Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Telecommunications and Media Informatics

Asad Idrees Razak

IMPLEMENTING A DEEP REINFORCEMENT LEARNING MODEL FOR AUTONOMOUS DRIVING

SUPERVISOR

Dr. Toka László

BUDAPEST, 2022

Table of Contents

STUDENT DECLARATION

I, **Asad Idrees Razak**, the undersigned, hereby declare that the present BSc thesis work has been prepared by myself and without any unauthorized help or assistance. Only the specified sources (references, tools, etc.) were used. All parts taken from other sources word by word, or after rephrasing but with identical meaning, were unambiguously identified with explicit reference to the sources utilized.

I authorize the Faculty of Electrical Engineering and Informatics of the Budapest University of Technology and Economics to publish the principal data of the thesis work (author's name, title, abstracts in English and in a second language, year of preparation, supervisor's name, etc.) in a searchable, public, electronic and online database and to publish the full text of the thesis work on the internal network of the university (this may include access by authenticated outside users). I declare that the submitted hardcopy of the thesis work and its electronic version are identical.

Full text of thesis works classified upon the decision of the Dean will be published after a period of three years.

Budapest, 16th December 2022

...…….
Asad Idrees Razak

Abstract

Artificial Intelligence (AI) is growing extraordinarily in almost every area of technology, and research into self-driving cars is one of them. In the last few years, there has been a lot of interest in deep learning and reinforcement learning which has led to tremendous growth in this domain, especially in vision-based subsystems. Machine learning (ML) moves autonomous driving technology forward, and we have already seen several big players in the vehicle and AI industries use this to their advantage. In this thesis, we will take the liberty to utilize state-of-the-art methods to train an agent to drive autonomously using the Deep Reinforcement Learning (DRL) approach. We will use an open-source simulator, CARLA [1], to conduct our experiment, providing a hyper-realistic urban simulation environment to train our models. We cannot use our raw algorithms in the real world because they come with many risks and moral questions, so we use these simulators to help us test them.

Moreover, road geometry makes it very hard for autonomous driving systems to make decisions, and modern autonomous driving systems still need help dealing with complexity. The driving policy is mostly hand-crafted in many modern-day solutions, which could lead to sub-optimal solutions and are very expensive to develop, generalize, and keep up at a large scale. DRL has shown promising results in learning complex decision-making tasks, from strategic games to challenging puzzles. Here, we will look at how an on-policy DRL algorithm called Proximal Policy Optimization (PPO) will be used in a simulated driving environment to learn to navigate on a predetermined route. The primary goal of this thesis is to investigate how a DRL model can train an agent on a continuous state and action space. Our main contribution is a PPO-based agent that can learn to drive reliably in our CARLA-based environment. In addition, we also implemented a Variational Autoencoder (VAE) that compresses high-dimensional observations into a potentially easier-to-learn low-dimensional latent space that can help our agent learn faster. This work aims to develop an end-to-end solution for autonomous driving that can send commands to the vehicle to help it drive in the right direction and avoid crashes as much as possible. In this paper, we have summarized some results and analyses and discussed work to simplify this problem further.

Absztrakt

A mesterséges intelligencia rendkívüli mértékben fejlődik a technológia szinte minden területén, az önvezető autók kutatása is ezek közé tartozik. A közelmúltban nagy érdeklődés mutatkozott a mélytanulás és a megerősítéses tanulás iránt, ami óriási fejlődést eredményezett a területen, különösen a látásalapú alrendszerek esetében. A gépi tanulás az autonóm vezetési technológia motorja, és látható, hogy több jelentős szereplő is előnyére fordítja ezt a technológiát. Ebben a szakdolgozatban a legkorszerűbb módszereket alkalmazzuk egy autonóm vezetésre alkalmas ágens kiképzésére, Deep Reinforcement Learning megközelítéssel. Kísérletünk elvégzéséhez egy nyílt forráskódú szimulátort, a CARLA-t [1] fogjuk használni, amely egy hiperrealisztikus városi szimulációs környezetet biztosít a modelljeink betanításához. A tesztelésükhöz szimulátorokat használunk, mivel a nyers algoritmusainkat nem használhatjuk a való világban.

Ráadásul az út geometriája nagyon megnehezíti az autonóm vezetési rendszerek számára a döntéshozatalt, és a modern autonóm vezetési rendszereknek még mindig segítségre van szükségük a komplexitás kezelésében. A vezetési irányelvet többnyire manuálisan alakítják ki számos modern megvalósításban, amelyek nem optimális megoldásokhoz vezethetnek és a lépéstartás is költséges. A Deep Reinforcement Learning ígéretes eredményeket mutatott az összetett döntéshozatali feladatok tanulásában, a stratégiai játékoktól a kihívást jelentő rejtvényekig. Itt azt fogjuk megvizsgálni, hogy egy on-policy DRL algoritmus, amit 'Proximal Policy Optimization'-nek neveznek, hogyan lesz használva egy szimulált vezetési környezetben, hogy megtanuljon egy előre meghatározott útvonalon navigálni. A szakdolgozat elsődleges célja annak vizsgálata, hogy egy DRL modell hogyan képes egy ágenst autonóm vezetésre képezni. Fő hozzájárulásunk egy PPO-alapú ágens, amely képes megtanulni megbízhatóan vezetni a CARLA-környezetben. A munka célkitűzése egy olyan végponttól-végpontig tartó megoldás kifejlesztése az autonóm vezetéshez, amely képes parancsokat küldeni a járműnek, hogy az a megfelelő irányba haladjon, és a lehető legnagyobb mértékben elkerülje a baleseteket. Ebben a tanulmányban összefoglaltunk néhány eredményt és elemzést, valamint megvitattuk a probléma további egyszerűsítésére irányuló munkát.

1 Introduction

1.1 Motivation

In the modern day, there is a growing desire to automate our daily tasks and use technology to make the various tasks safer, more efficient, and more convenient. Similarly, there is an emphasis on minimizing road accidents and traffic congestion by creating intelligent vehicles. Ever since the breakthroughs in vision-based autonomous driving techniques back in the 1980s [4], the question of cars run by artificial intelligence still needs to be answered. However, there is still no large-scale implementation of completely autonomous vehicles. However, with the emergence of publicly accessible tools and knowledge, such as autonomous driving simulators, machine learning frameworks, and code-sharing platforms, it is now simpler to contribute to the aim of full large-scale autonomy, in contrast to the situation back in the 1980s.

In the field of autonomous driving, there are two main approaches. First, the modular approach is used by most of today's autonomous driving systems because it divides the driving system into several independent modules, such as the mapping, perception, planning, and control modules. Second, the end-to-end approach uses artificial neural networks that learn a driving policy that directly maps observations to vehicles' actions. Moreover, there have also been studies on two specific methods of learning an end-to-end driving policy: imitation learning (IL) and reinforcement learning (RL).

Imitation learning is now one of the most practical methods for developing autonomous vehicles because it requires a substantial amount of expert demonstration data [6]. However, there is a caveat to this approach because those autonomous vehicles are trained substantially less on managing life-or-death situations. Furthermore, since they can never outperform an expert agent, gathering expert demonstrations for all possible scenarios is unfeasible. As a result, scaling and generalizing these systems to new situations is difficult. On the other hand, using deep learning methods in conjunction with reinforcement learning [10], the agent can learn the policy on its own

by receiving reward signals from the environment for the actions it has performed at each timestep.

Deep Reinforcement Learning (DRL) in today's time has enabled computers to play Atari games [13], Deepmind's AlphaGo [11], and even racing games like TORCS [12] without human intervention. They are a good substitute for IL approaches for urban autonomous driving because they do not require human guidance, saving both time and resources. Now the question is, can reinforcement learning (RL) be used to develop autonomous vehicles while being motivated by the superhuman performance of these learning paradigms? The use of reinforcement learning in autonomous driving has been encouraged in a few recent publications [1][9][15][16]. However, in the actual world, applying RL directly for autonomous driving is challenging because of the safety concerns and poor sample complexity of the most advanced RL algorithms. As a result, a growing amount of research is taking place on simulators that can eventually apply to real-world settings, that includes TORCS [12] and CARLA [1].

1.2 Autonomy in driving

Numerous businesses, including Volvo, Tesla, and Google, are already working on developing autonomous vehicles [17]. Meanwhile, cars observe the environment using sensors like radar or cameras. The movement control system predicts the surroundings and decides how to apply movement based on sensor data. However, we have not fully reached automation yet; therefore, these solutions can still improve, and the term "autonomous" in the context of self-driving cars merely refers to "self-governing." When we talk about self-driving cars, we frequently mean those that need little to no human input in the vehicle's control system to travel from place A to place B. For self-driving cars, the Society of Automotive Engineers [2] has created a hierarchy of increasing levels of automation. These levels consist of the following:

- **Level 0** - No Automation: In this case, the driver interacts with the vehicle's control system. A system with integrated warning signals or intervention mechanisms but still considered at level 0.
- **Level 1** - Driver Assistance: A human driver and automated technology share control of the vehicle. An automobile with Adaptive Cruise Control, a system in which the car sets its speed, but the driver is responsible for steering.

- **Level 2** - Partial Automation: The system controls the steering and speed of the vehicle, but a person must assist promptly if the system malfunctions.
- **Level 3** - Conditional Automation: The driver is not required to be prepared for intervention but must respond within a specified timeframe when the system requires it.
- **Level 4** - High Autonomy: The vehicle may run in confined geographic areas or driving scenarios without driver interaction. If the vehicle is unable to move, it will park itself until the driver intervenes.
- **Level 5** - Full Automation: Absolutely no human interaction is necessary.

A combination of level 0 ("no automation") and level 1 ("hands-on") autonomous vehicles makes up most of the cars on the road today. Level 2 ("hands off") automation features, including autonomous lane change, have been implemented into certain commercially available vehicles by automakers like Tesla and a few more. Additionally, there are a few examples of both commercial and non-commercial vehicles with level 3 ("eyes off") automation, such as the Audi A8's "Traffic Jam Pilot." Although level 4 ("mind off") and level 5 ("steering wheel optional") cars are not yet commercially accessible, research on such vehicles is ongoing. Waymo's self-driving car is one example of a self-driving vehicle without a steering wheel. So naturally, level 5 automation is the goal of research into driverless vehicles. For reference, the paper cited [2] explores more in-depth.

1.3 Components of an Autonomous Car

A self-driving car requires several systems or components to perform their distinct functions. Such systems are divided as follows:

Sensors and Control

What components are necessary for autonomous driving? When making an autonomous car, we should look at the components and sensors needed for optimal driving. We should also ensure that our car and its processes are energy-efficient for long-distance travel. We must also build software and interfaces the vehicle uses to connect with its onboard sensors.

Planning and Control

Planning is about controlling the vehicle's acceleration, brakes, and steering to get from place A to place B. The vehicle should use the information from its sensors or perception modules to figure out a set of control signals that will drive it where it needs to go. The car should be able to drive without hitting other cars, bikes, people, or other dynamic or static objects.

Mapping and Localization

Mapping and localization involve locating a vehicle on a high-definition map and assessing its topography. High-definition map localization may use GPS and environmental data. The car can pinpoint its exact location by a rough estimate of its location on a map and comparing its observations with its projections of where adjoining structures are on its internal map. In addition, it needs to interpret sensor input data and predict static and dynamic objects around it. For example, a perception module like RGB cameras, infrared sensors, and LiDAR should locate static things like lane lines and signage and understand their meaning.

Simulators

Before implementing our control and perception algorithms on an actual vehicle, it is crucial that we have methods for testing and validating them in a simulated environment; this is the primary function of the simulators. Furthermore, since we utilize simulators to validate our algorithms before deployment, it is vital that the simulator accurately replicates the physics and look of the actual world. There are several high-fidelity, open-source simulators for autonomous driving research. CARLA [1] is our simulator of choice for this thesis project.

1.4 Objective

In this thesis, we will try to seek answers to the following research questions:

- What are the ideas and methods behind deep reinforcement learning, and can modern reinforcement learning methods teach a car to drive in a regulated and dependable manner?
- How can we construct reward functions that motivate reinforcement learning to exhibit the desired driving behavior?

