Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Marcus Loo Vergara

Accelerating Training of Deep Reinforcement Learning-based Autonomous Driving Agents Through Comparative Study of Agent and Environment Designs

Master's thesis in Computer Science
Supervisor: Frank Lindseth
June 2019

NTNU
Norwegian University of Science and Technology

Marcus Loo Vergara

Accelerating Training of Deep Reinforcement Learning-based Autonomous Driving Agents Through Comparative Study of Agent and Environment Designs

Master's thesis in Computer Science
Supervisor: Frank Lindseth
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Abstract

In this thesis, we will be investigating the current landscape of state-of-the-art methods using deep reinforcement learning for the purposes of training self-driving cars. Autonomous driving has garnered the interest of researchers, governments, and private companies as of late, as such technologies promise to solve several problems that are prominent in modern society. Examples of such problems include individuals spending a lot of their time in traffic due to congestion, and people and institutions having to financially support costly car accidents made by human errors. Advancements in machine learning is what drives autonomous vehicle technology forward, and we have already seen several big actors in the automobile and artificial intelligence industries take advantage of this; having autonomous vehicles drive several miles on public roads without incidents. The primary goal of this thesis is to provide a comprehensive analysis of current methods in deep reinforcement learning for training autonomous vehicle agents, and our main contribution comes in the form of providing a working example of a Proximal Policy Gradient (PPO) based agent that can reliably learn to drive in the urban driving simulator, CARLA. Through our work, we provide two OpenAI-like environments for CARLA that we have designed to (1) minimize overall training time, and to (2) provide the necessary metrics for comparing models across runs. One of these environments is only concerned with following a predetermined lap, while the other is focused on navigating arbitrary paths provided by a topological planner – similar to how we would navigate in real-life. In creating these environments, we provide some analysis as to how various environment design decisions – such as training with different reward formulations, training in asynchronous/synchronous environments, or using environments with or without checkpoints – affect the resulting agent. Furthermore, we will be presenting various experiments on the use of variational autoencoders in the training pipeline, and show how we were able to significantly improve the quality of our agent by training a variational autoencoder to reconstruct semantic segmentation maps rather than training it to reconstruction the source RGB images themselves. For the lap environment, we will provide a couple of models that reliably learn to drive along the 1245m lap in approximately 8 hours. For the route environment, we will show that we can train a PPO network with multiple policy networks to create an agent that is able to follow the commands of a topological planner to moderate success.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

In the digital age, we have seen an increasing desire to automate our every-day tasks, along with a desire to use technology to make the world a safer and more reliable place to live. Creating intelligent systems that are able to safely drive a car from point A to point B without human intervention is an example of this. This is the problem we wish to solve when we create autonomous vehicles.

While the idea of creating autonomous vehicles is old and well-known, it is only recently that we have seen has been public companies and researchers starting to consider these technologies seriously. Along with other computer vision and artificial intelligence problems, creating autonomous vehicles is now considered to be viable due to the multitude of advancements we have seen in artificial intelligence through deep learning.

1.2 Autonomous Driving

Autonomous, in the context of self-driving vehicles, means "self-governing." An autonomous car is a complex system that operates with some level of automation. When we talk about

self-driving vehicles, we typically talk about vehicles that drive from some starting location A to some target location B with limited to no intervention made to the control system of the vehicle by a human driver. The *Society of Automotive Engineers* has defined a set of levels of increasing automation when it comes to self-driving cars [SAE14]. These levels are as follows:

- Level 0 - No Automation: Here the human driver is the only one who interacts with the vehicle's control system. A system is still considered level 0 if it has some warning signals or intervention systems integrated.
- Level 1 - Driver Assistance: The human driver shares control of the vehicle with the automated system. An example of a level 1 autonomous car is a car with *Adaptive Cruise Control*, a system where the car determines it's own speed while the human driver is responsible for steering the vehicle.
- Level 2 - Partial Automation: The system takes full control of the cars steering and speed, however a human needs to be prepared to intervene immediately whenever the system fails.
- Level 3 - Conditional Automation: The driver does not need to be prepared for immediate intervention, however they will need to respond within a limited time when the system calls for it.
- Level 4 - High Automation: The car can drive without the drivers intervention within limited geographical areas or in limited types of driving scenarios. If the car cannot proceed, the car will safely park the vehicle unless the human driver takes control.
- Level 5 - Full Automation: No human intervention required at all.

Today's automobiles consist primarily of a mix of level 0 ("no automation") and level 1 ("hands on") autonomous vehicles. Some car manufacturers such as Tesla, have produced vehicles that are commercially available which have features that are considered level 2 ("hands off") automation, such as automatic lane changing. There are also a few examples of commercial and non-commercial vehicles featuring level 3 ("eyes off") automation, such as Audi's A8's "Traffic

Jam Pilot;" capable of autonomously driving the car at speeds up to 60 km/h in traffic jams. There are no commercially available level 4 ("mind off") or level 5 ("steering wheel optional") automobiles, however there is on-going development for such vehicles, and Waymo's self-driving car is an example of a self-driving car that has no steering wheel. Level 5 automation is naturally the ultimate goal in autonomous driving research and development.

Why is autonomous driving interesting?

Driving has become an essential part of modern life. People have a need to move long distances over short periods of time, so cities have been constructed and redesigned in such a way to accommodate fast-moving automobiles. Traffic rules have been put in place to make driving a safe and reliable way to commute. However, modern day driving is not perfect, and introducing autonomous driving systems promises to solve a couple of problems we face in today's society:

Automobile Accidents: Humans are unreliable drivers. In 2017, Norway saw 106 deaths and 665 critical injuries due to traffic [Sen]. However, Norway is not a very densely populated country, and our traffic rules are relatively strict, so we have significantly lower per-capita numbers compared to the world. On a world basis, we saw 1,250,000 deaths in 2015 according to WHO [Org], a figure that is 8x larger than Norway's per-capita number when adjusted by population size. Autonomous vehicles promises to push these figures towards zero through automation. Computers cannot get sleepy. Computers cannot get angry. Computers can react faster than humans. They can observe their environment with multiple sensors at once, and are not limited to using only light and color sensors, but can for example use depth and distance sensors to get a better understanding of the environment than their human counterparts. Therefore, we expect such systems to be a substantially more safe way to transport people.

Commuting Time and Emissions: Computers can also communicate with each other wirelessly, and plan their behaviours in unison to an extent humans are incapable of. An autonomous car on the road can know where its nearby autonomous cars are going, and can therefore plan their trajectories accordingly. This would reduce commute time, CO2 emission, and have overall benefits to society. With eyes-off automation, the passengers could even do their own work

and be productive during their commutes. If autonomous cars were efficient, ride-sharing services may be implemented to eliminate people's need to own cars altogether, potentially cutting down on the environmental costs of producing cars and the financial expenses of owning a vehicle. Reducing the number of cars in circulation could mean that a lot of physical space can be saved by reducing the need for large parking spaces.

1.3 Components of an Autonomous Car

An autonomous car require several systems or components which solve their own individual task. We may divide such a system into the following categories:

Sensors, Compute and Control

What components are required to make a car drive intelligently? In designing an autonomous vehicle, we should consider what components and sensors are necessary to drive optimally; while also ensuring that our vehicle and methods are energy efficient enough to drive for long periods of time with limited battery capacity. We also need to develop the algorithms that control the car and the interfaces it uses to communicate with the on-board sensors.

Mapping and Localization

Mapping and localization is about finding out how to localize the car on a high-definition map and to interpret the topology of the car's surrounding environment. Localization on high-definition maps may use positioning systems such as Global Positioning System (GPS) and may also combine this information with information it can extract from the environment. For example, given a rough estimate of its location on a map, the car may be able to determine a more precise location by using information it can gather from its surroundings, for example, by correlating its observations with its estimates on where nearby buildings should be according to the vehicle's internal map.

Perception and Prediction

This is about making sense of sensor input, and generating predictions of various types of properties for both static and dynamic objects in the environment, *e.g.* object location in 3D space, object trajectories, object types, etc. Cars, buses, trucks, bicycles and pedestrians are examples of dynamic objects that should be tracked, and static object may include traffic lights, signs, lamp posts, etc. A perception module should be able to determine the location static object such as lane lines and signs, in addition to also understanding the meaning of various road signs and signals. Perception information can be extracted with traditional RGB cameras, infrared cameras, LiDAR laser scanners, etc.

Planning and Control

Planning is all about finding out how to manipulate the acceleration, break and steering to navigate the vehicle from point A to point B. To control, the vehicle should use the output from its sensors, or potentially the output of a perception module, to determine a series of control signal that will lead the vehicle to its goal. Ideally, the vehicle should be able to drive without any incidents with other vehicles, bikes, pedestrians, etc.

Simulators for Training and Validation

It is vitally important that we have ways to test and verify our control and perception algorithms, before deploying them onto a real vehicle. This is the main purpose of creating simulators. Since we use simulators to verify our algorithms before deployment, it is also vitally important that the simulator emulates the physics and appearance of real-life as closely as possible. There are several open-source, high-fidelity, simulators for autonomous driving research. The most prominent ones are CARLA [DRC+17] and AirSim [SDLK17]. We will discuss CARLA a bit in later sections.

1.4 Other Topics in Autonomous Driving

There are also several interesting philosophical, juridical, and societal topics to consider when it comes to autonomous vehicles.

Safety and Security

Assuring the safety of an autonomous system is important for deployment of autonomous systems. Simulators are useful in validating methods, however, deep learning based agents are often hard to interpret. In the case of an accident, we need to have ways to understand why the agent failed. So safety is, among other things, about building interpretability into our systems. Security is also a very important topic in the safety of autonomous vehicle; we have to ensure that our autonomous vehicle systems cannot be exploited or intercepted by outside actors.

Collaborating Vehicles

Collaboration is an important step in maximizing the efficiency of autonomous cars on the road. If cars can communicate, they can plan together, allowing less congestion in traffic and thereby faster commuting and less resource usage.

Ride Sharing

Autonomous vehicles may eliminate the need for individuals to own cars altogether. This reduces the need to produce and own cars, and we can instead focus on providing robust and safe ride sharing solutions.

Privacy and Ethics

Maintaining the privacy of the users and the surrounding people is also important to consider. When every vehicle is equipped with out and inward-facing cameras, people can easily

be tracked and identified; something that could violate existing privacy laws. Additionally, autonomous vehicles may have to make ethical decisions in certain situations. The most commonly posed scenario is that in which the car finds itself in a dangerous scenario, and has to decide whether to sacrifice a innocent bystander vs. the owner of the vehicle.

1.5 Objective

Our main objective in this thesis will be to compare and contrast multiple aspects of agent and environment designs (*e.g.* reward functions, environment objectives, agent parameters, state representation learning through variational autoencoders, etc,) with the goal of finding agent-environment setups which reliably create agents that can maneuver complex (simulated) 3D environments. In the pursuit of goal, we set out to answer the following research questions:

- What are the principles and methods behind deep reinforcement learning, and are state-of-the-art reinforcement learning methods able to learn how to drive a car in a controlled and reliable manner?
- What objective, metrics, and state representations should a reinforcement learning environment provide to best fit the needs of autonomous driving researchers?
- How do we design reward functions to bring out desired driving behaviour from reinforcement learning agents?
- How influential is environment design to the training speed and overall performance of a reinforcement learning based agent?
- How can we generate information rich state representations, and to what extent do good state representations accelerate agent learning?
- Can we use deep reinforcement learning to train agents to drive in environments that require the agent to be able to perform multiple types of maneuvers?

Chapter 2

Background

2.1 Machine Learning

2.1.1 Introduction

Machine learning is a sub-field of artificial intelligence that focuses on developing algorithms and systems that learn to solve problems by examples. Machine learning has garnered a lot of interest over the last decades, and its popularity stems from its simplicity and the empirical success of recent machine learning-based approaches. For problems that exhibits some degree of non-triviality, machine learning often outperform hand-crafted algorithms, for example in the case of classifying objects in images [DDS+09]. Since the theory behind machine learning algorithms is often general-purpose, the same theory can often be re-purposed for any number of domains, given that the necessary data for the given problem exists.

Since machine learning involves learning from examples, it is essential to collect the required data for training and testing. "Data" refers to anything that represents some type information that has meaning to machines or humans, for example, digital images, stock prices, etc.

Supervised learning is the most straight-forward application of machine learning, where the goal is to create regression models that maps some input data to some output data – in other

words, create a function F, such that $F(X) \approx Y$, where X represents the domain of the input, *e.g.* "pictures of animals," and Y represents the corresponding labels for each X. In order to apply supervised learning to a problem, a reasonably sized subset of X and Y pairs needs to be gathered and manually labeled by an expert. Given that our dataset (the complete set of labeled data-points) is well-formed, it is possible to create regression models that approximates, and generalizes $F(x)$. This is often done with artificial neural networks (ANNs). Gathering this data is a laborious process, so many researchers have decided to make their manually gathered datasets available for the public and other researchers. Some popular ones include MNIST [LBBH98] and ImageNet [DDS+09] for image classification, and MS COCO for semantic segmentation. There are also datasets made specifically for self-driving cars, such as KITTI [GLSU13], Apollo [HCG+18], etc.

In addition to supervised learning, there are also machine learning tasks that work unsupervised or semi-supervised. In unsupervised learning we wish to solve the same regression problem of $F(X) \approx Y$, *without* any labeled data, Y. A typical example of unsupervised learning is clustering, where we will try to group data points based on some statistical properties of the data. Semi-supervised learning leaves us somewhere in between supervised and unsupervised, where only a small subset of X is labeled.

In this report, we will explore another machine learning approach called *reinforcement learning*. Reinforcement learning features an intuitive learning framework, where we are considering an *agent* that lives in an *environment* in which the agent is able to act. The goal in reinforcement learning is to train the agent to maximize its "usefulness," or *utility*, with respect to some goal that was determined beforehand. In the case of autonomous vehicles, the goal could for example be to reach some destination B, without crashing into any objects. In this scenario, the utility of the agent is some quantity that describes how well it can drive from A to B without crashing, and the goal of the agent is to maximize this quantity. We will discuss reinforcement learning further in Section 2.3.

2.1.2 Artificial Neural Networks

Artificial Neural Networks (ANNs) are computational networks of nodes and connections resembling that of biological neural networks. Formally, artificial neural networks in machine learning refer to directed-acyclic graph with weights along each edge of the graph. When a datapoint x_i is fed to an ANN, it will propagate thought the network, and we end up with some prediction $F(x_i) = \hat{y}_i$. The goal in machine learning is to find the best configuration of network weights, often denoted W, with respect to some objective. This objective may, for example, be to minimize the error of a class predictor to ground truth labels, that is, minimize $L = \sum_i (\hat{y}_i - y_i)^2$ over all labeled data, $x_i \subset X$ and $y_i \subset Y$.

Gradient Decent: Gradient decent is the most common optimization technique used with ANNs. It works by calculating the gradient of the loss function, L, with respect to the weights of the network W. Once we have the gradients, we nudge our weight variables in the direction of greatest decent as follows:

$$w_{ij} \leftarrow w_{ij} - \alpha \frac{\partial L}{\partial w_{ij}} \tag{2.1}$$

Where α is a hyperparameter that determines how much we should nudge our weights in the direction of steepest decent per step, and $\frac{\partial L}{\partial w_{ij}}$ represents the direction of steepest ascent along the loss function's surface with respect to a particular weight $w_{ij} \in W$.

2.1.3 Deep Learning

The simplistic formulation of ANNs given above suffers from an inability to generalize to new data, because this type of ANN is essentially no different than a linear combination of transformations on input X over the layers' weight matrices W. It turns out that constructing deeper, non-linear models resolves this issue. In deep learning, we combine these ideas by constructing deep networks where we apply non-linear activation functions in our neurons to let the network extrapolate non-linear relationships in the input data. Common non-linear activation functions

include the sigmoid function and the Rectified Linear Unit (ReLU) function.

Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are neural networks that apply a convolutional operation to the input in some form. In image processing, convolution refers to a common operation that was historically used to extract features such as edges, and to blur and sharpen images. Convolution identifies spatial relationships in the pixel values of an image by sliding a 2-dimensional filter ω over the input image, calculating the dot-product between the filter and the pixels in the image. These filters were typically hand-crafted for a specific task, however, with CNNs these filters are learned. The idea of learned convolutional filters is an older idea from 1989 [LBD+89] that has proven to be very useful in most recent computer vision related tasks.

2.2 Imitation Learning for Self-Driving Vehicles

There has also been a lot or research in algorithms specific for self-driving vehicles. In this section, we will take a closer look at the ideas behind end-to-end imitation learning, and then later discuss how these methods compare to deep reinforcement learning methods for self-driving vehicles.

2.2.1 Background

There is a long history of research in end-to-end methods for self-driving vehicles. The idea of training an autonomous car by optimizing a neural network dates back almost three decades to Pomerleau's thesis on the *Autonomous Land Vehicle in Neural Network* (ALVINN) system [Pom93]. ALVINN demonstrated that it is possible to use machine learning-based models to train cars to follow roads, and later works have build this idea. An example of such work originates from military research, where the Defence Advanced Research Projects Agency (DARPA)

published a report in 2004 detailing their project called DARPA Autonomous Vehicle, or DAVE. DAVE was a small remote-controlled (RC) vehicle that learned to drive 20 meters autonomously in complex environments without crashing. DAVE was equipped with two front-facing cameras, and would be operated by an expert driver to generate data. The data would consist of the left and right images, in addition to the control signal of the expert. After sufficient data was collected, they would fit a model to map the input images to their respective steering commands; in other words, they were solving a supervised learning problem. Using the input of an expert operator to guide the learning is known as *imitation learning* or *behaviour cloning*.

2.2.2 End-to-End Imitation Learning

Researchers at NVIDIA built on the ideas presented in DAVE in their 2016 paper titled *End to End Learning for Self-Driving Cars* by Bojarski *et al.* [BTD+16]. In their work, they built a DAVE-like system that utilizes the advancements in deep learning to teach a full-scale vehicle to drive. They call this new system DAVE-2, and similar to DAVE, it learns to drive by training on data generated by an expert driver. Unlike DAVE, DAVE-2 uses a deep, CNN-based architecture to extract features from its input images. In their "on-the-road" test, they found that DAVE-2 was able to drive autonomously 98% of the time, excluding time spent changing lanes or time spent turning from one road to another. These results show that end-to-end learning is quite powerful in training autonomous vehicles.

Let's consider the objective of an imitation learning system like DAVE-2 from a mathematical stand point. Say that we have a neural network parameterized by θ, and that we have collected some images (observations), $\mathcal{O}$, and sampled their corresponding control signals, $\mathcal{A}$, from the expert. Ideally, we want to make the agent perform the same action as the expert, $a_i \in \mathcal{A}$, when it is presented with the same observation $o_i \in \mathcal{O}$. We formulate this as an optimization problem, where we want to minimize the error between the agent's predicted control vector, $F(o_i; \theta)$, and the expert's "ground truth" control vector a_i:

$$\underset{\theta}{\text{minimize}} \sum_{i} L(F(o_i;\theta), a_i) \tag{2.2}$$

Where L is some discrepancy measure, such as the squared error between the prediction and the ground truth, $L = (F(o_i;\theta) - a_i)^2$. Equation 2.2 can then be optimized with stochastic gradient decent, and given that we have enough data, our agent may generalize to new observations that are not in $\mathcal{O}$.

This simplistic version of imitation learning has some major drawbacks, however. First, without any examples of "bad" or off-center driving in our data, the car will not learn to correct itself once it finds itself in an unfavorable position. This generally leads to situations where small perturbations can make the car to "spiral out of control." DAVE-2 and similar systems relieve this problem by having 2 off-center cameras (left and right), and add those images to the data set with augmented, "corrective," control vectors. This, coupled with other image augmentation techniques led to an agent that in the end was pretty robust to these types of perturbations. Imitation learning still has a pretty significant limitation in terms of self-driving: it assumes that the optimal action can be inferred from the camera observations alone. This, however, is not true in the case of an agent making a turn at an intersection, where the optimal action depends on other factors such as the destination of the passengers. Codevilla *et al.* aims to correct this in their 2018 paper *End-to-end Driving via Conditional Imitation Learning* [CMD+17].

2.2.3 Conditional Imitation Learning

Codevilla *et al.* [CMD+17] introduces a conditional imitation learning architecture that aims to address the issue of not being able to issue navigation commands to a vehicle trained with imitation learning. What Codevilla *et al.* propose, is a modified model that learns sub-policies for the three different commands: {*turn left, turn right, continue straight*}. This system works for the most part similar to the system described in Section 2.2.2, but they introduce a set of discrete commands $\mathcal{C} = \{c^0, \ldots, c^K\}$, where $K = 3$ in this case, since there are three commands. In their network architecture, they construct one branch for every command c^k, and optimize

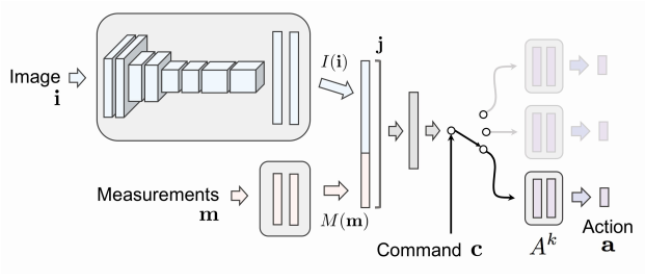


Figure 2.1: Codevilla's *et al.* conditional imitation learning model. Measurements m is a one-dimensional vector that consists of various measured properties of the system such as speed, position, throttle, etc. Notice how the fully-connected layers in each A^k branch are conditioned on the input command c.

the branches separately depending on which command is currently active. Each sub-policy, A^k, has two fully-connected layers (see Figure 2.1,) ensuring that each branch learns to use the features that are most relevant to each sub-policy's respective task. During training, we will determine which branch to optimize based on the type of action the expert is performing. We optimize the branches with Equation 2.2 as we did before. This way we have constructed a model that is conditioned on the current command, meaning we are able to maneuver the car by enabling the desired sub-policy.

It is worth mentioning that the authors tested their model both in the driving simulator CARLA, and on a small RC car. In their evaluation, the model reached an episodic accuracy of 88% in the *Town 1* environment, and 64% in *Town 2*. For comparison, the non-conditional version got 20% and 26% respectively. For the physical system, they measured the percentage of missed turns, and got 0% of missed turns on the branched version. A consequence of this architecture is that the planning algorithm is now off-loaded to a separate planning system. This may be beneficial, as the model is not trying to learn too many things at once. A disadvantage of this methods is that it requires a lot of data to become reliable for all sort of uncommon traffic setups and different environments and weather. Another disadvantage is the "blackbox" nature these types of end-to-end deep learning models. When something goes wrong, it is very hard to interpret the cause. Interpretability is an ongoing area of research, and there are similar end-to-end imitation learning methods for self-driving cars that have interpretability in mind, such as [MSS18].

2.3 Reinforcement Learning

2.3.1 Introduction

Reinforcement learning refers to a specific group of tasks that exhibit the following properties:

1. There exists an **agent** which is able to take actions in an **environment**.
2. When the agent acts it gets some feedback from the environment, referred to as **reward**.
3. The agent can observe the **state** of the environment in order to make decisions.

Let's consider the example of an autonomous vacuum cleaner:

The vacuum cleaner lives in a grid-based world, and has the ability to move in any of the cardinal directions – north, east, south, west – relative to its current position. Once the vacuum cleaner visits a dirty cell, that cell becomes clean and the vacuum cleaner receives a positive signal. In terms of the reinforcement learning concepts introduced above, we may say that the vacuum cleaner is the **agent**, and its set of possible **actions** are {*move north, move east, move south, move west*}. The **environment** is the room in which the agent was deployed, and the **state** of the environment represents which cells are dirty, which cells are clean, in addition to which cell the vacuum cleaner is located in. When the agent moves to a dirty cell, it gets a positive **reward** signal from the environment, signifying that the agent made a beneficial move. The rules which the agent follows to make decisions is the **policy** of the agent. Whenever a problem can be described by the properties mentioned above, we are talking about a reinforcement learning problem. We generally conceptualize this framework as a loop, as shown in Figure 2.2.

Processes that are formulated in terms of agents, environments, and rewards are more formally known as a *Markov Decision Process* (MDP). A Markov decision process is a *time discrete stochastic control* process that can be modelled by the 4-tuple $(\mathcal{S}, \mathcal{A}, P, r)$:

- $\mathcal{S}$ is the set of all possible states in the MDP.

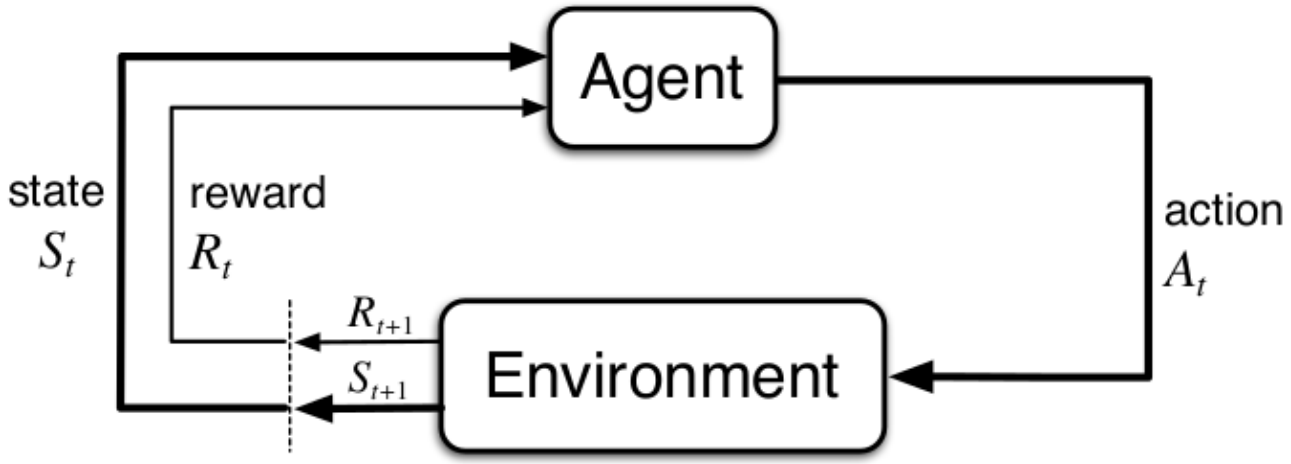


Figure 2.2: Reinforcement learning loop. Starting at time-step t, the agent observes the state s_t and reward r_t. Whenever an agent takes an action, a_t, the environment returns a new state, s_{t+1}, and a scalar reward value, r_{t+1}, representing the state and reward in time-step $t+1$.

- $\mathcal{A}$ is the set of all possible actions in the MDP. Also denoted $\mathcal{A}_s$, representing all the possible actions in state s.
- $P(s'|s,a)$ is the probability of transitioning to state $s' \in \mathcal{S}$ when taking action $a \in \mathcal{A}$ in state $s \in \mathcal{S}$.
- $r(s,a)$ is the immediate reward signal for taking action $a \in \mathcal{A}$ in state $s \in \mathcal{S}$.

When we say that a MDP is *time discrete*, it refers to the fact that we move step-wise through time; for example, an agent takes action a_0, the environment returns a reward r_1, the agent takes another action a_1, the environment returns another reward r_2, and so forth. Formally, we represent the current time-step as $t \in \mathbb{Z}$. In *episodic environments*, we typically start at an initial time-step and end once a terminal state has been reached. We call a run from $t = 0$ to $t = terminal$ an *episode*. In *continuous environments*, the process will never reach a terminal state, so the process will run until $t \to \infty$.

When we say that a MDP is a *stochastic* process, it means that environment can be modelled by transition probabilities. If we are in a state s and take action a, there is some probability that we will end up in state s'. Formally, this is modelled with a transition function, $P(s'|s,a)$.

A MDP is also a *control* process, because the goal in a MDP is to determine the optimal control *policy* for selecting actions. The policy represents the rules which the agent follows to determine

its action given a state, and is typically denoted as $\pi(a|s)$. The optimal policy is the policy that maximizes cumulative reward, also known as *return*. Mathematically speaking, to find the optimal policy, we need to find the policy that maximizes the following equation:

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \tag{2.3}$$

$R(\tau)$ is the return along *trajectory* τ. The trajectory $\tau = \{(s_0, a_0), (s_1, a_1), ...\}$ is the series of action-state pairs the agent took from $t = 0$ to $t \to \infty$. $R(\tau)$ is computed as the sum of the rewards $r(s_t, a_t)$ along the trajectory, that is, the sum of the rewards obtained by taking action a_t in state s_t for every state-action pair along the trajectory, multiplied with a discount factor $0 \leq \gamma < 1$. Since a_t is determined by policy π, it is possible to find a function for π that maximizes Equation 2.3. Solving this optimization problem will essentially leave us with an agent that maximizes total cumulative reward; in other words, an agent that follows an optimal policy. Discount factor γ is applied to future reward to makes sure that immediate rewards are prioritized over future rewards.

Reinforcement learning concern a subset of MDP problems where the state-transition probabilities $P(s'|s, a)$ and the reward function $r(s, a)$ are unknown to the agent. This means that the agent needs to explore the environment in order to find correlation between state-action pairs and reward. This brings us to the following reinforcement learning concepts:

Exploration vs. exploitation

This is the idea that an agent needs to both explore and exploit the environment to arrive at an optimal policy. Exploring entails taking either random or "non-optimal" actions in order to observe the consequences. Exploitation means taking the optimal action under the current policy. Exploration is essential during training, because the state transitions and rewards are unknown. When the agent has observed multiple state transitions and rewards, it can exploit the environment to come closer to a solution. We say that there is trade-off between exploring the environment and exploiting learned knowledge, and generally, we want to explore more at

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

Fully observable vs. partially observable environment

In a MDP, the environment can be fully observable or partially observable. When the environment is *fully observable*, the agent can observe the complete state of the environment at any point (the agent is "omnipresent.") An example of this type of environment is the game of chess. A chess playing agent will be able to observe location and types of all the chess pieces at any point in the game, so we say that the environment is fully observable. When the environment is *partially observable*, only a subset of the state is known. This is the most common case for real-life agents, such as an autonomous car. The car may only be able to observe the environment through a front-facing camera, meaning it cannot know or observe the states the of objects behind it. In other words, this environment is partially observable. A MDP that is partially observable is also known as a Partially Observable Markov Decision Process (POMDP).

Discrete vs. continuous state and action spaces

Depending on the environment, the state and action spaces may be discrete or continuous. 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 both continuous state and action spaces; the state and actions are real-numbered values, *e.g.* the state consists of the proximity values in meters, and the action consists of the vehicle's steering angle in degrees.

Model-based vs. model-free

Model-based methods are methods that attempt to create a model of how the environment works. If an agent is in state s and takes action a, it will get a new state s' and reward r.

Intuitively, these observations could be used to approximate $P(s'|s,a)$ and $r(s,a)$; that is, we can approximate a model of the environment. After a model has been approximated, we could use a MDP-solving algorithm such as Bellman's value iteration algorithm or Howard's policy iteration algorithm to derive the optimal policy given our current model of the environment.

Model-free methods are a methods that optimize the policy directly without modelling the environment. State spaces and action spaces in many reinforcement learning problems are continuous, and therefore we have an infinite number of possible values, rendering it impossible to efficiently create MDP models without making approximations. Optimizing the policy directly also allows us to find good policies through function optimization techniques, such as gradient decent.

Deterministic vs. stochastic policy

An agent's policy may be deterministic or stochastic. When we have a *deterministic policy*, the agent will always choose the same action when it is presented with the same state, assuming that the policy stays the same. The agent may alternatively follow a *stochastic* policy, where the agent will choose an action stochastically when it is presented with the same state. An example of a stochastic policy would be "given state s, go left 30% of the time and to go right 70% of the time." Deterministic policies make sense in fully observable, deterministic environments, where taking an action always lead to the same result (*e.g.* chess). Stochastic policies are useful in partially observable environments and in *stochastic environments*. Using a stochastic policy in a partially observable environment allows the agent to model some part of the hidden state of the environment as part of its stochasticity, making it more robust to hidden information. When an agent acts in a stochastic environment, taking an action may have different results from one time to another. This is common in robotics and other real-life control environments, where time measurements and control signals may be inaccurate. Using a stochastic policy in these cases is necessary, as a deterministic policy will be unable to find direct mappings from state-action pairs to their resulting states due to the stochasticity of the environment.

