Autonomous Vehicle End-to-End Reinforcement Learning Model and the Effects of Image Segmentation on Model Quality

By

Grant Michael Fennessy

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Computer Science

May 10th, 2019

Nashville, Tennessee

Approved:

Xenofon Koutsoukos, Ph.D.

Gautam Biswas, Ph.D.

ABSTRACT

Autonomous driving has the potential not only to transform people's lives, but also save them. Fully understanding state of the art autonomous driving architectures, however, requires a wide breadth of knowledge on available sensors, perception, image segmentation, localization, path planning, neural networks, convolution, and more. This thesis proposes a simple end-to-end architecture that has promising behavioral results. Two novel techniques are also introduced: a new exploration algorithm that seeks to produce more robust training behaviors over simple linear decay models, and a new data splitting technique that splits a layer into multiple semantically meaningful layers in an attempt to improve feature recognition by a convolutional neural network. A series of end-to-end models are trained with access to either a ground truth semantic segmentation perceptor or an image camera perceptor with a semantic segmentation predictor model. Models are evaluated and results are compared to see which approach is superior. The perceptor configuration on the trained models is switched and evaluation is run again to see how the it reacts to a change in perceptor quality. This thesis hypothesizes that models should be trained on ground truth semantic segmentation data, even if the trained model will ultimately be evaluated with a semantic segmenter model, as the model quality should prove superior and training time can be reduced substantially.

TABLE OF CONTENTS

Page

ABSTRACT ii

LIST OF TABLES v

LIST OF FIGURES vi

1 Introduction 1

2 Related Work 4

2.1 Simulators 4

2.2 End-to-End Architectures 5

2.2.1 Behaviors 7

2.2.2 Exploration Techniques 8

2.3 Vehicle Perceptors 9

2.3.1 Advanced Vision 11

2.3.2 Active Perceptors 12

3 Background Material 14

3.1 Reinforcement Learning 14

3.2 Deep-Q Network 16

3.2.1 Queues 18

3.2.2 Dueling DQN 19

3.3 Convolutional Neural Networks 20

3.4 Carla 23

3.4.1 Simulator Outputs 24

3.4.2 Simulator Inputs 26

4 System Architecture 27
4.1 Perception 28
4.1.1 Vehicle Cameras 28
4.1.2 Semantic Segmentation Perceptor 29
4.2 Data Fusion 30
4.2.1 Depth Splitting 30
4.2.2 Append Speed 32
4.2.3 Frame Stacking 33
4.3 Decision Making 34
4.4 Controller 36
5 Hypothesis and Training 38
5.1 The Deeplab Model 38
5.1.1 Probabilistic Exploration 40
5.2 The Reward Function 41
5.3 Training Configuration 43
5.4 Implementation 45
6 Experiments and Results 46
6.1 Evaluation 47
6.2 Depth Splitting 49
6.3 Exploration Style 52
6.4 Deeplab Perception 54
6.5 Post-Train Perception Adjustment 57
6.6 Model Behavior Success 59
7 Conclusion 62
7.1 Future Research 63
7.2 Final Comments 65
BIBLIOGRAPHY 67