- What effect does environment design have on an agent's training pace and overall performance based on its reinforcement?
- How can we produce state representations that are rich in information?
- Can we utilize deep reinforcement learning to teach agents how to drive in surroundings that demand them to execute various maneuvers?

1.5 Structure of the work

This dissertation is structured into five chapters. These chapters are organized in the following order:

1. *Introduction*: This chapter introduces the thesis, its purpose, and its motivation to set the stage and clarify the problem statement.
2. *Background and Related Work*: This chapter will cover some of the theoretical background of this thesis, including machine learning, deep learning, and reinforcement learning. This chapter will also cover some works relevant to this thesis.
3. *Methodology* - This chapter describes the architecture, techniques, and experiments implemented. In addition, it presents the proposed solution to our problem statement.
4. *Results* - This chapter describes the experiment's findings and the discussion surrounding it.
5. *Conclusion* - This chapter wraps up the results and discussion of this thesis. It also gives ideas for further research.

2 Background and Related Work

In this chapter, we lay out some of the theoretical foundations and related works in the autonomous driving domain.

2.1 Machine Learning

2.1.1 Introduction

Machine learning (ML) is an artificial intelligence subfield that focuses on creating problem-solving algorithms. ML has received much interest recently because of its simplicity and the empirical success of its techniques. ML often beats hand-crafted solutions in situations with some degree of non-triviality. Furthermore, given that the theory underlying machine learning algorithms are typically general-purpose, the same theory may be reused for any domain if the required data for the job is available. For example, given observed data, a computer-based agent can construct a model that fits the observed data, which can then be utilized as a hypothesis about the agent's reality to assist in solving problems [19].

This dissertation will explore a machine-learning approach called Reinforcement Learning (RL). RL provides an intuitive learning framework in which an agent observes an environment where it does some actions to achieve its goal. RL aims to educate the agent to maximize its utility, which is some quantity that specifies how successfully it managed to do its job.

2.1.2 Artificial Neural Networks

An artificial neural network is a mathematical model that solves problems resembling a biological brain network. Modern neural networks are non-linear statistical tools that construct models based on mathematical and statistical learning methods. Therefore, the first step is to derive analysis and practical application from a large amount of statistical data.

However, this is different in artificial intelligence for artificial perception. Artificial perception makes decisions using statistics, which allows artificial neural

networks to make simple decisions and judgments like humans. This method is more useful than calculus inference. Like other learning methods, neural networks solve many problems, including autonomous driving, speech recognition, stock prediction, and other complex problems because the simulated artificial neural networks find the rules underlying the problem to solve it.

2.1.3 Deep Learning

Deep learning refers to a class of machine learning methods that use deep neural networks. Deep neural networks are networks of multiple layers of simple, adjustable computing elements such as weights and biases [19].

The structure and flow of the human brain and its brain cells inspired the development of deep neural networks. The neural net mimics the brain's neurons in the form of a perceptron, which is a crucial component. Figure 2.1 depicts how a neural network combines nodes in a connected and layered structure. The network can learn a mapping between an input X and an output Y using these nodes, and their activation functions, weights between layers, and biases. Deep learning, one of the most common approaches to machine learning, is a versatile and widely applicable method. It can handle complex data and plays an essential role in this thesis due to its application in reinforcement learning [19].

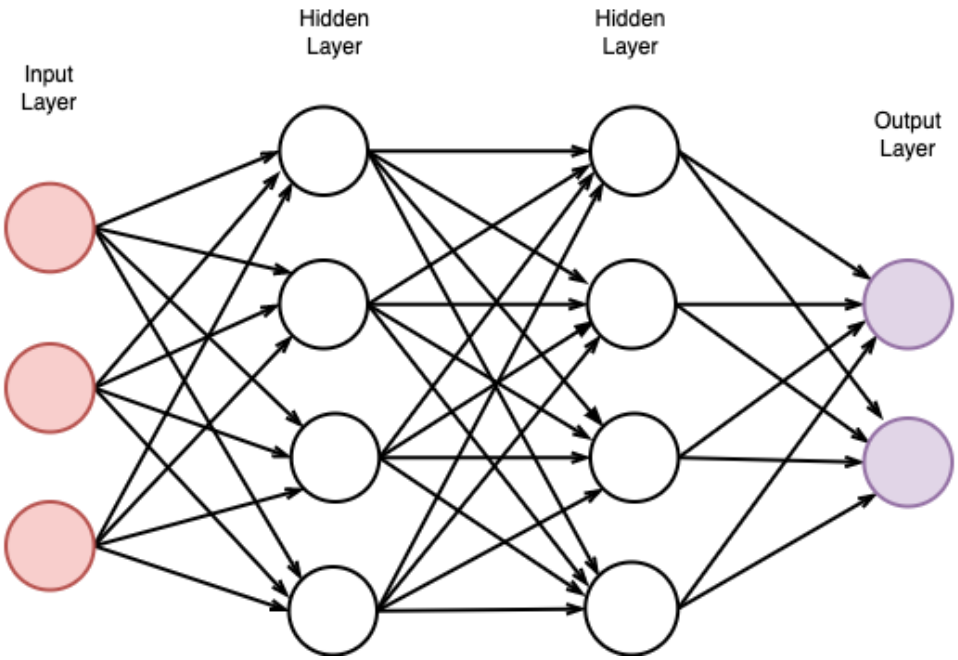

Figure 2.1 A two-layered neural network with three input and two output nodes.

2.1.4 Imitation Learning

Imitation learning technique attempt to mimic human behavior while performing a task by learning a mapping between observations and actions; an agent (a learning machine) is training to perform a task from demonstrations. The learning-by-imitation paradigm is gaining popularity because it allows teaching complex tasks with little expert knowledge. Generic imitation learning methods can potentially reduce the problem of teaching a task to that of provided demonstrations without the need for explicit programming. In addition, imitation learning provides a paradigm for developing a policy that, given corresponding input, will mimic the expert's actions [21]. These demonstrations are obtained offline (either in the real world or through simulation) and comprise a series of state observations and expert actions.

Moreover, in a study concerning our autonomous driving domain called “Conditional Affordance Learning” [23], the authors suggested an architecture that generalizes a direct perception method to the urban environment. They provide intermediate representations in the form of affordances suited for urban navigation based on the image's convolution properties. These affordance models have been taught to respond to directional commands at the highest level.

2.2 Reinforcement Learning

2.2.1 Introduction

In contrast to imitation learning [21], which teaches an agent how to respond based on a labeled dataset, reinforcement learning allows an agent to interact with its environment and receive regular incentives based on its behavior. The agent can use this reward to consider whether it behaved appropriately and whether its policy needs to be altered. In the case of imitation learning, an agent is not expected to outperform the thing it is copying, which in the case of autonomous cars is frequently a human being. However, in theory, an agent can explore and eventually outperform a person through reinforcement learning [19].

The main purpose of a reinforcement learning algorithm is to maximize the expected total of rewards. Typically, the environment takes the form of a Markov decision process (MDP), in which the agent must choose a course of action based on the

current state of the environment, with certain states yielding "better" or "worse" rewards. Depending on whether the training is off-policy or on-policy, the agent's policy (denoted as π) determines which action the agent takes both outside and during training. To maximize its predicted reward sum, the agent must maximize this strategy throughout training [19].

2.2.2 Markov Decision Process

A Markov decision process is a stochastic sequential decision problem with a transition model that specifies the probabilistic outcomes of any action and a reward function that specifies the reward from each state [19]. For example, given an agent in a current state s_t (in a state space **S**) with its reward r_t, the agent can choose to perform an action a_t (from an action space **A**) and based on a transition probability function p and a_t, the agent next timestep $t + 1$ end up in a new state s_{t+1} with its reward $r_t * p$ which is defined in equation 2.1, cited [3].

$$p(s'|s,a) = Pr(s_{t+1} = s'|s_t = s, a_t = a)$$

(Equation 2.1)

where $p(s'|s,a)$ is the probability that action a in state s at time t will put the agent in the next state s' at time $t + 1$. The Markov property is satisfied by the transition probability function because it only depends on the current state s and the action a of the agent and is independent of all earlier states and actions. From a given first state s_0 and to some final state s_n, and the actions chosen in between, we can define an agent's trajectory τ, $where\ \tau = \{(s_0 a_0 r_0), \dots, (s_n a_n r_n)\}$. We think of this framework as a close connected loop, illustrated in Figure 2.1.

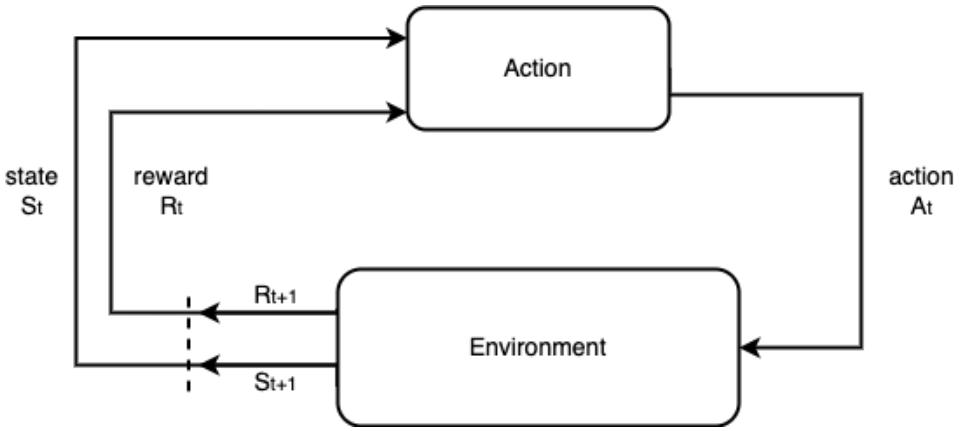

Figure 2.2 MDP: reinforcement learning loop. Starting at timestep t, the agent observes the state st and reward rt. Then, when an agent performs an action, at, the environment returns a new state, st+1, and a scalar reward value, rt+1, which reflects state and reward at timestep t+1.

Reinforcement learning concerns a subset of MDP problems in which the agent is unaware of the state-transition probabilities $P(s'|s,a)$ and the reward function $r(s,a)$. This means that the agent must explore its surroundings in order to discover a link between state-action pairs and reward.

2.2.3 Bellman equation

A common task in reinforcement learning is to estimate an action-value function $Q(s,a)$ for each state s in the environment. The Bellman equation can be used to retroactively update an action-value function based on an agent's trajectory. In a deterministic environment, the Bellman equation:

$$V(s) = \max_{a \in A(s)} \sum_{s'} [R(s,a) + \gamma V(s')],$$

(Equation 2.2), cited [3]

where V is the value function that gives the value of a given state s, and γ is a discount factor. The Bellman equation computes the value of a state s by taking the reward of the next step and multiplying it by the expected value from the following states [9]. The action-value function $Q(s,a)$ is also directly related to the value function as follows, cited [3]:

$$V(s) = \max_{a \in A(s)} Q(s,a),$$

(Equation 2.3)

2.2.4 Reward Function

One of the fundamental components of reinforcement learning is when an agent maximizes its expected reward; the use and design of a suitable reward function are critical for the RL algorithm's success. As reinforcement learning usually implies that an agent starts with a policy that gives random actions, a good reward function is expected to give the agent an indication whether which actions are good and which are bad. In scenarios where the rewards are too low or too difficult to obtain, more than random actions may be required for the agent to converge toward a good policy.

A reward at a given time step t is usually defined as $r_t = R(s_{t+1}, s_t, a_t,)$, where R is a reward function. However, a reward can also be defined over a trajectory τ, where we can sum all rewards over a finite and sequential amount of steps T in the environment, which can be written as:

$$R(\tau) = \sum_{t=0}^{T} r_t$$

(Equation 2.4), cited [3]

Using trajectories, we can put rewards from states and actions in relation to each other w.r.t time and not just evaluate the agent's policy at given isolated states and actions. Additionally, we can add a discount factor γ_t to value actions closer in time.