On-policy vs. off-policy

In on-policy methods, the same policy used to determine the value of the policy, and to control the agent. Proximal Policy Gradient is an example of an on-policy method (Section 2.3.3). Off-policy methods use different policies for evaluating the policy and for controlling the agent. Q-learning is an example of an off-policy method (Section 2.3.2).

Monte-Carlo vs. temporal difference

In methods using Monte-Carlo roll-outs, we compute a complete trajectory before we optimize, while in methods using n-step temporal difference, we only use n number of steps along a trajectory before we take an optimization step. Temporal difference (TD) methods have the advantage that they support non-episodic environments, *e.g.* a stock marked value predictor. TD-learning methods are, however, more susceptible to bias originating form the initialization of the agents parameters, while Monte-Carlo methods are less biased but have higher overall variance during training.

Reinforcement learning – Justification

Is this a reasonable way to model a learning framework? A common analogy to reinforcement learning is that of biological learning. When people and animals grow up, they gradually learn what actions are good and what actions are bad based on the positive and negative signals our brain receives from sensory organs. For example, if we were to touch a hot stove top with our hands, the body will send a big negative reward signal to the brain in the form of pain to deter us from repeating the same, dangerous action in the future. If we eat a calorie rich piece of cake, the body will send a positive reward signal to the brain, to teach us to repeat that action in the future. It stands to reason that the reinforcement learning framework is a logical way to formulate these types of problems following this analogy.

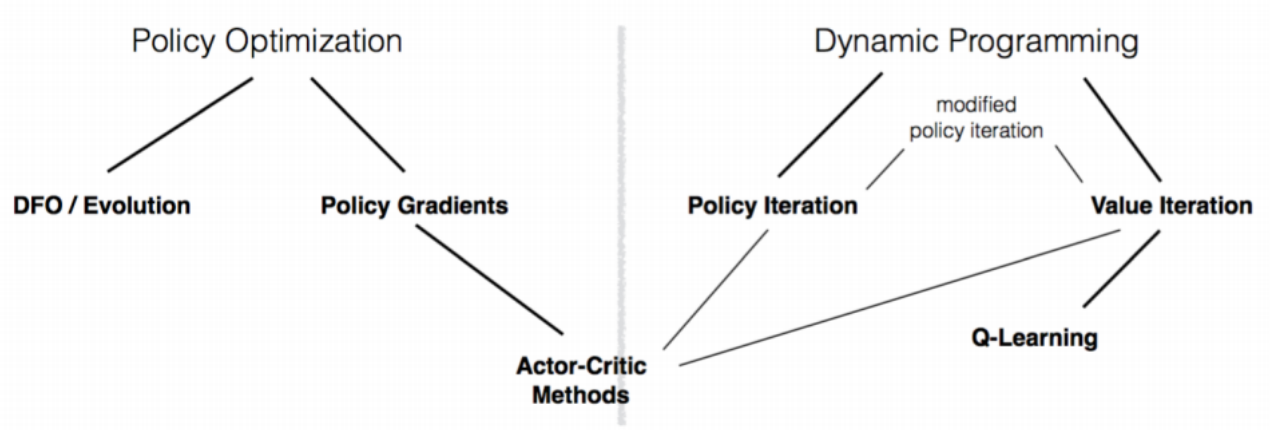


Figure 2.3: Classes of reinforcement learning algorithms. Illustration borrowed from presentation by Peter Abbeel of UC Berkeley.

2.3.2 Reinforcement Learning Algorithms

Figure 2.3 shows a hierarchy of classes of reinforcement learning algorithms, and how they relate. In this section, we will look into both policy optimization and dynamic programming algorithms, and later explain how these relate to the current state-of-the-art general-purpose reinforcement learning method, Proximal Policy Optimization.

Simplest Approach – Brute Force Search

One way to solve reinforcement learning problems is to search for the best policy through brute-force search. This involves evaluating several trajectories from a start to finish and picking the policy that yielded the highest return. This approach requires an evaluation of every possible trajectory to converge, and is therefore exponential in number of states, number of actions, and trajectory length. It may only work in episodic environments where only a few time steps is required to reach a terminal state, and environments with few and discrete states and actions; for example, tic-tac-toe.

Policy Gradient

A better approach would be to optimize the policy directly. Policy optimization methods search for a policy in a subset of the policy space. This can be done with both gradient-based optimization and gradient-free optimization. We will focus on gradient-based methods, however, gradient-free methods such as evolutionary computation have also shown to be successful as well.

Policy gradient optimization was introduced by Williams in 1992 [Wil92] in a paper titled *Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning*. In the paper, Williams derives the mathematical background of gradient-based policy optimization methods, and subsequently introduces a class of general-purpose algorithms called *REINFORCE* that uses gradient-following in connectionist networks – more commonly known today as artificial neural networks (Section 2.1.2) – to optimize the parameters of the policy. Intuitively, these methods work by collecting a bunch of trajectories, and then uses gradient ascent to make the good trajectories more probable in the policy.

Likelihood Policy Gradient

Assume that we have a neural network that parameterizes the stochastic policy $\pi_\theta(a|s)$ by parameters θ. We define the objective function as follows:

$$J(\theta) = \mathbb{E}\left[R(\tau)|\pi_\theta\right] \tag{2.4}$$

Where $\mathbb{E}[\,\cdot\,|\pi_\theta]$ denotes the expectation of $\cdot$ conditioned on π_θ and $R(\tau)$ is the return along trajectory τ (Equation 2.3.) The trajectory τ is conditioned on the policy π_θ, and the probability of following a specific trajectory under policy π_θ is given by $P(\tau|\pi_\theta)$. Using this, we can expand Equation 2.4 as follows:

$$J(\theta) = \mathbb{E}_{\tau \sim P(\tau|\pi_\theta)} \left[\sum_{t=0}^{T-1} \gamma^t r(s_t, a_t) \right] = \sum_{\tau} P(\tau|\pi_\theta) R(\tau) \tag{2.5}$$

Where T denotes the *horizon*, *e.g.* the number of time-steps we take in a fixed-time step environment. $J(\theta)$ represents the expected return of following policy π_θ. To find the optimal policy we have to find the policy that maximizes expected return, $J(\theta)$, with respect to θ. As such, we can formulate the problem as the following optimization problem on $J(\theta)$:

$$\max_{\theta} J(\theta) = \max_{\theta} \sum_{\tau} P(\tau|\pi_\theta) R(\tau) \tag{2.6}$$

We can solve this optimization problem with gradient ascent (Section 2.1.2), and to do so, we need to compute the gradient $\nabla_\theta J(\theta)$. We can derive this gradient with the help of the likelihood ratio trick:

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \sum_{\tau} P(\tau|\pi_\theta) R(\tau) \\
&= \sum_{\tau} \nabla_\theta P(\tau|\pi_\theta) R(\tau) \\
&= \sum_{\tau} \frac{P(\tau|\pi_\theta)}{P(\tau|\pi_\theta)} \nabla_\theta P(\tau|\pi_\theta) R(\tau) && \text{Multiply by identity } 1 = \frac{P(\tau|\pi_\theta)}{P(\tau|\pi_\theta)} \\
&= \sum_{\tau} P(\tau|\pi_\theta) \frac{\nabla_\theta P(\tau|\pi_\theta)}{P(\tau|\pi_\theta)} R(\tau) && \text{Likelihood ratio trick:} \\
&= \sum_{\tau} P(\tau|\pi_\theta) \nabla_\theta \log P(\tau|\pi_\theta) R(\tau) && \nabla_\theta \log P(\tau|\pi_\theta) = \frac{\nabla_\theta P(\tau|\pi_\theta)}{P(\tau|\pi_\theta)} \\
&= \mathbb{E}_{\tau \sim P(\tau|\pi_\theta)} \left[\nabla_\theta \log P(\tau|\pi_\theta) R(\tau) \right]
\end{aligned} \tag{2.7}$$

The reason we compute this gradient with the likelihood ratio trick, is because analytically calculating $\nabla_\theta P(\tau|\pi_\theta)$ is non-trivial – the dynamics model $P(\tau|\pi_\theta)$ is often a highly discontinuous function. The final step of Equation 2.7 shows that the gradient $\nabla_\theta J(\theta)$ is an expectation. A direct consequence of this is that we are now able to approximate the gradient empirically by

sampling m trajectories under policy π_θ:

$$\nabla_\theta J(\theta) \approx \hat{g} = \frac{1}{m} \sum_{i=0}^{m-1} \nabla_\theta \log P(\tau^{(i)}|\pi_\theta) R(\tau^{(i)}) \tag{2.8}$$

We know that $\frac{1}{m} \sum_{i=0}^{m-1} \nabla_\theta \log P(\tau^{(i)}|\pi_\theta) R(\tau^{(i)}) \to \nabla_\theta J(\theta)$ when $m \to \infty$ from the law of large numbers. This makes $\hat{g}$ is an *unbiased* estimator of $\nabla_\theta J(\theta)$. $\hat{g}$ is unbiased, even if $R(\tau)$ is discontinuous, unknown, or discrete.

However, we still need to be able to compute the gradient of the probability of a single trajectory, $\nabla_\theta \log P(\tau^{(i)}|\pi_\theta)$, in order to solve $\hat{g}$. We can do this by decomposing the trajectory into its state-action pairs as follows:

$$\begin{aligned}
\nabla_\theta \log P(\tau^{(i)}|\pi_\theta) &= \nabla_\theta \log \left(\prod_{t=0}^{T-1} \underbrace{P(s_{t+1}^{(i)}|s_t^{(i)}, a_t^{(i)})}_{\text{Dynamics model}} \cdot \underbrace{\pi_\theta(a_t^{(i)}|s_t^{(i)})}_{\text{Policy}} \right) \\
&= \nabla_\theta \left(\sum_{t=0}^{T-1} \log P(s_{t+1}^{(i)}|s_t^{(i)}, a_t^{(i)}) + \sum_{t=0}^{T-1} \log \pi_\theta(a_t^{(i)}|s_t^{(i)}) \right) \\
&= \nabla_\theta \sum_{t=0}^{T-1} \log P(s_{t+1}^{(i)}|s_t^{(i)}, a_t^{(i)}) + \nabla_\theta \sum_{t=0}^{T-1} \log \pi_\theta(a_t^{(i)}|s_t^{(i)}) \\
&= \sum_{t=0}^{T-1} \nabla_\theta \log P(s_{t+1}^{(i)}|s_t^{(i)}, a_t^{(i)}) + \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t^{(i)}|s_t^{(i)}) \qquad (2.9) \\
&= \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t^{(i)}|s_t^{(i)}) \qquad (2.10)
\end{aligned}$$

Note that $\nabla_\theta \log P(s_{t+1}^{(i)}|s_t^{(i)}, a_t^{(i)})$ disappears in Equation 2.9 because the dynamics model, $P(s_{t+1}^{(i)}|s_t^{(i)}, a_t^{(i)})$, is not parameterized by θ – policy gradient methods are model-free. So the gradient becomes zero, and we are left with a gradient that depends on the parameterized policy only. Inserting this derivation into the expectation of the objective gradient (Equation 2.8), we get the following model-free gradient estimator:

$$\hat{g} = \frac{1}{m} \sum_{i=0}^{m-1} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t^{(i)}|s_t^{(i)}) R(\tau^{(i)}) \tag{2.11}$$

Using this equation to optimize $J(\theta)$ with gradient ascent is know as *vanilla policy gradient*. Vanilla policy gradient is, unfortunately, very sample inefficient due to high variance; a single sample of $R(\tau)$ can have too great of an effect on the gradient. To reduce the variance, Williams introduces the idea of a *baseline* value, b. The goal of the baseline value is to center the reward signal around a zero-mean, meaning we will only update the parameters θ if the return along a trajectory $R(\tau)$ is better or worse than the baseline. This is the equation for the policy gradient with a constant baseline[1]:

$$\hat{g} = \frac{1}{m} \sum_{i=0}^{m-1} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t^{(i)}|s_t^{(i)}) (R(\tau^{(i)}) - b) \tag{2.12}$$

The baseline value can be estimated several ways, but the simplest way would be to calculate the expected return over multiple trajectories $b = \mathbb{E}[R(\cdot)] \approx \frac{1}{k} \sum_{i=0}^{k-1} R(\tau^{(i)})$. We will take a closer look at another baseline in Section 2.3.3.

It is also worth mentioning common representations of π_θ. In discrete action spaces, it is common to express π_θ as a *softmax* policy. The softmax function is $S_j = \frac{e^{o_j}}{\sum_{k=0}^{K-1} e^{o_k}}$, where j represents the j$^{\text{th}}$, and o_k is the k$^{\text{th}}$ output of a neural network parametarized by θ. Computing the derivative of the softmax policy yields:

$$\nabla_\theta \log \pi_\theta(a_j|s) = \sum_i a_i \frac{\partial S_i}{\partial o_j} = a_j - S_j \tag{2.13}$$

In other words, the difference between the softmax-probability of the taken action and the softmax-probability of the expected action.

There is also the *Gaussian* policy, commonly used with continuous action spaces. The Gaussian, or normal distribution is given by $\mathcal{N}(\mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$. Calculating the derivative of a

[1] Williams proves that the baseline estimator is also unbiased in [Wil92]

Gaussian policy with respect to the mean, μ gives us:

$$\nabla_\theta \log \pi_\theta(a|s)_{a\sim\mathcal{N}(\mu,\sigma)} = \frac{\partial}{\partial\mu} \log\left(\frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(a-\mu)^2}{2\sigma^2}}\right) = \frac{a-\mu(s)}{\sigma^2} \tag{2.14}$$

In other words, the gradient of the Gaussian policy is the difference between the taken action, a, and the mean action given state s, $\mu(s)$. If the network outputs the standard deviation σ alongside the mean, we need to calculate the derivative of the Gaussian distribution with respect to σ:

$$\nabla_\theta \log \pi_\theta(a|s)_{a\sim\mathcal{N}(\mu,\sigma)} = \frac{\partial}{\partial\sigma} \log\left(\frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(a-\mu)^2}{2\sigma^2}}\right) = \frac{(a-\mu(s))^2-\sigma^2}{\sigma^3} \tag{2.15}$$

Q-learning

Q-learning is a dynamic programming based approach to solve reinforcement learning problems. It works by maintaining a table of Q-values, or quality-values. The Q-table maps a state-action pair to a Q-value that represents the expected return of taking a specific action a in state s. This table is also denoted as $Q(s,a)$.

Q-learning is a 1-step temporal difference learning algorithm that works in both continuous and episodic environments, but works only with discrete state and action spaces. It is based on the Bellman equation, which goes as follows:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[r(s,a) + \gamma \max_{a'} Q(s',a') - Q(s,a)\right] \tag{2.16}$$

This is an equation that can be solved by dynamic programming; this, because we need the solution of $Q(s',a')$ in order to solve $Q(s,a)$. It is also an iterative algorithm that learns over time. $\max_{a'} Q(s',a')$ is the maximum Q-value of the next state, so $r(s,a) + \gamma \max_{a'} Q(s',a')$ represents the current estimate of the maximum return of taking the best action in the next state s'. $r(s,a) + \gamma \max_{a'} Q(s',a') - Q(s,a)$ represents how different our current estimate of

$Q(s, a)$ is, from how we think it should be given what we have learned about the next state and the reward we received this time-step. This is the *temporal difference* from state s to s'; also known as 1-step temporal difference. $0 < \alpha < 1$ is a scalar that determines how quickly the values in the Q-table should be updated based on the temporal difference – in other words, the learning rate. $0 < \gamma < 1$ is the discount factor, which works as described in Section 2.3.1. When this equation is applied over several iterations, the Q-values in the Q-table will converge to values that accurately predicts the return from taking an action in a given state.

Algorithm 1 Q-learning algorithm

```
Initialize Q(s, a) arbitrarily
for each episode do
    Initialize s
    for each step of episode do
        Choose a from s using policy derived from Q (e.g. ϵ-greedy)
        Take action a, observe r, s'
        Update Q(s, a) ← Q(s, a) + α [r(s, a) + γ max_a' Q(s', a') − Q(s, a)]
        s ← s'
    end for
end for
```

Algorithm 1 shows the complete Q-learning algorithm. A critical part of this algorithm is the exploration vs. exploitation trade-off introduced by an ϵ-greedy policy. An ϵ-greedy policy is a policy that will pick the optimal, or "greedy," action with a probability of $1 - \epsilon$, and a random action with a probability of ϵ, where $0 \leq \epsilon \leq 1$. Typically, we want to initialize $\epsilon = 1$ and anneal its value towards $\epsilon \to 0$ over the run of episodes. This ensures that our agent will explore the environment sufficiently before converging to a final policy. Because Q-learning uses an ϵ-greedy policy for selecting actions, while it uses a $\max_{a'} Q(s', a')$ policy to determine the value of the current policy, Q-learning is an off-policy reinforcement learning algorithm.

2.3.3 Deep Reinforcement Learning

As deep learning became more wide-spread, researchers started looking into ways to utilize deep learning in reinforcement learning. The first successful attempt of this is known as *Deep Q-learning*.

Deep Q-learning

Deep Q-learning, or Deep Q-network (DQN), was introduced in the paper *Playing Atari with Deep Reinforcement Learning* by Mnih *et al.* 2013 at DeepMind Technologies [MKS+13], and it is the earliest example of a deep reinforcement learning model that successfully learned control policies from high-dimensional state spaces. The approach has shown to be successful in playing a wide range of Atari games using only the input frames and a reward signal to train. DQN achieved scores that were on par with human performance, and it even got better scores than human players in three of the games that they tested.

Deep Q-learning works by training a convolutional network to accurately *predict* the Q-values of every action, given a state. That is, instead of maintaining a $|\mathcal{A}| \times |\mathcal{S}|$ table, we now approximate the table with a deep neural network. Intuitively, we can say that the network learns to predict the quality of actions by correlating states, actions and rewards pairs from its experiences.

Algorithm 2 Deep Q-learning with experience replay

Initialize replay memory $\mathcal{D}$ with capacity $\mathcal{N}$
Initialize action-value function Q with random weights
for episode = 1, M **do**
    Initialize sequence $s_1 = x_1$ and prepossessed sequence $\phi_1 = \phi(s_1)$
    for $t = 1, T$ **do**
        With probability ϵ select a random action a_t
        otherwise select $a_t = max_a Q^*(\phi(s_t), a; \theta)$
        Execute action a_t in emulator and observe reward r_t and image x_{t+1}
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1}$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathbb{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathbb{D}$
        Set

$$y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$$

        Perform gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
    end for
end for

Algorithm 2 shares several similarities to the original Q-learning algorithm. We are still working with an ϵ-greedy policy – to encourage exploration – and we are still calculating 1-step temporal differences. We also require discrete action-spaces, however the state-space can now

be continuous or high-dimensional. The method scales linearly with the state-space, as opposed to Q-learning, which scales exponentially.

The first difference to note is the addition of *replay memory*. Replay memory, $\mathcal{D}$, is a *deque* of predetermined size $\mathcal{N}$ that stores previous experiences of the agent. An experience is a 4-tuple of $(\phi_t, a_t, r_t, \phi_{t+1})$, where ϕ_t is the original state before action a_t is taken, r_t is the reward the agent received from taking the action and ϕ_{t+1} is the next state. Note that $\phi_t = \phi(s_t)$, where ϕ is some function used to preprocess the observation. Adding experiences to the replay buffer as we explore ensures that the agent reinforces both new and past experiences, so it does not forget as easily. It also allows us to do minibatch training to reduce the amount correlations being found by the model, and to speed up training.

However, the most important difference is how the agent is trained in DQN. Instead of iteratively maintaining a complete Q-table, we are now using stochastic gradient descent to optimize the network parameters, θ, with respect to the squared error between y_j and $Q(\phi_j, a_j; \theta)$, where j represents the randomly sampled indices for the minibatches. Following the same logic as with Q-learning, we can say that y_j represents the expected value of ϕ_t, given that we take the action that we currently believe is the best action in ϕ_{t+1}, while $Q(\phi_j, a_j; \theta)$ represents the current prediction of the value of the current state according to the network. In other words, $y_j - Q(\phi_j, a_j; \theta)$ tells us how incorrect our current estimate of the current state's value is, taking the current reward and the value of the next step into consideration. Using this as the neural network's loss function, we will eventually learn to accurately predict the values for each action given any state, leaving us with a policy where we can simply pick actions greedily to solve problems with high-dimensional state-spaces, such as Atari games.

In their experiments, they found it to be necessary to do prepossessing of the input images, denoted as $\phi(s)$ in Algorithm 2. In their case, they apply a prepossessing step that transforms the 210×160 128-bits images into 84×84 greyscale images, to reduce the computation load of backpropagation. More importantly, their preprocessing step also stacks the 4 last frames, turning the state-space into $84 \times 84 \times 4$. The authors found this modification to be crucial, as the network could not learn to predict motion otherwise. If you can only see a single image of

the game of *Pong*, it is impossible to predict what the best action is, because a single frame does not encode the motion of the ball.

It is also worth noting that there exists several improved adaptations and alterations of DQN, such as: DQN with Fixed Q-targets, Double DQN [Has10], Dueling DQN [WdFL15], DQN with Prioritized Experience Replay [SQAS15], among others.

Asynchronous Advantage Actor Critic

Deep Q-learning has a big limitation: it requires discrete action spaces. Reinforcement learning with discrete action spaces is useful for playing games – where the actions comes in the form of discrete button presses – but it can be difficult to adapt DQN to, say, control a robotic hand, or in our case, controlling a vehicle with continuous acceleration, breaking and steering angle variables.

Asynchronous advantage actor citric (A3C) by Mnih 2016 *et al.* [MBM+16] at DeepMind represents one of the more recent successes in deep reinforcement learning. A3C is an *actor-critic* method – a reinforcement learning formulation that was first introduced by Barto *et al.* in 1983 [BSA83]. The *actor* represents the parameters that determines the policy of an agent while the *critic* represents the parameters that are responsible for predicting the value of being in any given state. The "critic" part of A3C does essentially the same as DQN, however, unlike DQN we do not pick our actions based on the value function. Instead, the critic helps our agent make more accurate advantage estimates, which will in turn guide the training of the policy parameters. Optimizing the actor separately from the critic allows the actor to explore the environment independently of the critic, and it also allows for policies that output continuous action values since we can optimize the policy directly, as we did in Section 2.3.2 regarding policy gradient methods.

When compared to earlier deep learning-based actor-critic methods such as Deep Deterministic Policy Gradient [LHP+15], Mnih *et al.* credits most of the success of A3C to the "asynchronous" part of the algorithm. Instead of maintaining a replay memory of previous experiences, A3C

works by running multiple environments simultaneously. Each agent has its own copy of the network parameters, and they calculate their respective gradients over several time-steps asynchronously. These gradients are periodically applied to a global copy of the network parameters, and synchronized across the parallel agents. The benefit of training with parallel agents rather than replay memory is that all the samples we train on will use a recent version of the policy, meaning we are more likely to train samples that are probable under our current policy, thus improving the speed of training. *A2C* is the synchronous variant of A3C, and OpenAI has shown that it has equal to or better convergence properties than A3C [Ope17]. In A2C we run the environments synchronously to gather the samples we use in a minibatch. An advantage of this is that we can utilize the GPU for performing batched updates with large batch sizes. Most things that we will be referring to regarding A3C will also hold for A2C.

A3C is an *advantage actor-critic* method. In practice, this means that the policy gradient, $\nabla_\theta \log \pi_\theta(a_t|s_t)$ (see Section 2.3.2,) is weighed by the actors *advantage* rather than its return. We can formulate this as the following policy loss function:

$$L_\pi = -\mathbb{E}_t \left[\log \pi_\theta(a_t|s_t) A(\tau_t; \theta_v)\right] \tag{2.17}$$

Where θ are the actor parameters and θ_v are the critic parameters (in practice these may be shared,) and $\mathbb{E}_t\left[\,\cdot\,\right]$ denotes the expectation over all time-steps $t \in [0, \ldots, T]$. Note that the policy gradient from Section 2.3.2, $\log \pi(a_t|s_t;\theta) A(\tau_t;\theta_v)$, is negated in L_π because we want to express it as a *loss function* instead of an objective function (maximizing an objective is equivalent to minimizing the negated value of the objective.) $A(\tau_t;\theta_v)$ is the measured advantage of the agent along trajectory τ, and it is given by:

$$A(\tau_t;\theta_v) = \overbrace{\sum_{i=0}^{k-1}\left(\gamma^i r_{t+i}\right)}^{R_t(\tau)} + \overbrace{\gamma^k V(s_{t+k};\theta_v)}^{\text{bootstrap}} - \overbrace{V(s_t;\theta_v)}^{\text{baseline}} = R_t(\tau) - V(s_t;\theta_v) \tag{2.18}$$

Conceptually, advantage represents how much better the agent did compared to a *baseline* expectation, as discussed in Section 2.3.2. There are several ways to calculate this advantage,

and in A3C, the advantage is calculated by how much better the agent did compared to what the critic expected, as we can see in Equation 2.18. $R_t(\tau)$ is the *return* of the agent when following trajectory τ, and $V(s_t;\theta_v)$ is the critic's prediction, or the baseline. Note that for the return value of a trajectory to be accurate, we need to consider the trajectory as $t \to \infty$. However, because we are doing TD-learning, we instead bootstrap the last reward value with a discounted prediction of the future return given the last state, $\gamma^k V(s_{t+k};\theta_v)$. This is reasonable, as $V(s_t;\theta_v) \approx R_t(\tau)$. The purpose of training with advantage rather than return, is that it will normalize the reward signal during training. This reduces the variance and stabilizes the training process, as the gradient descent steps are now zero-mean [Wil92].

In addition to optimizing the policy, gradient updates are also applied to the critic parameters, θ_v, by minimizing the mean-squared error between the observed n-step return and estimated value over all the time-steps: $L_V = \mathbb{E}_t\left[\left(V(s_t;\theta_v) - R_t(\tau)\right)^2\right]$. They also introduce an entropy loss term, L_S, to encourage exploration. The entropy of a normal distribution is given by $-\frac{1}{2}\left(\log\left(2\pi\sigma^2\right)+1\right)$. Note that that the entropy should be maximized, so the equivalent loss function becomes negated, $L_S = \frac{1}{2}\left(\log\left(2\pi\sigma^2\right)+1\right)$. This gives us the following, combined loss function for optimizing a A3C-based network that outputs normal variables:

$$L = L_\pi + \alpha L_V + \beta L_S =$$
$$\mathbb{E}_t[-\log\pi(a_t|s_t;\theta)A(\tau_t;\theta_v) + \alpha\left(V(s_t;\theta_v) - R_t(\tau)\right)^2 + \beta\,\frac{1}{2}\left(\log\left(2\pi\sigma_t^2\right)+1\right)] \tag{2.19}$$

Where α and β are value and entropy loss scaling factors respectively. Applying stochastic gradient decent to this loss function will make the policy parameters move in the direction of policies of higher advantage, minimize the value loss to make the critic more accurate, and also maximize the entropy to maintain some level of uncertainty in the policy (maintain a level of σ in the case of a Gaussian policy.)

Proximal Policy Optimization and Trust Region Policy Optimization

Proximal Policy Optimization (PPO) by Schulman 2017 *et al.* [SWD+17] at OpenAI is currently considered the best baseline for reinforcement learning research. It has much better convergence properties than previous reinforcement learning approaches, due to a clever combination of clipping the policy loss, and calculating the loss in terms of a probability ratio instead of optimizing the policy's log likelihood directly. PPO is an actor-critic method similar to A3C, that combines *trust region optimization* with gradient decent to stabilize training; creating a loss function that nearly guarantees that the agent's policy will improve monotonically. PPO attempts to improve on *Trust Region Policy Optimization* [SLM+15] (TRPO) by Schulman *et al.*, which formulates an objective function that constraints the update step within some pessimistic lower-bound called a *trust region*.

Trust region optimization methods are optimization methods that optimize a function by computing a local, but accurate estimate of a function at a specific point, and derives a trust region from the upper-bound error of the objective function. In TPRO, this error is derived by the Kullback-Leibler divergence, or KL-divergence. KL-divergence is a measure of how much one probability distribution differs from another. The intuition is that by constraining the optimization subject to the KL-divergence between the old and the current policy, we are ensuring that the new policy is not diverging too far from the original; in other words our new policy will be within the old policy's trust region. This constraint allow us to perform multiple update steps per sample, because we know the new policy will not diverge too far from the old one in any one step, increasing the sample efficiency of our method significantly. In order to measure how much our policies are diverging, we need to express the optimization problem in terms of the current policy and the old policy. This is the basis of the TRPO objective:

$$\begin{aligned} \underset{\theta}{\text{maximize}} \quad & \mathbb{E}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t \right] \\ \text{subject to} \quad & \mathbb{E}_t \left[KL \left[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t) \right] \right] \leq \delta \end{aligned} \tag{2.20}$$

Where θ_{old} is the policy parameters before the update. TRPO, however, suffers from being

complicated to implement, and incompatible with models that have noise (such as dropout), or models that share parameters between the policy and value function. This is due the fact that Equation 2.20 needs to be optimized with second-order optimization methods such as the conjugate gradient algorithm, rather than first-order optimization techniques such as gradient decent. PPO aims to correct these shortcomings by reformulating the objective as a clipped objective function that we can optimize with gradient decent. Let's, however, start by reformulating the above objective as an unconstrained loss function:

$$L^{IS}_{\theta_{old}}(\theta) = \mathbb{E}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t \right] \tag{2.21}$$

Where IS stands for importance sampling. This comes from the fact that this loss function can be interpreted in terms of importance sampling [SLM+15]. Recall from Section 2.3.2, that in order to directly optimize the policy of an agent by first-order optimization, we need to calculate $\nabla_\theta \log \pi_\theta(a|s)$. The loss function $L^{IS}_{\theta_{old}}(\theta)$ is, however, a loss function expressed by the ratio between the old an new policy. We can prove, by the help of the chain rule, that these gradients are actually the same [KL02]:

$$\nabla_\theta \log \pi_\theta(a|s)|_{\theta_{old}} = \frac{\nabla_\theta \pi_\theta(a|s)|_{\theta_{old}}}{\pi_{\theta_{old}}(a|s)} = \nabla_\theta \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)} \right) |_{\theta_{old}} \tag{2.22}$$

This means that optimizing $L^{IS}_{\theta_{old}}(\theta)$ with gradient decent is equivalent to optimizing the policy gradient, $\nabla_\theta \log \pi_\theta(a|s)$. The importance of this reformulation of the policy gradient is that we can now use gradient decent, while imposing a trust region constraint on the loss function in terms of the new and old policy. Let $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$. The authors propose the following clipped loss function:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta)\hat{A}_t, clip\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t \right) \right] \tag{2.23}$$

Where ϵ is a hyperparameter that determines how much the new policy can diverge, per update, from the old policy in the direction of improved policies; in other words, the size of the trust

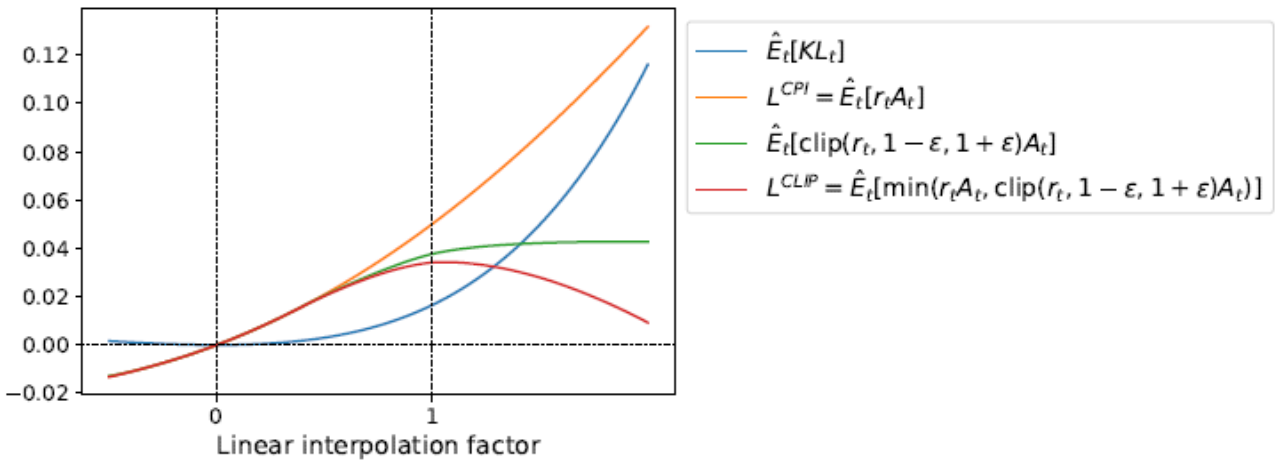


Figure 2.4: Shows the response of the different loss functions as policy θ is linearly interpolated to θ_{old}. Notice how $L^{CLIP} \to 0$ the more θ deviates from θ_{old}.

region (see Figure 2.4.) The min in the objective computes minimum of the unclipped and clipped objective. The unclipped objective is the regular $L^{IS}_{\theta_{old}}$ loss, and the clipped objective clips the probability ratio r_t to the interval $[1-\epsilon, 1+\epsilon]$ to ensure conservative changes. The min term makes it so that whenever the new policy is advantageous, that is, if $A > 0$ && $r_t(\theta) > 1$ or $A < 0$ && $r_t(\theta) < 1$, we constraint our updates by the clipped objective. Otherwise, if the new policy is detrimental, that is, $A < 0$ && $r_t(\theta) > 1$ or $A > 0$ && $r_t(\theta) < 1$, we will push the current policy parameters towards the parameters of the previous policy, essentially reverting changes of the previous update. The authors of PPO also tried to use a KL-divergence penalty with the loss function instead, but found the clipped loss to give overall higher return in their experiments.

