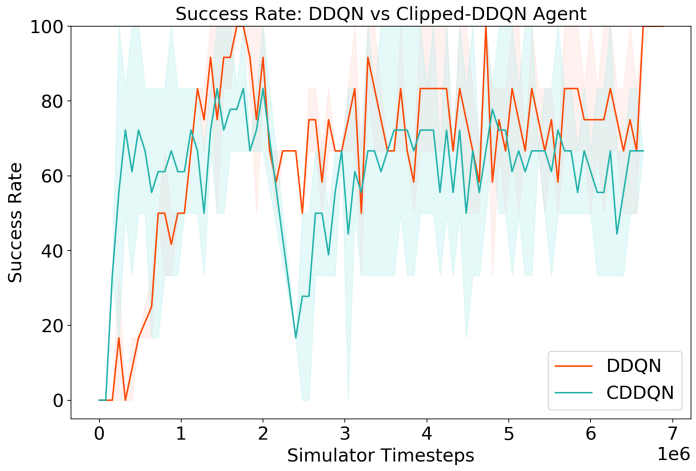

Figure 5.6: Training Success Rate plot for DDQN and clipped-DDQN agents on *Straight-Dynamic-TJunction* task. We observe both the agents report high performance for around 100-200K training steps initially during training, and thereafter become unstable. Although clipped-DDQN seems to marginally slow down the rate of Q-value divergence, it does not address seem to address the underlying issue causing the divergence.

Algorithm 1 Deep Q-Learning with Backward Sampling.
The Backward-Sampling related changes in the original DQN algorithm are shown in red.

Initialize replay memory $\mathcal{D}$ capacity $= N$
Initialize Buffer.MaxStepsToTerm = K
Initialize the list of sampling proportions for the k steps-to-termination values to L $= [P_0, P_1, ..., P_k]$
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta' = \theta$
Initialize a local Episode Transitions List, E as an empty list
for *episode* $= 1, M$ **do**
    Reset $E = [\,]$
    for $t = 1, T$ **do**
        //Note: T denotes the end of episode, and its value can vary across episodes
        Select a random action a_t with probability ε
        Otherwise, select $a_t = \arg\max_a Q(s_t, a; \theta)$
        Execute action a_t, collect reward r_{t+1}, observe next state s_{t+1}, done d_t
        Store the transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in E
        // Sample using BackwardSample
        Sample a mini-batch of size M of transitions $(s_j, a_j, r_{j+1}, s_{j+1}, d_j)$ from $\mathcal{D}$ using Buffer.BackwardSample(M, L)
        Set $y_j = \begin{cases} r_{j+1}, & \text{if } s_{j+1} \text{ is terminal} \\ r_{j+1} + \gamma \max_{a'} \hat{Q}(s_{j+1}, a'; \theta'), & \text{otherwise} \end{cases}$
        Perform a gradient descent step using targets y_j with respect to the online parameters θ
        Every C steps, set $\theta' \leftarrow \theta$
    end for

    // Compute Steps-To-Termination for each transition after the episode termination and Add to replay buffer $\mathcal{D}$
    Reset steps-to-termination variable, $h = 0$
    for $i = T, 1$ **do**
        $(s_t, a_t, r_{t+1}, s_{t+1}, d_t) = E[i]$
        if $h < K$ **then**
            Store the transition $(s_t, a_t, r_{t+1}, s_{t+1}, d_t, \text{StepsToTerm} = h)$ in $\mathcal{D}$
        else
            Store the transition $(s_t, a_t, r_{t+1}, s_{t+1}, d_t, \text{StepsToTerm} = -1)$ in $\mathcal{D}$
        end if
        Increment h by 1
    end for
end for

Figure 5.7: Episodic Average Q-values and Discounted Returns on *Dynamic-Navigation* task. We observe that somehow the DDQN agent does not face the Q-value divergence issue when training on a larger number of scenarios in the *Dynamic-Navigation* task.

Algorithm 2 Replay Buffer: BackwardSample

Initialize Buffer.MaxStepsToTerm = K
procedure BACKWARDSAMPLE(N, L)
    Inputs: BatchSize, N
    L : List of sampling proportions for the K StepsToTerm values = $[P_0, P_1, ..., P_K]$

    // Sample based on Steps to Termination (StepsToTerm)
    Initialize SamplesCount, $s = 0$
    for i = 0 to K **do**
        $N_i = P_i * N$
        Samples, S_i = Buffer.Sample(N_i, StepsToTerm=i)
        s += N_i
    end for

    // Sample remaining transitions uniformly
    $S_{uniform}$ = Buffer.Sample(N - s, StepsToTerm= -1)

    return All Samples: $[S_0, S_1, ..., S_K, S_{\text{uniform}}]$
end procedure

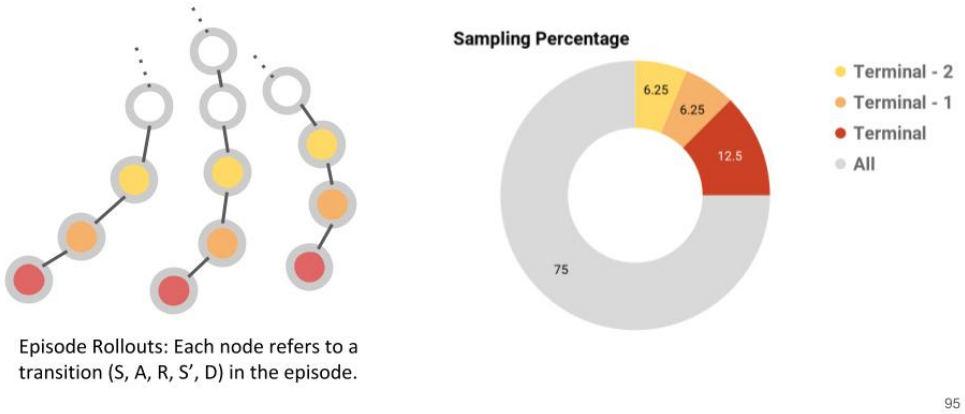

Figure 5.8: Pictorial Explanation of Backward Sampling Approach: Episode rollouts are shown in the left figure where each line graph corresponds to an episode and each node corresponds to a transition in the episode. These transitions are color-coded as red nodes (Terminal), orange nodes (Terminal-1), yellow nodes (Terminal-2) and so forth. Grey color corresponds to all the nodes including the terminal nodes. The idea behind Backward Sampling is to sample the transitions close to the termination state more frequently from the replay buffer. An example of sampling proportion used in Backward Sampling is shown on the right, where the transitions close to termination depicted by red, orange and yellow nodes are sampled in proportions of 12.5%, 6.25%, 6.25% respectively. This is augmented with uniform sampling from all the transitions, depicted by 75% sampling proportion of all the nodes denoted by grey border.