2.2.5 Reinforcement Learning Concepts

Exploration vs. Exploitation

In order to develop an ideal strategy, an agent must explore and exploit. Exploration requires the agent to perform either random or non-optimal actions to observe and better generalize as per the environment. Exploitation requires the agent to take the best possible action under the current policy. It is because state transitions and rewards are unknown during training; therefore, exploration is essential. After observing multiple state transitions and rewards, the agent can use the environment to get closer to an optimal policy. Of course, there is a trade-off between exploring the environment and exploiting learned knowledge. In general, we want to explore more at

the beginning of the training process and gradually start exploiting more towards the end to arrive at an optimal policy [5].

Fully observable vs. Partially observable environment

The environment in an MDP so it can either be fully or partially observable. When an environment is fully observable, an agent can see its entire state at any time. Chess is one example of this type of environment. At any point in the game, a chess-playing agent can observe the location and type of all the chess pieces, so we say the environment is fully observable. On the other hand, only a subset of the state is known when the environment is partially observable, e.g., self-driving cars. It is because the car may only be able to observe its surroundings via a front-facing camera, which means it cannot know or observe the states of objects behind it. In other words, this environment can be seen in parts. A partially observable Markov Decision Process is another name for a partially observable MDP (POMDP) [5].

Discrete vs. Continuous state and action spaces

The state and action spaces may be discrete or continuous depending on the environment. Chess is an example of a game with discrete state and action spaces; there are a countable number of possible states and actions at any given time. Autonomous driving with a proximity sensor is an example of an environment with continuous state and action spaces; the state and actions are real-numbered values, such as the proximity values in meters and the vehicle's steering angle in degrees [5].

Model-based vs. Model-free

Model-based methods attempt to build a model of how the environment works. For example, if an agent is in state s and performs action a, it will enter state s' and receive reward r. Intuitively, we could use these observations to approximate $P(s'|s,a)$ and $r(s,a)$; that is, we could approximate a model of the environment. After approximating a model, we could use an MDP-solving algorithm, such as Bellman's value iteration algorithm, to determine the best policy given our current model of the environment[5].

Model-free methods are those that optimize policies without first modeling the environment. For example, many reinforcement learning problems have continuous

state and action spaces, resulting in an infinite number of possible values, making it impossible to efficiently create MDP models without approximations. Direct policy optimization also allows us to find good policies using function optimization techniques such as gradient descent [5].

Deterministic vs. Stochastic policy

The policy of an agent can be deterministic or stochastic. When we have a deterministic policy, the agent will always choose the same action when presented with the same state, assuming that the policy stays the same. The agent can also follow a stochastic policy, in which it randomly chooses an action when presented with the same state. Deterministic policies make sense in fully observable and deterministic environments. It is where doing something always results in the same outcome (e.g., chess). Stochastic policies are helpful in both partially observable and stochastic environments. In a partially observable environment, using a stochastic policy allows the agent to model some of the hidden states of the environment as part of its stochasticity, making it more robust to hidden information. When an agent acts in a stochastic environment, the outcome of an action may vary from one time to another [5].

On-policy vs. Off-policy

In on-policy methods, the same policy is used to determine the value of the policy and control the agent. Proximal Policy Optimization is an example of an on-policy method. Meanwhile, Off-policy methods use different policies to evaluate the policy and control the agent. Q-learning is an example of an off-policy method [5].

Monte-Carlo vs. Temporal difference

In methods that use Monte-Carlo rollouts, we compute the entire trajectory before optimizing, whereas methods that use n-step temporal difference use n steps along a trajectory before taking an optimization step. Temporal difference (TD) methods have the advantage of being able to predict things in non-episodic environments. However, TD-learning methods are more vulnerable to bias caused by the initialization of the agents' parameters, whereas Monte-Carlo methods are less biased but have a higher overall variance during training [5].

2.2.6 Deep Reinforcement Learning

Deep reinforcement learning is a reinforcement learning method that uses deep neural networks. The name is a blend of reinforcement and deep learning. Deep neural networks are typically used in reinforcement learning algorithms to serve as an agent's policy, mapping states to actions. It allows agents to handle more complicated and changeable inputs, such as images or other continuous state spaces, and to output into a more complex action space.

Policy Representation: A Policy is a mapping of states and actions. After receiving a state from its environment, an agent decides on an action depending on that state and its policy. The states of an environment can be represented in a variety of ways, some in tabular form, discrete, as illustrated in Figure 2.3 cited [3], and others can be represented with continuous values. Action spaces can also have both discrete and continuous values.

Table policy: A good policy mapping can also be expressed as a table, given the problem of finding an optimal path from start to finish in a discretized environment, such as the checkered board illustrated in Figure 2.3. The positional coordinates of the agent on the board are the input to the mapping in the table, and the output is an action in either of the four directions. In this scenario, the state and action spaces are discrete.

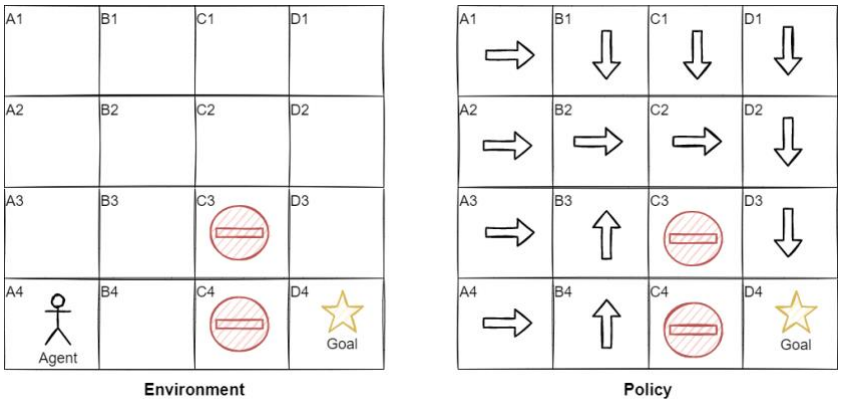

Figure 2.3 On the left is an optimal path issue setting, while on the right is a possible optimal table policy, illustration is taken from [3].

Policy networks: In scenarios where an agent is learning to drive in urban surroundings, inputs from the environment come in the form of continuous, high-dimensional data such as speeds, steering values, locations, accelerations, and images.

Unfortunately, most continuous, and high-dimensional spaces will be too complex for a table policy to manage efficiently and update correctly. As a result, neural networks or policy networks can be used to map between states and actions.

Policy Gradient Methods

Policy gradient methods are reinforcement learning techniques that use gradient descent to optimize policies with respect to expected rewards. In terms of how the policy changes, the methods use an estimate of the gradient of the expected rewards to determine if the policy changes would be advantageous or not. An estimated gradient for one of the basic gradient methods, vanilla gradient method, is defined by:

$$\hat{g} = \hat{E}_t\left[\nabla_\theta log\pi_\theta(a_t|s_t)\hat{A}_t\right]$$

(Equation 2.5), cited [3]

where $\hat{A}_t$ is the advantage function estimator at timestep t, and $\hat{E}_t$ is the empirical average along a finite batch of sampled trajectory, τ. The advantage function measures how much a specific action in each state is either good or bad. It can be defined as follows:

$$A(s,a) = Q(s,a) - V(s)$$

(Equation 2.6), cited [3][24]

where $Q(s,a)$ is the expected reward of an action a from a state s, and V(s) is the expected reward from state s prior to the action. With equation 2.5, we can compute a loss function for a policy network in terms of its parameters, represented as θ:

$$L^{PG}(\theta) = \hat{E}_t\left[\nabla_\theta log\pi_\theta(a_t|s_t)\hat{A}_t\right]$$

(Equation 2.7), cited [3]

Proximal Policy Optimization (or PPO) [20][25] is a policy gradient method that builds upon the works of the Trust Region Policy Optimization (TRPO). PPO offers some of the same advantages as TRPO, but it is considerably easier to implement, can be used to a bigger range of problems, and has better sampling complexity [13]. Researchers in the PPO paper [13] introduces a set of PPO methods, but for this thesis, we will only look at the method that uses a clipped surrogate objective. This clipped surrogate objective function is defined in equation 2.8.

$$L^{CLIP}(\theta) = \hat{E}_t\left[\min\left(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t\right)\right]$$

(Equation 2.8), cited [3][5]

Where $r_t(\theta)$ denotes the probability ratio $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$, which means that $r_t\ (\theta_{old}) = 1$. Meanwhile, ε is our clipping parameter; refer to the end of this document in Table 2 to see the choice of hyperparameters used in our experiments. Moreover, $r_t(\theta)$, is used in conjunction with the clipping function to discourage and prevent large deviations from the original θ_{old}.

As mentioned before, formulating the optimization problem in terms of a differentiable loss function allows us to use gradient descent. This means that, unlike TRPO, we can now parameterize a critic in terms of θ. PPO optimizes the critic the same way A3C, another DRL algorithm, does, by introducing a value function loss $L^{VF} = (V(s_t;\theta_v) - R_t(\tau))^2$. We also add the entropy term, $-\frac{1}{2}(log\ (2\pi\sigma^2) + 1)$, like in A3C. The final loss function is defined as:

$$L^{CLIP+VS+F}(\theta) = -\widehat{E}_t\left[L^{CLIP}(\theta) - \alpha L^{VF}(\theta) - \beta\frac{1}{2}(log(2\pi\sigma^2)+1)\right]$$

(Equation 2.9), cited [5]

2.3 Approaches to autonomous driving

This section will discuss the two main approaches to autonomous driving, modular and end-to-end, as well as their benefits and drawbacks. Paper cited [26] provides a more detailed description of techniques to autonomous driving.

Modular

The modular approach means that the autonomous system that drives a car is made up of separate modules, and each performs separate tasks required for the car to act properly in its environment. Modules can be created by humans or ML-based as there is no constraint over that, but these modules are typically organized in a pipeline. For example, these modules may handle navigation, lane orientation, perception, vehicle control, object detection, and other tasks. The modular approach is the industry standard in autonomous driving [26].

The pipeline extracts human-interpretable intermediate representations from modules, allowing people to gain insights into probable causes of system failure [26]. The modular approach demands more domain expertise compared to an end-to-end approach. However, in a commercial context, the system's modular architecture allows several teams to work on and specialize in different areas of the driving system simultaneously.

End-to-end

The end-to-end approach, as opposed to a modular approach, focuses on learning a driving policy that maps directly from environmental observations to vehicle actions. Typically, this driving policy is implemented as a deep neural network that generates either discrete or continuous vehicle control values. The neural network can be trained using either imitation learning or reinforcement learning. However, some approaches combine reinforcement learning and imitation learning, by combining exploration and expert data benefits.

A big problem with the end-to-end approach is that the neural network functions as a "black box," making it difficult for humans to comprehend the reasoning behind the network's inferences. In situations where the system's actions are flawed, it is consequently more difficult to comprehend why these actions were chosen. The training of RL-based end-to-end systems is expected to take place in a simulated environment because RL actions can be dangerous in real-world situations. The end-to-end approach requires a different domain knowledge level than the modular approach.

Reinforcement Learning for Self-Driving Vehicles

Most of the reinforcement learning research we have looked at so far has been focused on tackling locomotion difficulties in video games and robotics. So far, there have been very few studies on deep reinforcement learning for autonomous driving since an agent needs to explore the environment to learn. It is challenging to train a reinforcement agent in the real world safely. However, there is recent important work in the paper cited [9].