As mentioned before, formulating the optimization problem in terms of a differentiable loss function allows us to use gradient decent. This means that, unlike TRPO, we can now parameterize a critic in terms of θ. PPO optimizes the critic the same way A3C does, by introducing a value function loss $L^{VF} = (V(s_t; \theta_v) - R_t(\tau))^2$ (see Section 2.3.3). We also add the entropy term, $-\frac{1}{2}(\log(2\pi\sigma^2) + 1)$, like in A3C. The final loss function becomes:

$$L^{CLIP+VF+S}(\theta) = -\hat{\mathbb{E}}_t \left[L^{CLIP}(\theta) - \alpha L^{VF}(\theta) - \beta \frac{1}{2} \left(\log\left(2\pi\sigma^2\right) + 1 \right) \right] \tag{2.24}$$

Algorithm 3 PPO, Actor-Critic Style

1: **for** iteration=1,2,... **do**
2: **for** actor=1,2,...,N **do**
3: Run policy $\pi_{\theta_{old}}$ in environment for T timesteps
4: Compute advantage estimates $\hat{A}_1, ..., \hat{A}_T$
5: **end for**
6: Optimize L^{CLIP} wrt θ, with K epochs and minibatch size $M \leq NT$
7: $\theta_{old} \leftarrow \theta$
8: **end for**

Algorithm 3 is the algorithm for PPO. This algorithm is almost identical to the one of A2C; the standard policy gradient loss function is replaced with the clipped loss function, there is a loop repeating the gradient update on random minibatch samples over K epochs, and advantage estimate $\hat{A}_t$ uses the more accurate *generalized advantage estimation* (GAE) calculation instead:

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + (\gamma\lambda)^{T-t+1}\delta_{T-1}, \tag{2.25}$$
$$\text{where} \quad \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

Note that for A2C and earlier policy gradient based methods, it is not well-justified to run multiple batches on data sampled under the same policy. This is because the advantage estimate $\hat{A}_t$ is a noisy function, so running multiple batches based on a single sample of the advantage function is going to drive the likelihood of the respective action to infinity when $\hat{A} > 0$, or to zero when $\hat{A} < 0$.

2.3.4 Reinforcement Learning for Self-Driving Vehicles

Most research in reinforcement learning we have looked at so far have primarily focused on solving video games and robotics' locomotion problems. So far there has been quite limited research in use of deep reinforcement learning for autonomous driving, perhaps due to the fact that it is hard to safely train a reinforcement agent in the real world, since such an agent needs to explore the environment to learn. There is, however, recent notable work done by Kendall

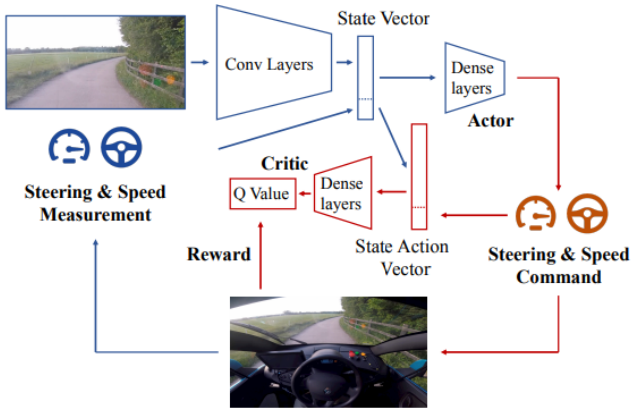


Figure 2.5: Kendall's *et al.* *Learning to Drive in a Day* model. This is an actor-critic based reinforcement learning model that learns to output steering and speed given a monocular input image, and given the vehicle's current steering and speed measurements.

et al. in their 2018 paper titled *Learning to Drive in a Day* [KHJ+18]. In their work they were able to teach a full-scale autonomous car to reliably follow a country-side road with only a single monocular front-facing camera, and only needing to train it over a handful of episodes.

Applying a deep reinforcement learning model to train self-driving vehicles is not trivial. As we discussed in Section 2.3.1, reinforcement learning solves Markov decision processes. This entails that the task can be modeled as a process where we have an agent that takes actions in the environment, and receives rewards form its actions. In practice, this means that we need to design a reward signal that is sufficient to teach the agent to solve the problem at hand. In the case of lane-following, we can imagine a reward signal to be a function of the car's offset from the center of the lane. However, this approach is limited in scale according to Kendall, as it may be difficult to extract the center point of the lane with high accuracy on a multitude of roads. Instead, Kendall *et al.* define the reward as the forward speed of the car, and they will terminate the episode whenever the car drifting too far to the sides of road (termination signal is given by a human safety driver.) Defining the reward this way encourages the car to cover as much distance as possible, because $velocity * time = distance$, and terminating early means a reduction in *time*. Given this reward formulation, they apply the actor-critic method DDPG [LHP+15] out-of-the-box with no further task-specific modifications made.

Figure 2.5 shows the model they used. The action space, or output, of the model is a continuous two-dimensional vector representing the steering angle, and speed in km/h. It is interesting to

note that outputting the angle and speed directly, and letting the controller manage turning and throttle, will most likely smooth out noise coming from the network. In their experiments, they have tried to use convolutional layers as part of the actor-critic network, in addition to trying to use a pre-trained variational autoencoder (VAE) to encode image information. They found using a variational autoencoder to encode the images to vastly improve the performance of the model, compared to training the convolutional layers alongside the rest of the actor-critic parameters. A simulator was adopted to tune the hyperparameters and to verify that their model works. The final model was able to learn to follow a 250m road in 11 epochs, or 15 minutes in real-time.

2.3.5 Reinforcement Learning with Variational Autoencoders

Apart from Kendall's work, we have also seen other examples of variational autoencoders being used with reinforcement learning, suggesting that VAEs will have an central place in the field of deep reinforcement learning. Other examples include *Disentangled Representation Learning Agent* by Higgins *et al.* [HPR+17], and *World Models* by Ha *et al.* [HS18]. In this section we will take a look at what autoencoders are, and how they are useful to reinforcement learning agents.

Variational Autoencoder

An 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. Mathematically, we may denote the input vector as $x \in X$ and the reconstructed signal as $\hat{x} = p(q(x))$, where q and p denote the encoder and decoder respectively. The encoder and decoder are non-linear functions that can be optimized by backpropagation, typically multi-layered perceptrons or CNNs. We will denote the the latent vector at the bottleneck as $z = q(x)$, and it has a size of z_{dim}. The goal of the autoencoder is to minimize the

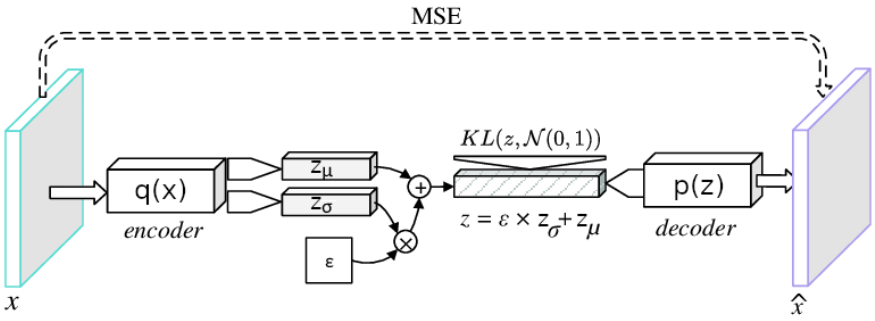


Figure 2.6: Shows the variational autoencoder architecture. The encoder, q, takes input vector x and outputs two vectors, z_μ and z_σ, through two parallel fully-connected layers. z is then sampled from $\mathcal{N}(z_\mu, z_\sigma)$ and passed through decoder p, producing reconstructed signal $\hat{x}$. Figure adapted from [PC18].

reconstruction loss, L_{rec}, for all x and $\hat{x}$ in our dataset X. L_{rec} is the loss function, and is we typically use mean squared error or binary cross-entropy. After training the VAE over a large dataset, we should end up with a model that first encodes (or compresses) input data into its most essential components, and then attempts to reconstruct the original signal by decoding the compressed representation.

In the paper *Auto-Encoding Variational Bayes*, Kingma *et al.* [KW14] describes an alternative autoencoder architecture they named a *variational autoencoder*. A variational autoencoder is similar to a regular autoencoder, however, a variational autoencoder has an added Gaussian sampling step in the bottleneck to make the model more robust to noise in the input and in the encoded representation. Instead of producing a single latent vector z at the bottleneck, we now produce two latent vectors, z_μ and z_σ, representing the mean and standard deviations of a multivariate Gaussian distribution respectively. We use this multivariate Gaussian distribution to sample a z_{dim}-sized vector that is then feed to the decoder, as is illustrated in Figure 2.6. During training we try to minimize the reconstruction loss, L_{rec}, just like in a regular autoencoder.

Kingma *et al.* also introduces a KL-divergence loss that serves to limit the divergence of the Gaussian distributions. As mentioned in Section 2.3.3, Kullbak-Leibler divergence is a measure of the divergence between probability distributions. In our case, we want to minimize

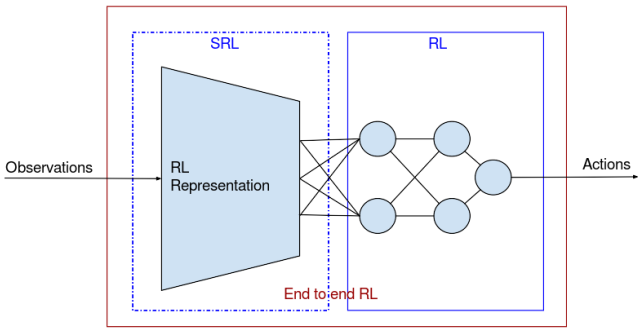


Figure 2.7: Shows a visualization of how state representation learning (SRL) can be used with reinforcement learning (RL). Note that the SRL module is typically pre-trained and frozen before we apply end-to-end reinforcement learning to the parameters of the RL module. Figure borrowed from [RHT+19].

the distance between the distributions given by $\mathcal{N}(z_\mu, z_\sigma)$ and $\mathcal{N}(0, 1)$, effectively pushing our Gaussian distributions towards zero-mean normal distributions. This forces the model to diverge as little as possible from a simple $\mathcal{N}(0, 1)$ distribution; making the model simpler and more robust. The KL-loss is given by:

$$L_{KL} = -\frac{1}{2} \sum_{i=0}^{z_{dim}-1} 1 + \log(z_{\sigma,i})^2 - z_{\mu,i}^2 - z_{\sigma,i}^2 \tag{2.26}$$

Where we have summed the KL-loss over z_{dim} different distributions. Adding the reconstruction loss, L_{rec}, to L_{KL} gives us a final loss of:

$$L = L_{rec} + \beta L_{KL} \tag{2.27}$$

Where β is a factor regulating the strength of the KL-divergence loss (set to 1 in the original VAE paper.)

In Reinforcement Learning

Variational autoencoders are used in reinforcement learning as a type of feature extractor. The idea is that 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. Learning on the

state representations produced by a VAE is known as *state representation learning*. Raffin *et al.* [RHT+19] argue that state representation learning improves the quality of an agent's learning by giving the agent a state space that ignores distractors, and is disentangled in its feature representations. Disentanglement refers to the idea that each latent space variable encodes some essential, uncorrelated variable in the system, *e.g.* the x or y-positions of the agent. The more disentangled the state representations are, the easier it should be for the agent to solve the environment.

Typically, the VAE is trained separately from the policy, such as in *DARLA: Improving Zero-Shot Transfer in Reinforcement Learning* by Higgins *et al.*. Higgins also experimented with the effect of changing β to encourage increased disentanglement in the state representation. Increasing β forces the VAE to find simpler models to explain the data, hence it is reasonable to expect stronger disentanglement.

World Models by Ha *et al.* [HS18] is another example of how VAEs can be used to aid the training of an agent. In their model, they add the output a recurrent network of LSTMs to the state representation to give the agent memory. The LSTM tries to predict the next state that will be output by the VAE after some action is taken, and will essentially give us a state representation model that is able to predict changes in the environment based on the agent's actions. In their results, they also note that their agent is the only entry on the CarRacing-v0 leaderboard that has effectively solved the task; achieving an average score of 906 over 100 randomly generated tracks.

Chapter 3

Proximal Policy Optimization in Driving-Like Environments

3.1 Introduction

As we discussed in Section 2.3.5, there is evidence that better state representations can vastly reduce training time and improve the quality of our agent. However, in this thesis, we wish to investigate all aspects of how we may setup a reinforcement learning problem to accelerate the learning of autonomous vehicles. In particular, we will be looking at the results and methods of Kendall *et al.* [KHJ+18], and we will try to identify which of their design decisions had the greatest impact when it comes to using deep reinforcement learning for autonomous vehicles. We will compare different variational autoencoder models, different reward functions, the effect of early termination in the environment, the effect of environment's visual complexity and task complexity. The ultimate goal of these experiments will be to demonstrate that deep reinforcement learning can be used to drive in visually complex and realistic settings, and we will provide some analysis of the effects of the various design decisions we make in our environments and agent models along the way.

This chapter is divided into four sections. The first section concerns the implementation details of the Proximal Policy Optimization implementation we used for most of the experiments. The

following sections are concerned with the three different environments we conducted experiments in: (1) a modified version of CarRacing-v0, and (2) a CARLA based environment that focuses on following the road (3) a CARLA based environment where we try to follow a route to reach a goal. We will discuss environment design decisions and experiments later in this chapter, and we will provide the results of our experiments in Chapter 4.

3.2 Implementation Details

For our experiments, we have decided to use Proximal Policy Optimization, as it appears to be one of the most consistent deep reinforcement learning algorithms for continuous control tasks [SWD+17] [BESK18]. As discussed in Section 2.3.3, PPO is a model-free, policy gradient based reinforcement learning algorithm that employs a first-order trust region criteria to prevent divergence. In this section, we will lay out the details of our implementation of PPO.

3.2.1 Setup

The implementation was written in Python 3.6 with TensorFlow 1.13. The simulations and training were run on a system with a single Nvidia GTX 970 with 4 GB of video memory, a 4-core CPU, and 23GB of RAM. Complete code of the PPO implementation, the experiments, and the new environments can be found at *https://github.com/bitsauce/RoadFollowing-ppo*.

3.2.2 Algorithm

Algorithm 3 shows the general outline of the PPO algorithm. Recall that PPO works by optimizing current policy π_θ with respect to its deviation from the previous policy $\pi_{\theta_{old}}$. The training data for an optimization step is sampled by running a single or multiple environments in parallel with the old policy $\pi_{\theta_{old}}$. As we are trying to emulate Kendall's result, we opted to optimize the algorithm to run with a single environment. For each optimization step, we simulate a T-step trajectory, and store $(s_t, a_t, r_t, d_t, V(s_t; \theta_v))$-tuples for each state-transition.

d_t is a variable that represents whether or not this state is a terminal state, and it is only used when we calculate the advantage estimates. Note that s_t corresponds to the latent space vector z produced by the encoder of a variational autoencoder when a VAE is used. After we have obtained a trajectory, we calculate advantage estimates with the generalized advantage estimation equation (Equation 2.25), where λ is an interpolation factor that serves as a trade-off between bias and variance in the advantage estimates [SML+15], and γ is the reward discount factor. The GAE-calculation is also bootstrapped with the critic's value prediction of the last state in the trajectory, as shown in Equation 2.18. 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 minibatches of $(s_{n,t}, a_{n,t}, R_{n,t}, \hat{A}_{n,t})$-tuples are feed 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 (Equation 2.24.) 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. In the following sections, we will go into further details on the exact PPO network architectures that were used in the experiments.

3.2.3 Actor and Critic Architecture

As discussed in Section 2.3.3, PPO needs an actor $\pi(a_t|s_t;\theta)$ network and a critic $V(s_t;\theta_v)$ network. In our implementation, we implement these as a two distinct multi-layer perceptrons. We make use MLPs here, since the input, s_t, is a vector (see Equation 3.3.) The actor MLP consists of three fully-connected layers of sizes 500, 300, and a_{dim}, where a_{dim} is the size of the action space. The activation function is ReLU for the first two layers, and the final layer uses no activation. 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$^{\text{th}}$ action. To get the scaled means, we must first point out that each of the agent's actions are limited to a predetermined range of valid values in all our environments. As a result, it reasonable to scale the output of the MLP to the range of

each actions respective range. We do this through the following transformation:

$$\mu_i = a_i^{min} + \frac{\tanh(o_i) + 1}{2} * (a_i^{max} - a_i^{min}) \tag{3.1}$$

By passing the raw outputs of the last fully-connected layer through the hyperbolic tangent function (tanh) we end up with values in the range $[-1, 1]$. Adding 1 and dividing by 2 puts our values in $[0, 1]$ range. Finally, we do a linear interpolation between the i^{th} action min and max value, resulting in $a_i^{min} \leq \mu_i \leq a_i^{max}$. Note that this transformation is absent from the PPO paper [SWD+17], however, we have previously shown (Appendix A.1) that this modification gives substantial improvements – a claim that is also supported by [RHT+18]. To define the multivariate Gaussian distribution that we sample actions from, we also provide a trainable parameter σ_i. In A3C, σ_i is predicted alongside μ_i by a fully-connected layer that is parallel to the μ_i layer; making σ_i a function of s_t. However, we found this to produce erratic agent behaviour, and opted to have a trainable variable for each action's standard deviation instead. Note that these trainable variables, which we will call $\sigma^{(\log)}$, actually represent the logarithm of σ. So to retrieve σ we do $e^{\sigma^{(\log)}} = e^{\log(\sigma)} = \sigma$. This is done to ensure that the standard deviation is never negative, and that the standard deviation will increase faster when more exploration is necessary. Each σ_i is also 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 [GB10] with a scaling factor of 0.1. This is done to lower the chances of having the initial weights influence the policy too much. Our environments use continuous action spaces, so to pick actions during training we simply sample the multivariate Gaussian distribution given by $a_i \sim \mathcal{N}(\mu_i, \sigma_i)$, while, in evaluation mode, we simply pick $a_i = \mu_i$.

The critic is a simple MLP with 3 fully-connected layers of sizes 500, 300, and 1, where the output will represent $V(s_t; \theta_v) \approx R(s_t)$. We use ReLU for the two first layers, and no activation for the final layer. Having no activation in the last layer makes it so that the critic is able to represent any possible value of $R(s_t)$.

Optimization: With the action means and standard deviations from the network we can

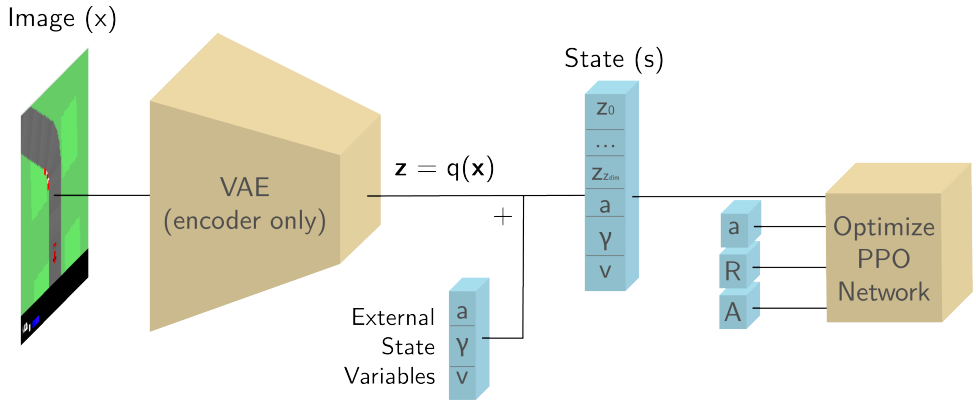


Figure 3.1: Shows the PPO+VAE training pipeline. Note that the subscript t is omitted in all the variable names. External state variables refer to acceleration (throttle,) steering angle and speed, respectively (see Section 3.3.1 for more information.)

calculate the log probability $\log \pi_\theta(a|s)$ of any action a under policy π given state s. Recall that we need to calculate $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ in order to compute the clipped loss, L^{CLIP}. We can do this with the help of the logarithm quotient rule:

$$\log \pi_\theta(a_t|s_t) - \log \pi_{\theta_{old}}(a_t|s_t) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} = r_t(\theta) \tag{3.2}$$

Thus, we compute the combined loss, $L^{CLIP+VF+S}$, and optimize it with the Adam optimizer.

3.2.4 Variational Autoencoder

As discussed in Section 2.3.4, Kendall showed that using a variational autoencoder was instrumental in reducing the training time. The idea is that the VAE will serve as a feature extractor, making the state space we train our agents on more disentangled and therefore easier to predict. We wish to investigate and confirm this, and have therefore implemented and integrated a VAE into the learning pipeline of our agent. We will further discuss how the encoded latent representation helps the agent learn faster in Section 4.1.3. Figure 3.1 shows the PPO+VAE training pipeline we will be using in the experiments.

Architectures: There are several ways to construct the encoder-decoder architecture of an autoencoder. In our experiments, we have tried using a multi-layer perceptron (MLP) for the

encoder and decoder, and we have also tried a convolutional neural network for the encoder-decoder pair. The MLP VAE's encoder takes a preprocessed $w \times h$ image with pixel values in the $[0, 1]$ range as input. The image is flattened and passed through two fully-connected layers of sizes 512, 256, and finally two parallel layers of z_{dim} units; where z_{dim} is a constant denoting the size of latent space of the VAE's bottleneck. The layers use a ReLU activation, except the last layer that has no activation function. 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 is interpreted as a the standard deviation – and we pass the sampled latent vector through the decoder. The decoder consists of two fully-connected layers of sizes 256, and 512, with ReLU, and finally a fully-connected layer of size $w \times h$ with a sigmoid activation function. The sigmoid squashes the output values to ensure that the range of the output pixels are the same as the input ($[0, 1]$ range.) The VAE is optimized by minimizing using either a binary cross-entropy loss (BCE) or a mean squared error loss (MSE) between the input and output, as described in Section 2.3.5.

The CNN version of the VAE is inspired by Ha *et al.* [HS18]. This model starts with a encoder of four convolutions of 32, 64, 128, 256 filters, with 4x4 kernels, a stride of 2 and ReLU activations. The output of the last convolution is flattened and fed into two parallel fully-connected layers of size z_{dim}. Similarly to the MLP model, 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 serves to resize 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 128, 64, 32 and 1 filters, strides of 2, and 4x4, 5x5, 5x5, 4x4 kernel sizes. The kernel sizes were selected this way to make the size of the output of the final convolution a $w \times h$ image. The VAE is then optimized with BCE or MSE like before.

Dataset: The datasets that we used to train the VAEs with were obtained by driving around in the environment manually, collecting images from the front facing camera as we go. In all the environments, we collected 10,000 images, and split out dataset into 9000 training images and 1000 validation images. Another alternative we briefly tried was to train the VAE and

agent simultaneously, thereby eliminating the need to collect any data for pretraining. We will discuss the datasets more in Section 3.3.2.

Training: To train the VAE, we train the model to reconstruct the input image from the sampled Gaussian latent vector, and minimize the reconstruction loss as described in Section 2.3.5. We use the Adam optimizer with a learning rate of α, learning rate decay of γ, and a batch size of N. We end the training if the validation loss has been increasing for the 10 last epochs.

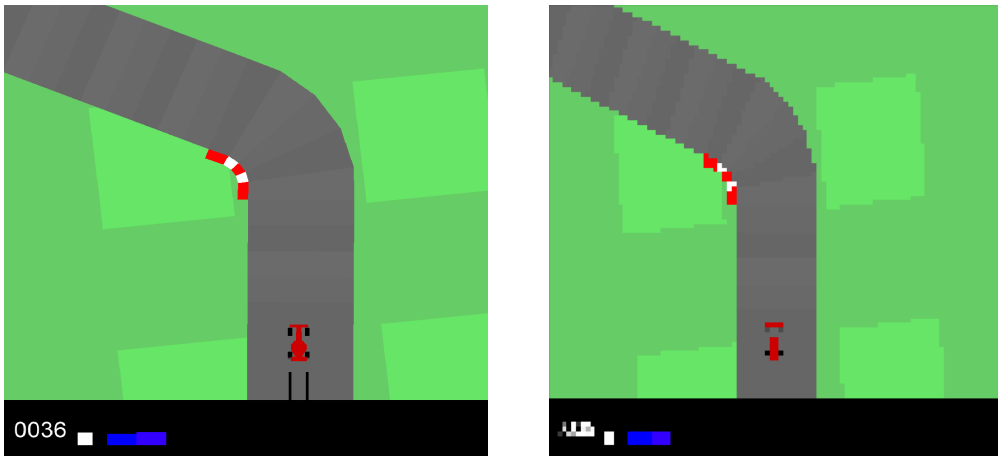
KL-tolerance: Note that in World Models [HS18], they use a KL-tolerance factor to make the optimizer only apply KL-loss once the KL-loss exceeds some tolerance factor. We feel like this is not well justified, and did not find any other examples of this implementation detail, so we opted to train without a KL-tolerance factor.

3.3 CarRacing-v0 Experiments

The initial goal of the thesis was to explore ways the we can reduce the amount of training time needed. As we have shown in our previous work (Appendix A.1,) training agents in CarRacing-v0 can take up to 2 days to converge. At the same time, we have seen recent work that is able to train a car to follow a road in only 15 minutes. With this as our motivation, we will discuss some key differences we found in the way their agent was trained compared to ours, and attempt to identify what measures can be changed to get results closer to theirs.

3.3.1 Environment

As discussed in Section 1.3, there exists a wide range of open source simulators ready to be used for the task of reinforcement learning. OpenAI's *CarRacing-v0* is one of these, and its strength lies being easy to use with reinforcement learning compared to other driving-simulators, making it ideal for quickly testing and iterating on our hypotheses. CarRacing-v0 features a racing car



(a) (b)

Figure 3.2: (a) Screenshot of the CarRacing-v0 environment as rendered on the screen. (b) Example of the 96x96 state space as seen by the agent.

on a procedurally generated racing track, viewed from a top-down 2-dimensional ("birdseye") view. The car is controlled by a (γ, a, b) triplet of continuous action values, where:

- $-1 \leq \gamma \leq 1$ is the steering angle (in radians)
- $0 \leq a \leq 1$ is the acceleration
- $0 \leq b \leq 1$ is the break

The physics of CarRacing-v0 are simulated with *Box2D*, a dedicated 2D physics simulator, making the simulation as realistic as possible. The environment considers several aspects of the dynamics of the car: it is a rear-wheel drive car with a speed sensor, ABS sensors, and a gyroscope. It also simulates the friction on the ground, so turning sharply and breaking on the grass will make the car skid.

The environment generates a randomly generated track for every episode, and the environment is considered "solved" if the agent achieves an average score greater than 900 over 100 iterations. The agent receives a reward of $1000/N$ every time it reaches a different "tile" or segment of the track, where N is the number of segments in total. The agent also looses 0.1 points for every frame, which equate to -5 points every second. This means that the maximum score the agent

can get is $1000 - 5t$ where t is the minimum possible time for this particular track in seconds. We will use this metric (which we will refer to as "score") when we compare models later.

Figure 3.2(b) shows an example of the state space of CarRacing-v0. The state space of the environment consists of a 96x96 pixel RGB image, where the speedometer, ABS sensors, steering angle and brake sensors are encoded into the image itself as vertical and horizontal bars in the lower 96x12 pixels of the image. The idea is that by encoding it in the image, we will only need to train a convolutional model, as the agent will learn to interpret these measurement directly from the image. However, we found this to be a bit limiting for some of the experiments we wanted to conduct, so we opted to create a modified version of the environment that we will discuss next.

Environment Modifications

CarRacing-v0, as its name suggests, is an environment primarily focused on creating agents that drive fast. In the environment, the car's acceleration and maximum possible speed are quite high. The environment also features hard turns that necessitates braking. To make the environment more similar to Kendall's, where the goal is to simply follow a straight road, we decided the following modifications:

Maximum Speed: To give us better control over the car and to reduce the maximum speed, we decided to make it so that throttle is only applied when the speed is less than 30 pixels/timestep. Practically, this will limit the max speed to 30 pixels/timestep, making it more in line with Kendall, who set a hard limit on 10 km/h. Additionally, we apply a 0.1 throttle scale (meaning 1 in throttle is 10% of the original value.)

Action Space: Like Kendall, we decided to reduce the action space by removing the brake action. That leaves us with a (γ, a) control vector.

Softer Turns: To make the environment more like a road-following environment rather than a racing environment, we decided to soften the turns, meaning we will have less use of the brake.

State Modifications: Furthermore, we do a simple preprocessing step in order to reduce the

size of the state space we feed to the VAE. We opted to crop and convert the 96x96 RGB frames that we are given by the environment into 84x84 greyscale images, giving us a state space of $84 \times 84 \times 3 = 27,648$ values. The cropping removes 6 pixels from the left and right sides, because these pixels typically do not contain information that is immediately relevant to the argent, and we also crop the lower 12 pixels from the image because these pixels encode the dashboard parameters, and are not necessary since we are appending these measurements to the state vector instead. The measurements we append are a, γ, v, so the states the PPO agent sees looks like follows:

$$s_t = \{z_0, ..., z_{z_{dim}-1}, a, \gamma, v\} \tag{3.3}$$

Where, $z = q(x)$ is the output of the encoder, a is current throttle, γ is current steering angle, and v is speed, normalized by $max\ speed = 30$. Figure 3.1 shows how this state vector is constructed.

3.3.2 Experiments

To identify the contributing factors in reducing training time in Kendall, we devised experiments that compare individual aspects of their design. Here are the experiments we conducted:

Variation Autoencoder: Kendall *et al.* suggested that using a pretrained variational autoencoder for feature extraction was the most impactful design decision, and [RHT+19] supports this claim. However, we decided to dive a bit deeper, and have analyzed the effect of choice of loss function (BCE vs MSE,) choice of latent space size (z_{dim},) and choice of β. Note that [HPR+17] suggests that increasing β will help the VAE generalize better, by forcing the VAE to produce more disentangled state representations, so we will also test this claim. Finally, we will also evaluate the effectiveness of VAE models as a whole, by comparing an agent that uses a randomly initialized VAE and an agent trained directly on pixel values (with an architecture matching the encoder of the VAE.)

PPO Hyperparameter Search: We conducted a quick hyperparameter search, comparing the Atari and MuJoCo parameters from the PPO paper [SWD+17], and wanted to investigate the effect of using a finite vs. an infinite horizon in the environment. It is known that infinite horizons make reinforcement learning agents more biased [BB11], however it may also lead to shorter training times as a result. Also note that Kendall use an infinite horizon in their experiments, since it is the most natural way to train an agent in real-life where it is impossible to pause in the middle of an episode to train.

Early Termination: Early termination is a term that 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 that is smaller than the original distribution. In Kendall's environment, the expert driver terminates the episode once the car drives of the road. This is different from CarRacing-v0, where the car must to drive until a timer has expired, even when it has driven off the road. Kendall does not directly address the effect of this decision in their paper, so we will investigate it this report.