LIST OF TABLES

Table | Page

6.1 Results: No Depth Splitting 50

6.2 Results: Depth Splitting 50

6.3 Results: Linear Exploration 53

6.4 Results: Probabilistic Exploration 53

6.5 Results: Ground Truth Perceptor 56

6.6 Results: Deeplab Perceptor 56

6.7 Results: Ground Truth to Deeplab Perceptor 58

6.8 Results: Deeplab to Ground Truth Perceptor 58

6.9 Results: Best Model Extra Training 60

6.10 Results: Behaviors 61

LIST OF FIGURES

Figure Page

1.1 Vehicle Inputs and Outputs 2

2.1 Vehicle Sensors Comparison 12

3.1 DQN Atari Game 17

3.2 DDQN Atari 19

3.3 Understanding CNNs 21

3.4 Carla Urban Landscape 24

3.5 Carla Cameras 25

4.1 Agent Architecture 27

4.2 Segmentation Selector 30

4.3 Depth Splitting 31

4.4 Q Network Architecture 35

5.1 Deeplab Conversions 39

6.1 Depth Splitting Reward Curves 49

6.2 Depth Splitting Comparison 51

6.3 Exploration Style Reward Curves 52

6.4 Exploration Type Comparison 53

6.5 Deeplab Reward Curves 55

6.6 Deeplab Results Comparison 56

6.7 Semantic Segmentation Switch 59

Chapter 1

Introduction

A Utopian world of self driving cars has been promised for years, but always seems another several years away. Research interest has increased steadily over the last decade, in no small part due to the economic incentives from major technology companies, the requirements for entry into the field are incredibly demanding. Assembling an autonomous vehicle requires a significant amount of knowledge from several fields, along with the mechanical engineering expertise of actually modifying a vehicle, be it an small radio controlled car or full sized automobile. Simulators can prevent the need for physical hardware, but the transition from simulation back into real life work is not always straightforward. This thesis seeks to introduce several of the key technologies behind autonomous driving, presents a simple working architecture, then runs a collection of experiments to demonstrate driving behaviors. During experimentation, a few simple yet novel techniques are presented and analyzed.

It is difficult to understate the potential value of fully autonomous vehicles. Removing the human from the loop can save lives, as humans are known to be erratic drivers who have difficultly paying attention when mobile phones and other distractions are present. A fleet of vehicles that can remove human drivers from the loop, all while promising a higher degree of safety, has the potential to dramatically reduce the number of road related deaths across the world. A secondary benefit to autonomous driving is increased asset utilization. Most vehicles are only used for a fraction of the day, while a fully autonomous vehicle can continue to drive all day, likely even providing income for its owner.

Achieving this dream, however, has proven elusive. A driving agent must have a capacity to detect upcoming obstacles, determine exactly what those obstacles are, understand how fast those obstacles are moving relative to itself, and then use all that information to

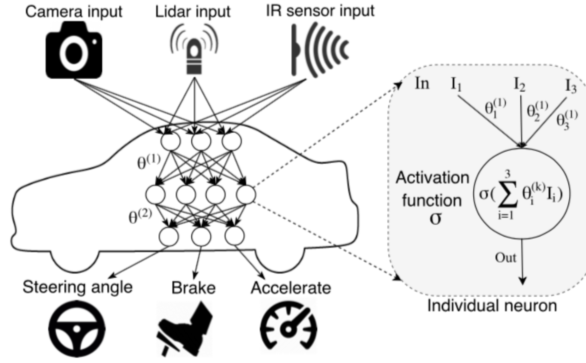

Figure 1.1: An illustration of what an end-to-end autonomous agent's inputs and outputs might look like [1].

turn the steering wheel and apply throttle and brakes. When compounded with handling navigation, localization, speed limits, traffic lights, and informative signs, it is clear that the problem space is incredibly large.

Training models to navigate tight urban roadways and intersections is extremely challenging. Even when many of the complexities are stripped away, such as pedestrian and other vehicle avoidance, navigation, and traffic rules, the simple act of remaining within the lane is easier said than done. This thesis aims to address this challenge by introducing a simple end-to-end architecture that produces modest driving behaviors. Specifically, the goal of the autonomous agent is to drive as close to 25km/hr as possible without leaving narrow urban roads. This requires navigating sharp turns, detecting and understanding lane markings, and making sense of T intersections.

This architecture is then used as a platform to test two novel techniques: a new exploration algorithm that seeks to produce more robust training behaviors over simple linear decay models, and a new data splitting technique that splits a layer into multiple semantically meaningful layers in an attempt to improve feature recognition by a convolutional neural network (CNN).

A comparative study is then launched which focuses on the semantic segmentation perceptor. Two otherwise identical models are trained, where one uses ground truth semantic

segmentation, and the other uses Deeplab v3 [2, 3] for semantic segmentation. Each model is then evaluated while using the other perceptor, emulating either a jump from simulation grade perceptors to real life perceptors, or the improvement in quality of an existing perceptor.

The proposed architecture is evaluated on how well it can produce the desired driving behaviors within the allotted training time. The study completes with analysis on where the architecture succeeded, how the proposed techniques helped reduce training time, and how the architecture can be augmented moving forward.