the observed transitions in a temporary Episode Transitions List during the episode rollout (line 9), and after the episode terminates, we compute the *StepsToTerm* for each transition and add it to the Replay Buffer (lines 22-32 in Algorithm 1). We define $MaxStepsToTerm = K$ as the maximum steps before termination which we would like to sample more frequently, and need to choose sampling proportions for each of the K values as $[P_0, P_1, ..., P_K]$, based on which they are sampled in a mini-batch from replay buffer along with the uniform samples comprising the remaining proportions. We also add support for Backward-Sampling in replay buffer, for which the pseudocode is shown in Algorithm 2. The main support added is the ability to sample transitions with a particular *StepsToTerm* value. For this, we maintain additional reverse index lists to store indices of the transitions for each of the K *StepsToTerm* values (less than *MaxStepsToTerm*), so that it is computationally efficient to backward-sample the transitions at runtime. We use the value of *StepsToTerm = -1* to refer to transitions further than *MaxStepsToTerm* which are not required to be stored in the reverse index lists. We also use the value of *StepsToTerm = -1* to sample uniformly from all transitions in the buffer.

5.3.2 Clipped Double DQN

To reduce overestimation bias in Double DQN, we formulate and use the clipped Double Deep Q-Learning (clipped-DDQN) formulation, which extends from clipped Q-learning formulation in actor-critic setup in TD3 [14].

We use two networks each for the current (Q) and target (Q^T) action-value functions: Q_1, Q_2, Q_1^T, Q_2^T. s, a denote the current state and action, and s', a' denote the next state and action. γ refers to the discount factor and done refers to boolean variable denoting if s' is a terminal state. The targets (y_i) for each of the Q-value functions are defined in the following equation 5.2:

$$\begin{aligned} y_1 &= r(s,a) + (1 - \text{done}) * \gamma * \min_{i \in [1,2]} Q_i^T(s', a_1^{'}) \\ y_2 &= r(s,a) + (1 - \text{done}) * \gamma * \min_{i \in [1,2]} Q_i^T(s', a_2^{'}) \\ &\text{where } a_i^{'} = \arg\max_{a'} Q_i(s', a') \end{aligned} \tag{5.2}$$

We use the Q_1 action-value network for the agent's policy for training and Q_1^T target action-value network for the final agent's policy for evaluation.

5.4 Reinforcement Learning with Expert Demonstrations

5.4.1 Motivation

Many self-driving car companies have millions of miles of collected data (logs) [55] annotated comprehensively with sensor inputs, system states and the driver's (human or self-driving car) actions. The sensor inputs typically consist of readings from a diverse set of sensors, including cameras, LiDARS, RADARS, GPS, IMU and Encoders. The system states include state-variables from different subsystems in the self-driving car pipeline, including localization, perception, prediction, planning and control. The driver actions comprise the high-level decision making such as lane-change, merging, turning, and also the low-level control actions such as throttle, brake and steer.

Also, many self-driving car companies have a system of *Log-Replay* in their simulators for testing and evaluation. *Log-Replay* is a system to replay the real-world collected logs, where it is possible to update the self-driving agent with a new version of driving policy and test the same. Although *Log-Replay* is not completely interactive when compared to standard RL environments as the other actors' behaviors remain the same during the replay, it is reasonable to utilize it for short snippets of time, and collect some on-policy data for training.

This motivates the design of off-policy RL algorithms to utilize both the limited amounts on-policy data and large amounts of existing expert's driving data to extract policies with the maximum possible utility which can work in the real world. There is existing work in the domain of learning from demonstration [15, 21], and a recent thrust in Offline RL research [30].

We explore a preliminary idea in this direction by collecting driving policy data from an expert agent including the rewards in the CARLA environment and augment it with the agent's experiences in the replay buffer and train using the RL objective.

5.4.2 Expert Agent Details

We build an expert agent in CARLA using the existing automatic agent control logic. For the goal-directed navigation task, the expert agent first generates an optimal path of waypoints from the source to the destination. It then estimates the steer action value using a PID controller with input as the angle between the current vehicle orientation and the next waypoint. It detects a stopping condition due to presence of dynamic actors or a traffic light using information from the simulator, and outputs a target speed of 0 kmph in those cases. In all other cases when it needs to keep driving, it outputs a fixed target speed of 20 kmph. Another PID is used to compute throttle and brake actions from the target speed.

Addressing Action-Space Mismatch

We need to match the action space of the expert and the RL agent to use expert agent's policy data. Given our RL agent uses a discrete action space, as listed in Table 5.1, there is a mismatch with the expert agents which outputs continuous steer values from [-1.0, 1.0] and discrete target-speed action values [0, 20]. To address this, we map the expert agent's continuous steer actions to discrete steer actions using the mapping described in Table 5.3. We use a conservative mapping scheme such that steer values are mapped towards lower steer values such that the expert agent also works with frame-skip. We use the same PID controller parameters for both the RL agent and the expert agent to match the action of throttle computed using target speed. Also, we have the same proximity thresholds for other dynamic actors and traffic light for both the agents.

5.4.3 Expert Agent Demonstration Data Generation

The expert agent is run in the same RL-environment setup as the RL agent such that the state space, action space, reward function and episode termination are exactly same for both the agents. This enables us to use the experiences collected using the expert agent policy directly while training the RL agent. Although the expert agent internally uses more state information from the simulator for its policy, we save only the state information accessible to RL agent in the expert demonstration data.

Expert-Agent Continuous Steer Values Interval	RL-Agent Discretized Steer Values
[-1.0, -0.49)	-0.5
[-0.49, -0.29)	-0.3
[-0.29, -0.05)	-0.1
[-0.05, 0.05)	0.0
[-0.05, 0.29)	0.1
[0.29, 0.49)	0.3
[0.49, 1.0]	0.5

Table 5.3: The mapping function to map expert agent's continuous steer actions to discrete steer actions to match the action space of the expert and the RL agent.

The expert agent is run multiple times on the 25 scenarios corresponding to the Dynamic Navigation task in Town01 as part of the CARLA benchmark, explained in Section 3.5.1. A total of 100K experiences are collected and stored in an expert agent replay buffer. The average success rate of the expert agent in these scenarios is 73.3%, which corresponds to average success episodes of 18.33 out of 25. The success rate is computed using the definition of success in Section 5.1.4 which is used during training. We note that this differs from the success definition in the CARLA benchmarks as it counts lane-invasion, off-road and traffic light violations also as failure cases. The majority failure cases are lane-invasions and off-road infractions, which perhaps can be addressed with finer discretization of steer-values and further tuning of PID parameters. We believe the expert-agent has a reasonably good policy to evaluate the benefit of combining expert agent demonstration data in training an RL agent.