PPO, DDPG, SAC: We also wanted to compare deep reinforcement learning algorithms. Kendall used Deep Deterministic Policy Gradient [LHP+15] in their experiments, and [RS19] used the newly published Soft Actor-Critic method [HZAL18]. [SWD+17] showed that PPO consistently has better convergence properties than other state-of-the-art deep reinforcement learning algorithm, such as DDPG, but one might argue that imposing a trust region may slow down training in favour of stability. We wish to find this out by comparing the algorithms side-by-side.

Reward Function: For CarRacing-v0, we used Kendall's reward function only, however, we scaled the speed by a factor of 0.001. The full reward function is as follows:

$$r(v) = \begin{cases} -1 & \text{if driven off the road} \\ v * 0.001 & \text{otherwise} \end{cases} \tag{3.4}$$

Where v is the speed of the car in pixels/timestep.

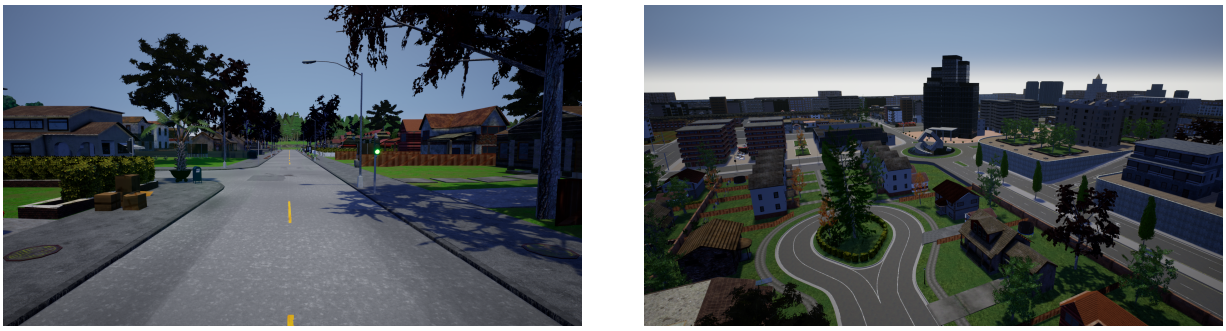
3.4 CARLA Lap Environment Experiments

Furthermore, we were interested to see whether this type of learning works in an environment that more closely resembles real-life driving. In the real world, images are not as clean and noise free as in CarRacing-v0, and we also have to account for *depth*. The agent needs to learn when to pay attention to, and when to ignore objects based on distance, as misinterpreting these distances can have catastrophic results to the agent. Additionally, roads in real life vary in width and length, they differ in their road markings and lane lines, and some roads may even lack markings and lane lines altogether. This is to say that the state space of real-life driving is certainly more complex than that of *CarRacing-v0*.

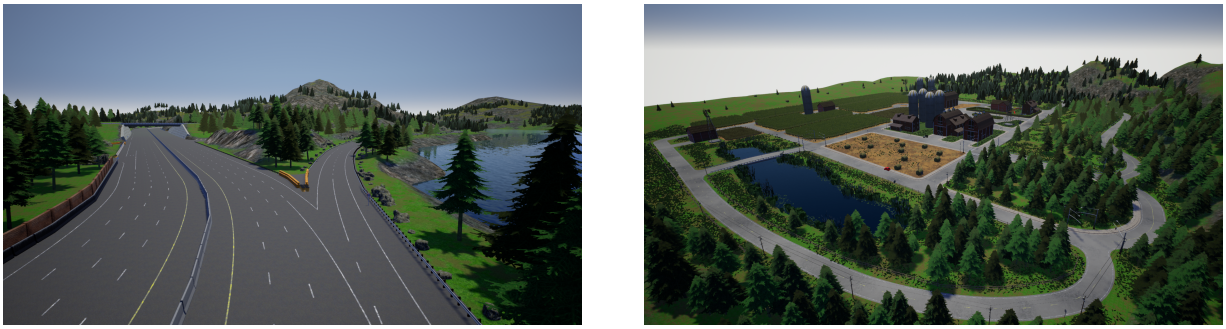
3.4.1 CARLA

We will be using the urban driving simulator, *CARLA* [DRC+17] (version 0.9.4 and 0.9.5,) to test our algorithm in a harder, more realistic driving environment. CARLA is an open-source simulator for autonomous driving research, build in *Unreal Engine 4*. The simulator is focused on simulating a realistic driving environment featuring common urban driving scenarios, and it is bundled with 7 different maps out-of-the-box. Additionally, it provides a general purpose API that allows us to spawn vehicles, cameras and other sensors that we can use however we like. Figure 3.3 shows screenshots from four of the maps.

In terms of reinforcement learning support, there is no built-in functionality or example code to train reinforcement learning agents. The authors of CARLA [DRC+17] provide their results for training a reinforcement learning agent with the A3C algorithm, however, only the code for running a fully trained agent is available online. Furthermore, we would like to be able to interact with a CARLA-based environment through an OpenAI gym-like interface. As such, we decided that we would write our own OpenAI-like environment for CALRA, and has made



(a) (b)



(c) (d)

Figure 3.3: Screenshot from (a) Town01, (b) Town03, (c) Town04, and (d) Town07. Demonstrates the variety in road types, lane markings, and landscapes we can find in CARLA's maps.

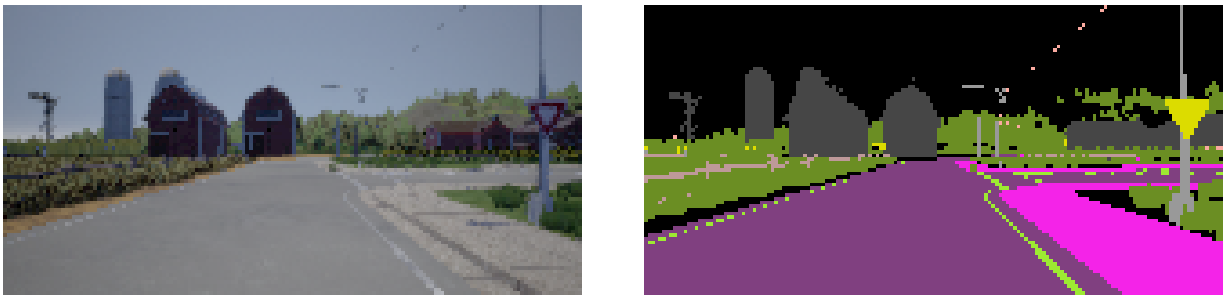
that code publicly available at *https://github.com/bitsauce/Carla-ppo*.

3.4.2 Environment Design

Map

The first step in designing our environment was for us to decide what map we were going to use. Since we are most interested in emulating the result of Kendall, we decided to go for *Town07* (Figure 3.3(d).) Kendall conducted their experiments on a straight country-side road, and we found Town07 to emulate this type of environment well. Town07 features both 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. There are some traffic lights, stop signs, and speed limit signs spread throughout the map, and there are no multi-lane roads. 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 similar to Kendall. The curvy up and downhill roads we

Figure 3.4: Example screenshot from the trailing camera.



(a) (b)

Figure 3.5: (a) Example image as seen by the front facing camera. (b) Corresponding segmentation image.

will be using poses an interesting difference compared to Kendall. The map is also filled with several structures of different shapes and sizes, and additionally features a pond and diverse vegetation. Differences in vegetation is useful in letting us know if our agent has generalized to small perturbations in the scenery. Finally, the interconnected roads in the center opens up the opportunity to try to train our agent to navigate intersections and more complicated road structures, an idea which we will discuss more in Section 3.5.

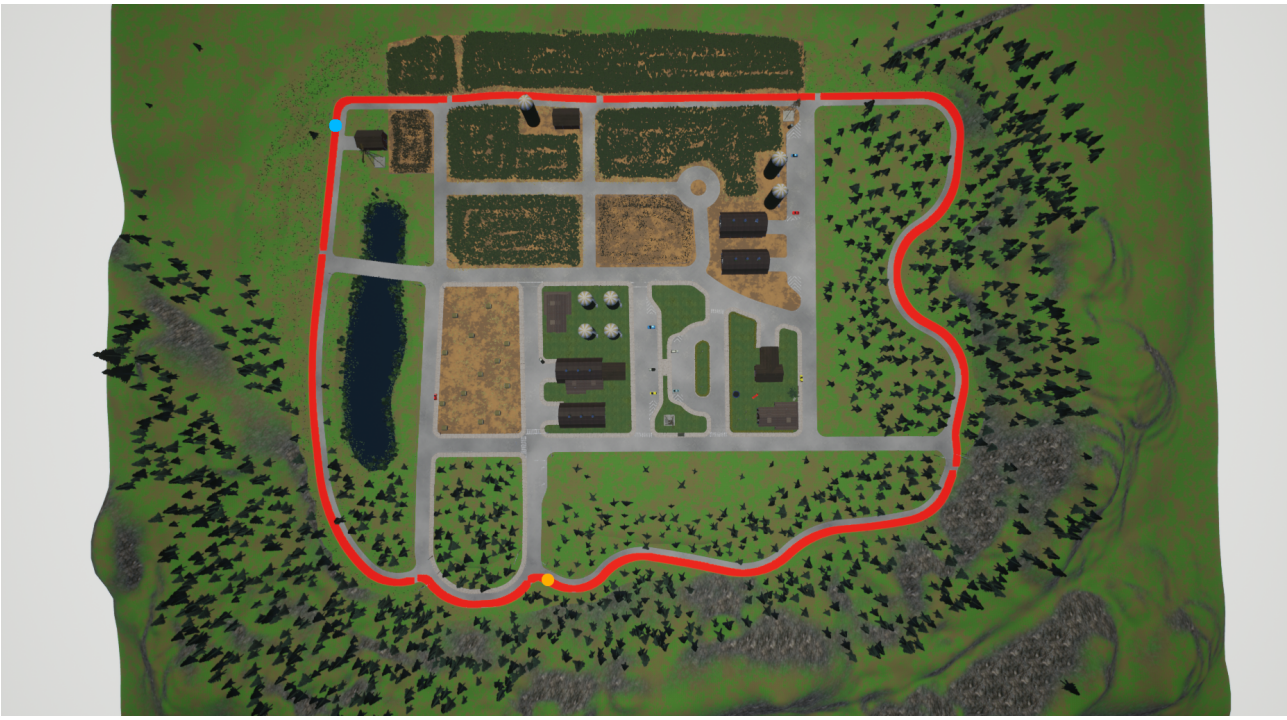


Figure 3.6: Shows a top-down view of the map with the lap highlighted in red. The orange dot marks the starting location that was used in all the experiments using CARLA 0.9.4, and the blue dot marks the starting location used in all the experiments that uses CARLA 0.9.5.

Vehicle and Sensor Setup

In this section, we will describe the vehicle and sensor setup used in our environment. The car we used is a black *Lincoln MKZ 2017* with automatic transmission. The vehicle is equipped with a front facing camera that is attached to the outside on the front of the car, and a trailing camera for spectating purposes. The front facing camera outputs 160x80 RGB, which is a common resolution for autonomous driving purposes [BTD+16]. The front facing camera also has the capability to extract segmentation maps from the environment, which is a feature we will use in Section 3.4.3. Figure 3.4 and 3.5 show examples of output from the front facing and trailing camera.

Environment Inner-Workings

Objective: The objective of agents in our environment is to follow a predetermined lap on the outskirts of Town07, without deviating too far form the center of the lane. The lap is depicted in Figure 3.6, and it passes through seven intersections. For every intersection, the

agent should drive straight (or right if there is no road straight ahead.) This means that there will be no ambiguity in what the end goal of the agent is whenever it encounters intersections. The length of the lap is 1245m, and the environment will consider the completion of three laps as a successful run and terminate. The starting position of the agent during evaluation is marked by the orange dot, however, note that we moved the starting position to the blue dot in all the experiments that uses CARLA 0.9.5. The agent is expected to drive in a clock-wise manner along the marked route.

Action Space: The environment expects actions of $(-1 \leq \gamma \leq 1, 0 \leq a \leq 1)$-tuples, just like the modified CarRacing-v0 environment. We decided to omit braking again, as we will be driving at low speeds with no other cars present, and with no consideration of traffic rules – eliminating the need for braking.

Termination Criteria: The termination criteria for the environment are as follows:

1. Have we deviated more than 3m away from the center?
2. Are we driving slower than 1.0 km/h after the first 5 seconds of the episode has passed?
3. Three laps were completed – success.

The first criteria is imposed on the agent to ensure that the agent is following the road, and to terminate the agent early so it can resume from a good state. We determine the distance to the center of the road by finding the shortest distance between the location of the car in 3D space, and the line that is drawn between the previous and next waypoint on the route. Waypoints are generated along the center of the outermost lane, with a distance of 1m between each waypoint. The environment keeps track of the previous and current waypoint, by checking if our vehicle has passed the next waypoint on the route. We do this by help of the dot product between the waypoint's forward vector, w_{fwd}, and the vector between the vehicle and the waypoint, $c_{pos} - w_{pos}$. If we denote the dot product as $d_{w,c} = w_{fwd} \cdot (c_{pos} - w_{pos})$, then we know that the car has passed the line that is orthogonal to the waypoint's forward vector when $d_{w,c} > 0$. We use this rule to keep track of the current and previous waypoints.

The second criteria is imposed on the agent to make sure that the agent will terminate if: (1) the agent gets stuck in a local minima where the throttle is always equal to zero, (2) if the agent gets caught on some object and is unable to recover.

Checkpoints: In order to facilitate the concept of making the agent "fail faster," we place periodic checkpoints along the track (in training mode only.) We save a new checkpoint every 50m traveled, and the agent is reset to the previous checkpoint upon reaching a terminal state. (Note: the vehicle is always reset to the center of the lane.) The idea behind making the environment push the agent to fail faster is that we will be able to learn faster by skipping straight to the parts of the track that the agent is currently struggling with. Making the agent drive all the way to the difficult part of the track takes a fair amount of time, and the data we accumulate on this trek may not be very useful to the agent in solving the current challenge. We will discuss the effects of this design decision a bit later in Section 4.2.2.

Metrics: The environment also collects a bunch of metrics that we use to compare results. The environment does not provide a default reward function, so these metrics can be used to compare models instead. We have designed the environment this way, so that we can try out different reward functions and still be able to compare models. The metrics we record are:

- **Total distance traveled:** The total distance traveled in meters.
- **Number of laps completed:** Number of laps completed. Updated every time the agent passes a waypoint.
- **Total and average deviance from the center of the lane:** Tells us how much we are deviating from the center of the lane. Average center deviance is averaged over the number of timesteps of the current episode.
- **Average speed:** Average speed, averaged over the number timesteps of the current episode.

We will be using these metrics in our comparisons section. Note that all metrics are reset at the beginning of a new episode.

Synchronous vs Asynchronous: Finally, we have to consider if we want our environment to be synchronous or asynchronous. In reinforcement learning, a synchronous environment will wait until we call the step function before it updates the state of the environment. Synchronous environments are more predictable, and they also allow us to train with learning processes that run slower than the simulation rate of the environment, with is useful when our hardware is weak or the training process is slow. However, there is no requirement that we must use a synchronous environment, so let us consider an asynchronous version of the environment. In an asynchronous environment, the environment will not wait for the step call before updating the state of the environment; the environment will update independently and at variable rates. In terms of autonomous driving, this may actually have some benefits. Training process of training an agent in an asynchronous environment is similar to training an agent real-world, since there is no way to halt the environment to do computations in real-life. Therefore, it may be reasonable to say that if we are able to train an agent to solve the asynchronous version of the environment, that we may also be able to train the same agent it in a real-life setting. Another way of looking at it is that training agents in asynchronous environments make them robust to temporal noise, which is often desirable. In our reward function experiments, we will be using an asynchronous environment, however, in some of the later experiments we will be using a synchronous environment instead.

3.4.3 Experiments

Variational Autoencoder Training

The VAE training process is the same as in *CarRacing-v0*. We start by driving around manually, collecting 10,000 160x80 RGB images that we will use for training. In addition to the RGB images, we also collect the corresponding 10,000 160x80 segmentation maps. Figure 3.5(b) shows an example of a segmentation map, with the corresponding RGB image depicted in Figure 3.5(a). We will be using the full 160x80 RGB image as the input to the variational autoencoder, which is a relatively big increase compared to CarRacing-v0 (CarRacing-v0: $84 \times 84 \times 1 = 7056$, CARLA: $160 \times 80 \times 3 = 38400$.) We believe that this is a necessary change to compensate

for the increase in complexity that the CARLA environment brings. Since we changed the dimensions of the input of the VAE, we also had to change the size of the kernel in the second deconvolutional layer of the decoder to 4×4, to make the dimensions add up.

In the following experiments, we have used two different VAE models, one we will call the *rgb-vae* model, and the other the *seg-vae* model. These VAE models were trained with the same parameters as the best performing CarRacing-v0 VAE (Table 4.2.) The only difference between the rgb-vae model and the seg-vae model, is that the seg-vae model has been trained to reconstruct the *segmentation maps* corresponding to the RGB input, while the rgb-vae tries to reconstruct the RGB images themselves. The idea here is that if our VAE is able to reconstruct the segmentation maps from the RGB images, it will learn to encode features that are more relevant to understanding the semantics of the environment. Using autoencoders for semantic segmentation is a well known idea (*e.g.* [NHH15],) however, no one – to the best of our knowledge – have tested the hypothesis that reinforcement learning agents benefit from learning on state representations that have stronger emphasis on representing the semantics of the environment, which, in our case is achieved by training a VAE to reconstruct segmentation maps.

Reward Functions

We wanted to explore how the reward formulation affects both the end behaviour and the training speed of our agent. We have devised six different reward formulations, and compared them in the CARLA environment.

Kendall:

As a baseline, we wanted to train an agent using the Kendall reward formulation. Kendall simply gives a reward that is proportional to the speed:

$$r(v) = \begin{cases} 0 & \text{if } v > v_{target} \text{or on infraction} \\ v & \text{otherwise} \end{cases} \tag{3.5}$$

Where v is the current speed of the vehicle in km/h, v_{target} is the target speed (the speed we want the agent to drive at after finishing training,) and an *infraction* refers to termination criteria 1 and 2 from Section 3.4.2. Like in Kendall, we also stop the training whenever it goes above the target speed, which we have set to 20 km/h (different from Kendall's 10 km/h.)

Reward 1 – No Termination Over Target Speed:

To give the agent a bit more leniency, we decided to remove Kendall's speed termination criteria, and instead devise a reward function where $v = v_{target}$ is the speed that will give the agent the highest reward:

$$r(v) = \begin{cases} -10 & \text{on infraction} \\ v_{norm} & v_{norm} \leq 1 \\ (2 - v_{norm}) & v_{norm} > 1 \end{cases} \tag{3.6}$$

Here, the term $v_{norm} = \frac{v}{v_{target}}$ is the speed, normalized such that $v_{norm} = 1$ when $v = v_{target}$. This term will grow linearly from 0 to 1 as the car reaches $v = v_{target}$, and beyond that, the reward will decrease linearly from 1 to 0 until $v = 2 * v_{target}$, and continues to negative values beyond that. This reward function will encourage the agent to stay as close as possible to v_{target}, as this is the only value of v that will yield a reward of 1. Additionally, we give a reward of -10 on infractions to deter the agent from going into states that lead to infractions.

Reward 2 – Keep Centered:

To properly utilize the power of running in a simulator, we wanted to try some reward functions that take advantage of being able to measure precise distances between all objects in the environment. In particular, we will be using the distance between the center point of the car to the center of the lane.

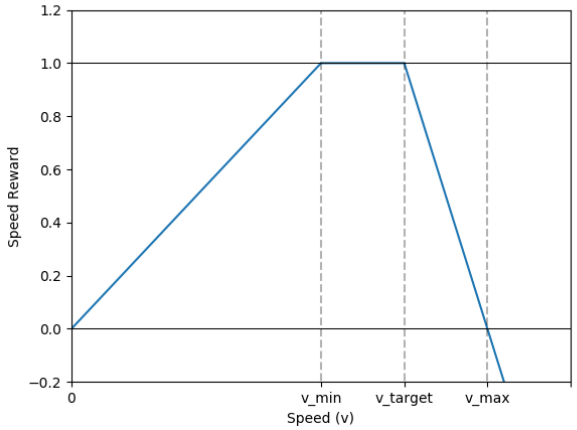


Figure 3.7: Shows how speed is rewarded in reward function 3, 4, and 5.

$$r(v,d) = \begin{cases} -10 & \text{on infraction} \\ v_{norm} + (1 - d_{norm}) & v_{norm} \leq 1 \\ (2 - v_{norm}) + (1 - d_{norm}) & v_{norm} > 1 \end{cases} \tag{3.7}$$

The reward function consists of two terms: the first term, v_{norm}, is a function of the speed v, and the other term, $1 - d_{norm}$, is a function of the distance to the center of the lane, d. The distance term $1 - d_{norm}$ (where $d_{norm} = \frac{d}{d_{max}}$) is simply a function that is inversely proportional to the distance d. This term will have a value of 0 when $d = d_{max} = 3$. The idea behind this reward function is that we want our agent to minimize the distance between the center of the lane and our car (maximize $1 - d_{norm}$,) while we also want to encourage the agent to maintain speeds as close to target speed, v_{target}, as possible.

Reward 3 – Leeway for Speeds Close to Target Speed:

$$r(v,d) = \begin{cases} -10 & \text{on infraction} \\ \frac{v}{v_{min}} + (1 - d_{norm}) & v < v_{min} \\ 1 + (1 - d_{norm}) & v_{min} \leq v < v_{target} \\ (1 - \frac{v - v_{target}}{v_{max} - v_{target}}) + (1 - d_{norm}) & v \geq v_{target} \end{cases} \tag{3.8}$$

With this reward function, we want to give the agent some leeway whenever its speed is close to the target speed. One problem the earlier agents have, is that the throttle signal needs to be very precise for the car to achieve maximum reward. Furthermore, the noisiness originating from the the speed term makes it harder for the agent to learn the correlations between steering angle and centering reward, leading to unstable steering. Giving the agent a range of speed values that all give the maximum reward, allows the agent to focus on steering once it has learned to maintain speeds within the given range.

In this reward function, we have introduced two new variables: v_{min} and v_{max}. Figure 3.7 shows how these relate to v_{target}. In short, v_{min} denotes the smallest speed that will give full reward, and any speed in the range $[v_{min}, v_{target}]$ will also give a reward of 1. For any speeds over v_{target}, the rewards will be interpolated from 1 (when $v = v_{target}$) to 0 (when $v = v_{max}$,) and then to negative values beyond that.

Reward 4 – Additional Reward for Being Aligned With the Road:

Furthermore, we want to add an additional reward signal to encourage the car to be aligned with the road. The idea behind this addition is that we can improve steering behavior by discouraging the agent from turning away from the direction of the road in the first place. By placing a stronger emphasis on this, we hope to see an agent that is less erratic in its steering behaviour.

$$r(v,d) = \begin{cases} -10 & \text{on infraction} \\ \frac{v}{v_{min}} + (1 - d_{norm}) + \alpha_{rew} & v < v_{min} \\ 1 + (1 - d_{norm}) + \alpha_{rew} & v_{min} \leq v < v_{target} \\ (1 - \frac{v - v_{target}}{v_{max} - v_{target}}) + (1 - d_{norm}) + \alpha_{rew} & v \geq v_{target} \end{cases} \tag{3.9}$$

In this reward function, we have introduced an angle term α_{rew}, which is calculated as the angle difference between the vehicle's forward vector, and the current waypoint's forward vector, normalized such that an angle difference of 0° gives a reward of 1, while an angle difference of α_{max} will give a reward of 0. Mathematically:

$$\alpha_{rew} = \begin{cases} 1 - |\frac{\alpha_{diff}}{\alpha_{max}}| & |\alpha_{diff}| < \alpha_{max} \\ 0 & \text{otherwise} \end{cases} \tag{3.10}$$

Where α_{diff} is the angle difference between the vehicle's forward vector and the current waypoint's forward vector.

Reward 5 – Multiplied Centering, Angle and Speed Rewards:

$$r(v,d) = \begin{cases} -10 & \text{on infraction} \\ \frac{v}{v_{min}} * (1 - d_{norm}) * \alpha_{rew} & v < v_{min} \\ 1 * (1 - d_{norm}) * \alpha_{rew} & v_{min} \leq v < v_{target} \\ (1 - \frac{v - v_{target}}{v_{max} - v_{target}}) * (1 - d_{norm}) * \alpha_{rew} & v \geq v_{target} \end{cases} \tag{3.11}$$

In the final reward function, we wanted to test if multiplying the reward terms may work better than adding them together. Since all of our reward terms have been normalized into $[0, 1]$ range (except for the speed term when $v > v_{max}$,) we speculate that our agent can learn better if we interpret the reward function in terms of binary logic. In a driving scenario, we really want the car to drive forward at the same time as it is centered and aligned with the road. In binary logic, we might say that we "want the vehicle to have a speed close to the target speed" AND "be centered in the lane" AND "be aligned with the road." With the previous reward functions, being completely centered is just as good as driving at target speed completely off-center (effectively an OR-statement,) and we would like to counteract this. Therefore, we multiply the different factors – since multiplication of discrete boolean values acts as an AND statement – leading to a reward function that only gives high rewards if multiple criteria are met to a certain extent. Note that we are still using continuous values for each term, so it is not exactly analogous to a boolean AND, but it may still exhibit some of the same properties.

Sub-policies

Finally, we would like to explore the idea of training sub-policies for the four maneuvers we want the car to be able to make. Our method is inspired by Codevilla [CMD+17], who teaches a car to follow a pre-planned path by switching out their acting network based on the current maneuver the agent should make – in their case with imitation learning. In our case, we want to try a similar approach with reinforcement learning, so we trained one PPO actor-critic network for each of the following maneuvers:

1. Follow the road
2. Go straight at the intersection
3. Turn right at the intersection
4. Turn left at the intersection

As we can see in Figure 3.6, the agent encounters several difficult intersections along the lap – particularly the two intersections close to the orange marker. Since our PPO agent has to 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 these intersections. We hope that by introducing additional policies, we will remove some ambiguity from these situation, resulting in a agent that learns more reliably, faster. Furthermore, a sub-policy agent will also be able to drive arbitrary routes by following the high-level commands of a route planner, similar to what Codevilla *et al.* did in [CMD+17]. This is an idea we will explore further in Section 3.5.

3.5 CARLA Route Environment Experiments

For the route environment experiments, we simply wanted to test if it is possible to use PPO to train an agent that is able to follow a path in CARLA. To drive from arbitrary location A to arbitrary location B is the ultimate goal of most autonomous driving systems, so we will

design an environment where we may be able to train a deep reinforcement learning agent to do this. In this section, we will explain the design decisions we made for this environment, and furthermore describe the final experiment we have conducted.

3.5.1 Environment Design

In this environment, the map, vehicle setup and action space is the same as in Section 3.4.2. Furthermore, the environment is still customizeable, allowing us to change the reward function and state space representation to fit our needs. The inner-workings of the environment has changed, however, and we will discuss these changes next.

Environment Inner-Workings

Objective: The objective is the part that differs the most between the *Route environment* and the *Lap environment* (Section 3.4.2.) In the *Lap environment* we were only concerned with a single route along the perimeter of Town07, however, in this environment, there are $\binom{127}{2} = 8001$ possible routes the agent should be able to drive. The objective of the agent is to drive starting at point A to point B without any infractions. The points are randomly selected (without replacement) from a list of 127 manually placed spawn points, without replacement. After the points have been selected, we use a global route planner to calculate the path between the two points (it uses the A* pathfinding algorithm [HNR68] internally,) and we generate waypoints 1m apart along the track as we did before (we use these waypoints to track the progress of our agent.) Be aware that the episode does not end when the agent reaches point B, instead a new route is generated (as described above.) The episode will end once the agent has traversed 3000m in total, and the environment will terminate successfully (*e.g.* environment was "solved.") The reason we designed it this way is because it gives us a metric that is easy to compare between episodes. Completing a route is not meaningful if the route was a simple 200m stretch of a straight road. However, consistently driving 3000m on a random assortment of routes means our agent has generalized well.

Termination Criteria: Similarly to the *Lap environment*, this environment will terminate once the agent is 3m away from the center of the road, or it has been driving with a speed less than 1.0 km /h for the last 5 seconds. The environment is considered solved once the agent has traversed 3000m, and will be successfully terminated upon completion.

Checkpoints: Since this environment will expose the agent to a variety of driving scenarios – by the nature of randomly selecting routes on each reset – we find it to be unnecessary to use checkpoints like we did before. Therefore, this environment does not feature any checkpoints.

Metrics: The metrics we use are the same as in the *Lap environment*.

Maneuver: The environment tells the agent what the current maneuver should be at every timestep. The state of the current maneuver is changed approximately 5m before the maneuver should be executed, to leave some room for the agent to adjust itself.

3.5.2 Experiments

For this environment, we simply take and run the best performing model from the *Lap environment*, to see how well it performs. Since we want to follow a route, we have to change the underlying PPO model into a PPO model with sub-policies. To summarize the main elements of this model:

- This model will use a variational autoencoder that is trained on segmentation maps.
- We will be using *Reward 5* form Section 3.4.3.
- We will use four sub-policies – one for each maneuver the car should be able to make (follow the road, go straight, left or right.)

Using a sub-policy based model to switch the agent's behaviour depending on the current maneuver is necessary to make the agent able to follow a path. Without it, the agent will not know which direction to take in the intersections it encounters, and, as shown by Codevilla

[CMD+17], introducing sub-policies is an effective solution to this problem. Furthermore, using sub-policies eliminates the need for the agent to do any route planning at all, relaying all that work to the global route planner.

Chapter 4

Results

4.1 CarRacing-v0

Here we will present our findings in our modified CarRacing-v0 environment. First, we will look into VAE training.

4.1.1 Variational Autoencoder Comparisons

To start of, we trained five different variational autoencoders. Figure 4.1, 4.2, 4.3, 4.4, and 4.5 shows the effect of annealing latent space vector z for each of the trained models. The x-axis in these figures represents a ± 1 perturbation to the i$^{\text{th}}$ latent space variable, where i is shown on the y-axis. Also note that the initial z vector was set by passing a typical image from the

Loss Function	Architecture	z_{dim}	β	Reconstruction loss
BCE	CNN	64	1	4623
BCE	MLP	64	1	4623
BCE	CNN	10	1	4620
BCE	CNN	64	4	4630
MSE	CNN	64	1	18.39

Table 4.1: Final reconstruction loss on the validation set for each trained VAE model. Note that the loss of the MSE model is significantly smaller, because it is measuring a different quantity.

Hyperparameter	Value
Learning rate α	1e-4
Batch size N	100
Loss function	BCE
Architecture	CNN
z_{dim}	64
β	1

Table 4.2: Default VAE parameters.

environment (an image of a straight road) through the encoder; meaning that the models have been seeded similarly.

Baseline Model: Table 4.2 shows a list of all the parameters that were used when training the baseline model. The parameters for the baseline model were selected based on a similar implementation of a VAE [RHT+18]. Figure 4.1 shows some reconstructions for this model. Looking at the figure, we can see that the hypothesis that a VAE can learn disentangled features is correct (Section 3.2.4.) For example, we can see that $z_{index} = 7$ corresponds to the inward curvature of the center of the road, $z_{index} = 26$ and $z_{index} = 27$ corresponds to the outward and inward curvature of the top of the road, $z_{index} = 55$ corresponds to the x-position and angle of the road. Worth noting here is that our VAE appears to have combined the angle and x-position into a single variable, meaning these features are somewhat entangled. This could be an issue caused by a lack of data (maybe the road is normally angled whenever the car is offset on the road in the dataset,) or it may be an issue that we can solve by altering the model. We will be using the baseline parameters for all the models discussed in this section, unless otherwise is stated.