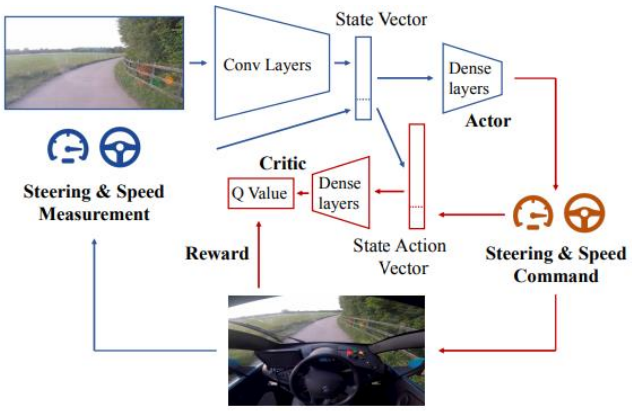

Figure 2.4 cited from paper [9], an actor-critic-based reinforcement learning model that learns to output steering and speed from a single input image and the vehicle's current steering and speed information.

Reinforcement learning solves Markov decision processes, proving that the task can be described as a process in which an agent makes actions in its environment and receives rewards for those actions. In practice, this means that we need to design a dense reward signal to teach the agent to solve the problem.

Figure 2.4 shows the model used in the research paper [9]. The model's action space, or output, is a continuous two-dimensional vector representing the steering angle and speed in km/h. It is interesting to observe that directly outputting the steer and speed and having the controller handle turning and throttle will likely reduce network noise. In their experiments, they attempted to encode visual data using both convolutional layers and a pre-trained variational autoencoder. They discovered that encoding the images with a variational autoencoder greatly improved the model's performance compared to training the convolutional layers alongside the remaining actor-critic parameters. Recent developments in DRL approaches and the fact that RL techniques depend on trial and error have piqued the interest of many researchers in testing these techniques in the domain of autonomous driving.

Reinforcement Learning with Variational Autoencoders

An variational autoencoder is a type of generative neural network model that consists of an encoder network followed by a decoder network. The idea is to use backpropagation to train the network to reconstruct a high-dimensional input signal after it has been compressed into some low-dimensional vector.

Variational autoencoders (VAEs) are used as a type of feature extractor in reinforcement learning. We can help a reinforcement learning agent learn by compressing high-dimensional observations into a low-dimensional latent space that is likely easier to learn. Understanding the state representations created by a VAE is referred to as *state representation learning.* [14] State representation learning improves the quality of an agent's learning by providing the agent with a state space detached from its feature representations and disregards distractors. The concept of disentanglement states that each latent space variable stores some essential, uncorrelated variable in the system, such as the agent's x or y position. The more disentangled the state representations are, the easier it should be for the agent to solve the environment. In addition, we have observed numerous instances of variational autoencoders used in reinforcement learning, which suggests that VAEs will play a central role in deep reinforcement learning.

3 Methodology

3.1 Introduction

In this chapter, we will look at how to set up a reinforcement learning problem to accelerate the learning of an autonomous vehicle. The main goal of this chapter is to demonstrate how deep reinforcement learning agents can drive in visually complex and realistic environments by analyzing the design decisions we make for our environment, agent, and network models.

3.2 Implementation Details

3.2.1 Setup

The implementation is in Python 3.7 with the package and dependency manager Poetry and PyTorch "1.12.0+cu113." Simulations and training were run on a system with a single Nvidia Quadro P5000 with 6 GB of video memory, a 4-core CPU, and 12 GB of RAM. A PPO implementation and complete source code for our performed experiments and CARLA environment setup are available at github.com/idreesshaikh/Autonomous-Driving-in-Carla-using-Deep-Reinforcement-Learning .

3.2.2 Algorithm

We chose Proximal Policy Optimization (PPO) as the deep reinforcement learning algorithm for continuous control problems [25] that worked best in our tests. PPO is a model-free reinforcement learning algorithm based on policy gradients that stops divergence with a first-order trust region criterion. In this part, we will outline the specifics of our PPO implementation.

PPO works by optimizing current policy π_θ with respect to its deviation from the previous policy $\pi_{\theta_{old}}$. Here, we will run our optimization step in a single environment to make implementation easier. We simulate a T-step trajectory for each optimization step and store $\left(s_t, a_t, r_t, d_t, V\left(s_t;\ \theta_v\right)\right)$ −tuples for each state-transition to create a small training buffer. In our implementation, the length of the training buffer was ten

episodes, which means that the value of T is not constant but instead changes. It also means that our training buffer grows bigger with time since our episodes get longer due to the agent's improving performance. Note that s_t corresponds to the latent space vector z produced by the encoder of a variational autoencoder plus some external features. Once T steps have been computed, we end up with T samples, which we stochastically sample minibatches of size $M \leq T$ from, for K number of epochs. These mini-batches of $(s_{n,t}, a_{n,t}, R_{n,t}, \hat{A}_{n,t})$ – tuples are fed through an actor-critic network which will optimize the parameters θ and θ_v with the Adam optimizer according to the $L^{CLIP+VF+S}$ loss function in equation 2.9. Recall that the $L^{CLIP+VF+S}$ introduces a clipping parameter ε for the L^{CLIP} loss – which ensures that our new policy after optimization is not too far from our current policy – and that we apply a value loss scaling factor α and entropy loss scaling factor β to the final loss.

3.2.3 Actor and Critic Architecture

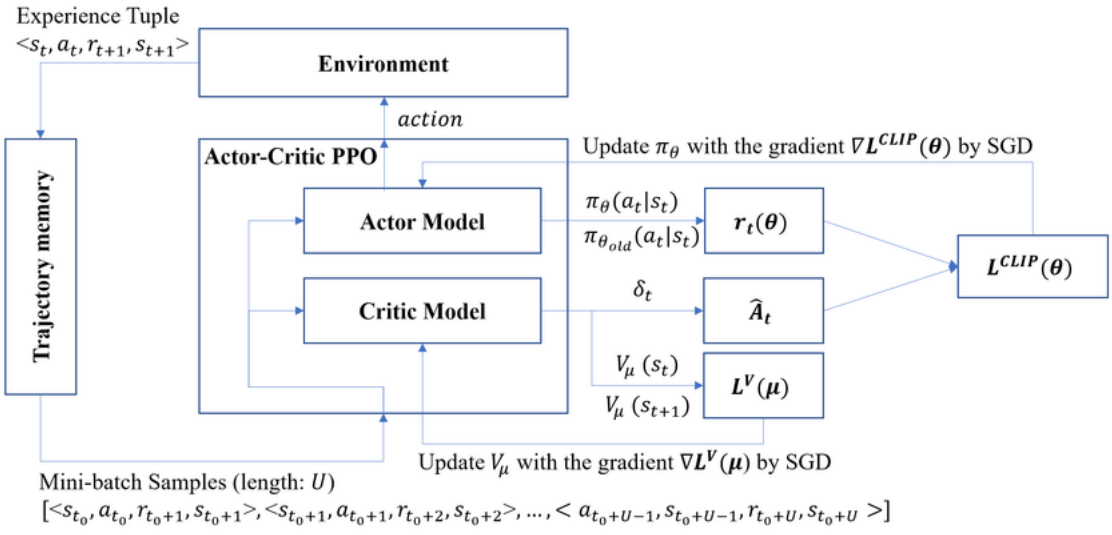

Figure 3.1 An actor-critic layout of PPO, the diagram is taken from "Federated Reinforcement Learning for Training Control Policies on Multiple IoT Devices" paper, cited [33].

PPO needs an actor $\pi(a_t|s_t;\theta)$ network and a critic $V(s_t;\theta_v)$ network. We implement these as two distinct Multi-Layer Perceptrons (MLP) — a better representation of actor-critic architecture shown in Figure 3.1. We use MLPs here since

the input, s_t, is a vector. The actor MLP consists of four fully-connected layers of sizes 500, 300, 100, and a_{dim}, where a_{dim} is the size of the action space. The activation function is used for $tanh$ for all of the layers. The output of the last layer in this MLP represents the unscaled means of the Gaussian distributions which we sample actions from, and we will denote the unscaled mean as o_i and the scaled mean as μ_i for the i^{th} action. To get the scaled means, we must first point out that each agent's actions are limited to a predetermined range of valid values. As a result, it is reasonable to scale the output of the MLP to the range of each action's respective range. We do this with the following transformation:

$$\mu_i = (\max(min(o_i, 1), -1) + 1)/2$$

(Equation 3.1)

By passing the raw outputs of the last fully connected layer through our clipping function (Equation 3.1), we get a value in the range of [−1, 1]. Adding one and dividing by 2 outputs our values in the [0, 1] range. Meanwhile, to define the multivariate Gaussian distribution that we sample actions from, we also provide a trainable parameter σ_i. Each σ_i is initialized to a value of our choice, which we will call σ_{init}. The weights in the μ_i layer are also initialized with variance scaling with a scaling factor of 0.2, see Table 2. This is done to reduce the possibility of the initial weights influencing the policy too much. Our environments use continuous action spaces, so to pick actions, we sample the multivariate Gaussian distribution given by $a_i \sim N(\mu_i, \sigma_i)$, and in evaluation mode, we pick $a_i = \mu_i$.

The critic is a simple MLP with four fully-connected layers of sizes 500, 300, 100, and 1, where the output will represent $V(s_t; \theta_v) \approx R(s_t)$. We use tanh for the first three layers and no activation for the final layer. Having no activation in the last layer makes it so that the critic can represent any possible value of $R(s_t)$.

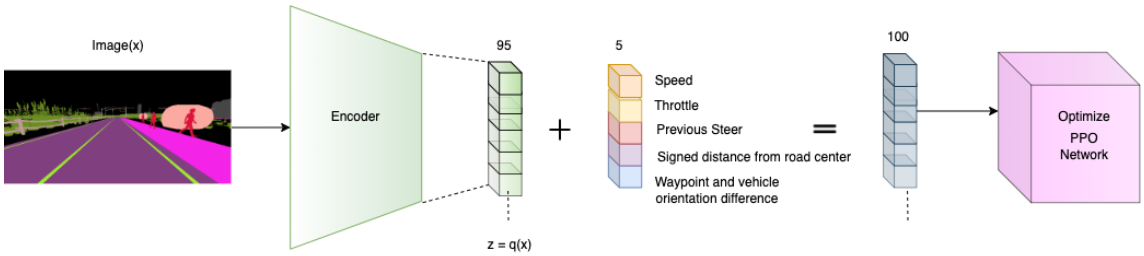

Figure 3.2 This diagram depicts the PPO+VAE training pipeline. Note: all the variable names are missing the subscript t. Diagram inspired by the works of [5].

In order to optimize the network, we use the action means and standard deviations from the network, we then calculate the log probability $log\pi\theta(a|s)$ of any action a under policy π given state s. Remember we need to calculate $rt(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ in order to compute the clipped loss, L^{CLIP} . We can do this with the help of the logarithm quotient rule:

$$log\ \pi_\theta(a_t|s_t) - log\ \pi_{\theta_{old}}(a_t|s_t) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} = r_t(\theta)$$

As a result, we compute the combined loss, $L^{CLIP+VF+S}$, and use the Adam optimizer to optimize it.

3.2.4 Variational Autoencoder

The VAE will serve as a feature extractor, making the state space we train our agents on more disentangled and eventually easier to predict. We created and integrated a VAE into our agent's learning process. Figure 3.2 illustrates the PPO+VAE training pipeline that we use in our setup.

Architecture

The encoder-decoder model architecture is better illustrated in Figure 3.3. We use a four-layered CNN as an encoder with channels ranging from 32, 64, 128, and 256. The kernel is configured with a size of 4x4, 3x3 alternatively, with a stride of 2. After the second and fourth layers, batch normalization overcomes the overfitting problem and improves regularization. The output of the last convolution is flattened and fed into a fully connected layer of 1024 units and then to two parallel fully connected layers of

size z_{dim}. After which, there are two parallel layers of z_{dim} units, where z_{dim} is a constant denoting the size of the latent space of the VAE's bottleneck. The layers use a LeakyReLU activation inspired by the paper cited [8][30]. We sample the latent space – interpreting the output of one of the parallel heads as the mean of a Gaussian distribution, while the other head interprets it as a standard deviation. We use the output of the parallel heads to sample vector z from a Gaussian distribution, which we then pass to the decoder. The decoder starts with a fully-connected layer that resizes z to the same size as the output of the final convolution of the encoder. We do this so that it will be easier to restore the image to its original size through transposed convolutions. We apply four transposed convolutions of 256, 128, 64, and 32 filters, strides of 2 and 3 x 3, and 4x4 kernels alternatively plus LeakyReLU activations. The kernel sizes were selected this way to make the size of the output of the final convolution a $w \times h$ image. With a sigmoid activation function, we squash the output values to ensure that the range of the output pixels is the same as the input ([0, 1] range.) The VAE optimizes by minimizing the loss using a mean squared error loss (MSE) between the input and output. With a learning rate of 1e-4, the model is trained using the Adam optimizer. Figure 4.3 shows the results with regular and reconstructed images. Our VAE implementation is inspired by the works of [25][7][8].