5.4.4 Using Expert Agent Demonstration Data

We use the Reinforcement Learning with Expert Demonstrations (RLED) framework [5, 27, 41], where the rewards are also available in the demonstration data along with states and actions. We follow a similar approach to Human Experience Replay [23], where we maintain two replay buffers, one containing expert agent demonstration data, and the other containing the RL agent's experience. During training, we sample a minibatch comprising both expert agent and RL agent transitions. We evaluate with

different ratios [0%, 25%, 50%, 100%] of expert to agent transitions in a minibatch and report our findings.

There are improved recent approaches that combine imitation learning and reinforcement learning to further improve performance such as DQfD [21]. There are other approaches like Normalized Actor-Critic (NAC), that effectively normalizes the Q-function, reducing the Q-values of actions unseen in the demonstration data robust to sub-optimal demonstration data. We acknowledge that ours is a preliminary step, and should be extended to evaluate with the improved approaches.

5.5 Experimental Evaluation

The performance of RL agents trained with the different approaches described in the previous Sections 5.3 and 5.4 is evaluated on the CARLA CoRL2017 and NoCrash benchmarks. The details of the benchmarks can be referred from Section 3.5. In the following sub-sections, we discuss the details on the agent training, and the baselines used for evaluation.

5.5.1 Agent Training and Evaluation

All the agents are trained on the *Dynamic Navigation Task* of the CARLA CoRL2017 benchmark. During training, a random scenario is sampled from the 25 benchmark scenarios in the training town (Town 01), which is used to set the source and destination location for the agent in an episode. A randomized number of dynamic actors are spawned in each episode to help train the agent in both regular and dense traffic scenarios. During training, the agent is evaluated on all the 25 scenarios of the *Dynamic Navigation Task* after every 40K training timesteps. For evaluation, the epsilon-greedy exploration is disabled so that the agent uses a deterministic greedy policy, thereby removing variability in evaluation due to exploration. The average success rate over 3 seeds is shown in the Training Success Rate plots (Figures 5.9, 5.10, 5.11 and 5.12) for each agent. The training success rate plots are added to show the stability and time-complexity of learning, along the variability in performance observed across different seeds.

For quantitative comparison on the benchmarks, the best agent model over the

course of training is chosen based on the average success rate during training. The best model is evaluated 4 times on the benchmark scenarios, and the final mean and standard errors for the success rate from 3 different seeds are reported as the metrics for comparison.

5.5.2 Baselines

We compare the performance of RL agents trained with our different approaches and group them in 3 categories listed below.

- **DDQN Agent** : Agents trained using Double DQN (DDQN) with different replay buffer sampling strategies: Uniform Sampling (DDQN-U), Backward Sampling (DDQN-B), Prioritized Experience Replay (DDQN-P).
- **cDDQN Agent** : Agents trained using Clipped-Double DQN (cDDQN) with different replay buffer sampling strategies: Uniform Sampling (cDDQN-U), Backward Sampling (cDDQN-B), Prioritized Experience Replay (cDDQN-P).
- **DDQN-Expert Agent** : Agents trained using Double DQN along with the expert agent demonstration data. We compare agents trained with different percentages of the expert demonstration data [0%, 25%, 50%, 100%].

We note that the existing state-of-the-art baselines on the benchmarks use different state inputs, viz. RGB images as sensor input and high level navigation commands as routing input, in contrast to the low-dimensional state input and precise low-level waypoint features as routing input in our approach. Since the published works do not open-source their implementations, we are unable to evaluate them on our state inputs. We acknowledge these differences and nevertheless add the published numbers from the baselines as a reference to compare our agent's performance. The baselines used are listed below and their details can be referred from Section 4.3.

- Modular Pipeline Approaches: CARLA *MP* [12].
- Imitation Learning Approaches: CARLA *IL* [12].
- RL Approaches: CARLA *RL* [12], *CIRL* [32], *IA* [51].

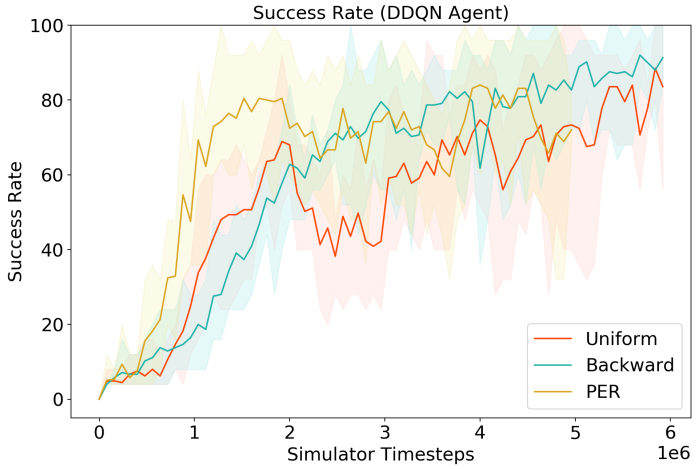

Figure 5.9: Training Success Rate plot for DDQN agent : The figure shows the average success rate during training for the DDQN agent using different methods to sample from the replay buffer: Uniform Sampling, Backward Sampling, Prioritized Experience Replay (PER). The solid lines show the average success rate and the shaded regions show the range spanning minimum and maximum success rates over 3 seeds. Comparable performance is observed across all the sampling approaches. PER is the fastest to achieve a high success rate (around 80%) within 2 M simulator steps, while Uniform Sampling takes longer and reaches a higher performance of 88% in 6 M simulator steps and shows some instability initially in training. Backward Sampling approach seems to learn faster than Uniform Sampling and reaches marginally higher success rate (90%) eventually.

5.6 Results and Discussion

In this section, we present the results of the agents on the experimental setup described in the previous section. The results are divided into 3 subsections for each of our approaches: DDQN agent, Clipped-DDQN agent, DDQN agent with Expert Demonstrations.

5.6.1 DDQN Agent

Looking at the success rate plot for DDQN agent in Figure 5.9, we observe that the agent learns to navigate with comparable performance across all the sampling approaches. PER is the fastest to achieve a high success rate (around 80%) within

2 M simulator steps after which its performance becomes somewhat unstable. Uniform Sampling takes longer and reaches a higher performance of 88% in 6 M simulator steps and shows some instability initially in training. Backward Sampling approach seems to learn faster than Uniform Sampling and reaches marginally higher success rate (90%) eventually. The variability in all the agents is shown by the shaded regions spanning the minimum and maximum success rate values across 3 seeds.