CNN vs MLP: In the first set of comparisons, we will look at how the convolutional architecture compares to the multi-layer perceptron architecture. We can see in Table 4.1, that the CNN model has equal reconstruction loss to the MLP model. However, in terms of parameters, the CNN model has $1.9M$ parameters, while the MLP model has $7.5M$ parameters; almost $4\times$ as many. The fact that the CNN model is comparable to the MLP model even when it has much fewer parameters is to be expected, as CNNs have been experimentally shown to excel at feature extraction in images [KSEH12]. However, if we compare Figure 4.1 and Figure 4.2, we can see that, visually speaking, the MLP model is able to reconstruct some details that the CNN

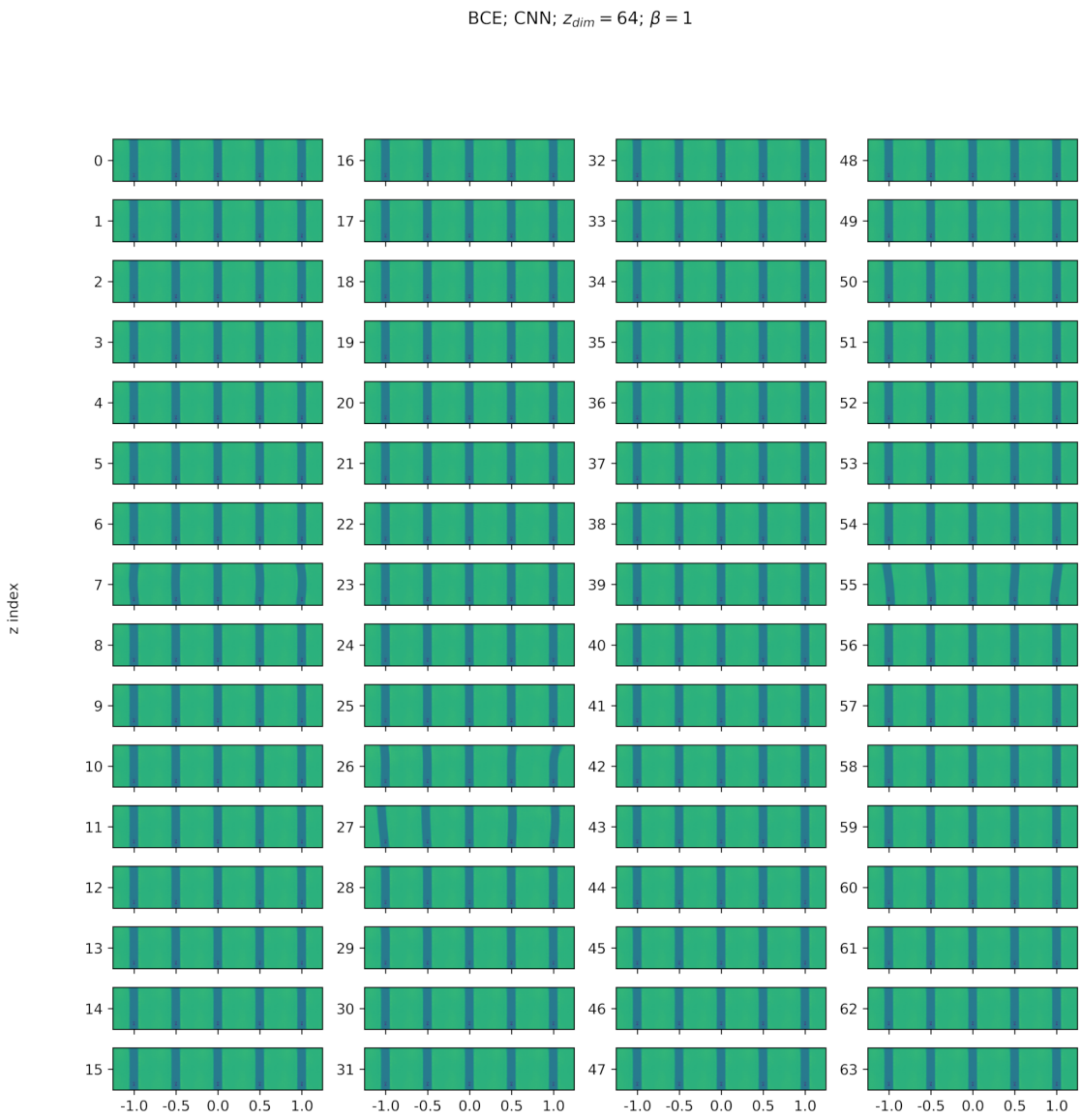


Figure 4.1: Shows the reconstructions generated by the VAE model trained with BCE loss, CNN network architecture, $z_{dim} = 64$, and $beta = 1$ as we anneal the latent space vector z by ± 1, where each figure represents one dimension of z. We can see that $z_{index} = 7$ corresponds to the inward curvature of the center of the road, $z_{index} = 26$ and $z_{index} = 27$ corresponds to the outward and inward curvature of the top of the road, $z_{index} = 55$ corresponds to the x-position and angle of the road.

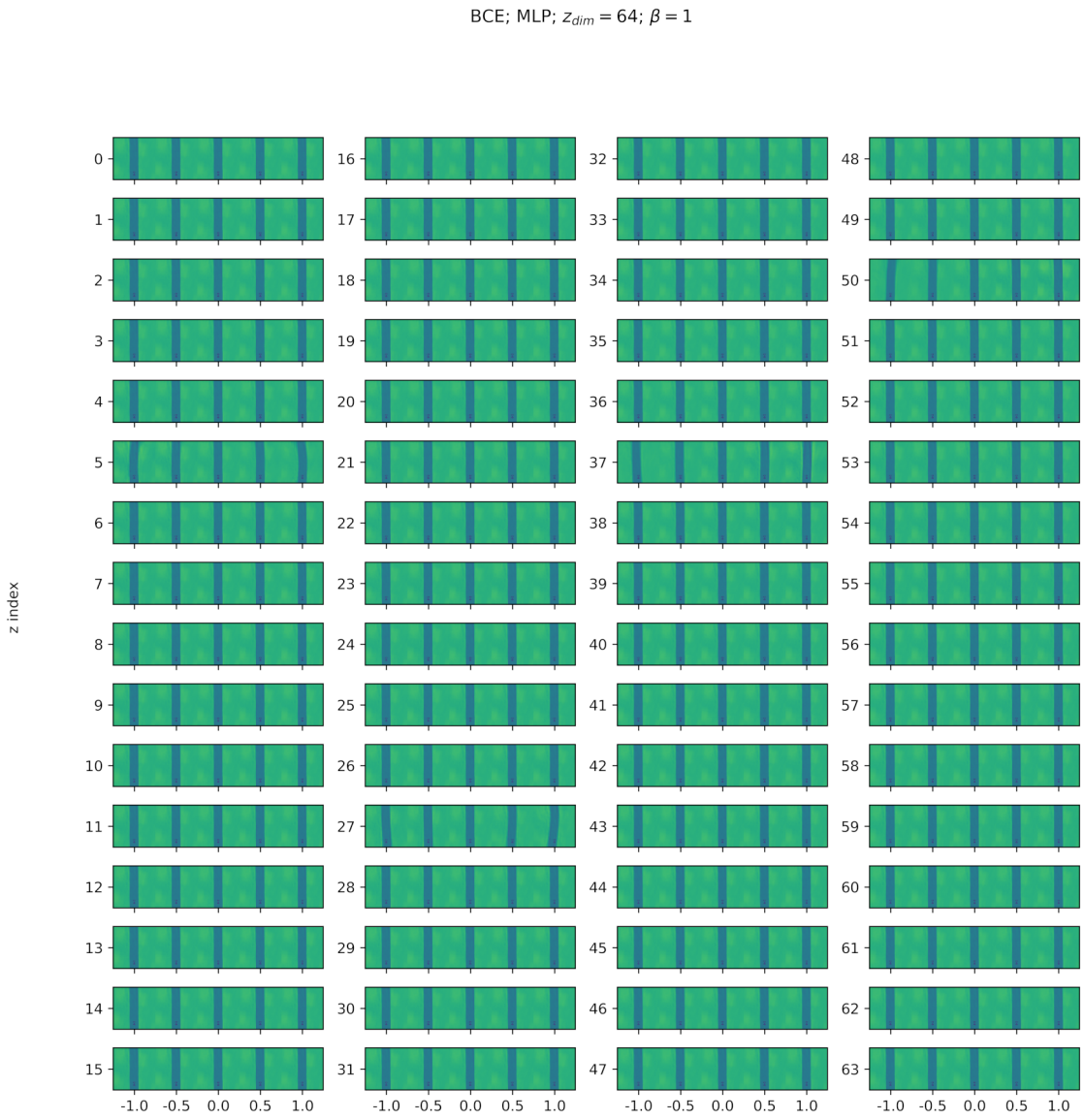


Figure 4.2: Shows the reconstructions generated by the VAE model trained with BCE loss, MLP network architecture, $z_{dim} = 64$, and $beta = 1$ as we anneal the latent space vector z by ± 1, where each figure represents one dimension of z. We can see that $z_{index} = \{7, 26, 27, 55\}$ are the features with the strongest effects, for example, $z_{index} = 5$ clearly corresponds with road curvature.)

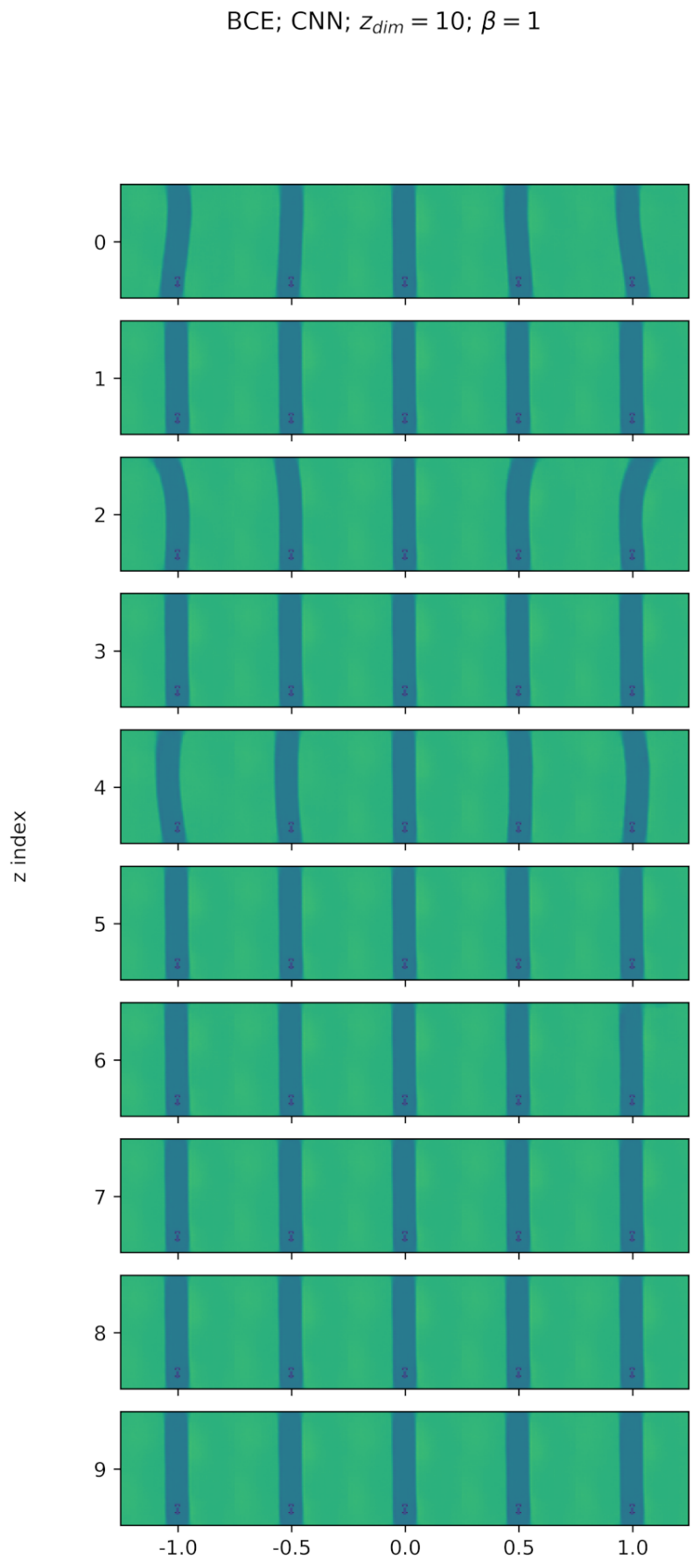


Figure 4.3: Shows the reconstructions generated by the VAE model trained with BCE loss, CNN network architecture, $z_{dim} = 10$, and $beta = 1$ as we anneal the latent space vector z by ± 1, where each figure represents one dimension of z. We can see that the model is still able to learn relevant features, even though $z_{dim} = 10$. Also observe that this model only has three dimension that have a significant impact on the reconstructions ($z_{index} = \{0, 2, 4\}$.)

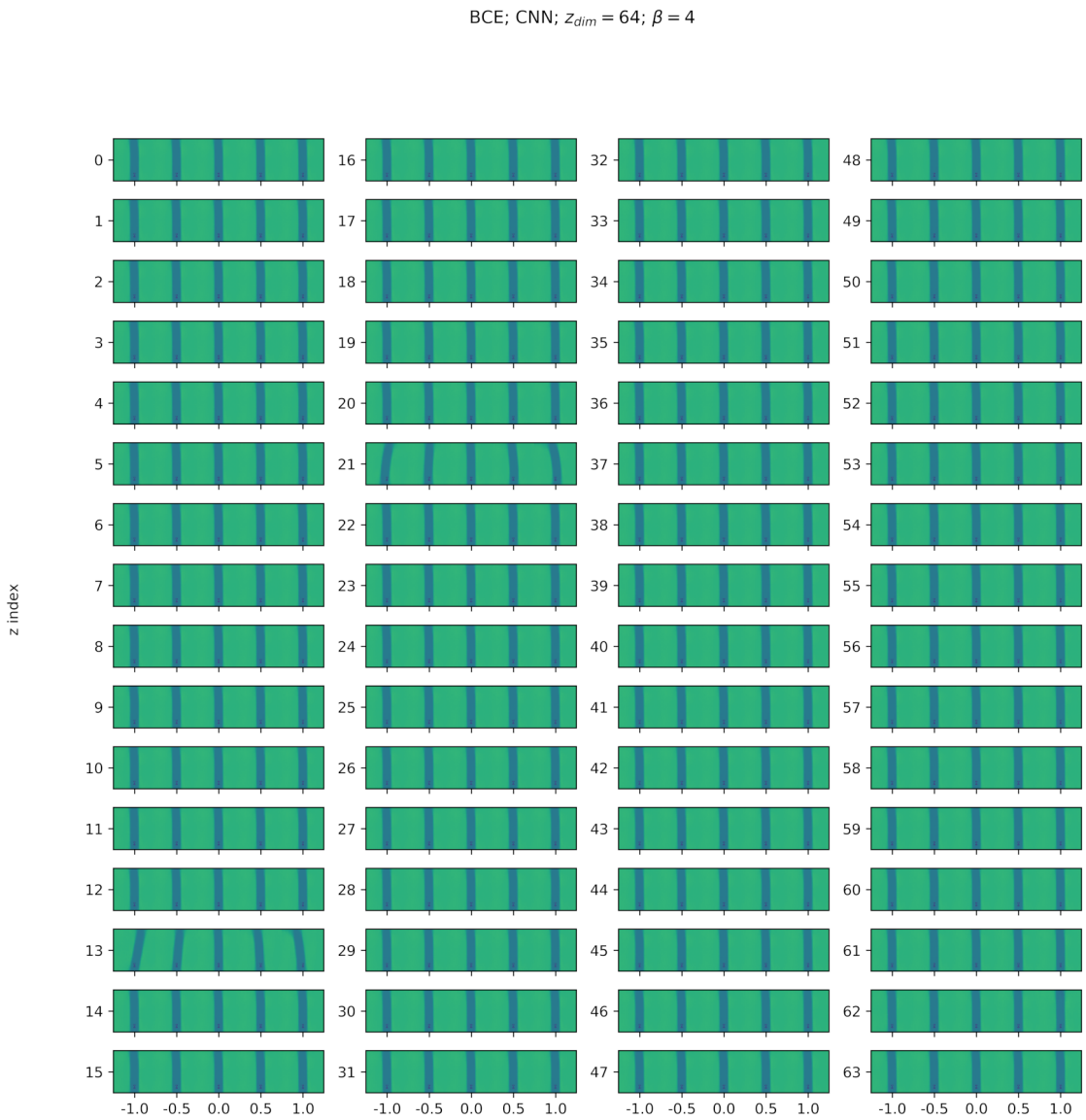


Figure 4.4: Shows the reconstructions generated by the VAE model trained with BCE loss, CNN network architecture, $z_{dim} = 64$, and $beta = 4$ as we anneal the latent space vector z by ± 1, where each figure represents one dimension of z. We can see that only two dimensions have an impact on the reconstructions ($z_{index} = \{13, 21\}$.)

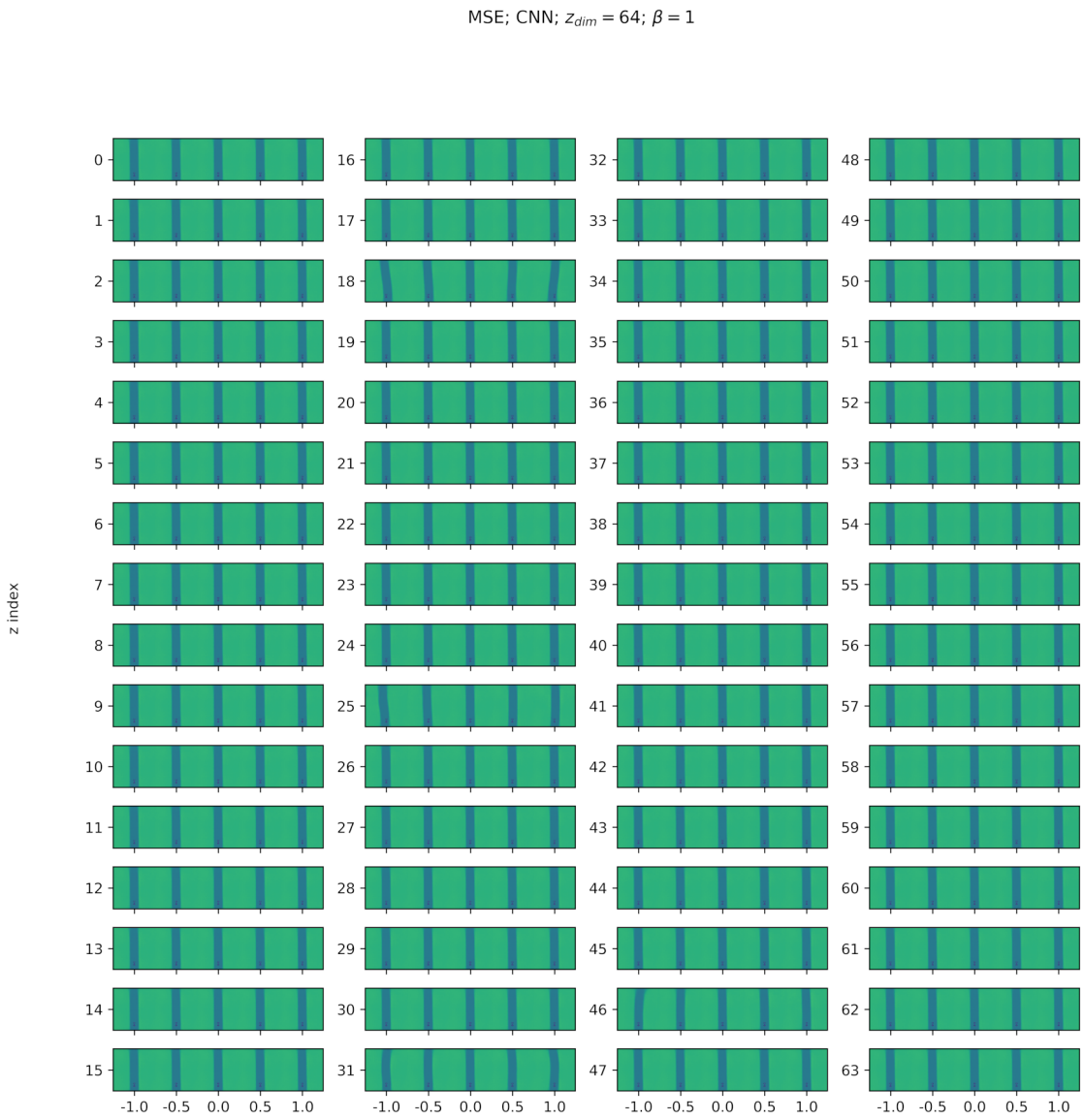


Figure 4.5: Shows the reconstructions generated by the VAE model trained with MSE loss, CNN network architecture, $z_{dim} = 64$, and $beta = 1$ as we anneal the latent space vector z by ± 1, where each figure represents one dimension of z. We can see that $z_{index} = \{18, 25, 31, 46\}$ are the dimensions with greatest effects, and that these dimensions produce reconstructions that are visually similar to the significant features in Figure 4.1.

seemingly ignores. For example, we can see that the yellow spots on the grass come through in the MLP model, while in the CNN model they do not. The MLP model is, however, more noisy in its output in general. Having noisy state representations may be a downsides in terms of trying to train an agent to act, because it means that the agent will need to learn to ignore the noisy features of the state space. Finally, if we look at $z_{index} = \{7, 26, 27, 55\}$ in Figure 4.1, and compare them with $z_{index} = \{5, 27, 37, 50\}$ in Figure 4.2, we can see that the latent variables have much stronger effects in the CNN model (for example, $z_{index} = 5$ clearly corresponds with road curvature.) This could be a result of the CNN model being more strongly disentangled than the MLP model – perhaps because the MLP model ends up correlating features such as the curvature of the road with the yellow patches on the grass. As a result, we will be using CNN models for the rest of the experiments.

BCE vs MSE: As we see in Table 4.1, the models that were trained with BCE loss have much higher reconstruction losses than the MSE model. This is because these two loss functions are measuring different qualities of the predictions of our VAE. As such, these two models cannot be directly compared in terms of reconstruction loss, so we will instead look at the quality of the reconstructions, and the quality of the agents later on. Figure 4.5 shows the reconstructions when we use MSE. If we compare this to the BCE reconstructions in Figure 4.1, it is not very obvious which one is better. One might argue that some of the reconstructions have higher quality with BCE, such as $z_{index} = 7$ of BCE versus $z_{index} = 31$ of MSE (which seem to correspond to road curvature in the respective models,) however, there is not sufficient evidence to claim that either model is superior. We will stick with BCE for the rest of the models.

$z_{dim} = 10$ **vs** $z_{dim} = 64$**:** One question that we posed was if reducing the size of the bottleneck in the VAE may force the model to learn more important features, since it has fewer variables to work with. Figure 4.3 show the result when we set $z_{dim} = 10$. We can see here that the model was, indeed, still able to learn relevant features. The features differ a bit from the features when $z_{dim} = 64$; in particular, we can see that the $z_{dim} = 64$ model has four variables that have significant impact on the reconstructions ($z_{index} = \{7, 26, 27, 55\}$), while the $z_{dim} = 10$ model only has three ($z_{index} = \{0, 2, 4\}$.) Note that the $z_{dim} = 10$ has very clear and strong symmetries

in its state space when compared to $z_{dim} = 64$. It is not clear if having several strong features in the state space is more beneficial to the agent than having strictly symmetrical features. If the features are non-symmetrical, the agent may be able to carefully combine non-symmetrical features and learn non-symmetrical relationships in the state space. Note, however, that the $z_{dim} = 10$ model reached a lower reconstruction loss in Table 4.1. As such, we have reasons to suspect it to perform similarly or better than the $z_{dim} = 64$ model.

$\beta = 1$ vs $\beta = 4$: Finally, we will investigate the effect of increasing the β parameter. Table 4.1 shows us that setting $\beta = 4$ increases the reconstruction loss, which is to be expected as the VAE needs to learn simpler models to compensate for the increase in KL-loss. Looking at the $\beta = 1$ reconstructions in Figure 4.4, we can see that only two dimensions, $z_{index} = \{13, 21\}$ have a major effect on the output. This confirms the idea that increasing β forces a simpler model, however, the features seem more entangled when compared to $\beta = 4$. The model appears to have combined x-position and road curvature, and furthermore, the two dimensions seem to encode very similar types of curvature. As such, we do not expect this model to perform very well, however, it may be true the model works better with transfer learning as claimed in [HPR+17] (we will not test this claim in this report.)

4.1.2 Proximal Policy Optimization – Hyperparameter Search

Before making any model comparisons, we wanted to find the hyperparameters that give the fastest learning. In this section, we have applied all the environment modifications discussed in Section 3.3.1, and we have used the default VAE (Table 4.2) with MSE instead of BCE, although this is unlikely to affect our learning. Figure 4.6 shows the validation score of running five different models. Note that validation was run every fifth episode, and that a score over 500 is already fairly good in terms of driving behaviour.

Atari vs MuJoCo: We can observe in Figure 4.6, that the Atari parameters greatly outperformed the MuJoCo parameters. It seems that increasing the horizon (Atari: $T = 128$, MuJoCo: $T = 2048$) can make the training less table (more biased,) and the learning rate may also be too high (Atari: 2.5×10^{-4}, MuJoCo: 3.0×10^{-4}.) Be aware that we have removed

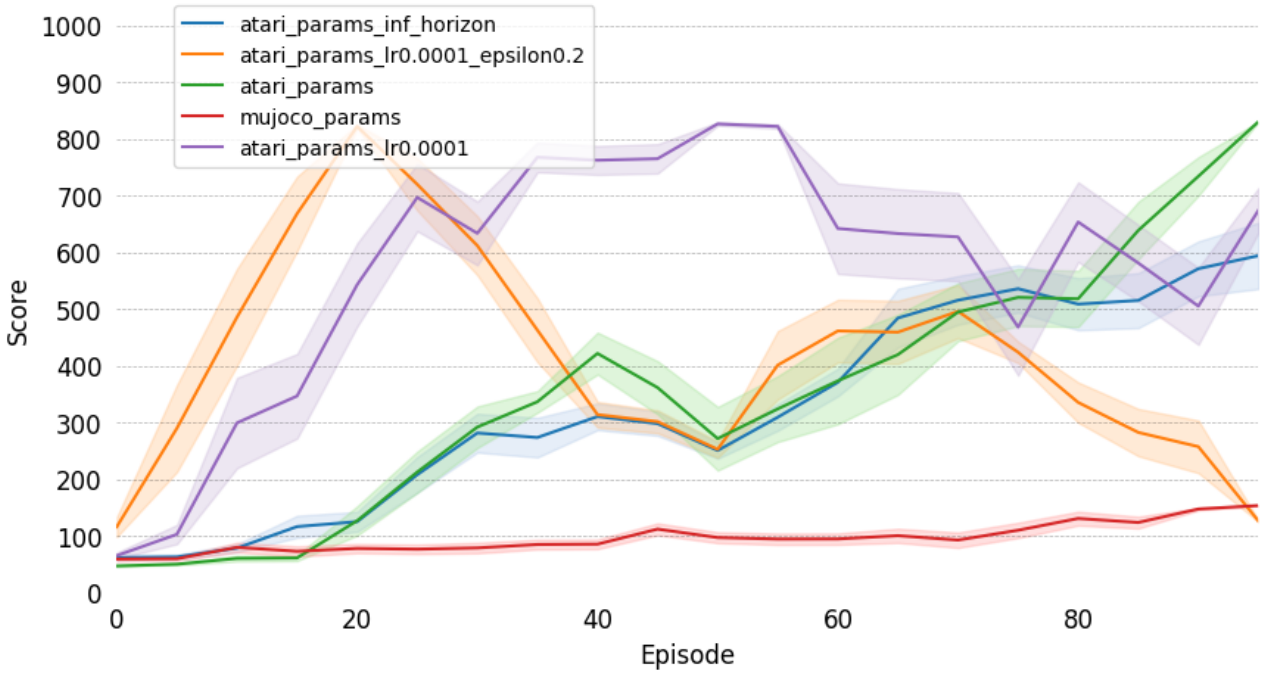


Figure 4.6: Shows episodic scores of five models different models. Note that a sliding window of ± 2 was applied to smooth out all the score graphs. Also note that while the y-axis (episode number) does not equate to wall-time, the models that learn in fewer episodes are more sample efficient and will learn faster in environments that are harder.

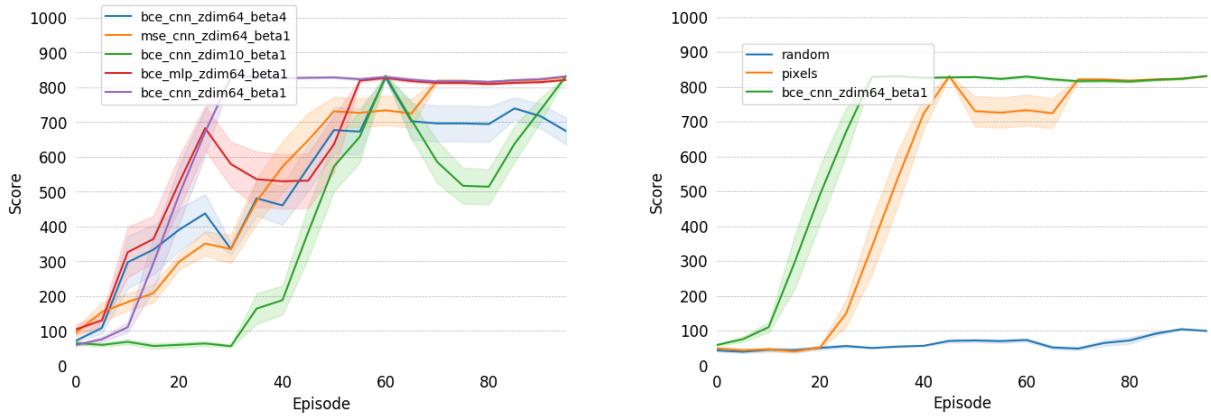
the learning rate and epsilon decay from the Atari model, (since we are only training for 100 episodes,) and that the Atari example used 8 parallel environment, while we are only using 1. Atari games are generally more difficult and have sparse reward functions, so reducing the number of environment (and consequently reducing the amount of exploration,) is not a major issue to us. Since the Atari parameters performed much better, we will use them as our baseline from now on.

Learning Rate: Since we removed the learning rate decay, we decided to also lower the learning rate to compensate. We lowered the learning rate from 2.5×10^{-4} to 1.0×10^{-4}, and we can see in Figure 4.6 that this reduced the training time further.

Epsilon Modification: Our theory here was that increasing epsilon would speed up the training, since it would increase the size of the trust region, thus allowing greater divergence in policy. We can see in Figure 4.6 that this is partially true; it did reach a higher score faster, however it also started dropping down after the initial spike. This is most likely due to the trust region being too big, allowing the model to take destructive steps when optimizing the

Hyperparameter	Value
Horizon T	128
GAE parameter λ	0.95
Discount factor γ	0.99
Clipping parameter ϵ	0.2
Learning rate	1e-4
Value loss scale α	1.0
Entropy loss scale β	0.01
Initial noise σ_{init}	0.4
Number of epochs K	3
Batch size M	32

Table 4.3: Hyperparameters used in agent comparison experiments.



(a) (b)

Figure 4.7: (a) Episodic scores of five agents trained with different VAEs. (b) Episodic scores of three agents; green was trained with a VAE, orange was trained directly on images from the environment, blue was trained with a randomly initialized VAE.

policy. The initial spike may, however, be useful if we want to see results faster. As such, we will be using an epsilon of 0.2 in later chapters.

Infinite vs Finite Horizon: Since the Kendall environment was using an infinite horizon, we wanted to investigate what the result of this decision may have been. We can see in Figure 4.6 that, for PPO, optimizing with an infinite horizon does not have a significant impact on the training. We conclude that the length of the horizon is not too influential and we will stick with a finite horizon.

4.1.3 Agent Comparisons

At this point, we already have an agent that learns to drive pretty quickly. Most of the models in the last section learned to drive within 100 episodes, and the best model, in terms of reaching 800 points in the least time, needed only 15 episodes. This was the *atari_params_lr0.0001_epsilon0.2* model in Figure 4.6, and it reached a score of 810.5 in $\sim$ 4 minutes. As a result, we settled on using the parameters of this model in subsequent experiments, and the complete list of hyperparameters are shown in Table 4.3. These parameters are tailored to attain results as quick as possible, allowing for quicker iteration in more difficult environments.