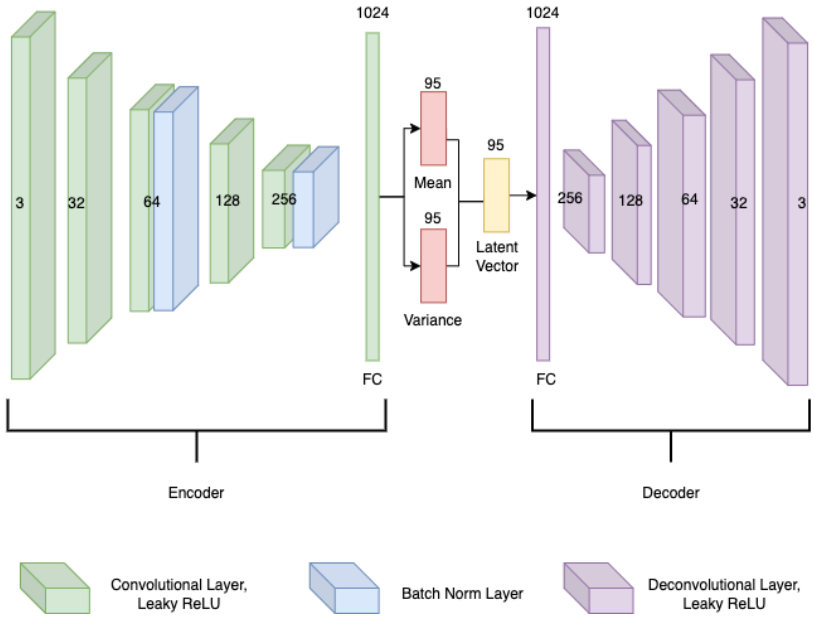

Figure 3.3 Architectural diagram of Variational Autoencoder.

Dataset: We gathered the datasets used to train the VAEs by driving around in the environment automatically and manually, gathering segmented images from the front-facing camera as we drove. We chose towns 1, 2, and 7, collected over 12,000 images and divided the dataset into 80 percent training images and 20 percent validation images.

Training: To train the VAE, we use the sampled Gaussian latent vector to reconstruct the input image and minimize the reconstruction loss. We use the Adam optimizer with a learning rate of $1e-4$, and a batch size of 32.

3.3 CARLA Environment

We are curious to test if this form of learning would work in a setting more like real-world driving. The agent needs to learn when to pay attention and when to disregard items based on distance, as misinterpreting these distances can have fatal repercussions for the agent. Furthermore, roads vary in width and length, as do their road markings and lane lines. Some roads may even be devoid of markings and lane lines entirely.

CARLA [1] (version 0.9.8), an urban driving simulator, will test our algorithm in a more challenging, realistic driving scenario. CARLA is an open-source simulator built in Unreal Engine 4 for autonomous driving research. The simulator focuses on reproducing a realistic driving environment with frequent urban driving scenarios. It comes with seven different maps out of the box, with three more offered separately and bringing the total to ten. As a server-client system, the server renders the simulation depending on commands from the client and the physics engine.

Meanwhile, the python-API-based client delivers steering, throttle, and braking commands to the server and receives sensor information. For example, the steering control has a range of [-1.0, 1.0], while the throttle and brake instructions have a range of [0.0, 1.0]. Furthermore, it provides a general-purpose API that allows us to spawn vehicles, cameras, and other sensors to use as we see fit.

We develop a CARLA-RL environment on top of the CARLA's API with reset and step methods to create an OpenAI gym-like RL environment in CARLA, which is not natively supported. Importantly, we run the simulator in asynchronous mode, which

means the simulator does not wait for the control message, which is more representative of the actual implementation of these algorithms. Moreover, we decided to write our Open AI gym-like environment for CALRA to easily leverage many different DRL algorithms. We have made this code publicly available at github.com/idreesshaikh/Autonomous-Driving-in-Carla-using-Deep-Reinforcement-Learning .

3.3.1 Environment Design

Map

The first step in designing our environment was to decide what map we would use. We decided to go for Town 7 (Figure 3.4) and Town 2 (Figure 3.5). Town07 more closely resembles a country-side road; it features long and short stretches of curved and straight roads, with a handful of intersections in the more densely interconnected roads in the center of the map. In this iteration of our environment, we will be ignoring traffic lights and other signage to make the scope of our agent smaller and more focused. The map has curved up and downhill roads and several structures of different shapes and sizes, and it additionally features a pond and diverse vegetation. Differences in vegetation help us know if our agent has generalized to small perturbations in the scenery. Whereas, Town02 is more urban, with less vegetation and more concrete buildings changing the scenario. The real intention behind these world-apart maps is that they will be using the same variational autoencoder. Therefore, we intend to make a generic VAE that works everywhere in every town.

Figure 3.4 Shows a top-down view of the map of Town 7 with the lap highlighted in blue. The orange dot marks the starting location, and the green dot marks the end location.

Figure 3.5 Shows a top-down view of the map of Town 2 with the lap highlighted in blue. The orange dot marks the starting location, and the green dot marks the end location.

Vehicle and Sensor Setup

Here, we will describe the vehicle and sensor setup used in our environment. The vehicle equips with a front-facing camera attached to the front of the car and a environment camera for spectating purposes. The front-facing camera outputs 160x80 semantically segmented images. Figure 3.6 show an example output of a spectating camera. CARLA provides a wide selection of sensors that may be mounted to the driving agent and receive data at each timestep.

Figure 3.6 Image from the environment spectating camera.

We use a camera sensor that produces Semantically Segmentation (SS) images, giving us 12 different classifications, e.g., lane marker, sidewalk, road, pedestrian, etc. In addition, we also use the Collision sensor to detect whether the driving agent collides with any static or dynamic objects in the environment. It records an event if a triggering condition is met, and that event may then be handled correctly.

Waypoints

Waypoints in CARLA are described as 3D-directed points representing the map's location and the lane's orientation, as shown in Figure 3.7. We calculate intermediate waypoints at a distance of one meter from each other between the start and the destination.

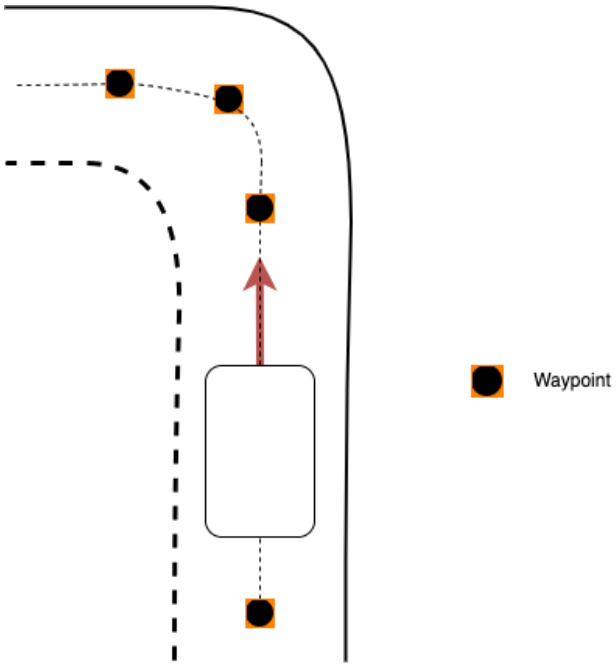

Figure 3.7 Waypoints (shown in red) are intermediate 3D-directed points that contains information of location and orientation between our start and an end position.

3.4 Reinforcement Learning setup

We will now present the reinforcement learning formulation for our problem of predetermined path navigation after laying out the necessary information for our environment design. It includes the state space, action space, and reward function definitions.

3.4.1 State Space

We use the front camera's semantically segmented image encoding and external features as the state inputs for the RL agent. These are described below:

Semantic Segmented Image

One of our state inputs is encoding a front Semantically Segmented (SS) image. The SS image can be simply retrieved using the Semantic Segmentation Camera Sensor, discussed in the previous section. Given the current state-of-the-art perception architectures, we think the segmentation encoding job can be learned in isolation; hence, we focus on learning control agents directly from SS images using DRL. We use a CNN-based variational autoencoder (VAE) for dimensionality reduction and its bottleneck embeddings as input to our agent policy network (Figure 3.2).

External Features

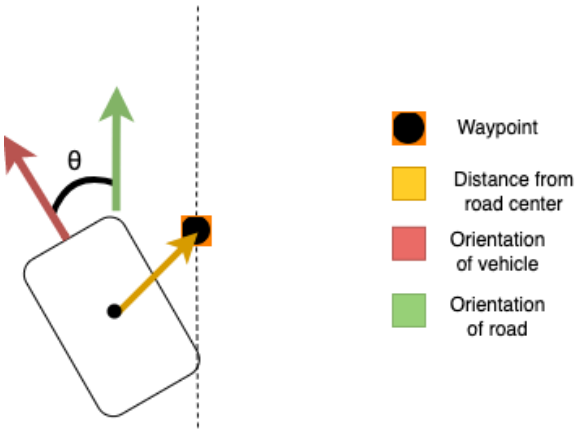

Figure 3.8 External features representation.

Figure 3.8 illustrates the external features we would use as our state input alongside the encoder front camera image, e.g., current speed, last steer value, throttle, distance from the trajectory, and the vehicle's orientation from the forward vector of the lane. These are described in detail below:

1. **Throttle:** It is the acceleration of our agent. It does not have to be normalized since it is already in the interval of [-1.0, 1.0].

2. **Current Speed:** The current speed of our agent is in km/h. However, it is preferred to be in the interval [0, 1], so we normalize it by dividing it by the target velocity: 20 km/h.
3. **Previous Steer:** The previous steer value is determined by the agent's steering action in the preceding timestep. We use the last steer action as a proxy to offer current state information related to steering because the CARLA simulator does not disclose the agent's current steer value. It ranges from [-1.0, 1.0].
4. **Distance from Waypoint:** The driving agent's signed perpendicular distance from the waypoints offers information on the agent's location relative to the best waypoint trajectory. Moreover, suppose the agent goes away from the trajectory, the value increases; therefore, It also serves as a termination criterion since we can detect whether or not the agent has deviated from the trajectory.
5. **Waypoint orientation:** It is the angle between the agent's forward vector and the forward vector of the waypoint, θ.

The encoded SS and external features make us our state s.

3.4.2 Action Space

In CARLA simulator, the actions for controlling our agent are usually defined as a tuple of three values (s, t, b). In that tuple s is our steer, t is the throttle and b is the brake. The s action value ranges between [-1.0, 1.0], and the t and b actions values range between [0.0, 1.0]. Our (Equation 3.1) is helpful here since it clips those values in those ranges.

The input representation is then fed into our policy network, which consists of a multi-layer perceptron and outputs $(\hat{s}, \hat{t})$, where $\hat{s}$ is the predicted steer action and $\hat{t}$ is the predicted throttle for that timestep, as shown in Figure 3.9. We decided to omit to brake, as we will be driving at low speeds with no other cars present or any dynamic actors, therefore not worrying about traffic rules – eliminating the need for braking.