The quantitative evaluation for the DDQN agent is reported in Tables 5.4 and 5.5. Looking at the CARLA CoRL2017 benchmark results in Table 5.4, we observe that the DDQN agent with all the sampling strategies reports near-perfect performance in the training town (Town01). It implies that the agent has reliably learned the task of navigation (without considering collision with dynamic actors as a failure) on the training town (Town 01). We see a performance drop on the testing town (Town 02), showing the gaps in generalization. Some of the error cases in the test town corresponds to the agent failing to take unseen sharp turns and colliding with the curb. The performance within all the three sampling strategies is comparable, with backward sampling showing marginally worse performance in the test town.

Looking at the NoCrash benchmark results in Table 5.5, we observe that the DDQN agent achieves comparable performance to the existing RL baseline of IA [51]. Although the comparable performance is possibly due to the simplified low-dimensional input representation in our approach compared to the baseline, it validates that our approach enables the agent to learn to navigate with dynamic actors with a reasonable performance. Although we observe better performance than the baseline on the *Dense* task, there is a scope for improvement. Some of the error cases with dynamic actors are due to the limitations in the hand-engineered front obstacle detector which fails at certain intersections and sometimes leads to creating a traffic jam.

5.6.2 Clipped-DDQN Agent

The average success rate during training for the clipped-DDQN (c-DDQN) agent using different replay buffer sampling strategies is shown in Figure 5.10. We observe that the c-DDQN agent shows comparable performance with Uniform and Backward sampling approaches reaching around 75%, while it achieves a lower performance of 56% with PER early in training after which its performance degrades. The variability

CARLA CoRL2017 Benchmark (% Success Episodes)								
Task	**Training Conditions (*Town 01*)**							
	MP	*IL*	*RL*	*CIRL*	*IA*	*DDQN-Uniform*	*DDQN-Backward*	*DDQN-PER*
Straight	98	95	89	98	**100**	$\mathbf{100 \pm 0}$	99 ± 1	$\mathbf{100 \pm 0}$
One Turn	82	89	34	97	**100**	$\mathbf{100 \pm 0}$	96 ± 3	$\mathbf{100 \pm 0}$
Navigation	80	86	14	93	**100**	$\mathbf{100 \pm 0}$	95 ± 2	99 ± 1
Dyn. Navigation	77	83	7	82	**100**	$\mathbf{100 \pm 0}$	97 ± 2	99 ± 1
Task	**Testing Conditions (*Town 02*)**							
	MP	*IL*	*RL*	*CIRL*	*IA*	*DDQN-Uniform*	*DDQN-Backward*	*DDQN-PER*
Straight	92	97	74	**100**	**100**	$\mathbf{100 \pm 0}$	95 ± 1	$\mathbf{100 \pm 0}$
One Turn	61	59	12	71	**100**	94 ± 2	93 ± 1	96 ± 1
Navigation	24	40	3	53	**100**	89 ± 6	83 ± 3	98 ± 0
Dyn. Navigation	24	38	2	41	98	91 ± 3	79 ± 6	$\mathbf{99 \pm 1}$

Table 5.4: Quantitative comparison of DDQN agents on the CARLA CoRL2017 Benchmark [12]. The comparison between DDQN agents using different replay buffer sampling strategies (Uniform, Backward, PER) is shown. The published numbers from the baselines, *MP* [12], *IL* [12], *RL* [12], *CIRL* [32], and *IA* [51], are also included for reference, acknowledging the input differences in baselines compared to our method. The table reports the percentage (%) of successfully completed episodes for each task for the training (*Town 01*) and testing town (*Town 02*). The results reported for the DDQN agents are the mean and standard errors of the success rate from 3 different seeds that are evaluated 4 times on the benchmark scenarios. The agents with the highest mean value of success episode percentage are shown in **bold**.

CARLA NoCrash Benchmark (% Success Episodes)				
Task	**Training Conditions (*Town 01*)**			
	IA	DDQN-Uniform	DDQN-Backward	DDQN-PER
Empty	**100**	98 ± 1	94 ± 4	98 ± 2
Regular	96	$\mathbf{97 \pm 2}$	92 ± 5	92 ± 6
Dense	70	$\mathbf{83 \pm 9}$	$\mathbf{83 \pm 7}$	80 ± 8
Task	**Testing Conditions (*Town 02*)**			
	IA	DDQN-Uniform	DDQN-Backward	DDQN-PER
Empty	**99**	88 ± 5	81 ± 7	98 ± 1
Regular	**87**	79 ± 9	73 ± 6	86 ± 9
Dense	42	$\mathbf{71 \pm 6}$	56 ± 7	70 ± 11

Table 5.5: Quantitative comparison of DDQN agents on the CARLA NoCrash Benchmark [9]. The comparison between DDQN agents using different replay buffer sampling strategies (Uniform, Backward, PER) is shown. The published numbers from the available RL-based baseline *IA* [51] are also included for reference, acknowledging the input differences in baselines compared to our method. The table follows the same notation and metrics as Table 5.4.

in all the agents is shown by the shaded regions spanning the minimum and maximum success rate values across 3 seeds. We see a large variability in the backward sampling agent due to poor performance of one of the seeds. Also, there is a possibility that c-DDQN agent requires many more timesteps to achieve a reasonable performance. We are able to report training results upto 5 M timesteps due to compute and time constraints. It requires further analysis to understand the cause of lower performance in the c-DDQN agents compared to DDQN agents with longer training.

The quantitative evaluation of the c-DDQN agent on the benchmarks is presented in Tables 5.6 and 5.7. We observe that c-DDQN agent achieves comparable performance to DDQN agent on the CoRL benchmark training town, while lower performance on most other tasks in the NoCrash benchmark. These benchmark results are in agreement with the observed low training performance of c-DDQN agent. Also, we note that the c-DDQN agent with backward sampling achieves higher performance than uniform sampling and PER across most tasks on both the benchmarks. Hence, it is not conclusive if there is a sampling strategy which performs the best across scenarios, and further experimentation and analysis across diverse scenarios are required.

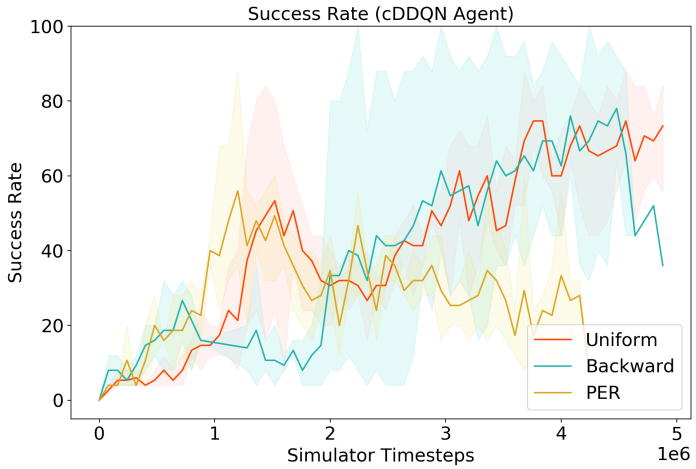

Figure 5.10: Training Success Rate plot for cDDQN: The figure shows the average success rate during training for the clipped-DDQN agent using different methods to sample from the replay buffer: Uniform Sampling, Backward Sampling, Prioritized Experience Replay (PER). The solid lines show the average success rate and the shaded regions show the range spanning minimum and maximum success rates over 3 seeds. We observe that the c-DDQN agent shows comparable performance with Uniform and Backward sampling approaches reaching around 75%, while it achieves a lower performance of 56% with PER early in training after which its performance degrades.