In addition to comparing the quality of the VAE models directly (Section 4.1.1,) we have also compared the performance of using these VAEs when training our agent. As discussed, we cannot compare the MSE and BCE losses directly, so training an agents with these losses may give us additional insight. Furthermore, reconstruction loss may not be sufficient to judge the agent's performance in the first place; for example, a higher value of beta will always have a higher reconstruction loss because of the restrictions it puts on the optimization of the VAE. Note that the default parameters of the trained VAEs are stated in Table 4.2, and we have used those values unless otherwise is stated.

Effect of Variational Autoencoder Model

CNN vs MLP: We can see in Figure 4.7(a) that the MLP and CNN models perform fairly similarly. The CNN model does a bit better, which could be a result of the fact that CNNs excel at extracting features from images, as mentioned. As such, we decided to use the CNN model for the later sections.

BCE vs MSE: We can see here that the MSE model performs slightly worse than the BCE model. As we discussed earlier, these two metrics cannot be compared directly, and even though the MSE reconstruction loss was orders of magnitude lower, the BCE model still performs comparably. As such, we decided to stick with BCE loss for the later sections.

$z_{dim} = 10$ **vs** $z_{dim} = 64$**:** Surprisingly, the $z_{dim} = 10$ model seem to perform a lot worse than

the rest. This may be due to the fact that even though the $z_{dim} = 10$ model appear to encode just as much information as the $z_{dim} = 64$ model (from Figure 4.3 and 4.1,) it may be that the $z_{dim} = 64$ model is actually encoding small perturbations that we are unable to spot with our eyes. The results suggests that $z_{dim} = 10$ is not sufficient to encode all the high-level features of the environment, and we will thus use $z_{dim} = 64$ in later sections.

$\beta = 1$ **vs** $\beta = 4$**:** Contrary to the results of [HPR+17], we could not get beta 4 to outperform beta 1. As we discussed earlier, when we set $\beta = 4$ it appears that the latent features become more entangled, and as a result, our agent will have a harder time interpreting state features. As such, we will use $\beta = 1$ in later sections.

Randomly Initialized VAE: To ensure that training the VAE is actually beneficial to the agent, we have also trained an agent with a randomly initialized VAE for comparison. Figure 4.7(b) shows the results. We can clearly see that training the VAE is vital to the quality of our agent; the random agent barely improved. This makes sense intuitively, as a random VAE is simply going to output random feature vectors, making it so that the agent needs to learn an to act in an excessively noisy state space.

Training on Pixel Values: Moreover, we also trained an agent to act directly on pixels values, much like the original PPO paper [SWD+17]. In this model, we have simply taken the convolutional layers of the VAE encoder and prepended it to the actor-critic architecture, and we optimize it with respect to the policy and value loss. We can see in Figure 4.7(b), that pre-training the VAE to reconstruct the input image does, in fact help. The VAE model is most likely learning faster because there are fewer parameters to optimize, and the state space is more stationary. Therefore, we will continue to use a pre-trained VAE for the later experiments.

Alternating Optimization: We also had the idea to train the VAE and policy in alternate phases instead of pretraining. The way this worked was that as the agent was exploring the environment and gathering the data needed to train the PPO model, we were also collecting all the images/observations. Once the trajectory was computed, we optimize the policy as described in Section 2.3.3, followed by optimizing the VAE to reconstruct the images it collected

for 10 epochs as described in Section 3.2.4. The experiment, was unsuccessful – it did not learn anything useful in 100 episodes – however, we also did not test this method beyond a simple experiment.

Effect of Reinforcement Learning Algorithm

PPO vs DDPG vs SAC: To make sure we are using the best algorithm for this problem, we attempted to compare the performance of our PPO implementation to the performance of the DDPG and SAC algorithms from OpenAI baselines [DHK+17]. Unfortunately, we were unable to get any decent results with DDPG or SAC. We would have to look into this further to verify that the VAE was integrated correctly into the baseline algorithms, however we decided to limit the scope of this project to PPO only.

4.2 Carla Lap Environment

So far, we have shown that providing better state representations through the use of a variational autoencoder helps speed up the training of a PPO-based deep reinforcement learning agent. Our agent learned to follow the road in CarRacing-v0 in approximately 4 minutes, and our analysis lead us to a set of hyperparameters and a training setup that we found to work well. Given these results, we want to find out if a similar learning setup can translate to a more complicated environment – namely, the CARLA lap environment discussed in Section 3.4. In this section, we will be testing and comparing different reward formulations, evaluating the effect of "failing faster," and testing our idea of training the variational autoencoder with segmentation maps as the target. Note that we have used the default parameters given in Table 4.2 for training the variational autoencoder, and we have used the PPO parameters given in Table 4.3, except that we have set $\sigma_{init} = 0.1$ because a higher amount of precision is required to solve this environment – particularly the asynchronous version.

We have compiled a video showcasing the results of our experiments. This video can be found

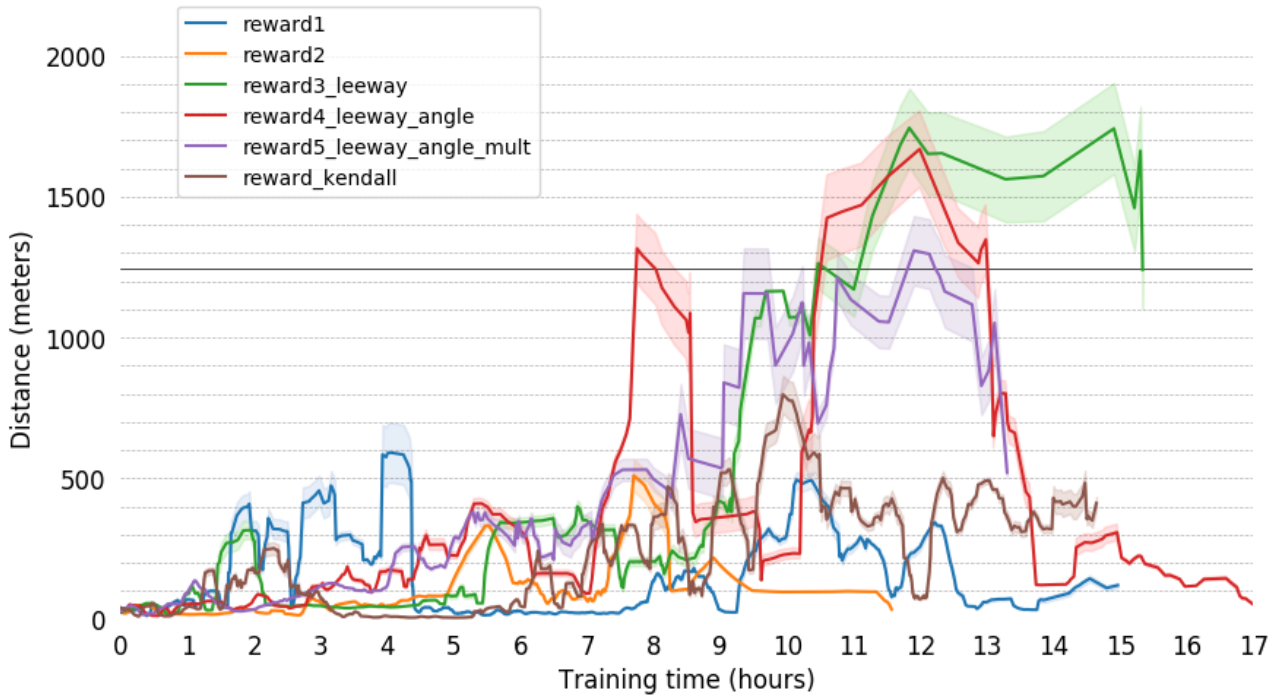


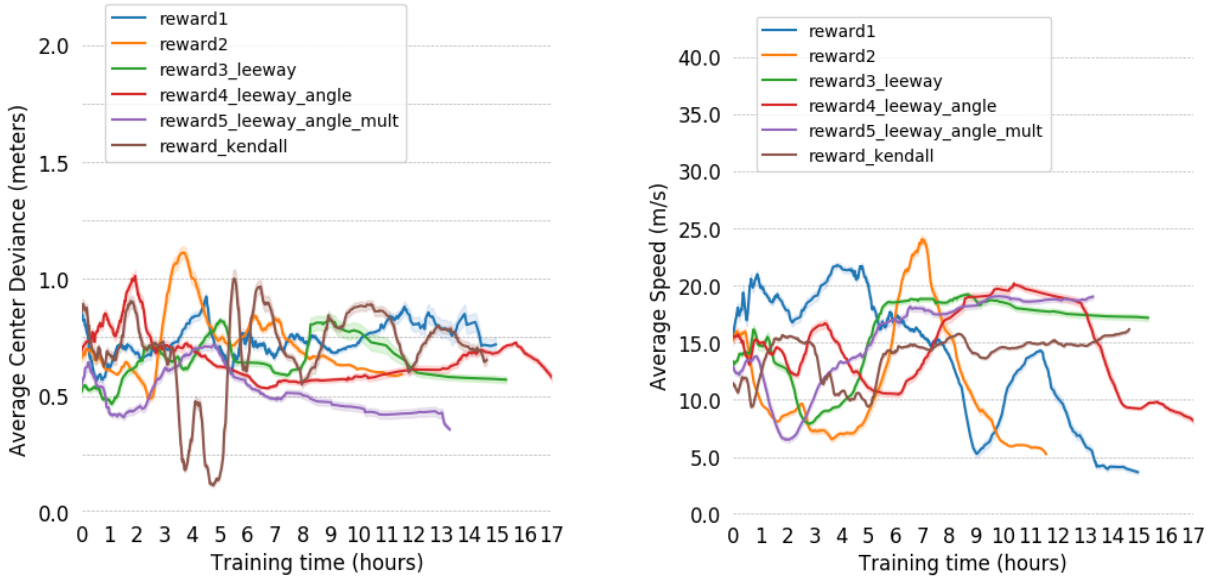
Figure 4.8: Shows the distance traveled by the agent before an infraction occurred, in evaluation mode. The graphs show the six reward formulations that we decided to compare.

at https://youtu.be/iF502iJKTIY, and throughout this and the next sections, we will be referencing specific behaviours that can be observed in the video in the following experiments.

4.2.1 Reward Functions

Figure 4.8 show a comparison of the performance of the six reward formulations in terms of distance traveled during evaluation over training time. We use distance traveled as a metric to compare agents here, because using total reward does not work when the reward functions themselves measure different aspects of the agent's behaviour. The section showcasing the reward functions starts at 44:14 in the video (note that all the timestamps in this report are clickable,) and in this section of the video we have showcased the best run for each of the six agents. Note that all agents in this section were trained in the asynchronous version of this environment, with a target frame rate of 15 FPS, and we will be using $v_{target} = 20km/h$ and $d_{max} = 3m$.

Kendall:



(a) (b)

Figure 4.9: (a) Shows episodic average centering deviation for the agent. (b) Shows episodic the average speed of the agent. Note that a sliding window of ±20 was applied to smooth the graphs.

Kendall's reward function was the first one we tested, and it achieved a maximum distance of 2431m, or 1.95 laps completed, in 10 hours (not visible on graph due to smoothing.) Considering that the lap, starting at the orange marker, is a fair bit more complicated than Kendall's straight road, it shows that Kendall's reward formulation can have success even on curvy up and downhill roads. However, the agent did not manage to complete the three laps we tasked it to do. Looking at some of the later results, we found that several of the runs were terminated early due to the agent slightly overstepping the speed limit we have imposed on it. This is most likely to happen when the agent experiences some temporal noise in its observations, caused by the asynchronous environment. As a result, we wonder if an agent that is more free to go over its desired target speed may perform better, which is what we will test in the next reward function.

Reward 1 – No Termination Over Target Speed:

Here we show the results of integrating the target speed into the reward function itself, creating a more lenient environment for the agent to explore. This agent reached a max distance of 3753m after $\sim$ 4 hours of training. This agent is the first agent to solve our environment, and

by inspecting the video we have observed the following:

- The agent is able to maintain speeds close to the target speed (20 km/h).
- The agent's steering is fairly unstable, both on straight (*e.g.* 49:49) and curvy (*e.g.* 49:15) roads.
- The agent is able to turn fairly reliably (*e.g.* 50:19, 50:50, and 50:55.) It does not cut corners nor does it make too wide turns.
- Speed and steering are slightly more unstable when going downhill compared to Kendall's reward function (*e.g.* Kendall: 46:59, *Reward 1*: 49:15.)

The results show us that the reward function was, indeed, successful at creating an agent that attempts to drives as close as possible to the target speed, and that aggressively terminating the environment may be detrimental to the agent's learning. However, we would like the car to have a greater focus on staying centered in the lane, to hopefully eliminate the unstable, sinusoidal-like steering behaviour. Therefore, we have experimented with ways we might incentivize the agent to staying centered in the lane in the following reward functions.

Reward 2 – Keep Centered:

In this reward function we add an incentive/reward for staying centered. This agent achieved a maximum score of 2115m after $\sim$ 8 hours of training, and did not improve beyond that. While it performed worse than the previous agent, the video reveals that this agent was able to learn some behaviours that we find to be desirable. For example, we can see several examples of the agent slowing down before turning to the right, and also slowing down before going downhill. It is not entirely clear why the modification led to these differences, other than staying centered gives the agent additional reason to drive more carefully. To summarize the what we see in the video:

- The agent is able to maintain speeds close to the target speed (20 km/h).

- The agent will slow down before making turns (*e.g.* 57:01.)
- The agent will slow down before going downhill (*e.g.* 55:20.) Recall that the agent is not able to brake in our environment, so doing this makes sense.
- The agent struggles to center itself (similar to *Reward 1*,) however, it does drive more centered in certain sections of the lap (*e.g. Reward 1*: 50:50, *Reward 2*: 57:04.)

We can see that our agent failed at the sharp, 90 degree turn on the second round, and it looks like it drove off the road because the red house on left confused the agent, making it drive unstably on the last stretch of road leading up to the turn. As to why these agent seem to have an unstable, almost sinusoidal, steering behaviour on straight roads, may be a result of the agents not understanding the physics of the environment. The agent knows that it should adjust its steering to move to the center, however, it tends to overcompensate, perhaps due of a lack of temporal information in the input state. Generally though, we think that there are ways to utilize the distance to the center in the reward function to encourage centering. As a result, we will be keep the centering term in the next experiments as well.

Note that we have also tried an alternate version of this reward function that simply gives no speed reward when we are driving above the target speed. This reward function is functionally more similar to Kendall's reward function, but we found that it was much more difficult to learn due to the drastic discontinuities in the reward when the agent was close to the target speed.

Reward 3 – Leeway for Speeds Close to Target Speed:

Providing some leeway in the speed term of the reward function appears to have improved the results of *Reward 2* by a fair amount. As discussed in Section 3.4.3, the idea behind this reward function is that if we provide some leeway in the speed term of the reward function, we will give the agent a chance to explore how the steering angle affects the centering term of the equation without interference from the speed term. For these experiments, we have set $v_{min} = 15$ and $v_{max} = 25$. We can see in Figure 4.8, that, unlike *Reward 2*, this one does not collapse after the initial increase in performance around the 8h mark. Instead, this model reaches average

evaluation distance of $\sim 1700m$, and achieves a max distance of 3774m (3 laps) at the 13 hour mark. If we look at the video, we can make the following observations:

- Driving behaviour is fairly similar to *Reward 2*, however, it is more reliable in sharp turns (it did not fail any sharp turns unlike *Reward 2.*)
- The changed did not improve the steering behaviour; it is still overcompensating its turns when it is trying to converge to the center of the lane (*e.g.* 59:12.)

Since this agent performed better and remained more stable in its performance after converging (when compared to *Reward 2*,) we conclude that providing leeway to the agent's speed term is a good idea. We will therefore be utilizing this speed term in the following reward functions as well.

Reward 4 – Additional Reward for Being Aligned With the Road:

This reward formulation appears to have many of the same benefits as *Reward 3*, with the biggest differences being that the steering is a lot smoother, and that the environment was solved even faster. In this and the next reward formulation, we have set $\alpha_{max} = 20$, meaning that the angle reward will equal zero if we deviate more than 20° from the direction of the road. The first observation we make when looking at Figure 4.8, is that this model had a significant peak at the 7h mark, achieving a max distance of 3714m in $\sim$ 8 hours. This model solved the environment 5h faster than *Reward 3*. However, after the initial peak in performance, we observed a dip, similar to that of *Reward 2*. This may have occurred due to some temporal noise interfering with the training process, however, the cause is not clearly known.

If we compare the driving behaviour of this agent to the previous agents, we find that:

- Steering behaviour has been significantly improved. The car is no longer overcompensating its steering when it is centering itself in the lane, and it is driving more centered as a result (*e.g.* 1:05:10.)

- Steering is, however, unstable when going downhill (*e.g.* 1:05:20.) We observe the car starting to turn left and right as it is going downhill, possibly in an attempt to slow itself down.

- The agent has a tendency to cut corners (*e.g.* 1:06:40, 1:06:48.)

In terms of driving behaviour, we would say that this is the best agent so far: it does not steer erratically, and keeps to the speed limit. Policies which overcompensate when they try to center themselves make the agent have an overall high angle difference compared to one that drives straight. We have shown here that simply adding a term that gives additional reward when we are perpendicular to the road is very effective at reinforcing desired driving behaviour. As a result, we will continue to use an angle reward in the following reward functions.

Reward 5 – Multiplied Centering, Angle and Speed Rewards:

As we discussed in Section 3.4.3, multiplying rewards may enforce the agent to make sure it fulfills multiple criteria at once while it is driving. In this case, we want the car to have speeds close to target speed, drive in the center of the lane, and drive with the body of the car aligned with the road.

This agent achieved a max distance of 3721m at the 9 hour mark. Looking at the video, we can make the following observations:

- The agent's steering is more stable than *Reward 4* (*e.g. Reward 4*: 1:05:10, *Reward 5*: 1:10:34.)

- The agent cuts fewer corners than *Reward 4*, but does make some wide turns – turning onto the grass or into the neighbouring lane (*e.g.* 1:11:38, 1:12:10.)

- Speed and steering much better when going down hill when compared *Reward 3* and *4* (*e.g.* 1:10:41.) This is probably one of the results of doing multiplication instead of addition, because in this reward formulation, the agent will be given a negative reward for any speed over the max speed, unlike *Reward 4*, where the final reward could potentially be positive depending on magnitude of the other terms in the equation.

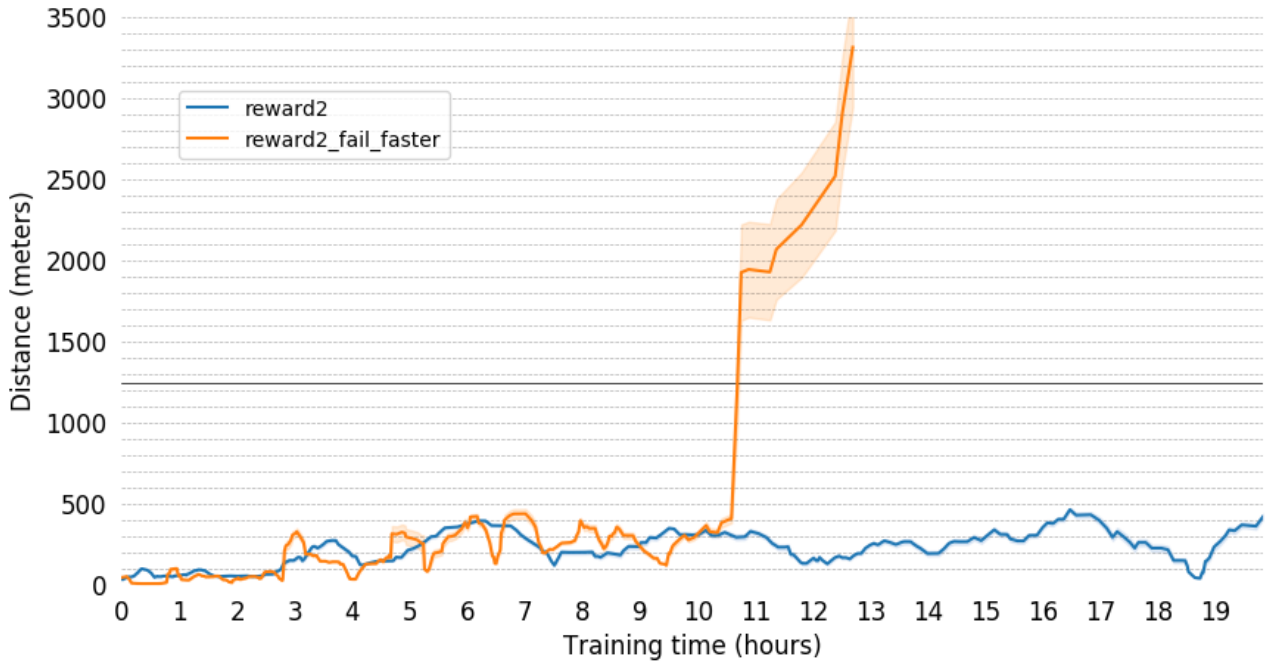


Figure 4.10: Shows the distance traveled by the agent before an infraction occurred, in evaluation mode. The graphs compare two agents that were trained using *Reward 2*; the *fail faster* agent is trained in an environment that resets the car to the latest checkpoint, while the other agent is reset to the track's starting position at the start of each episode.

Comment on Centering Metric:

In addition to recording the total distance traveled per episode, we also recorded the average distance to the center of the lane, averaged over all timesteps of the environment (Figure 4.9(a).) This sounds like a reasonable statistic in theory, however, if we look at the average centering deviation statistic for *Reward 3* and *Reward 4*, we find that *Reward 4*'s best run had a 0.53m average distance to the center, while *Reward 3*'s best run had a value of 0.54m. Looking at the video's side-by-side, we know that *Reward 4* has a much more desirable driving behaviour than *Reward 3* (*e.g. Reward 3*: 59:12, *Reward 4*: 1:05:10.), meaning that this metric is not very useful in comparing steering behaviours. Normalizing the mean by the variance of all sampled distances would most likely lead to a better metric.

4.2.2 Failing Faster

In this experiment we wanted to find out what the effect of resetting the agent in a way that makes the agent fail faster has on the overall training time. In early experimentation phases, we had started off by training agents in an environment that resets the agent to the start of the track each time it encountered an infraction (*Reward 2* in Figure 4.10 shows one of these experiments.) We noticed that the agent started to struggle at a very specific point on the track – the curvy road on the right side of the map – after approximately 3 hours. In each episode after that, we would see the agent spend $\sim$ 2 minutes to get back to the point it was struggling with, where it would then fail almost instantly (showcased in the video at 10:50.) In order to overcome this obstacle, the agent needs to either (1) repeatedly attempt this stretch of road, and, by the help of lucky values sampled from the exploration noise, sample actions that lead to better rewards, or (2) experience similar stretches of roads to eventually generalize to this road as well. The results in the graphs suggests that our agent is not able to generalize by redoing the first section of the lap. This might be a result of this turn being the first turn where buildings taking up major parts of the input image. Up to this point, the agent might not understand that the buildings it sees in the distance should not affect the policy, and it most likely has to redo this section several times over to learn this fact. Having to drive all the way to this point every time is, however, quite sample inefficient, as most of our samples along the path do not teach us how to overcome this particular obstacle.

To counteract this effect we introduce checkpoints, as described in Section 3.4.2. Figure 4.10 shows us a comparison of this environment design decision, and furthermore shows us that this is most likely a good idea. We can see here that, unlike *Reward 2*, the fail-faster model manages to complete a full three laps in $\sim$ 11 hours, while the *Reward 2* model is still stuck after 20 hours. *Reward 2* would likely manage the track given more training time, however, reducing training time is one of our main goals, so we used checkpoints in all the other experiments as a result.

Note that these agents were part of an earlier set of experiments where $v_{target} = 10$ and $\sigma_{init} = 0.4$.

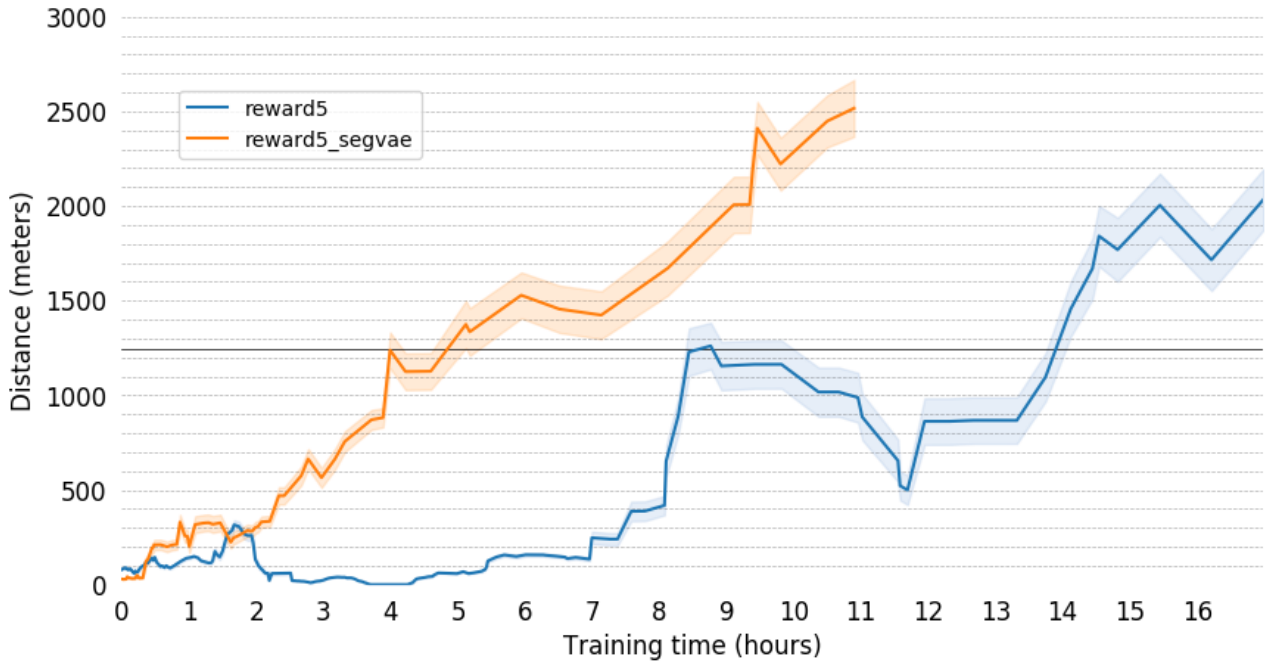


Figure 4.11: Shows the distance traveled by the agent before an infraction occurred, in evaluation mode. The graphs compare two agents trained with *Reward 5*; the *segvae* agent uses a VAE that was trained to reconstruct the semantic segmentation maps, while the other uses a regular RGB VAE.

4.2.3 Variational Autoencoder and Segmentation Maps

In this section, we will look at the effect of trying to train a variational autoencoder to extract stronger semantic features from the environment, by training it to reconstruct semantic segmentation maps from input images rather than using the standard image reconstruction pipeline. Figure 4.11 shows us the results of this experiment. The *seg-vae* model reaches a maximum score of 3745m at 10.5 hours, while the *rgb-vae* model reached a score of 3739m at 10.4 hours. The first thing to notice in the results, is that the *seg-vae* model has a very steady and reliable increase in performance from start to end, while the *rgb-vae* model is a bit more turbulent, reminiscent of earlier results. Furthermore, the *seg-vae* model learns to drive decently much faster than the *rgb-vae* model, suggesting that our hypothesis that an agent that understands the semantics of its environment is going to be more effective at solving the task. Intuitively, it is much more important for the agent to understand what the different object in the scenes represent in terms of semantics, rather that the general shapes and colors of said objects. By training the agent this way, we eliminate the need to represent texture and colors in the latent

Figure 4.12: Shows the reconstructions generated by the *rgb-vae* model as we anneal the latent space vector z by ± 10, where each figure represents one dimension of z. Note that the VAE we used in this plot was trained on data from the updated *Town07* (see Section 4.2.4.)

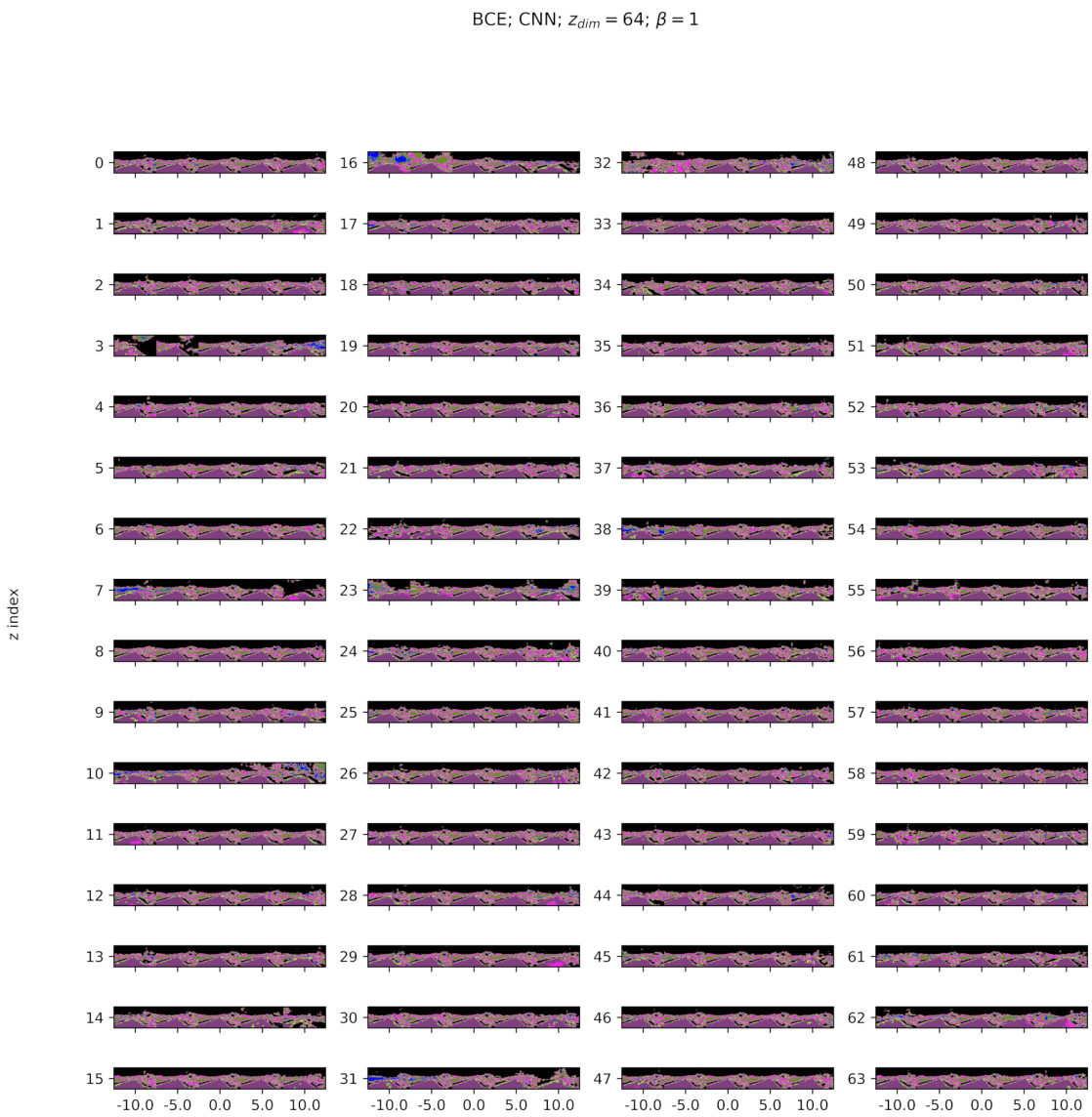


Figure 4.13: Shows the reconstructions created by the *seg-vae* model as we anneal the latent space vector z by ± 10, once for each z_{index}. Note that the VAE we used in this plot was trained on data from the updated *Town07* (see Section 4.2.4.)

space vector, significantly simplifying the latent space representation. We can see this effect in Figure 4.13; where the *seg-vae* reconstruction tend to have continuous flat surfaces in the reconstructions, while the *rgb-vae* have to use several dimensions that are used to represent noise and texture (*e.g.* $z_{index} = \{35, 36, 63\}$ in Figure 4.12).

Note that in this experiment and onwards, we have increased σ_{init} to 1.0 because we encountered some intermediate experiments failing due to getting stuck in local minima.