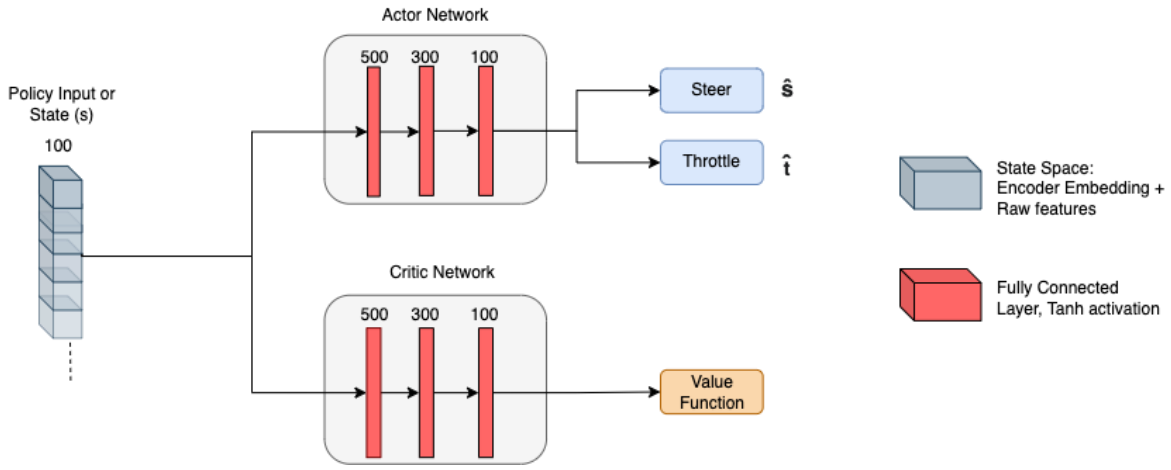

Figure 3.9 Proposed Architecture: Semantically segmented (SS) images and external features obtained from the CARLA simulator serve as inputs for the architecture. The SS pictures are encoded by a pretrained auto-encoder whose bottleneck encoding and external attributes are input to the policy network. The policy network generates the control actions ($\hat{s}$, $\hat{t}$), where $\hat{s}$ is the projected steering angle and $\hat{t}$ is the predicted throttle angle.

3.4.3 Extra Environment Techniques

Termination Criteria

We have listed the termination criteria for our environment underneath:

1. Have we deviated more than 3m from the center of the lane?
2. Are we driving slower than 1.0 km/h after the first 10 seconds of the episode has passed?
3. Have we incurred any other infractions like collision?
4. Are we driving at a speed that is more than our maximum speed?
5. Was our designed lap completed – success?

The above criteria are imposed on the agent to ensure that it follows the road and terminates the agent early so it can resume in a good state.

Early Termination

Early termination is a term we use to describe an environment that terminates the environment once the agent has reached a bad or unrecoverable state. From a training efficiency standpoint, the idea here is that we may learn faster by sampling more “good” states from a distribution smaller than the original distribution. For example, in our environment, we terminate the episode once the car drives off the road.

Checkpoints

We set periodic checkpoints along the track to make the concept of making an agent "fail quicker" easier. Every 100m, we save a new checkpoint, and the agent is reset to the previous checkpoint when it reaches a terminal state. The concept behind forcing the agent to fail quicker is that we will be able to learn faster by jumping right to the areas of the track where the agent is currently struggling. However, making the agent drive to the challenging stretch of the track takes a substantial amount of time. The data we acquire on this excursion may need to be more beneficial to the agent in completing the current problem.

Asynchronous

In an asynchronous environment, the environment does not wait for the step called before changing its state; instead, it updates autonomously and at varying rates. In terms of autonomous driving, this may offer certain advantages. Because there is no way to pause the environment to do calculations in real-life, training an agent in an asynchronous environment is similar to training an agent in the real world. As a result, it may be acceptable to conclude that if we can teach an agent to solve the asynchronous version of the environment, we may be able to train the same agent in a real-world scenario.

3.4.4 Reward Function

We formulate a dense reward function, R, but we want it to be a simple function that incentivizes our agent, enhances its behavior, and speeds up its training. In order to design an intuitive reward, we need to understand the things that improve the agent's performance. We came about the following 1) termination criteria, 2) the agent's speed, 3) the agent's distance from the waypoint, and finally, 4) the agent's orientation plays an important role. Let us look at the following list:

1. The current speed v of the vehicle is in km/h, and we want our agent to drive at a target speed v_{target} of 20km/h, and if our agent drives at the target speed, then it will get the highest reward. In order to achieve the maximum reward, our throttle value must be precise, and mostly this is not the case. Therefore, we must designate a range where our agent gets the maximum reward. After the target speed v_{target}, we introduce two new terms, min speed v_{min}, and max speed v_{max}.
2. Distance between the center of the vehicle and the center of the lane also plays a huge part. The closer our agent is to the center of the lane, the better it performs. The max agent can be far from the center d_{max} which is 3m as per our termination criteria, and $d_{norm} = \frac{1}{d_{max}}$.
3. Since we care how far our agent is from the center of the lane, we must also care how aligned our agent is to the lane's orientation. It will eventually improve the agent's steering behavior. Therefore, we introduce another term α_{diff}, calculated as the angle difference between the vehicle's forward vector and the current waypoint's forward vector. However, we should also define α_{max} which is the threshold maximum beyond which the reward value is just zero, and α_{max} is 20 degrees in our case. Here is α_{rew} , cited [5], that combines these terms as one:

$$\alpha_{rew} = \begin{cases} 1 - \left|\frac{\alpha_{diff}}{\alpha_{max}}\right|, & \alpha_{diff} < \alpha_{max} \\ 0, & otherwise \end{cases}$$

4. The first 3 points talk about the positive; however, we also need a negative reward when the agent performs poorly. Therefore, we put forth a reward of -10 for any infraction or accident defined in the previous section.

This eventually led us to this formulation, cited [5]:

$$R = \begin{cases} -10, & on\ infraction \\ \frac{v}{v_{min}} * (1 - d_{norm}) * a_{rew}, & v < v_{min} \\ 1 * (1 - d_{norm}) * a_{rew}, & v_{min} \leq v < v_{target} \\ \left(1 - \frac{v - v_{target}}{v_{max} - v_{target}}\right) * (1 - d_{norm}) * a_{rew}, & v \geq v_{target} \end{cases}$$

(Equation 3.2, cited [5])

3.5 Methodology

To start this sub-section, we will put every piece of the puzzle we have discussed so far together as a simple layout architecture. It outlines our novel approach of learning an end-to-end autonomous driving policy using our PPO agent. Figure 3.10 presents our overall architecture consisting of 3 important parts – 3.5.1. CARLA, simulation environment, 3.5.2. Variational Autoencoder to encode SS front camera images into latent space, 3.5.3. PPO agent to learn the driving policy and provide control values to our simulation.

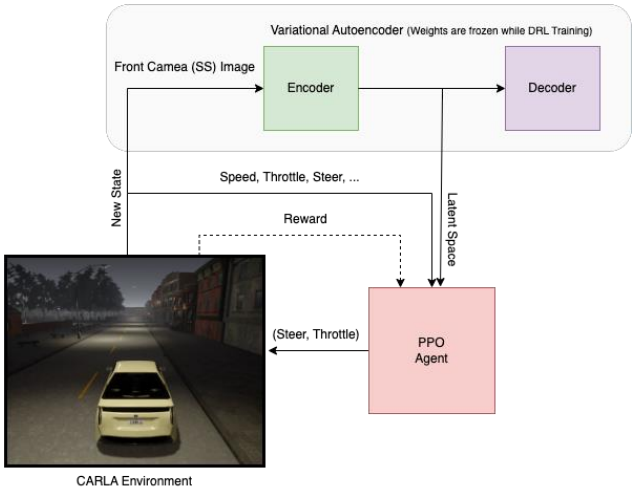

Figure 3.10 Architectural layout encapsulating all the three essential components: 1) CARLA Simulation, 2) VAE and 3) PPO Agent.

3.5.1 CARLA Environment

The goal of our agent is to navigate a predetermined route in Town 7 and Town 2 without deviating too far from the center of the lane. The laps for Town 7 and Town 2 are illustrated in Figure 3.4 and Figure 3.5 respectively, and it passes through multiple intersections. The laps for Town 7 and Town 2 are illustrated in Figure 3.4 and 3.5, respectively, and it passes through multiple intersections. For every intersection, the agent should drive straight (or left if there is no road straight ahead.) So there is no ambiguity in the agent's end goal whenever it encounters intersections. The lap length is 750m in Town 7 and 780m in Town 2, and the environment will consider the completion of three laps as a successful run and terminate. The orange dot marks the starting position of the agent.

3.5.2 Variational Autoencoder

The Variational Autoencoder (VAE) training process starts by driving around automatically and manually, collecting 12,000 160x80 semantically segmented images we will be using for training. Then, we will use the SS image as the input to the variational autoencoder ($h * w * c$ = 38400 input units). VAE's weights are frozen while our DRL networks. See section 3.2.4 for the architecture of VAE, and for the results check out section 4.1.

3.5.3 Proximal Policy Optimization

To train our RL agent, we use a cutting-edge on-policy reinforcement learning technique called Proximal policy optimization (PPO) [25]. Each training episode is on a predetermined route for each defined driving task in Towns 2 and 7, illustrated in Figure 4.8 and Figure 4.9. If the agent arrives at the destination, the episode is considered a success. On the other hand, it ends as a failure if the agent crashes into something or does not reach the target within the maximum number of timesteps (t_{max} =7500).

4 Results

4.1 Variational Autoencoder

Training Parameters

Table 1 details the training parameters utilized during the training of our variational autoencoder network (VAE):

Hyperparameter	Value
Learning rate α	1e-4
Batch size N	32
Loss Function	MSE
Architecture	CNN
z_{dim}	95
epochs	50

Table 1 VAE parameters.

Figure 4.1 shows the training loss per epoch of our VAE model, and Figure 4.2 shows the validation loss per epoch of our VAE model. We ran both the training and validation for 50 epochs with the above-mentioned parameters. The details of the VAE architecture are mentioned in the previous chapter.

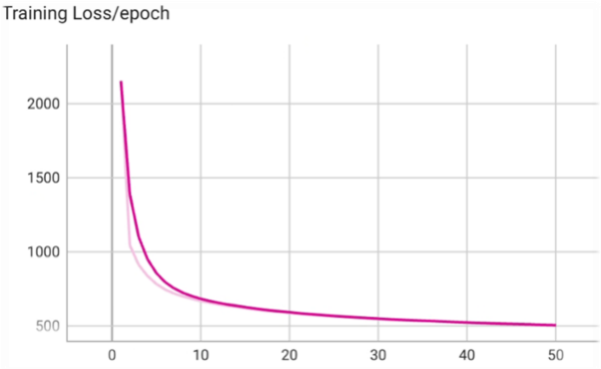

Figure 4.1 Training Loss/Epoch of VAE.

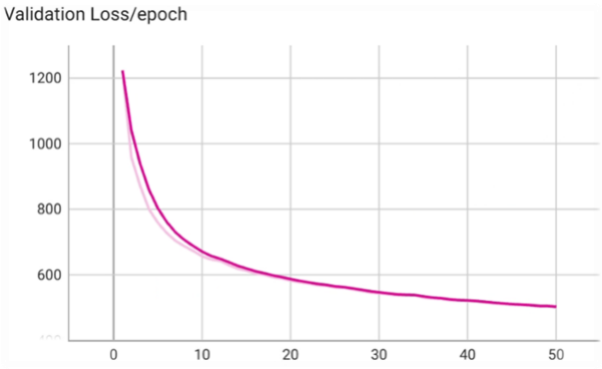

Figure 4.2 Validation Loss/Epoch of VAE.

Meanwhile, Figure 4.3 shows the image transformation that takes place after it has been fed into the VAE. Our reconstructed image is not as clear, but we do not really care about the output of the decoder since the latent space vector z_{dim} is what we need, and that is what will be used in our reinforcement pipeline.