CARLA CoRL2017 Benchmark (% Success Episodes)						
Task	**Training Conditions (*Town 01*)**					
	DDQN-U	DDQN-B	DDQN-PER	cDDQN-U	cDDQN-B	cDDQN-PER
Straight	$\mathbf{100 \pm 0}$	99 ± 1	$\mathbf{100 \pm 0}$	98 ± 1	$\mathbf{100 \pm 0}$	96 ± 2
One Turn	$\mathbf{100 \pm 0}$	96 ± 3	$\mathbf{100 \pm 0}$	96 ± 2	99 ± 0	96 ± 2
Navigation	$\mathbf{100 \pm 0}$	95 ± 2	99 ± 1	94 ± 2	98 ± 2	92 ± 2
Dyn. Navigation	$\mathbf{100 \pm 0}$	97 ± 2	99 ± 1	96 ± 1	97 ± 1	92 ± 3
Task	**Testing Conditions (*Town 02*)**					
	DDQN-U	DDQN-B	DDQN-PER	cDDQN-U	cDDQN-B	cDDQN-PER
Straight	$\mathbf{100 \pm 0}$	95 ± 1	$\mathbf{100 \pm 0}$	99 ± 1	98 ± 2	95 ± 4
One Turn	94 ± 2	93 ± 1	$\mathbf{96 \pm 1}$	91 ± 4	93 ± 2	82 ± 9
Navigation	89 ± 6	83 ± 3	$\mathbf{98 \pm 0}$	80 ± 13	82 ± 0	68 ± 11
Dyn. Navigation	91 ± 3	79 ± 6	$\mathbf{99 \pm 1}$	82 ± 13	82 ± 0	69 ± 10

Table 5.6: Quantitative comparison of Clipped-DDQN (cDDQN) agents on the CARLA CoRL2017 Benchmark [12]. The comparison between cDDQN and DDQN agents using different replay buffer sampling strategies (Uniform, Backward, PER) is shown. The table follows the same notation and metrics as Table 5.4.

CARLA NoCrash Benchmark (% Success Episodes)						
Task	**Training Conditions (*Town 01*)**					
	DDQN-U	DDQN-B	DDQN-PER	cDDQN-U	cDDQN-B	cDDQN-PER
Empty	$\mathbf{98 \pm 1}$	94 ± 4	$\mathbf{98 \pm 2}$	92 ± 4	93 ± 4	91 ± 2
Regular	$\mathbf{97 \pm 2}$	92 ± 5	92 ± 6	87 ± 4	92 ± 4	87 ± 6
Dense	83 ± 9	83 ± 7	80 ± 8	$\mathbf{85 \pm 0}$	77 ± 17	81 ± 5
Task	**Testing Conditions (*Town 02*)**					
	DDQN-U	DDQN-B	DDQN-PER	cDDQN-U	cDDQN-B	cDDQN-PER
Empty	88 ± 5	81 ± 7	$\mathbf{98 \pm 1}$	73 ± 13	84 ± 1	66 ± 12
Regular	79 ± 9	73 ± 6	$\mathbf{86 \pm 9}$	74 ± 10	76 ± 5	66 ± 12
Dense	$\mathbf{71 \pm 6}$	56 ± 7	70 ± 11	67 ± 6	46 ± 14	54 ± 9

Table 5.7: Quantitative comparison of Clipped-DDQN (cDDQN) agents on the CARLA NoCrash Benchmark [9]. The comparison between cDDQN and DDQN agents using different replay buffer sampling strategies (Uniform, Backward, PER) is shown. The table follows the same notation and metrics as Table 5.5.

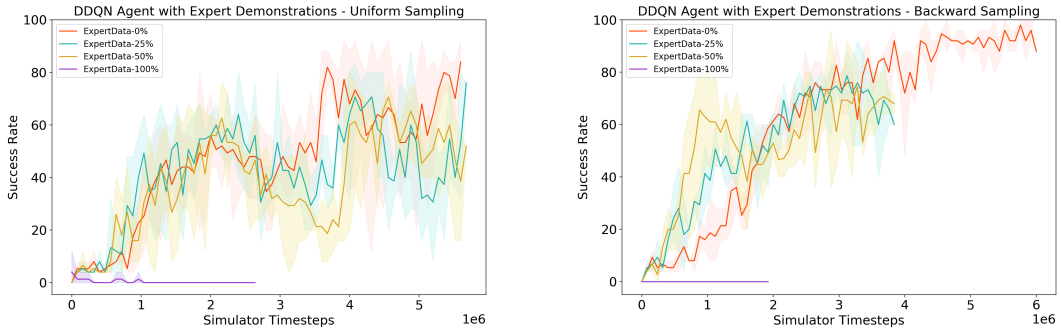

Figure 5.11: Training Success Rate plot for DDQN-Expert agent: The figure shows the average success rate during training for the DDQN agent using different amounts of expert demonstration data [0%, 25%, 50%, 100%]. The solid lines show the average success rate and the shaded regions show the range spanning minimum and maximum success rates over 3 seeds. We observe that the agent does not learn at all using 100% expert data. Left figure: Agent uses uniform sampling and it is observed that using 25% or 50% expert agent data does not help in faster or improved training. Right figure: Agent uses backward sampling and it is observed that adding 25% or 50% helps to reach higher initial success rate faster (within 1M timesteps).

5.6.3 DDQN Agent with Expert Demonstrations

We evaluate the DDQN agent using different amounts of expert demonstration data [0%, 25%, 50%, 100%] as shown in Figure 5.11. We observe that the agent does not learn at all using 100% expert data, which is expected given the state-space and action-space distribution mismatch between the RL agent and the expert agent. With uniform sampling, it is observed that using 25% or 50% expert data, the agent achieves similar performance to the baseline of 0% expert data. With backward sampling we observe that adding 25% or 50% expert demonstrations seems to help reach higher initial success rate close to the expert agent performance. However, adding expert demonstrations does not seem to help improve the eventual performance or stability of learning.