If we look at the video results of the *seg-vae* and the *rgb-vae* models, we can make the following observations:

- Both agents maintain speeds that are stable and close to target speed, even when going downhill.
- Both models have smooth turns and do not cut corners or make wide turns.
- The *rgb-vae* agent overcompensates more than *seg-vae* agent when it is trying to center itself in the lane (*e.g.* 13:30.) This may be a result of the *seg-vae* model having a better and less noisy understanding of where the road ends and begins. The *seg-vae* model is therefore more aligned with the center of the lane overall.
- Both agents change their steering angle at periodic intervals. This seems to be more of an effect of increasing $\sigma_{init} = 1.0$ than anything else, since this is the only difference between *Reward 5* of the previous section, and the *Reward 5* in Figure 4.11.

4.2.4 Sub-policy Model

Environment Changes

As we were writing the code for this experiment, we found a need to update to CARLA 0.9.5 due to some issues in the path planner in 0.9.4. Along with the update, the map we use, *Town07*, also got a visual upgrade. Since this new map emulates the visuals of a real-life driving scenario more closely, we have decided to use the update map for the rest of the experiments. Since the

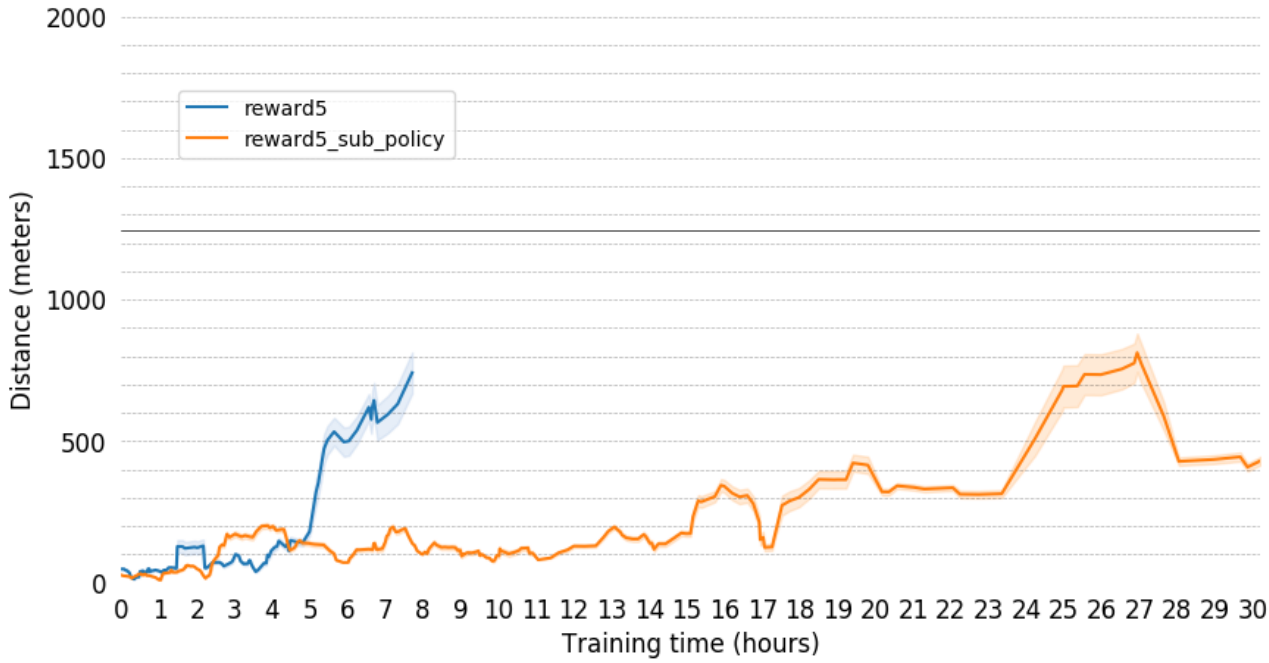


Figure 4.14: Shows the distance traveled by the agent before an infraction occurred, in evaluation mode. The graphs compare two agents trained with *Reward 5*; the "sub-policy" agent has four sub-policy networks – one for each maneuver – and the other agent uses a single policy like before.

way we train our agent remains practically the same, we have no reason to expect the overall conclusions of the earlier experiments to be drastically different in the new map. Furthermore, we also decided to change our environment into a synchronous environment, meaning that the simulation will wait until we submit an action each step. We do this to eliminate the chance of temporal noise interfering with our results in the rest of our comparisons. For all of the experiments using a synchronous environment we set $FPS = 30$, meaning that $1/30s$ will pass between each step.

Results

Figure 4.14 shows the result of training two *Reward 5* agents; one with a single PPO network, and one with four PPO networks. The vanilla network achieved a max distance of 2075m at $\sim$ 7.7 hours, while the sub-policy network achieved a max distance of 2232m in $\sim$ 26 hours. Note that we decided to train the sub-policy network a fair bit longer, because we practically have to train four separate PPO networks. Furthermore, we decided to increased α_{max} to 180°,

to make sure that the agent will be given some reward signal, even when $|\alpha_{diff}| > 20$.

If we look at the video result, we can make the following observations:

- The agent is able to learn in the new map; it keeps a speed around the target speed and is able to follow the road and make turns (*e.g.* 21:46)
- Steering is, however, more noisy in both of these experiments when compared to *Reward 5* from Section 4.2.1. This might be a result of trying to learn in an environment that is more visually complex environment, or it can be a side-effect of training in a synchronous environment, as we will discuss later.
- Both of the agent's best runs ended because the agent collided with a sign pole or other objects along the roadside (Single policy: 26:47, Sub-policies: 27:34.) We could have prevented this by, for example, reducing d_{max}, or by giving the agent a negative reward on collision.
- The models have similar turning patters; they make some wide turns, but generally turn smoothly.

The results show that training separate PPO network for different maneuvers is a valid strategy. The sub-policy agent reaches a max distance comparable to the vanilla network. While the sub-policy network takes more than 3x as much time to train, we now have the option to train agents that have the ability follow commands, a la Codevilla [CMD+17]. This opens up the possibility to train agents that can navigate from a point A to another point B, by use of a global route planner that finds the best route between A and B, and then having the PPO agent execute the driving part. We will explore this option in Section 4.3.

4.2.5 Note on Exploration Noise

During all of the CARLA experiments, we tried a variety of different values for σ_{init} with several different models. Each value of σ_{init} had its own pros and cons, and we were unable to find the

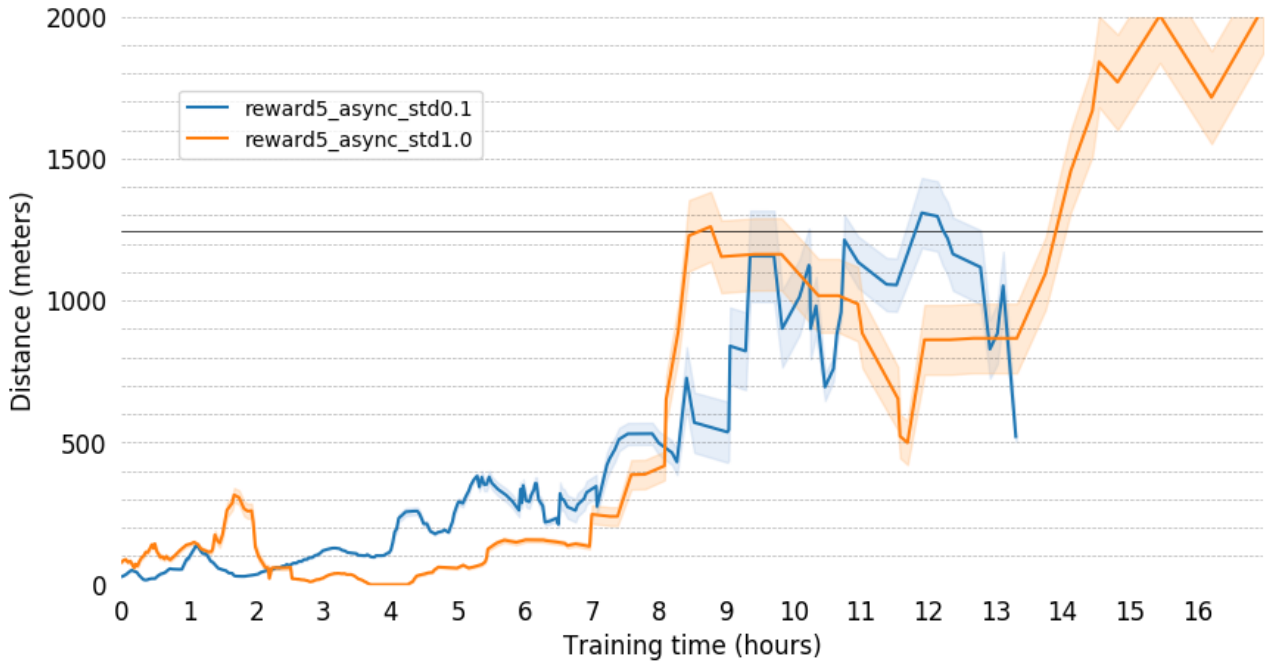


Figure 4.15: Shows the distance traveled by the agent before an infraction occurred, in evaluation mode. The graphs compare two agents trained with *Reward 5*; one with $\sigma_{init} = 0.1$ and the other with $\sigma_{init} = 1.0$.

perfect value for σ_{init}. When σ_{init} is low (*e.g.* $\sigma_{init} = 0.1$,) 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 a local minima, and training will become slower because the sampled values are generally close to the mean. When σ_{init} is high (*e.g.* $\sigma_{init} = 1$,) we end up with an agent that is more jerky in its steering, but we also reduce the risk of ending up in local minima and the agent will train faster. *Reward 4* from Section 3.4.3 shows that we can counter some of the steering angle instabilities by designing reward functions that take this effect into account, however, more work is necessary to counteract this behaviour completely (*e.g.* we could try to keep a history of recent values for the distance to center, and penalize the agent for having high variance in these values.)

Another issue we observed with the standard Gaussian exploration noise, is that it is difficult to make the agent "commit" to a control signal long enough for the agent to observe its effects. For example, if the agent is stuck on a sharp turn, what will happen with regular Gaussian noise is that our agent will most likely repeat bad actions and be slowly pushed to go in the right direction over time due to minor perturbations in the actions. However, when here are

second-order relationships between the actions and the changes in states, we need to be lucky enough to repeat the same action multiple times in a row in order to observe these changes, which is unlikely when we are sampling from a Gaussian. In our case, the speed of the car is the derivative of the throttle, and the momentum (direction) of the car depends on a combination of momentum, torque and steering angle. If the agent would able to commit to an action for longer time (*e.g.* "this time, try to turn right more than left for some period of time,") it is more likely to reach a good policy faster. In DDPG [LHP+15], this fact is actually accounted for by the use of Ornstein-Uhlenbeck noise instead of Gaussian noise. An Ornstein-Uhlenbeck process models the velocity of a Brownian particle under friction, and can be used to sample random variables that have a second-order relationship to the underlying state, *e.g.* how should we sample velocity when we want to sample the position from a normal distribution. Using Ornstein-Uhlenbeck noise with PPO is not as easy as using it with DDPG, because PPO needs to calculate the log-probability of an action given a policy, that is calculate $\log \pi(a_t|s_t)$. Chen *et al.* [YTXC09] has derived the formulas for calculating the log probability of a Ornstein-Uhlenbeck distribution, and it is certainly an interesting extension to consider.

Figure 4.15 shows that setting $\sigma_{init} = 0.1$ generally achieves the same distances as $\sigma_{init} = 1$, however, the best run of $\sigma_{init} = 0.1$ has a centering deviation of 0.33m, which is lower than the best run of $\sigma_{init} = 1$, which has an average centering deviation of 0.56m. The increase in average centering deviation is to be expected, because the policy is has a more aggressive exploration rule. Comparing the video results, we can also confirm that setting σ_{init} to 1 results in an agent that only changes its steering angle periodically (see video at 27:58.) The cause for this is unknown, however we speculate that the $\sigma_{init} = 1$ agent is unable to find policies that use fine steering controls due to the aggressive noise, leading to an agent that only adjusts the steering angle in bursts.

4.2.6 Note on Environment Synchronicity

Intuitively, we expect a synchronous environment to generally perform better than a asynchronous environment, due to the temporal stochasticity that is inherently introduced in an

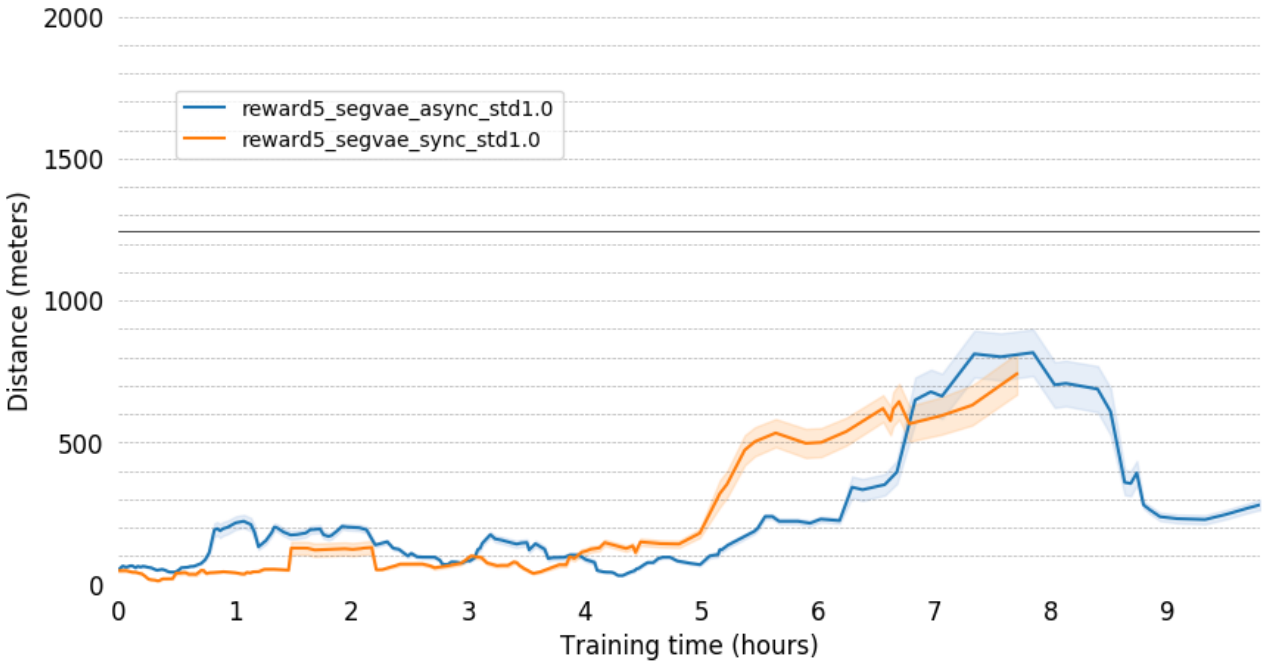


Figure 4.16: Shows the distance traveled by the agent before an infraction occurred, in evaluation mode. The graphs compare two agents trained with *Reward 5*; one agent in a synchronous environment and the other in an asynchronous environment.

asynchronous environment. Initial experimentation showed, however, that only some configurations of reward functions and hyperparameters were easier to solve in the synchronous environment, while most configurations were actually harder and did not amount to well performing agents. We suspect that the temporal noise in the asynchronous environment may actually have been to the agent's benefit, as the agent would generalize better when the outcome of an action is non-deterministic, giving the agent more variety in its data. As discussed in Section 3.4.2, training asynchronously has some benefits in terms creating agents that could, theoretically, learn to drive in real-time. However, we think it is more valuable to train synchronous agents in autonomous driving research, as it makes it easier to compare and contrast results between studies.

Figure 4.16 shows the results for *Reward 5*. Inspecting the graph, we can see that the agents in both the synchronous and asynchronous environments learn at a similar pace; with the synchronous environment increasing in performance slightly faster than the asynchronous one. The synchronous environment reached a max distance traveled of 2075m in $\sim$ 7.7 hours, while the asynchronous one reached a peak distance of 2708m in $\sim$ 7.9 hours. Comparing the video

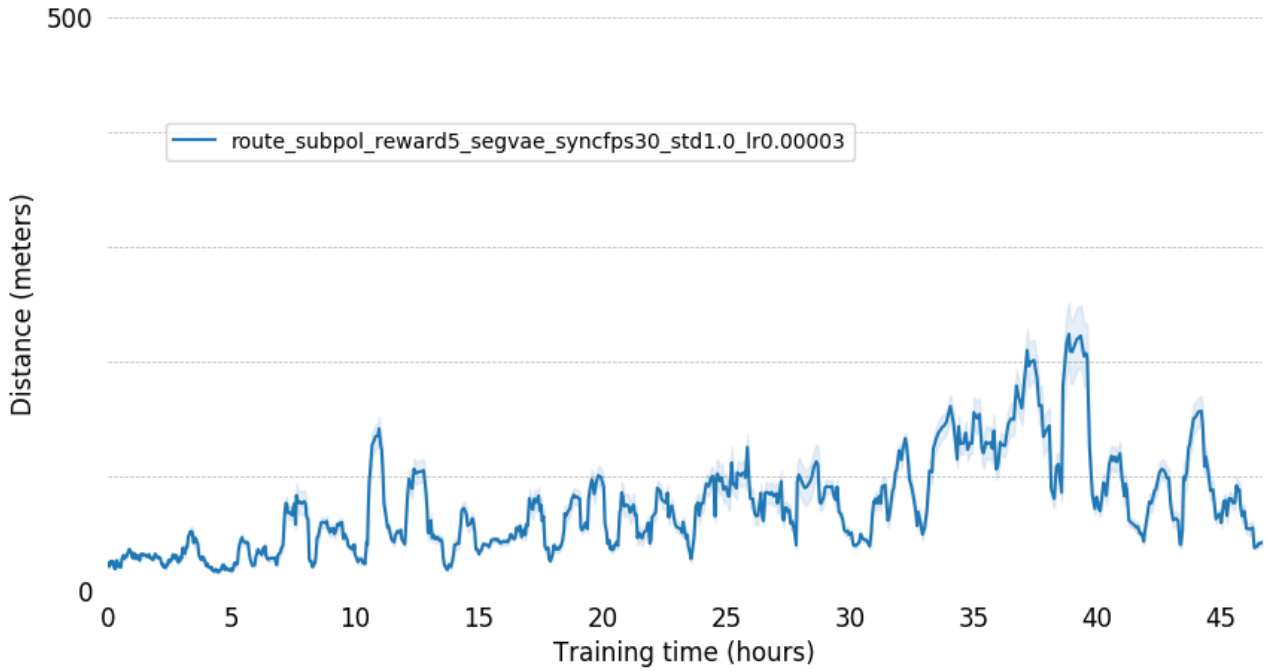


Figure 4.17: Shows the distance traveled by the agent before an infraction occurred, in evaluation mode. The graphs shows the result of agents trained with *Reward 5* in the CARLA route environment. Note that a sliding window of ±5 was applied to smooth out the graphs.

results, we can actually see that temporal noise is what caused the asynchronous agent to fail in this case (start watching 44:04, lapse happens at 44:07.) A couple of frames before the episode ends, we can see that there is a short lapse in the recording – likely due to a lost step/frame. A lapse of one frame has the duration of approximately $1/15s = 67ms$, which, in an environment where fast reaction times are vital, is certainly enough time for the agent to reach unrecoverable states. In the video we see that an ill-timed lapse was, indeed, enough to cause the agent to crash into a pole on the side of the road. In terms driving behaviour, the asynchronous agent tends to overcompensate less in its turns. The reason for this is not clear, however, it could be a result of noise working in the benefit of the agent; the agent has some probability to recover from an unstable state when a timestep is shorter or longer than expected.

4.3 Carla Route Environment

Finally, we tested our agent in the CARLA route environment. As stated in Section 3.5.2, we will be using the *seg-vae* variational autoencoder, *Reward 5* with $\alpha_{max} = 180°$, and four

sub-policies. Similar to the *Lap environment* sub-policy model, we let this model train for ~ 48 hours, and we additionally lowered the learning rate to 3e-5, to compensate for the fact that finer controls are most likely necessary to for agents to be successful in this environment.

Figure 4.17 shows the results for this agent. Looking at the graph, we can see that there is a clear upward trajectory in performance, and by inspecting the videos (starts at 6:47,) we can see that the sub-policy model is able to learn the different maneuvers. However, the agent is still quite a ways from completing 3000m, with the best run of this agent finishing 930m at the $\sim$ 39 hour mark. If we look at the path this and other high performing agents take, we quickly realize that the agents who complete the longest distances are also the agents that are tasked to drive long stretches of easy to drive, straight roads. Therefore, we have also included additional video examples that better showcases the agent in action.

By looking at the videos, we observe that:

- The agent is able to make a variety of left and right turns, including a right turn onto a small roundabout (10:22,) keeping to the right or left lane at branches (10:19,) driving any direction in open four-way intersections (9:50,) etc.
- The steering overcompensates in a similar way to what we saw in the sub-policy model in the *Lap environment*.
- The agent does, however, occasionally cut corners, driving on the grass (*e.g.* 9:50.)
- The agent sometimes crashes into sign poles and fences (*e.g.* 10:35,) however it also recovers from these crashes occasionally (*e.g.* 10:22.)

Chapter 5

Conclusion

In this thesis, we have explored the several aspects of how deep reinforcement learning can be used to solve and create autonomous driving agents. We have verified previous research which suggests that variational autoencoders can be used to help deep reinforcement learning agents learn faster, and provided extensive analysis as to how different state representations affect the overall performance of agents. Furthermore, we have shown that this method is not only limited to 3D environments that are visually simple – such as Raffin's [RS19] Donkey Car simulator and Wayve (Kendall) [KHJ+18] custom in-house simulator. While we encountered some challenges in setting up the final environment in CARLA, we have ultimately created a self-contained deep reinforcement learning system that works with CARLA, and have provided in-depth analysis of the effects of multiple parameters of the system, such as the choice of reward function, standard deviation, environment synchronicity, and environment checkpoints. Finally, the experiments in the *Route environment* serves as a simple proof of concept showing that an agent with sub-policies can be used to learn how to navigate to a destination waypoint by switching out the current policy according to the route planner's commands.

5.1 Contributions

In the following list, we have summarized our contributions to the field of deep reinforcement learning for autonomous vehicles:

- We have created two gym-like environments for CARLA, one that focuses on following a predetermined lap, and another that is focused on training agents that can navigate from point A to point B. The code for both of the environment are publicly available at https://github.com/bitsauce/Carla-ppo. While we can find existing examples of gym-like environments for CARLA, there is no implementation that is officially endorsed by CARLA. Furthermore, most of the third-party environments do not provide an example of an agent that works out-of-the-box, or they may use outdated reinforcement learning algorithms, such as Deep Q-learning.

- We have provided extensive analysis of various PPO parameters and environment design decisions, with the aim of finding the optimal setup to train reinforcement learning based autonomous driving agents.

- We have provided in-depth analysis of several aspects of the design of reward functions for the purposes of training autonomous driving agents.

- We have provided an example that shows how VAEs can be used with CARLA for reinforcement learning purposes.

- We have shown that how we train and use a VAE can be consequential to the performance of a deep reinforcement learning agent.

- We found that major improvements can be made by training the VAE to reconstruct semantic segmentation maps instead of reconstructing the RGB input itself. Training the VAE this way ensures that the VAE has a greater focus on encoding the semantics of the environment, which further aids in the learning of state representation learning-based agents.

- We have used our findings to devise a model that can reliably solve the CARLA lap environment in approximately 8 hours on average.

- We have provided an example of how sub-policies can be used to navigate with PPO, and we found it to have moderately success in the route environment.

5.2 Discussion and Future Work

In this section, we will be discussing and summarizing our findings, and furthermore, we will be discussing what improvements can be made to our work, and give our thoughts on what we believe future work in deep reinforcement learning for autonomous vehicles should pursue in order to elevate the field.

5.2.1 Variational Autoencoders with Deep Reinforcement Learning

As we have discussed in Section 3.2.4, we can use a variational autoencoder that tries to reconstruct its input image, as a way to compress our input image into some low-dimensional feature space that a deep reinforcement learning agent can use to train with. We have shown that employing a VAE certainly helps compared to training directly on input pixel values. Furthermore, we have shown that how we train the VAE also affects the end result of the agent, and that a trained VAE performs much better than a random VAE, suggesting that only having a stationary state space is not sufficient for an agent to learn – the features also need to have some semantic meaning. Out of the VAE variables we tested, we found the size of the VAE's encoded space, z_{dim}, to be one of the most influential variables, and we found that it should be large enough for the VAE to encode all possible features of the input. In visualizing the reconstructions, we saw that even though it might appear like the $z_{dim} = 10$ model encodes just as much information as the $z_{dim} = 64$ model, and that the $z_{dim} = 64$ has a lot of features that do not encode anything apparent, the $z_{dim} = 64$ performs much better; something that we think could a result of some features being correlated – and therefore will have no effect in

the absence of another – or that the features might model noise or other perturbations that are hard to catch with the eye. Additionally, we also provided some high-level analysis of what the individual encoded features mean in the *CarRacing-v0* environment, and also showed that models with higher β values or lower z_{dim} values tended to generate features that are more entangled than others. We found BCE loss and MSE loss to have similar results, and showed that pre-training the VAE on manually collected data worked a lot better than optimizing the VAE and the PPO agent in alternate training phases.

In Section 4.1.2 we also looked at optimizing PPO to be used with a VAE in the *CarRacing-v0* environment, and we demonstrated that the on-policy deep reinforcement learning algorithm also works in the VAE state representation learning pipeline. In our hyperparameter search, we compared the Atari and MuJoCo parameters from the PPO paper [SWD+17], and found the Atari parameters to work the best with our implementation. We are uncertain why the MuJoCo parameters performed so poorly, and suspect that learning rate is the biggest factor, as we found the learning rate to be one of the most influential variables in our Atari experiments. We found that lowering the learning rate helped significantly, however, we did also not use a decaying learning rate like they did in the PPO paper, because we were unsure how long we would need to run the simulations for. This might be the biggest factor as to why we had to lower the learning rate.

As we had shown in our earlier report (Appendix A.1,) squashing the action space to the minimum and maximum values for each action substantially aids in learning. This is also corroborated by OpenAI's Spinning Up documentation [Ach19]. Furthermore, we found little difference in using a finite or an infinite horizon, which means that we could most likely use our method to train a PPO driving agent in real-life by using an infinite horizon, as using infinite horizons allows the agent to finish the entire episode before optimizing.

Some experiments that would have been interesting to try out, would be, for example, to run our implementation with the best parameters in an unmodified *CarRacing-v0*, to evaluate what the effect of making the environment "easier" to solve had on the training time. It is not obvious that modifying the environment itself (*e.g.* making the turns softer or limiting the speed) had

a great impact on the training time, and, furthermore it is also not clear what the additional complexity of adding another action is to a PPO agent, even when that action is rarely required. Lastly, we are not entirely certain why we got different results from Higgins *et al.* [HPR+17] when we tried to train an agent with $\beta = 4$. After running the *CarRacing-v0* experiments, we did not spend more time trying to finding the answers to these questions, since our main focus was to get a PPO agent working with CARLA.

5.2.2 CARLA Lap Environment

After we found a pipeline that seemed to perform reliably in *CarRacing-v0*, we set out to design an OpenAI gym-like environment, that we can use to train reinforcement learning agents in CARLA. The first environment we created is one that emulates *CarRacing-v0* and other driving environments where the goal is to simply follow the road (*e.g.* Kendall's and Raffin's,) and the details of our environment are laid out in Section 3.4.2. In our environment, we have had a focus on making it easy to try out custom reward functions and state representations, while providing several metrics that can be used to compare agents. Furthermore, we have provided a reward function that we found to perform reasonably well, and it could serve as a baseline for future experiments. In addition to providing some analysis in the choice of our final reward function, we also have included analysis of five other reward function in Section 4.2.1, including Kendall's speed-as-reward reward function. In this section, we showed that too aggressive termination can be detrimental to the learning, and that creating a reward function that gives maximum reward when our agent's speed is equal to the target speed is an effective way of controlling our agent's maximum speed. In later reward functions, we showed that providing some leeway in the speed term of the reward function gives the agent the opportunity to explore the effect of steering angle on the total reward, independent of the noise from the speed term. At this point we still had a problem with the agent overestimating how much it should be turning to center itself in the lane, leading to an agent that drives in a zig-zag pattern without reaching an equilibrium. To combat this, we tried to train an agent with a reward function that also considers the angle between itself and the road, and found that to have great effect in

stabilizing the agent's steering. Finally, we thought it would make sense to try and change the reward function into one that will only give high rewards when multiple criteria are met (*e.g.* "we are driving in the center," and "we are driving at target speed",) and tried to accomplish this by making a reward function that multiplies the different reward terms instead of adding them. We found this to help our agent learn faster without collapsing after the initial peak in performance, which could be useful for giving the agent time to fine-tune its behavior in the later stages of the training process.

One question we had after our initial reward function experimentation was why the PPO agent learns much slower in this environment, compared to *CarRacing-v0*. The PPO agent needed only 15 minutes to learn to drive in *CarRacing-v0*, while in our *CARLA lap* environment, the agent needed approximately 8 hours on average to learn to drive well. Looking at some of the results from the earlier episodes in the training of the CARLA agents, we actually see that the agent does not need a lot of time to learn to drive 200m on straight roads (approximately 15 minutes.) However, as soon as the agent encounters the first intersection, we saw the agent fail repeatedly for potentially many hours before being able to make the turn. This observation is what led us to adding checkpoints to the lap environment, as to encourage the agent to fail faster in training mode. In reality, we believe that the main reason it takes much longer to train in our environment, is because it is deceptively more difficult than *CarRacing-v0*. Our modified *CarRacing-v0* does not feature any sharp turns or branching roads at all, so holding the throttle at a constant level is a valid strategy in this environment. Furthermore, there are a lot more small details in the observations that have a lot of importance in the CARLA environment, compared to *CarRacing-v0*. The location of the lane lines, locations of object we can collide with, and the curvature of road that is off-camera are all necessary pieces of information that our agent needs be able to drive properly in CARLA, and a 160x80 pixel RGB image may simply not contain enough information. In general, we believe that improving the state representation by, for example, adding memory to the agent through recurrent networks, or fusing input between different types of sensors such as LiDAR and RGB cameras, can go a long way in improving the performance of our baseline agent.

To make sure that our final environment setup and agent parameters were justified before

continuing onto the route environment, we did some analysis of the effect of different values for σ_{init} and environment synchronicity. We found that using low values of σ_{init} (*e.g.* $\sigma_{init} = 0.1$) in our agents, made it more likely that they would get stuck in a local minima. Using higher values of σ_{init} (*e.g.* $\sigma_{init} = 1$) alleviated this problem, but had the side-effect of introducing undesirable steering patterns in our agents. We concluded that setting $\sigma_{init} = 1$ was still a worthwhile trade-off to make our agent's learning more consistent from run to run. In the same vein, we found evidence that training agents in asynchronous environments introduces temporal noise in the observations, which is undesirable as it can lead to inconsistencies between runs. The driving behaviour of the agents trained in the synchronous and asynchronous environment did not vary drastically, so we concluded that using a synchronous environment is more useful to autonomous driving researchers, as it makes it easier to reproduce and compare results.

In one of our experiments, we showed that training the VAE to reconstruct the segmentation maps, rather reconstruction the RGB input images, helps reduce training time by a significant portion. When we train the VAE on semantic segmentation maps, we ensure that the compressed state representations will encode the semantics of the roads and the objects in the scene, rather than the texture of surfaces or other noisy patters. We believe that the segmentation VAE's representations are more useful to the agent when solving an autonomous driving problem, and we think it would be interesting to explore similar ways of enforcing information rich state representations that can be used with deep reinforcement learning. For example, we have considered training a VAE model that only trains on segmentation maps with road and non-road classes only – since being able to distinguish between the 12 classes presented in the segmentation maps may not be important in determining the optimal driving behaviour (*e.g.* vegetation.) Other than that, we could also look into training models that are specialized for semantic segmentation, and try to use their compressed state representations when training the agent.

Another contribution of our work is that we have shown that it is possible to train PPO-based agents with multiple sub-policies, which lets us control the vehicle at intersections – inspired by the method of Codevilla *et al.* [CMD+17]. This is allows us to train agents that navigate from a starting point A to an end point B, with the help of a global path planner. We showed

that this model is able to perform similarly to the single-policy model, at the cost of having to train the agent for longer. We might have expected the sub-policy model to perform better, since it has overall more parameters, so it would be interesting to investigate why it did not happen in our case.