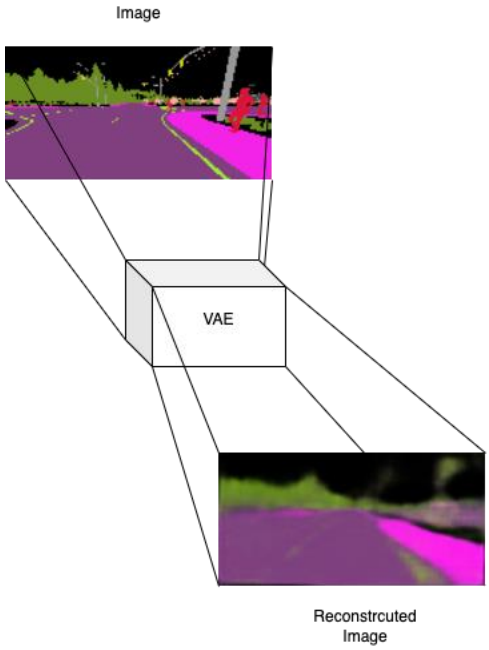

Figure 4.3 Image reconstruction from our VAE model.

4.2 Carla Environment

In this section, we shall be focusing on the factors that vary from training to training or within the training of our agent. Following the setup and methodology discussed in the previous section here, we will be focusing on the outcome and its variability in our training.

Throughout our training, we have focused on some very important variable factors/metrics that we shall be focusing on in this section:

- An average distance from the road's center during our training cycle
- Comparison between Town02 and Town07 (Urban and Semi-Urban), each of which has a distinctive landscape
- Episodic length as the training goes by
- σ_{init} (exploration noise) change within the training
- Asynchronicity in our environment setup

Let us start by looking at the bare-bone diagram of these two towns and our predetermined navigation route. In the map diagrams shown below in Figure 4.4 and Figure 4.5 cited [1], our agent is driving in the anti-clockwise direction onto the blue navigation route; It starts from the orange dot position and finishes at the green dot position.

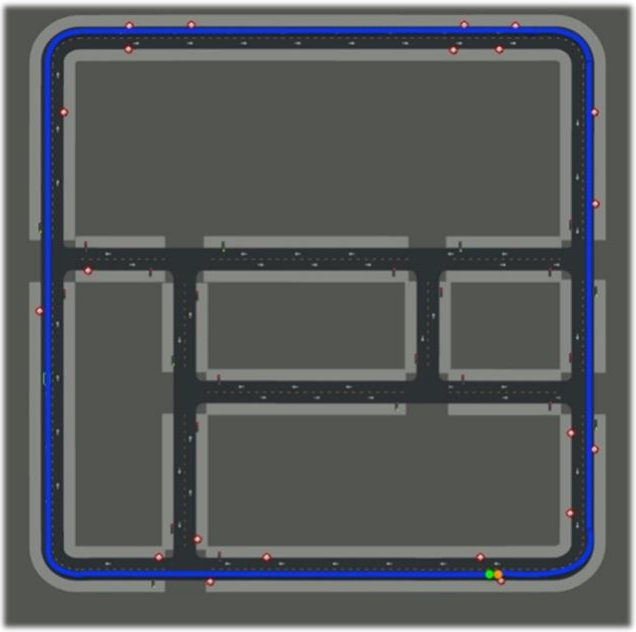

Figure 4.4 Town 2 top-view and path followed by our agent highlighted in blue. It is 780m long. Illustration cited CARLA Simulator documentation [1].

Figure 4.5 Town 7 top-view and path followed by our agent highlighted in blue. It is 750m long. Illustration cited CARLA Simulator Documention [1].

Our PPO agent must learn to follow the road even when there is no clear indication of which direction is the correct one; the agent can get stuck at intersections and junctions while navigating through the map. We hope that by introducing additional

policies, we will remove some ambiguity from these situations, resulting in an agent that learns more reliably and faster.

As it can be seen, our navigation path in Town 7 is trickier due to its topology and curvatures. It is also semi-urban and therefore lacks a lot of modern urban features; however, looking at Figure 4.6 we can deduce that in the first half of the training (before 1 million timesteps), our agent in Town 7 is performing. We deduce that setting σ_{init} (Gaussian exploration noise) to 0.2 gives our agent enough exploration range to converge toward a better policy due to the town's topology. Meanwhile, Town 2 is straighter and less curvy and is therefore not a fan of this exploration noise. However, after 1 million timesteps, we squeeze the exploration noise to 0.1, and it constricts its exploration range, therefore, giving the agent in Town 2 a fair bit of edge now as it starts to converge faster than the one in Town 7.

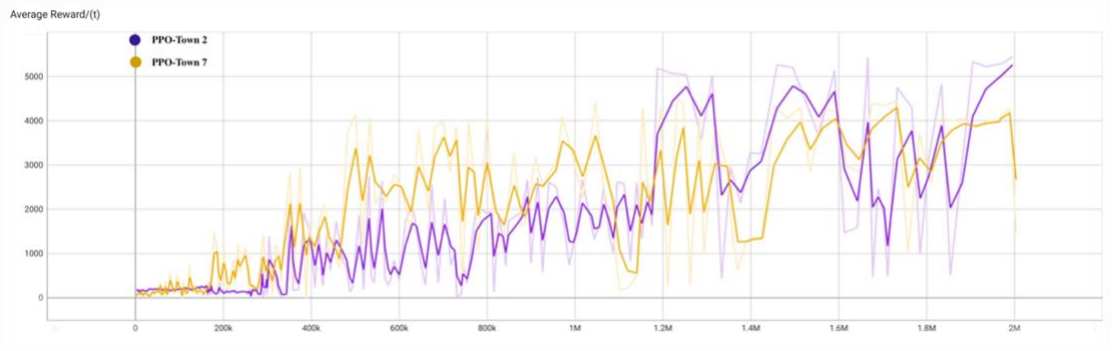

Figure 4.6 Average reward achieved in Town 2 and Town 2 over timestep t.

Moreover, we also recorded the average distance from the center of the lane, averaging over all timesteps of the environment (Figure 4.7). Here our relationship is reciprocal because the lesser the deviation from the lane center, the better the performance of the agent. We should point out again that in Town 7, the topology is not flat and straight at all but somehow has a deviation from the road center lesser than in Town 2. Our Gaussian noise parameter plays a huge part here as well, and since it works as an exploratory noise measure, it deviates our agent from straighter roads to explore more scenes making it more noticeable.

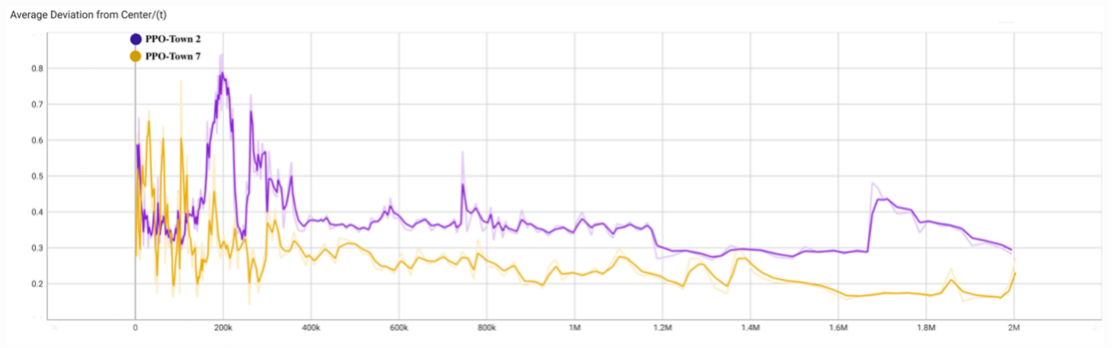

Figure 4.7 Average deviation from the center in Town 2 and Town 7 over timestep t.

In our experiments, we realized that when σ_{init} is low (e.g., σ_{init} = 0.1,) the resulting agent will drive more smoothly - the steering angle will be more stable - but the trade-off is that there is a greater chance that our agent will be stuck in local minima, and training will become slower because the sampled values are generally close to the mean. When σ_{init} is high, the steering of the agent is jerkier, but there is less chance of hitting a local minima, and the agent will learn more quickly. However, there is a trade-off to both situations; therefore, we decided to set σ_{init} to 0.2 at the start of the training and constrict it to 0.1 after a million timesteps. The choice to pick those numbers was motivated by the maps and predetermined navigational routes.

We also found that it is challenging to get the agent to "commit" to a control signal for long enough for the agent to observe its effects when using the standard Gaussian exploration noise. Regular Gaussian noise, for instance, will cause our agent to repeat bad actions most likely and gradually be pushed to move in the right direction over time due to minor perturbations in the actions if the agent is stuck on a sharp turn. As the agent would generalize better when an action's outcome is non-deterministic, giving the agent more variety in its data, we suspect that the temporal noise in the asynchronous environment may have been to the agent's advantage. Asynchronous training has some advantages for developing agents that could, in theory, learn to drive in real time.

Furthermore, in Figure 4.8 we are given the episodic reward of Town 7, which has a healthier incline even when the navigation route on the map is very complex. It can also be said that in Town 7, our agent learned to drive the entire navigation route in roughly around 900 episodes.

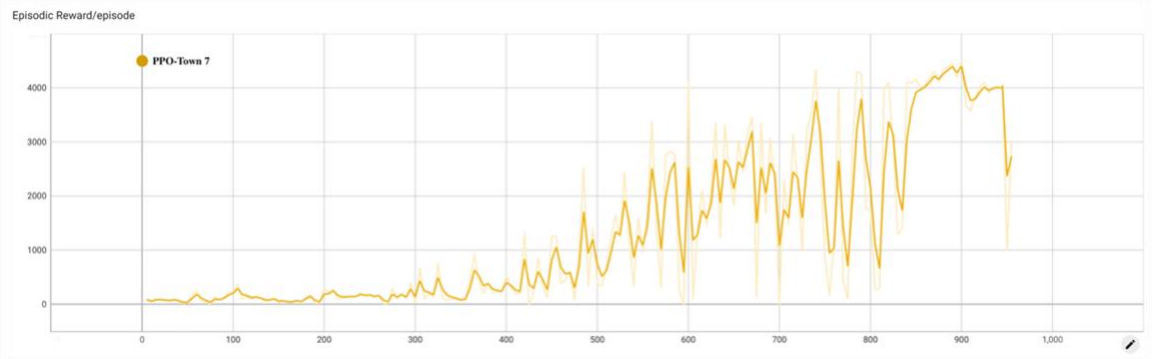

Figure 4.8 Town 7 average reward per episode.

Meanwhile, in Figure 4.9 we see the episodic reward of Town 2. We notice a sudden bump after roughly 1100 episodes because the navigation route is mostly going straight and taking a left turn. It can be deduced that by the 1100th episode, the agent has learned both driving features; therefore, it has achieved the maximum reward. Even if it looks like our agent is stuck in local minima before episode 1200, it is not hard for it to recover. In addition, Town 2 is simpler compared to Town 7, but it takes more episodes to learn an optimal policy, and the reason is that in Town 2, our agent fails faster, which is favorable for better policy convergence. Our agent recovers from a better state which eventually pushes it for a better policy, as can be seen by the episodic reward.

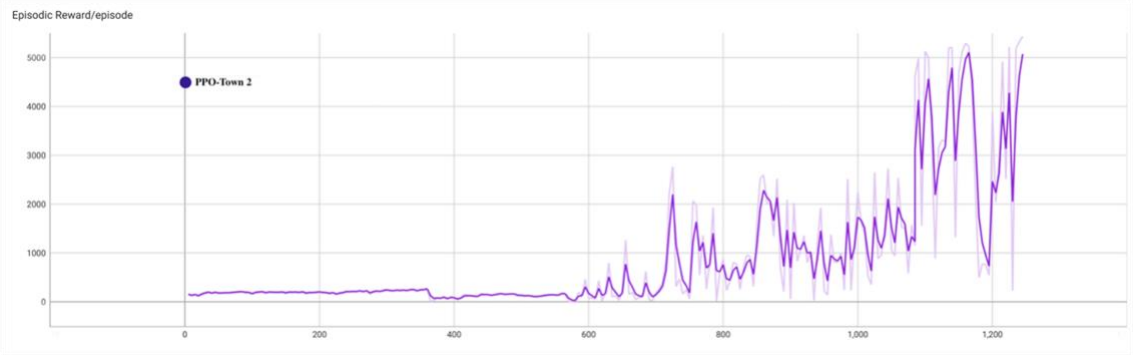

Figure 4.9 Town 2 average reward per episode.