We note that in the current setup, when we add the expert agent data along with the RL agent experience in a minibatch for training, we end up using reduced amounts of experience collected by the RL agent compared to the baseline in the same number of timesteps. For a fair comparison, it is reasonable to use the same amount of experience collected by the RL agent and evaluate how the additional expert

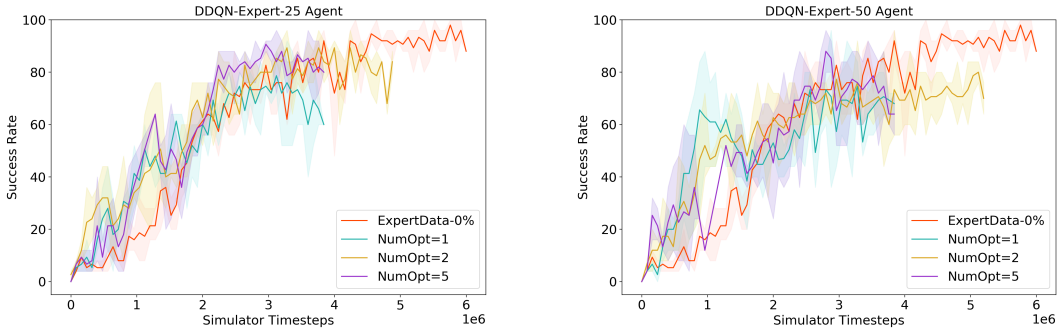

Figure 5.12: Training Success Rate plot for DDQN-Expert agent with different number of optimizations: The figure shows the affect of increasing number of optimization steps on the DDQN-Expert agent. The solid lines show the average success rate and the shaded regions show the range spanning minimum and maximum success rates over 3 seeds. In the left figure, we observe that the DDQN-B-Expert25 agent with NumOpt=5 reaches high performance in around 3M steps, while the baseline DDQN-B agent (0% expert data) takes around 5M steps to reach similar performance. Similary, in the right figure, DDQN-Expert-50 agent with OptEpochs=2 and OptEpochs=5 shows faster learning and reaches near the success rate of DDQN-B agent. This shows the addition of expert data helps in faster learning. We also observe that increasing the optimization epochs helps in faster learning without hurting the stability.

demonstrations help. To address this, we increase the number of optimization steps (referred as NumOpt) at each training step such that we use the same amount of RL agent experience as the baseline and add additional expert agent data. We evaluate this approach with different number of optimization steps ([1, 2, 5]). We maintain that the target Q-network is updated after 10K gradient steps in the current Q-network across the different optimization step settings. The training results of this approach are shown in Figure 5.12. We observe the addition of expert demonstrations with the similar amount of RL agent experience helps in faster learning compared to baseline. The DDQN-B-Expert25 agent with NumOpt=5 reaches high performance in around 3M steps, while the baseline DDQN-B agent (0% expert data) takes around 5M steps to reach similar performance. Similary, in the bottom figure, DDQN-Expert-50 agent with OptEpochs=2 and OptEpochs=5 shows faster learning and reaches near the success rate of DDQN-B agent.

We report the quantitative evaluation of the DDQN-B-Expert25 (with NumOpt=5), DDQN-B-Expert50 (with NumOpt=2) agents with the baseline of DDQN-B (0%

CARLA CoRL2017 Benchmark (% Success Episodes)			
Task	**Training Conditions (*Town 01*)**		
	DDQN-B	DDQN-B-Expert25	DDQN-B-Expert50
Straight	99 ± 1	$\mathbf{100 \pm 0}$	98 ± 2
One Turn	$\mathbf{96 \pm 3}$	94 ± 1	93 ± 3
Navigation	$\mathbf{95 \pm 2}$	93 ± 4	89 ± 7
Dyn. Navigation	$\mathbf{97 \pm 2}$	96 ± 2	85 ± 3
Task	**Testing Conditions (*Town 02*)**		
	DDQN-B	DDQN-B-Expert25	DDQN-B-Expert50
Straight	95 ± 1	$\mathbf{99 \pm 1}$	96 ± 2
One Turn	$\mathbf{93 \pm 1}$	88 ± 4	87 ± 5
Navigation	$\mathbf{83 \pm 3}$	75 ± 0	72 ± 6
Dyn. Navigation	$\mathbf{79 \pm 6}$	78 ± 4	67 ± 5

Table 5.8: Quantitative comparison of DDQN-Expert agents on the CoRL2017 CARLA Benchmark [12]. The comparison of DDQN-B-Expert25 (with NumOpt=5) and DDQN-B-Expert50 (with NumOpt=2) agents with the baseline of DDQN-B (0% expert data, NumOpt=1) agent is shown. The table follows the same notation and metrics as Table 5.4.

CARLA NoCrash Benchmark (% Success Episodes)			
Task	**Training Conditions (*Town 01*)**		
	DDQN-B	DDQN-B-Expert25	DDQN-B-Expert50
Empty	$\mathbf{94 \pm 4}$	91 ± 2	82 ± 6
Regular	92 ± 5	$\mathbf{95 \pm 2}$	80 ± 2
Dense	83 ± 7	$\mathbf{89 \pm 3}$	74 ± 5
Task	**Testing Conditions (*Town 02*)**		
	DDQN-B	DDQN-B-Expert25	DDQN-B-Expert50
Empty	$\mathbf{81 \pm 7}$	79 ± 3	62 ± 4
Regular	73 ± 6	$\mathbf{75 \pm 4}$	61 ± 3
Dense	56 ± 7	$\mathbf{62 \pm 6}$	50 ± 3

Table 5.9: Quantitative comparison of DDQN-Expert agents on the CARLA NoCrash Benchmark [9]. The comparison of DDQN-B-Expert25 (with NumOpt=5) and DDQN-B-Expert50 (with NumOpt=2) agents with the baseline of DDQN-B (0% expert data, NumOpt=1) agent is shown. The table follows the same notation and metrics as Table 5.5.

expert data, NumOpt=1) agent on the benchmarks in Tables 5.8 and 5.9. We observe that the DDQN-B-Expert25 agent achieves comparable performance to the baseline DDQN-B agent on the CoRL2017 benchmark and marginally better performance on the NoCrash benchmark. This suggests that adding expert demonstrations can help achieve comparable performance in less number of simulator timesteps. The performance of DDQN-B-Expert50 agent is worse than DDQN-B-Expert25 and DDQN-B agents on both the benchmarks which motivates further analysis into the optimal amount of expert demonstrations performance.