As we were developing the lap environment, and after having run some of the initial experiments, we found that several aspects of the environment and experiments could be improved. As seen in the video, many of the earlier experiments were run in the version of Town07 from CARLA 0.9.4, and also in the asynchronous version of the environment. As such, it would be ideal to rerun some of the earlier experiments to make them more reflective of how these agents would perform in a more visually complex environment. Also, changing to a synchronous environment would make the these results easier to reproduce, as we would eliminate temporal noise as discussed before. After we had updated to the new environment, we noticed that the new agents had a tendency to crash into obstacles along the sides of the roads, due to the way the car learns to steer itself. This is the main reason, we believe, that the agents trained in the updated map did not always learn to complete three laps like before. This is a problem that we would like to solve, and our initial thought is that this issue may resolved simply by reducing d_{max} or σ_{init}, or adding a negative reward on crashing (none of these have been tested.) Other improvements we have considered to the environment are, for example, to reintroduce braking to the action vector, adding orientation to the state vector (so that the agent can understand movement due to gravity,) implementing traffic rules, supporting multiple agents, etc. One result that is also worth revisiting is the fact that *Reward 1* actually performed better than *Reward 2*, suggesting that giving the agent a reward for staying centered may not be as useful as predicted. Furthermore, designing reward functions that considers temporal aspects of the agents behavior might be a good solution to prevent the agent from having unstable steering behaviour; *e.g.* one could take a sample of the 10 last distances to the center of the lane, and calculate a reward based on the variance of these datapoints.

Improving exploration is also something we believe could be effective, particularly because the agent is acting in an environment where there are second-order relationships between its actions and its future observations. When the agent applies throttle, a change in speed will

not be observed instantly, and how the agent should steer in the moment depends heavily on the momentum and other physical properties of the system. We argue that Gaussian noise is ill-equipped for these types of scenarios, because the agent needs to commit to a particular action for longer periods of time in order for the agent to experience the consequences of its actions. As such, we believe that changing the Gaussian noise in PPO with Ornstein-Uhlenbeck noise could help. Another idea would be to consider curiosity-based reward function to aid in exploration.

5.2.3 CARLA Route Environment

In our final experiment, we wanted to test the capabilities of our model in an environment that simulates real-life driving scenarios closer. In doing this, we had to create another version of the environment that tasks the agent to follow the commands of a global route planner, with the goal of navigating from some point A on the map to some other point B. The details of this environment is laid out in Section 3.5. Like before, this is an environment that is easy to customize and use, however, certain changes had to be made to accommodate for the fact that the randomly selected routes can have variable difficulties and lengths. To make evaluation runs easier to compare, we made it so that the agent will be given a new route upon route completion, until the agent has traveled 3000m in total – a successful terminal state. We demonstrated that the final agent we trained was able to follow the instruction of the global route planner, making somewhat complicated maneuvers, such as driving onto, following, and exiting a roundabout. However, the agent exhibited similar behaviour to the other agents trained in the updated Town07, crashing into object along the side of the road and having unstable steering. Like before, we believe that that a simple solution to the crashing could be to lower d_{max} or σ_{init}, however, improving the state space by adding memory, training a specialized VAE, or designing reward functions that consider temporal aspects of the agents behavior be more fruitful. Unique to the route environment is the fact that it is possible for the agent to end up ambiguous states which is a result of only having a single image as input. In certain configurations (even with a sub-policy model,) it is impossible for the agent to know

if it should turn or drive straight, because the start of a turn can look similar to the end of another turn, but in these cases the correct actions are different. This further reinforces the idea that the state space needs to encompass a temporal aspect.

Finally, we would like to mention some improvements that could be made to this environment. The main problem we encountered in using this environment, was the fact that is was difficult to properly evaluate an agent's performance based on any single metric. Like before, we provide a metric that tells us how far the agent is able to drive each episode, however, this metric does not immediately tell us how good the agent is at maneuvering intersections, which is the main challenge in this environment. There is a chance that the agent might get "lucky," and end up being tasked with navigating a route that consists of long stretches of easy, straight roads. Instead, we should implement a metric that accurately reflects the quality of our agent, for example, a metric telling us how many successful turns the agent was able to make in an episode.

5.2.4 Comparison to Similar Work

Our initial problem statement was been inspired by the works of Kendall *et al.* [KHJ+18], Raffin *et al.* [RS19] and CARLA's Dosovitskiy [DRC+17]. Since these works are fairly similar, we wanted to discuss some of the similarities and differences in our methods.

Kendall showed that it is possible to get good results in very few episodes (even in real-time,) given that we set up the driving environment in a way that best facilitates reinforcement learning. We have shown that by using their reward function, we can achieve similar results in CARLA, however, the method needs a lot more training in our environment because the steering behaviour required to navigate curvy roads and intersections are fairly more complicated than Kendall's straight roads. Furthermore, the length of our lap is 1245m, while their road is 250m – our agent has more room for error. Like in their work, we have also restricted our models to not use any temporal aspect of the observed states, and the same is true for our reward functions. Our agent needs approximately 8 hours of training time to reach an agent that solves the environment. Through our experiments, we have confirmed that our agent is also able to learn

to drive 250m on a straight road after approximately 15m of training, however, if we want to reduce the training time for the entire track, we believe that we need to create models which considers some temporal aspects of the environment as well. Other aspects that differ is, for example, that Kendall trained their VAE by random exploration, while we pretrained ours on manually collected data, and that their agent outputs an absolute speed instead of throttle. Instead of adopting these ideas, we decided to take inspiration from Raffin, who, similar to us, pretrained the VAE on data collected manually, and outputs the throttle. We did this, as their methodology sounded more robust to us, however, we did not test this claim.

Raffin has shown similar results to Kendall in their Donkey Car simulator; teaching a donkey car (a small remote controlled car) to follow a somewhat curvy road in 5-10m of training time. Raffin also showed that a more recent reinforcement learning algorithm, SAC, can work just as well or better than Kendall's DDPG. They did not provide any analysis of how much changing the algorithm helps, other than stating that it is easier to select hyperparameters for SAC. We have shown that PPO can also be used as well, and we believe that using on-policy learning algorithms has its benefits; particularly that we can employ trust regions in order to prevent the policy from degenerating quickly. Other than that, Raffin also found that training agents with Kendall's simple speed-as-reward (in their case, throttle-as-reward) reward function made their trained agents steer in zig-zag patterns, similar to what we found. Raffin provided some solutions to this problem, and their final solution involved keeping a history of steering commands and penalizing jerky steering, in addition to clipping the steering signal to impose continuity in the agent's actions. At one point, we tried to smooth our agent's actions in a similar fashion – in our case by linearly interpolating the new action with the agent's previous action – but this experiment did not yield any interesting results. Since our priority was to get the our route environment agent working, we decided to reduce our scope, and to instead work with models that only considers one step of the environment a time. Taking inspiration from Raffin's work should, however, be considered for future research.

Finally, we would like to add some remarks on CARLA's official reinforcement learning experiments [DRC+17]. In their experiments, they have shown that they were able to train an A3C deep reinforcement learning agent to navigate from point A to point B in a similar fashion to

us, however, their model has a fairly low success rate. One of the reason their agent appears to travel quite far compared to ours, is that they do not have any termination criteria besides a time limit, meaning that the agent is free to drive off the road during training and evaluation. We believe that this has a tendency to create lower quality agents, and we have shown in Section 4.1.3 that early termination can reduce training time by a bit. In CARLA's A3C experiments, Dosovitskiy's *et al.* [DRC+17] also used a reward function that is fairly different from ours, as their reward function consists of several terms that only give positive or negative rewards when a change in the state (such as change velocity, on collision, on intruding into lane going in the opposite direction,) have occurred since the last step. Our initial reaction when seeing their reward function is that it might be difficult select the right coefficients for each of the terms in the equation, however, we encourage future work to look into their style of reward function as well. CARLA's implementation of their A3C reinforcement learning agent is, unfortunately, not available online.

5.2.5 Closing Remark

Overall, we have been able to show that deep reinforcement learning agents can learn to drive in complicated environments using very sparse information. With only a 160x80 RGB image, in addition to knowledge of the velocity and the previous control signals, our agent was able to complete three 1245m laps on the outskirts of Town07. We have shown that minor modifications in the reward formulations can have a significant impact on the behaviour of our agent, and have provided some ideas as to how we can construct reward functions to ensure desirable driving behaviour from deep reinforcement learning based autonomous driving agents. We believe deep reinforcement learning has several advantages over imitation learning, however, we think there is still a lot of research needed before we are able create deep reinforcement learning agents that are able to drive every driving scenario that we may encounter in real-life. We hope that – by providing an implementation of a CARLA-based gym environment that is bundled with a working PPO-based agent that works out out-of-the-box – that we may have filled a gap that is currently present in the reinforcement learning for autonomous driving research community.

Appendix A

CarRacing-v0 with PPO – Non-VAE Experiments

A.1 Models

Appendix A describes the set of PPO experiments that we used to determine the effectiveness of various direct optimization models in CarRacing-v0 (direct optimization meaning that the agent needs to learn to do feature extraction without a pre-trained VAE.) The details of the PPO training pipeline used in this section is identical to that of Chapter 3, besides not using a VAE. Furthermore, other state and environment details are altered as described in this appendix.

As stated in Chapter 3, CarRacing-v0 features a car in a procedurally generated racing track, viewed from a top-down 2-dimensional ("birdseye") view. In these experiments, car is controlled by a (γ, a, b) triplet of continuous action values, where:

- $-1 \leq \gamma \leq 1$ is the steering angle (in radians)
- $0 \leq a \leq 1$ is the acceleration
- $0 \leq b \leq 1$ is the brake

The state space of this environment consists of an image, preprocessed in the same way as described in Section 3.3.1. The following measurements are also exposed:

- Speed of the car
- ABS sensor values for each wheel
- Steering angle
- Angular momentum

A.1.1 Implementation Details

Code for these experiments can be found at *https://github.com/bitsauce/CarRacing-v0-ppo*.

A.1.2 Experiments

Frame stack model

Stacking sequential frames is the most straight-forward and most common way to represent the state space for a reinforcement learning agent that take images as input. As we discussed in Section 2.3.3, Mnih *et al.* found it to be essential to stack 4 sequential to solve Atari-based environments with Deep Q-learning; because a stack of sequential frames store temporal information. In this model we also opted to create a frame stack of 4 sequential frames, and we use this as the input.

Image Preprocessing: When we stack 4 96x96 RGB images on top of each other, the resulting state space becomes quite highly dimensional ($96 \times 96 \times 3 \times 4 = 110,592$ state values.) In order to reduce the size of the state space, we opted to crop and convert the 96x96 RGB frames into 84x84 greyscale images, giving us a state space of $84 \times 84 \times 3 = 27,648$ state values. The cropping removes 6 pixels from the left and right sides because these pixels typically contain information about objects that are far away from the agent and are therefore not relevant to

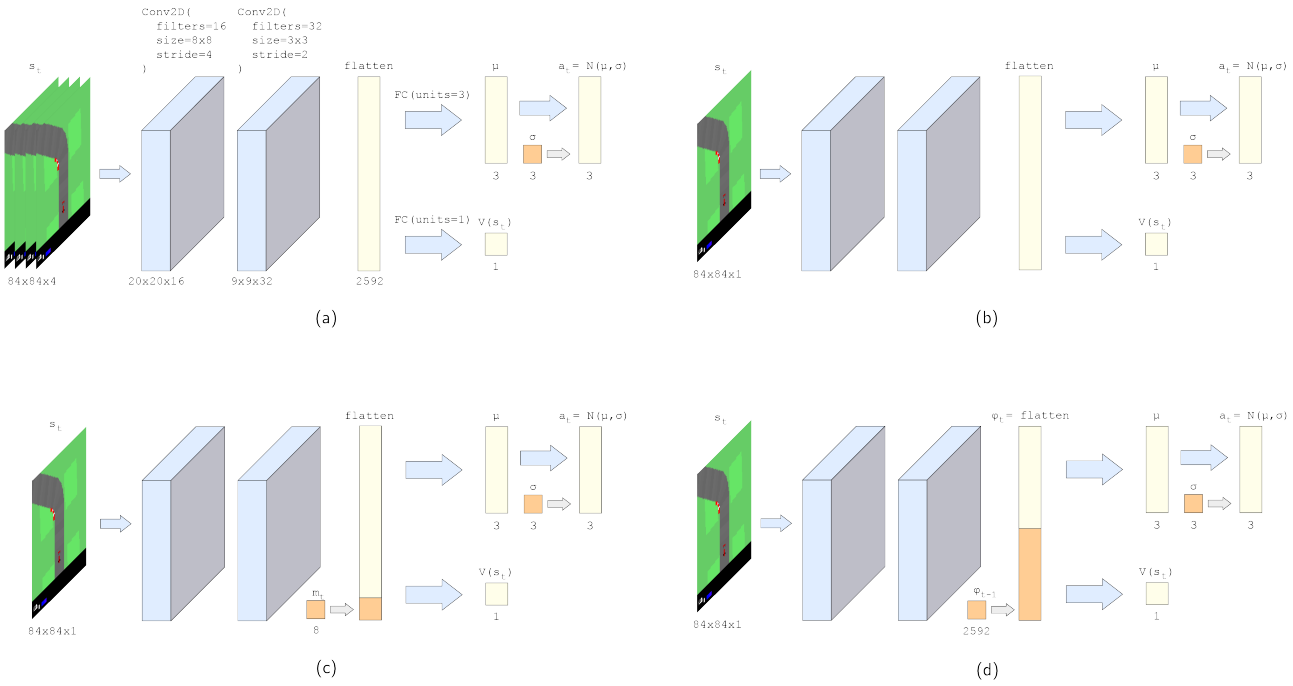


Figure A.1: Shows the 4 CarRacing-v0 models that were tested in the experiments. (a) Is the typical frame stack model with 4 stacked frames. (b) Same as (a) but with only one frame as input. (c) Car measurements are concatenated with the latent features ϕ. (d) Recurrent model where the previous step's latent features, ϕ_{t-1}, are concatenated with the current latent features, ϕ_t. Note that convolutional and fully-connected layers have the same kernel sizes and units in every model.

the agent at the moment, and we also crop the lower 12 pixels to from the image because these pixels encode the dashboard parameters, and should not be necessary when these parameters can be inferred directly from the sequential stack of frames.

Network Architecture: The network is a typical convolutional neural network with a fully-connected layer for the critic output and a fully-connected layer for the actor output. The convolutional part of the network architecture was inspired by a project report titled *Reinforcement Car Racing with A3C* [JML] by students at Stanford University's course CS234. In the report, they claim to have tested networks with 2 up to 7 convolutional layers of varying filter sizes and strides. In the end they found that deeper networks did not help the agent learn. This is probably because state space is not very complex to begin with – it consists of mostly straight lines and simple solid-colored geometric shapes as shown in Figure 3.2(b) – so a deeper network will provide little benefit. Another consideration is that deeper networks are harder to train and take longer to converge. Since is seems shallower architectures are faster to train and will preform better or similarly to deep architectures, we opted for the 2-layer architecture shown in Figure A.1(a). The first convolutional layer has 16 8x8 filters with stride 4, while the second layer has 32 3x3 filters with a stride of 2. They both use *leaky ReLU*, $f(x) = x$ if $x > 0$ else $0.01x$, as their activation functions. Note that the filters are quite big compared to the filters in deep architectures such as ResNet [HZRS15]. This is due to the fact that shallow convolutional networks need bigger filters to increase their receptive fields.

The flattened feature maps of the convolutions is also shared with the critic. The critic is represented by a fully-connected layer with one output value representing the value of being in state s, that is $V(s)$. This layer has no activation function because we want to be able to represent any possible value that R_t might have, in other words $-\infty \leq V(s) \leq \infty$.

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

Hyperparameter	Value
Horizon T	128
GAE parameter λ	0.95
Discount factor γ	0.99
Clipping parameter ϵ	0.2
Learning rate	3e-4, decaying by 0.85 every 10,000 epoch
Value loss scale α	0.5
Entropy loss scale α	0.01
Number of epochs K	10
Batch size M	128
Number of envs N	8

Table A.1: Hyperparameters used in the experiments.

$$\log \pi_\theta(a_t|s_t) - \log \pi_{\theta_{old}}(a_t|s_t) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} = r_t(\theta) \tag{A.1}$$

Thus, we compute the combined loss, $L^{CLIP+VF+S}$, and optimize it with the Adam optimizer.

Single Frame Model

The single frame model is the same as the frame stack model except that the state is represented by the current frame only (see Figure A.1(b)). We wanted to train this model to have a baseline to compare the rest of the models to.

Single Frame with Measurements Model

This model is also a single frame model, however, the speed, steering angle, ABS and angular velocity measurement of the vehicle, m_t, are concatenated with the latent space vector ϕ, as shown in Figure A.1(c). It is reasonable to believe that this state representation should encode as much information as the frame stack variant, since stacking frames simply aims to let the network infer temporal properties such as speed and angular momentum from the differences in the frames.

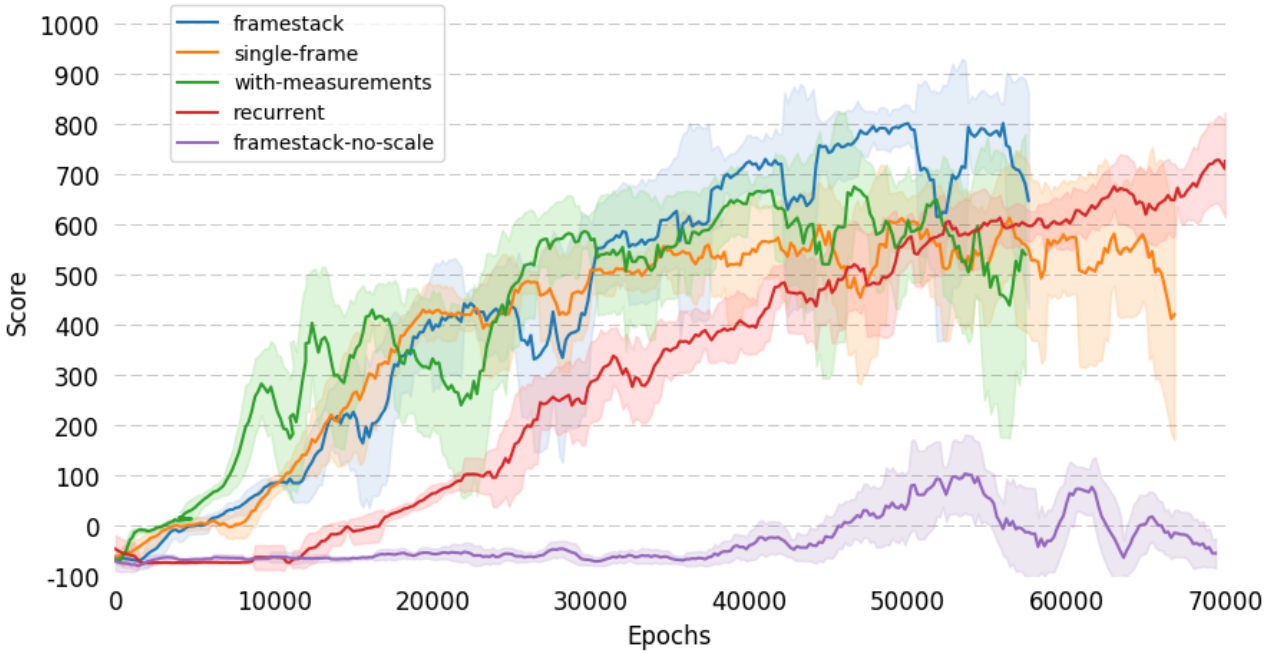


Figure A.2: Shows the cumulative reward of the models over number of epochs. Note that a sliding window of ±5 was applied to smooth out the graph.

Recurrent Model

For the recurrent model, we wanted to see if the agent performs better if it can utilize temporal information beyond 4 frames. In this model, we concatenated the latent space vector ϕ_{t-1} from the previous step to the latent space vector ϕ_t from the current step, as seen in Figure A.1(d).

Non-scaled Actions, Frame Stack Model

To know whether or not scaling the mean is actually a good idea, we trained the frame stack model without applying Equation 3.1. Instead of tanh, no activation function was used in the actors fully-connected layer, and no action scaling (Equation 3.1) is applied.

A.1.3 Results

A.2 Comparisons

Figure A.2 shows the cumulative reward of each of the models over number of epochs of training. The evaluation step was run every 200 epoch during training. Evaluation involves using the current policy over a complete run from start to end on one agent and recording its cumulative reward (the reward is calculated as explained in Section 3.3.1.) Note that these measurements are quite noisy because we are only evaluating the performance over a single trajectory. Because of this, a sliding window of ± 5 was applied to calculate the means and standard deviations shown in the figure.

Frame Stack Model

Frame stacking is a tried and true method in deep reinforcement learning, and using frame stacking in CarRacing-v0 appears to be no exception. This model achieved the highest score out of all the model – a score of 896.6 in its best run – and got an average score of 726.4 over the last 100 runs, as seen in Table A.2. While the model did not "solve" the CarRacing-v0 environment as far as OpenAI's criteria of achieving an average of 900 reward over 100 trials is concerned, it does reliably stay on the road and is able to make sharp turns while maintaining a constant speed throughout the run without any erratic behaviour. This, and most of the other models, were even able to *recover* from bad states, which is a property that is very relevant to self-driving vehicles. We believe that this model could learn to solve the environment if we make it a bit deeper by adding more convolutional or fully-connected layers; the current network is relatively shallow to favour training time. A video of the results can be found at *https://youtu.be/8X_LSy4TF84*, demonstrating the best run of this agent, in addition to showing runs of all the 5 models over 50k epochs of training, and an example of how the agent is able to recover from bad states.

Single Frame Model

The single frame model and the frame stack model have very similar developments up until about epoch 30,000, where we see the single frame model start converging. This model reached a final average score of 545.9, substantially less than the frame stack model. This goes to show that the temporal information we get from stacking the frames is essential to the success of a CarRacing-v0 agent.

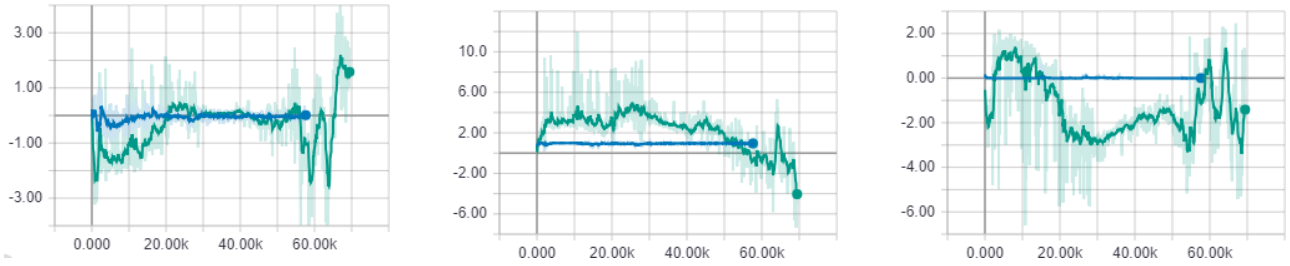
Single Frame with Measurements Model

Interestingly enough, this model appeared have faster initial improvement than the other models, but had a hard time converging later on. The initial improvement may have occurred because the agent learned more quickly that a high speed is good for getting high rewards, however, further investigation is required to verify this. We found it to be necessary to normalize the measurements to approximately $-1 \leq m_t \leq 1$ range, to reduce training variance due to fluctuations in measurement values. However, the agent still had the highest variance during training of all the models.

We suspect that the model was not able to converge because we were simply concatenating the 8 measurements to the latent vector, meaning the model had a hard time differentiating those parameters, and learning what their relationship is to the 2592 other parameters. This could be alleviated by adding fully-connected layers before and after concatenation with the latent space vectors as Codevilla *et al.* did [CMD+17], or by using m_t to direct an attention mechanism similar to Mehta *et al.* [MSS18].

Recurrent Model

The recurrent model had a slow start, but eventually achieved similar scores to that of the frame stack model. Looking at Figure A.2 and Table A.2, we may observe that this model has substantially less variance in its performance measures compared to the other models. Having low variance during training and testing is a very desirable property, and a particularly



(a) (b) (c)

Figure A.3: Shows the action means for (γ, a, b) over number of epochs. The blue graph is the frame stack model and green graph is the non-scaled frame stack model. (a) shows the steering angle γ, (b) shows the acceleration a, and (c) shows the braking b.

Model	μ	σ
Frame Stack	726.4	138.9
Single Frame	545.9	151.3
Frame + Measurement	590.9	159.3
Recurrent	624.7	75.6
No Action Scale	20.5	81.8

Table A.2: Mean and standard deviation of the cumulative rewards obtained by 100 evaluation runs after convergence.

important feature when it comes to training self-driving vehicles. This difference may originate from the fact that we have doubled the number of parameters in our fully connected layers, because the size of the shared feature vector is doubled after concatenation. Regardless, we find this model to be the most interesting model to investigate further.

Non-scaled Actions, Frame Stack Model

The non-scaled action mean version performed substantially worse than the scaled one given the same amount of training time. The agent did not learn anything for the first 40,000 epochs because the action means, particularly the steering angle, diverged too far away from the space of valid actions as seen in Figure A.3. We believe that the agent would reach the same performance as the scaled variant over time, but it is obviously slower to train and is therefore not favorable.

OpenAI Baseline Comparison

OpenAI's PPO implementation was run for comparison, with default parameters except for the network model, where we chose to use a convolutional neural network instead of a multi-layered perceptron for a more appropriate comparison. The parameters are, by default, set to the same parameters that they used to solve continuous control problems in the MuJoCo environment (Table 3 from [SWD+17].) If these parameters can solve MuJoCo control problems, it is reasonable to assume that they should also work for CarRacing-v0 as well. To summarize the most important differences between our models and theirs: their model only runs a single environment over 1 million frames before terminating, the horizon is $T = 2048$ and number of epochs is $K = 10$ with minibatch size $M = 64$, meaning that we end up running $\lfloor \frac{\#\text{steps}}{T} \rfloor * K = \lfloor \frac{10^6}{2048} \rfloor * 10 = 4880$ epochs in total.

The training was ran to completion, and the baseline model had not started to converge yet (about -60 score average.) 4880 epochs is admittedly a bit low compared to number of epochs that we trained the other models in this section for, however, we would expect it to at least achieve similar performance as our frame stack model – which at this point would get scores that are greater than zero. There are several reasons why this might have happened. First, we used 8 environments instead of one. This means that our agent will see much more variety in its input and target data, leading to faster convergence. Second, the horizon is pretty big compared to the minibatch size. The consequence of this is that the likelihood of picking important samples – *e.g.* samples of when the agent is about to go off-road, or samples that are pivotal in making an upcoming turn – is lowered. Third, the OpenAI implementation does not scale their action means either, so we should expect its performance to be similar to that of the non-scaled action mean frame stack model, which did not get a positive score until about epoch 40,000. In reality, we should probably run OpenAI's implementation with the same hyperparameters as our experiments for at least 70,000 epochs to make more accurate comparisons.

References

[Ach19] Josh Achiam. Spinning up documentation. `https://spinningup.openai.com/en/latest/`, 2019.

[BB11] Peter L. Bartlett and Jonathan Baxter. Infinite-horizon policy-gradient estimation. *CoRR*, abs/1106.0665, 2011.

[BESK18] Yuri Burda, Harrison Edwards, Amos J. Storkey, and Oleg Klimov. Exploration by random network distillation. *CoRR*, abs/1810.12894, 2018.

[BSA83] A. G. Barto, R. S. Sutton, and C. W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5):834–846, Sept 1983.

[BTD+16] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars. *CoRR*, abs/1604.07316, 2016.

[CMD+17] Felipe Codevilla, Matthias Müller, Alexey Dosovitskiy, Antonio López, and Vladlen Koltun. End-to-end driving via conditional imitation learning. *CoRR*, abs/1710.02410, 2017.

[DDS+09] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[DHK+17] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. OpenAI Baselines. `https://github.com/openai/baselines`, 2017.

[DRC+17] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, pages 1–16, 2017.

[GB10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR.

[GLSU13] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: The kitti dataset. *International Journal of Robotics Research (IJRR)*, 2013.

[Has10] Hado V. Hasselt. Double q-learning. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 2613–2621. Curran Associates, Inc., 2010.

[HCG+18] Xinyu Huang, Xinjing Cheng, Qichuan Geng, Binbin Cao, Dingfu Zhou, Peng Wang, Yuanqing Lin, and Ruigang Yang. The apolloscape dataset for autonomous driving. *CoRR*, abs/1803.06184, 2018.

[HNR68] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, July 1968.

[HPR+17] Irina Higgins, Arka Pal, Andrei A. Rusu, Loïc Matthey, Christopher Burgess, Alexander Pritzel, Matthew Botvinick, Charles Blundell, and Alexander Lerchner. Darla: Improving zero-shot transfer in reinforcement learning. In *ICML*, 2017.

[HS18] David Ha and Jürgen Schmidhuber. World models. *CoRR*, abs/1803.10122, 2018.

[HZAL18] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *CoRR*, abs/1801.01290, 2018.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[JML] Se Won Jan, Jesik Min, and Chan Lee. Reinforcement car racing with a3c.

[KHJ+18] 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.

[KL02] Sham Kakade and John Langford. Approximately optimal approximate reinforcement learning. In *Proceedings of the Nineteenth International Conference on Machine Learning*, ICML '02, pages 267–274, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[KSEH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Neural Information Processing Systems*, 25, 01 2012.

[KW14] Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. *CoRR*, abs/1312.6114, 2014.

[LBBH98] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

[LBD+89] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, Dec 1989.

[LHP+15] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015.

[MBM+16] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016.

[MKS+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[MSS18] Ashish Mehta, Adithya Subramanian, and Anbumani Subramanian. Learning end-to-end autonomous driving using guided auxiliary supervision, 2018.

[NHH15] Hyeonwoo Noh, Seunghoon Hong, and Bohyung Han. Learning deconvolution network for semantic segmentation. *CoRR*, abs/1505.04366, 2015.

[Ope17] OpenAI. OpenAI Baselines: ACKTR & A2C, August 2017.

[Org] World Health Organization. Who data - violence and injury prevention.

[PC18] Eduardo Pinho and Carlos Costa. Unsupervised learning for concept detection in medical images: A comparative analysis. *Applied Sciences*, 8, 07 2018.

[Pom93] Dean Pomerleau. *Neural network perception for mobile robot guidance.* Kluwer Academic Publishing, January 1993.

[RHT+18] Antonin Raffin, Ashley Hill, René Traoré, Timothée Lesort, Natalia Díaz-Rodríguez, and David Filliat. S-rl toolbox: Environments, datasets and evaluation metrics for state representation learning. *arXiv preprint arXiv:1809.09369*, 2018.

[RHT+19] Antonin Raffin, Ashley Hill, Kalifou René Traoré, Timothée Lesort, Natalia Díaz Rodríguez, and David Filliat. Decoupling feature extraction from policy learning: assessing benefits of state representation learning in goal based robotics. *CoRR*, abs/1901.08651, 2019.

[RS19] Antonin Raffin and Roma Sokolkov. Learning to drive smoothly in minutes. `https://github.com/araffin/learning-to-drive-in-5-minutes/`, 2019.

[SAE14] International SAE. Automated driving levels of driving automation are defined in new sae international standard J3016. 2014.

[SDLK17] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics*, 2017.

[Sen] Statisktisk Sentralbyraa. Veitrafikkulykker med personlig skade - aarlig - ssb.

[SLM+15] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015.

[SML+15] John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *CoRR*, abs/1506.02438, 2015.

[SQAS15] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *CoRR*, abs/1511.05952, 2015.

[SWD+17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[WdFL15] Ziyu Wang, Nando de Freitas, and Marc Lanctot. Dueling network architectures for deep reinforcement learning. *CoRR*, abs/1511.06581, 2015.

[Wil92] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. In *Machine Learning*, pages 229–256, 1992.

[YTXC09] Cheng Yong Tang and Song Xi Chen. Parameter estimation and bias correction for diffusion processes. *Journal of Econometrics*, 149:65–81, 04 2009.

NTNU
Norwegian University of
Science and Technology