Moreover, as soon as the agent reaches the first intersection, we observed that the agent repeatedly fails for perhaps hours before finally succeeding in making the turn. In order to encourage our agent fail more frequently in training mode, we added checkpoints in our environment. A 160x80 segmented image plus some external features might not contain all the necessary information for our agent to drive safely in

CARLA. In general, we believe that improving the state representation can significantly boost the performance of our baseline agent. For instance, adding memory to the agent using recurrent networks or combining input from various types of sensors, such as LiDAR and RGB cameras, are just a few examples.

We think that the VAE's representations are more beneficial to the agent when addressing the issue of autonomous driving, and we believe that it would be interesting to investigate comparable strategies for enforcing information-rich state representations that can be applied with deep reinforcement learning. Additionally, we could investigate training models that are tailored for semantic segmentation and attempt to train the agent using their compressed state representations.

Training parameters of our PPO-based agent are mentioned in Table 2 placed in Appendix A at the end of this paper for quick reference.

5 Conclusion

In this thesis, we have looked at a variety of ways in which deep reinforcement learning can be applied to the development of autonomous driving agents. We have confirmed in our findings that variational autoencoders can be used to speed up the learning process for DRL agents. Not only that we have also provided a thorough analysis of how different state representations impact the performance of agents. Despite some difficulties in establishing the final environment in CARLA, we have successfully developed a self-contained DRL system that can be used in conjunction with CARLA.

We present an approach of using waypoints as external input features alongside the control values and front camera images. We also designed an architecture to learn planning and control directly from semantically segmented images and external features using an on-policy DRL algorithm (Proximal Policy Optimization). Bear in mind that it was carried out without the presence of any dynamic actors, and the agent learned to navigate successfully. We propose using low-dimensional features to focus on policy learning in order to analyze further and decouple the issue of representation in policy learning.

Reinforcement learning is a self-learning algorithm that uses changes in environmental conditions and continuous action adjustment. This self-learning algorithm yields numerous methods, including value-based and policy-based algorithms, each of which is said to solve a different problem. To cite our ideas, we used various techniques in the CARLA environment to solve the problem of autonomous driving and created an autonomous driving agent. We confirm previous research by demonstrating that the policy-based algorithm is superior in assisting deep reinforcement learning in solving continuous state and action space problems such as autonomous driving.

5.1 Contribution

In the following list, we have outlined a summary of the contributions we made in this dissertation:

1. We have created a gym-like environment for CARLA that focuses on navigating a predetermined route both in Town 2 and Town 7, and it can also be expanded to different towns too with even more complex scenery.
2. We analyzed various environment design decisions in order to determine the best setup for training reinforcement learning-based autonomous driving agents. We have provided an example of how variational autoencoders (VAE) can be used in conjunction with CARLA.
3. We demonstrated how we train and use a VAE that can have an impact on the performance of a deep RL agent.
4. We discovered that training the VAE on semantically segmented (SS) images rather than the RGB input itself can result in significant improvements. This method of training the VAE ensures that the VAE is more focused on encoding the semantics of the environment.

5.2 Discussion and Future Work

This section will examine and summarize our findings, as well as offer suggestions for how to make our work better and what areas of autonomous driving research we should concentrate on in the future.

This work could be expanded in a variety of ways in the future. Experiment with other cutting-edge off-policy algorithms, such as TD3, SAC, and DDPG, that work in continuous domains and are more resistant to hyperparameters perturbations. The continuous action domain can also be discretized and formulated for Dueling Deep Q networks, Double Deep Q networks, etc.

To get the best results possible, it is also essential to solve some open problems, like formulating the optimization problem with reinforcement learning or imitation learning goals. This can be elevated even further by combining both these learning paradigms together.

5.3 Closing Remark

Together, our results demonstrate that deep RL agents can master driving in challenging scenarios with only limited training data. Our agent was able to finish a 750-meter track outside of Town 7 and a 780-meter track in the Town 2 using only a 160x80 image and some knowledge of external features. We conclude that small changes to reward formulations can significantly affect our agent's behavior. We suggest constructing reward functions to ensure good driving behavior in deep RL-based autonomous driving agents. While we appreciate the advantages of deep reinforcement learning over imitation learning, we feel that much more effort is required before we can construct deep reinforcement learning agents capable of handling all road conditions. We believe that we have helped the reinforcement learning for the autonomous driving research community by implementing a CARLA-based gym environment with a working PPO-based agent that can be used right away.

6 Acknowledgment

I thank my supervisor, Dr. Toka László, for his exceptional leadership and unwavering support during my thesis study. I experienced many ups and downs throughout my thesis, but his support was unparalleled.

I am also thankful to the faculty of BME VIK and its professors for their ongoing assistance and to the department of TMIT. I also want to thank the Stipendium Hungaricum Scholarship Program for allowing me to study at this excellent school for my bachelor's degree. Studying away from home in a foreign country was not an easier experience; however, my colleagues always ensured that I felt at home.

Finally, I will be eternally grateful to my family for their sacrifices and dedication to my education and progress. None of this would have been possible without their constant love, encouragement, and support.

Bibliography

[1] Alexey Dosovitskiy, Germa´n Ros, Felipe Codevilla, Antonio Lo´pez, and Vladlen Koltun. Carla: An open urban driving simulator. In CoRL. [Online]. Available: http://arxiv.org/abs/1711.03938, 2017.

[2] International SAE. Automated driving levels of driving automation are defined in new SAE international standard J3016, 2014.

[3] Alexander Stens, Iversen Szewczyk, Frank Lindseth, Gabriel Kiss. AI-agents Trained Using Deep Reinforcement Learning in the CARLA Simulator, 2022.

[4] M. Xie, L. Trassoudaine, J. Alizon, M. Thonnat and J. Gallice, 'Active and intelligent sensing of road obstacles: Application to the european eureka-prometheus project,' in 1993 (4th) International Conference on Computer Vision. DOI: 10.1109/ICCV.1993.378154, 1993.

[5] Marcus L. Vergara, Frank Lindseth: Accelerating Training of Deep Reinforcement Learning-based Autonomous Driving Agents Through Comparative Study of Agent and Environment Designs, 2019.

[6] 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, 2017.

[7] Eugenia Anello. Variational Autoencoder with Pytorch. [Online]. Available: medium.com/dataseries/variational-autoencoder-with-pytorch-2d359cbf027b, 2021.

[8] Arjit Sharma, Sahil Sharma. WAD: A Deep Reinforcement Learning Agent for Urban Autonomous Driving. [Online]. Available: https://arxiv.org/abs/2108.12134, 2021.

[9] 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.

[10] Richard S. Sutton and Andrew G. Barto. Reinforcement learning. Journal of Cognitive Neuroscience, 1999.

[11] David Silver et al. Mastering the game of go with deep neural networks and tree search. Nature, 2016.

[12] E. Espie´, Christophe Guionneau, Bernhard Wymann, Christos Dimitrakakis, Re´mi Coulom, and Andrew Sumner. Torcs, the open racing car simulator. 2005.

[13] David Gonza´lez, Joshue´ Pe´rez, Vicente Milane´s, and Fawzi Nashashibi. A review of motion planning techniques for automated vehicles. IEEE Transactions on Intelligent Transportation Systems, 2016.

[14] AntoninRaffin,AshleyHill,KalifouRen´eTraor´e,Timoth´eeLesort,NataliaD´ıaz Rodr´ıguez, and David Filliat. Decoupling feature extraction from policy learning: assessing benefits of state representation learning in goal based robotics. CoRR, abs/1901.08651, 2019.

[15] Qadeer Khan, Torsten Sch¨on, and Patrick Wenzel. Latent space reinforcement learning for steering angle prediction. arXiv preprint arXiv:1902.03765, 2019.

[16] Xiaodan Liang, Tairui Wang, Luona Yang, and Eric P. Xing. Cirl: Controllable imitative reinforcement learning for vision-based self-driving, 2018.

[17] Hendrickson, Josh. "What Are the Different Self-Driving Car "Levels" of Autonomy?" [Online]. Available: https://www.howtogeek.com/401759/what- are-the-different-self-driving-car-levels-of-autonomy, 2019.

[18] F. Codevilla, E. Santana, A. M. López and A. Gaidon, Exploring the limitations of behavior cloning for autonomous driving. DOI: 10.48550/ ARXIV.1904.08980, 2019.

[19] P. N. Stuart Russell, Artificial Intelligence: A Modern Approach (4th Edition). Language: English, 2020.

[20] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, YuhuaiWu, and Peter Zhokhov. OpenAI Baselines.https://github.com/openai/baselines , 2017.

[21] 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), DOI: 10.1109/icra.2018.8460487, May 2018.

[22] Hitesh Arora. Off-Policy Reinforcement Learning for Autonomous Driving, July 2020.

[23] Axel Sauer, Nikolay Savinov, and Andreas Geiger. Conditional affordance learning for driving in urban environments. arXiv preprint arXiv:1806.06498, 2018.

[24] Yesmina Jaafra, Jean Luc Laurent, Aline Deruyver 2, Mohamed S. Naceur. Robust Reinforcement Learning For Autonomous Driving, 2019.

[25] J. Schulman, F. Wolski, P. Dhariwal, A. Radford and O. Klimov, Proximal policy optimization algorithms. arXiv: 1707.06347 [cs.LG], 2017.

[26] J. Janai, F. Güney, A. Behl and A. Geiger, Computer vision for autonomous vehicles: Problems, datasets and state of the art. DOI: 10.48550/ARXIV. 1704.05519, 2017.

[27] Marin Toromanoff, Emilie Wirbel, and Fabien Moutarde. End-to-end model-free reinforcement learning for urban driving using implicit affordances, 2019.

[28] Óscar Pérez-Gil, Rafael Barea1, Elena López-Guillén1, Luis M. Bergasa1, Carlos Gómez-Huélamo1, Rodrigo Gutiérrez1, Alejandro Díaz-Díaz1.Deep reinforcement learning based control for Autonomous Vehicles in CARLA, 2021.

[29] Eric Yang Yu. Medium: Coding PPO from Scratch with PyTorch(1-4), 2020.

[30] Matt Vitelli, Aran Nayebi. CARMA: A Deep Reinforcement Learning Approach to Autonomous Driving, 2016.

[31] Nikhil Barhate. nikhilbarhate99/PPO-PyTorch, [Online]. Available: https://github.com/nikhilbarhate99/PPO-PyTorch, 2021.

[32] Nilesh Barla. Self-Driving Cars With Convolutional Neural Networks (CNN). [Online]. Available: https://neptune.ai/blog/self-driving-cars-with-convolutional-neural-networks-cnn, 2022.

[33] Lim, Hyun-Kyo & Kim, Ju-Bong & Heo, Joo-Seong & Han, Youn-Hee. Federated Reinforcement Learning for Training Control Policies on Multiple IoT Devices, 2020.

Appendix A

PPO Training Parameters

This appendix presents the training parameters used in our Proximal Policy Optimization (PPO) agent in this dissertation:

Hyperparameter	Value
Discount factor γ	0.99
Clipping parameter ε	0.2
Learning rate	1e-4
Value loss scale α	0.5
Entropy loss scale β	0.01
Number of epochs K	7
Initial noise σ_{init}	0.2 (squeezed to 0.1 after a million timesteps)
Horizon T/ Batch size M	$\sum_{i=0}^{n} x$; n=10, x = episodic length in timestep

Table 2 Hyperparameters of PPO-based agent.

Thesis Project

The project code is available at https://github.com/idreesshaikh/Autonomous-Driving-in-Carla-using-Deep-Reinforcement-Learning .

All the sources used in this work have been cited in Bibliography. In addition, the Project uses sources mentioned in the Bibliography for both theoretical and practical implementation.