Chapter 6

Conclusion

6.1 Summary

In the first part of the thesis, we present an approach to use waypoints as the navigation input and design an architecture to learn planning and control directly from semantically segmented images and waypoints using an on-policy RL algorithm (Proximal policy optimization). The agent successfully learns to navigate, except in the presence of dynamic actors. To analyze this further and decouple the issue in representation and policy learning, we propose to use low-dimensional features to focus on policy learning. We use the off-policy RL algorithm of Deep Q-Learning (DQN) for learning goal-directed navigation in the presence of other dynamic agents, present an analysis of the various issues in training instability and explores techniques to address them including n-step returns, dueling network architecture, clipped Double DQN and *Backward-Sampling*. We demonstrate the agent achieves comparable performance to the state-of-the-art baselines using low-dimensional state input representation. We also present a preliminary analysis on how combining expert driving policy data along with RL driving agent's experiences can aid in faster learning.

6.2 Future Work

There are multiple directions in which this work can be extended in the future. It would be worthwhile to investigate further and identify the root cause of the observed Q-value divergence in the *Straight-Dynamic-TJunction* task, and understand why clipped DDQN agent does not address the issue. One direction could be to explore the recent distributional RL extensions of DQN, such as C-51 [2], QR-DQN [10], and IQN [11], and evaluate if they address the stability issue and improve performance. Another direction is to understand the generality and limitations of the *Backward-Sampling* approach and identify the scenarios in which it would be helpful. It would be interesting to combine *Backward-Sampling* with PER, and also compare it with the recent related approaches of DisCor [28] and EBU [29]. It is also worth experimenting with other state-of-the-art off-policy algorithms that work in the continuous domains and are more robust to hyper-parameters such as TD3 [14], Soft-Actor-Critic [17] and DDPG [48].

It would be useful to extend the work on using expert demonstrations by evaluating existing state-of-the-art approaches in the domain of learning from demonstration such as DQfD [15]. It is also worthwhile to address the open questions to achieve optimal performance, such as how good should an expert agent be, how much expert demonstrations data to use, and how to formulate the optimization problem using reinforcement learning and imitation learning objectives.

Bibliography

[1] Tanmay Agarwal. On-policy reinforcement learning for learning to drive in urban settings. Master's thesis, Pittsburgh, PA, July 2020. 4.5

[2] Marc G Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. *arXiv preprint arXiv:1707.06887*, 2017. 6.2

[3] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016. 2.2

[4] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Junbo Jake Zhao, and Karol Zieba. End to end learning for self-driving cars. *CoRR*, abs/1604.07316, 2016. 1.1

[5] Jessica Chemali and Alessandro Lazaric. Direct policy iteration with demonstrations. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, page 3380–3386. AAAI Press, 2015. ISBN 9781577357384. 5.4.4

[6] Dian Chen, Brady Zhou, Vladlen Koltun, and Philipp Krähenbühl. Learning by cheating. In *Conference on Robot Learning*, pages 66–75, 2020. (document), 2.2, 4.3, 4.1, 4.2

[7] Felipe Codevilla, Matthias Miiller, Antonio López, Vladlen Koltun, and Alexey Dosovitskiy. End-to-end driving via conditional imitation learning. *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–9, 2018. 1.1

[8] Felipe Codevilla, Matthias Muller, Antonio Lopez, Vladlen Koltun, and Alexey Dosovitskiy. End-to-end driving via conditional imitation learning. *2018 IEEE International Conference on Robotics and Automation (ICRA)*, May 2018. doi: 10.1109/icra.2018.8460487. URL http://dx.doi.org/10.1109/ICRA.2018.8460487. (document), 2.2, 4.3, 4.1, 4.2

[9] Felipe Codevilla, Eder Santana, Antonio M López, and Adrien Gaidon. Exploring

the limitations of behavior cloning for autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 9329–9338, 2019. (document), 3.5.2, 4.3, 4.1, 4.2, 5.5, 5.7, 5.9

[10] Will Dabney, Mark Rowland, Marc G. Bellemare, and Rémi Munos. Distributional reinforcement learning with quantile regression, 2017. 6.2

[11] Will Dabney, Georg Ostrovski, David Silver, and Rémi Munos. Implicit quantile networks for distributional reinforcement learning. *arXiv preprint arXiv:1806.06923*, 2018. 6.2

[12] Alexey Dosovitskiy, Germán Ros, Felipe Codevilla, Antonio López, and Vladlen Koltun. Carla: An open urban driving simulator. In *CoRL*, 2017. (document), 1.1, 2.3, 3, 3.1, 3.5.1, 4, 4.1.1, 4.2, 4.3, 4.1, 5.5.2, 5.4, 5.6, 5.8

[13] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, et al. Noisy networks for exploration. *arXiv preprint arXiv:1706.10295*, 2017. 2.3.1

[14] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods, 2018. 5.2.3, 5.3.2, 6.2

[15] Yang Gao, Huazhe Xu, Ji Lin, Fisher Yu, Sergey Levine, and Trevor Darrell. Reinforcement learning from imperfect demonstrations, 2018. 1.1, 5.4.1, 6.2

[16] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: The kitti dataset. *I. J. Robotics Res.*, 32:1231–1237, 2013. 1.1

[17] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018. 6.2

[18] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. 3.3

[19] Hado V Hasselt. Double q-learning. In *Advances in neural information processing systems*, pages 2613–2621, 2010. 5.2.1

[20] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018. 2.3, 2.3.1

[21] Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Gabriel Dulac-Arnold, Ian Osband, John Agapiou, Joel Z. Leibo, and Audrunas Gruslys. Deep q-learning from demonstrations, 2017. 1.1, 5.4.1, 5.4.4

[22] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines. https://github.com/hill-a/stable-baselines, 2018. 4

[23] Ionel-Alexandru Hosu and Traian Rebedea. Playing atari games with deep reinforcement learning and human checkpoint replay, 2016. 5.4.4

[24] Takeo Kanade, Charles E. Thorpe, and William Whittaker. Autonomous land vehicle project at cmu. In *ACM Conference on Computer Science*, 1986. 1.1, 2.1

[25] Alex Kendall, Jeffrey Hawke, David Janz, Przemyslaw Mazur, Daniele Reda, John-Mark Allen, Vinh-Dieu Lam, Alex Bewley, and Amar Shah. Learning to drive in a day. *CoRR*, abs/1807.00412, 2018. 1.1, 2.3

[26] Qadeer Khan, Torsten Schön, and Patrick Wenzel. Latent space reinforcement learning for steering angle prediction. *arXiv preprint arXiv:1902.03765*, 2019. 1.1

[27] Beomjoon Kim, Amir-massoud Farahmand, Joelle Pineau, and Doina Precup. Learning from limited demonstrations. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 2859–2867. Curran Associates, Inc., 2013. URL http://papers.nips.cc/paper/4918-learning-from-limited-demonstrations.pdf. 5.4.4

[28] Aviral Kumar, Abhishek Gupta, and Sergey Levine. Discor: Corrective feedback in reinforcement learning via distribution correction, 2020. 2.3.1, 5.2.3, 6.2

[29] Su Young Lee, Sungik Choi, and Sae-Young Chung. Sample-efficient deep reinforcement learning via episodic backward update, 2018. 5.2.3, 6.2

[30] Sergey Levine, Aviral Kumar, George Tucker, and Justin Fu. Offline reinforcement learning: Tutorial, review, and perspectives on open problems, 2020. 1.1, 5.4.1

[31] Jesse Levinson, Jake Askeland, Jan Becker, Jennifer Dolson, David Held, Sören Kammel, J. Zico Kolter, Dirk Langer, Oliver Pink, Vaughan R. Pratt, Michael Sokolsky, Ganymed Stanek, David Stavens, Alex Teichman, Moritz Werling, and Sebastian Thrun. Towards fully autonomous driving: Systems and algorithms. *2011 IEEE Intelligent Vehicles Symposium (IV)*, pages 163–168, 2011. 1.1, 2.1

[32] Xiaodan Liang, Tairui Wang, Luona Yang, and Eric P. Xing. Cirl: Controllable imitative reinforcement learning for vision-based self-driving. In *ECCV*, 2018. (document), 1.1, 2.3, 4.1.1, 4.1, 4.3, 5.5.2, 5.4

[33] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg

Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015. 1.2, 5.2.1

[34] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016. 2.3

[35] P Read Montague. Reinforcement learning: an introduction, by sutton, rs and barto, ag. *Trends in cognitive sciences*, 3(9):360, 1999. 2.3.1

[36] Michael Montemerlo, Jan Becker, Suhrid Bhat, Hendrik Dahlkamp, Dmitri Dolgov, Scott Ettinger, Dirk Hähnel, Tim Hilden, Gabe Hoffmann, Burkhard Huhnke, Doug Johnston, Stefan Klumpp, Dirk Langer, Anthony Levandowski, Jesse Levinson, Julien Marcil, David Orenstein, Johannes Paefgen, Isaac Penny, Anna Petrovskaya, Mike Pflueger, Ganymed Stanek, David Stavens, Antone Vogt, and Sebastian Thrun. Junior: The stanford entry in the urban challenge. In *The DARPA Urban Challenge*, 2009. 1.1, 2.1

[37] Urs Muller, Jan Ben, Eric Cosatto, Beat Flepp, and Yann L. Cun. Off-road obstacle avoidance through end-to-end learning. In Y. Weiss, B. Schölkopf, and J. C. Platt, editors, *Advances in Neural Information Processing Systems 18*, pages 739–746. MIT Press, 2006. URL http://papers.nips.cc/paper/2847-off-road-obstacle-avoidance-through-end-to-end-learning.pdf. 1.1, 2.2

[38] The Mercury News. Google self-driving car brakes for wheelchair lady chasing duck with broom, 2016. URL https://www.mercurynews.com/2016/04/05/google-self-driving-car-brakes-for-wheelchair-lady-chasing-duck-with-broom/. 1.1

[39] Yunpeng Pan, Ching-An Cheng, Kamil Saigol, Keuntaek Lee, Xinyan Yan, Evangelos Theodorou, and Byron Boots. Agile off-road autonomous driving using end-to-end deep imitation learning. *arXiv preprint arXiv:1709.07174*, 2, 2017. 1.1, 2.2

[40] Fabio Pardo, Arash Tavakoli, Vitaly Levdik, and Petar Kormushev. Time limits in reinforcement learning, 2017. 5.1.4

[41] Bilal Piot, Matthieu Geist, and Olivier Pietquin. Boosted bellman residual minimization handling expert demonstrations. In Toon Calders, Floriana Esposito, Eyke Hüllermeier, and Rosa Meo, editors, *Machine Learning and Knowledge Discovery in Databases*, pages 549–564, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-662-44851-9. 5.4.4

[42] Dean A Pomerleau. Alvinn: An autonomous land vehicle in a neural network. In *Advances in neural information processing systems*, pages 305–313, 1989. 1.1, 2.2

[43] Nicholas Rhinehart, Rowan McAllister, and Sergey Levine. Deep imitative models for flexible inference, planning, and control. *arXiv preprint arXiv:1810.06544*, 2018. 1.1

[44] Martin A. Riedmiller, Michael Montemerlo, and Hendrik Dahlkamp. Learning to drive a real car in 20 minutes. *2007 Frontiers in the Convergence of Bioscience and Information Technologies*, pages 645–650, 2007. 1.1

[45] Axel Sauer, Nikolay Savinov, and Andreas Geiger. Conditional affordance learning for driving in urban environments. *arXiv preprint arXiv:1806.06498*, 2018. (document), 2.2, 4.3, 4.1, 4.2

[46] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *CoRR*, abs/1511.05952, 2016. 2.3.1, 5.2.3

[47] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017. 1.2, 4, 4.2

[48] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. 2014. 6.2

[49] Sebastian Thrun and Anton Schwartz. Issues in using function approximation for reinforcement learning. In *Proceedings of the 1993 Connectionist Models Summer School Hillsdale, NJ. Lawrence Erlbaum*, 1993. 5.2.3

[50] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005. ISBN 0262201623. 1.1

[51] Marin Toromanoff, Emilie Wirbel, and Fabien Moutarde. End-to-end model-free reinforcement learning for urban driving using implicit affordances, 2019. (document), 2.3, 4.1, 4.3, 4.2, 5.5.2, 5.6.1, 5.4, 5.5

[52] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI conference on artificial intelligence*, 2016. 1.2, 5.2.1

[53] Richard S. Wallace, Anthony Stentz, Charles E. Thorpe, Hans P. Moravec, William Whittaker, and Takeo Kanade. First results in robot road-following. In *IJCAI 1985*, 1985. 1.1, 2.1

[54] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003, 2016. 2.3.1, 5.2.1

[55] Waymo. Waymo safey report, 2018. URL `https://storage.googleapis.com/sdc-prod/v1/safety-report/Safety%20Report%202018.pdf`. 1.1, 5.4.1

[56] Grady Williams, Nolan Wagener, Brian Goldfain, Paul Drews, James M. Rehg, Byron Boots, and Evangelos A. Theodorou. Information theoretic mpc for model-based reinforcement learning. *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1714–1721, 2017. 1.1

[57] Bernhard Wymann, Eric Espié, Christophe Guionneau, Christos Dimitrakakis, Rémi Coulom, and Andrew Sumner. Torcs, the open racing car simulator. *Software available at http://torcs. sourceforge. net*, 4(6):2, 2000. 1.1

[58] Jiakai Zhang and Kyunghyun Cho. Query-efficient imitation learning for end-to-end autonomous driving. *arXiv preprint arXiv:1605.06450*, 2016. 1.1