In all, the following hypotheses are evaluated:

- Does probabilistic exploration yield better models over linear decay exploration?
- Does depth splitting yield better models over keeping depth as one layer?
- Does a model train better when using a ground truth segmentation perceptor or Deeplab v3?
- How well does a trained model function if its segmentation perceptor is swapped out?

A trained model is produced that is capable of driving around a small town within the Carla simulator [4] for 60km at 16km/hr without colliding with any objects, proving that a relatively simple reinforcement learning model is capable of attaining modest driving behaviors. Switching the perceptor on the trained model from ground truth to Deeplab yielded only slight changes in performance. Experiments with other models show that splitting the depth layer into multiple semantically meaningful layers prior to ingestion into a CNN has an outsized influence on model quality, increasing attained rewards by from 12.2% to 37.4%. The probabilistic exploration style also yielded model quality improvements, with rewards growing from 27.1% to 37.4% when switching from linear decay ε-greedy exploration to probabilistic ε-greedy exploration. In all, the proposed architecture proves to be a simple yet effective approach to slow speed autonomous driving.

Chapter 2

Related Work

The available literature for topics related to autonomous vehicles is extensive, especially considering the necessary underlying technologies. Simulator availability, autonomous vehicle architectures, common perceptors, desired vehicle behaviors, and studies on exploration vs exploitation are discussed in the following sections.

2.1 Simulators

Regardless of which architecture is selected for the autonomous agent, a vast amount of data needs to be collected for adequate training. Companies like Tesla, Uber, and Waymo [5, 6, 7] are making headway, but their datasets are proprietary. And even though some open source datasets do exist [8, 9], their sensor setup might be completely different from the desired architecture setup. If the two are not a strong fit, then training on one has no assurance of being useful on the other. Due to the inherent dangers of a vehicle, most or all training must happen offline, as any mistakes made by the model in practice can be expensive or at worst deadly.

Some simulators [4, 10, 11] have been created which allow a custom suite of sensors to be configured. These simulations tend to be faster than real life, can be widely parallelized for little cost, and completely remove the potential for loss of life if a collision were to occur. The trade-off is, of course, whether or not the simulators are realistic enough to pose any reasonable degree of comparison between the simulation and the real world. Post-processing algorithms have been crafted that attempt to convert the simulation generated images into ones more representative of real life conditions [12, 13], and some simulators [4, 11] leverage the powerful Unreal Engine 4 [14] to produce extremely high quality and realistic images. The end goal is to bring deltas between simulation data and real life data

to zero, or at least as close to zero as possible. Even if the simulation to real life crossover for an individual model is not perfect, these simulators have become powerful enough that proof of concept models can be trained within them and iterated upon very rapidly, making them fantastic testing grounds for model architectures.

TORCS [10] is an earlier simulator which focuses on race car driving, allowing simpler models to learn the importance of staying within lane, controlling speed, and avoiding other vehicles. DeepDrive [11] switches to single lane roads in a more realistic environment with significant foliage, altitude changes, and urban background scenery, all adding additional challenge to visual perception and model requirements. Carla [4], the simulator of choice for this thesis, is placed within a densely populated urban town, featuring other vehicles, pedestrians, navigation signs, working traffic lights, two lane roads, and a wide variety of urban roadside scenery.

Some recent work has been conducted on converting image segmentation data into life-like full color images [13, 12]. This would allow a simulator to be run which only outputs image segmentation data, from which highly realistic training data images could be generated. Others have repurposed simulators as dataset generators [15], using available simulation data to create images with perfect bounding boxes. This would remove the necessity of human annotators, thus making truly enormous training datasets plausible.

2.2 End-to-End Architectures

End-to-end autonomous architectures are those which directly take inputs from perceptors, then yield the vehicle action. This stands in contrast to more modular approaches, which may have multiple tasks in series or parallel that culminate in a final action decision. Having higher modularization is beneficial in that it can be easier to understand and verify, but is often more complex.

End-to-end approaches to autonomous driving are a bit of wishful thinking at the moment. Though the potential is certainly there [16, 17, 18], progress is a bit slow due to

the extraordinarily high state space of a driving agent. The main problem, however, is the difficulty of actually proving these models work properly given the seemingly limitless number of edge cases. Some visualization techniques have been posed to address this issue [19], but it is far from a solved problem. Given the high levels of safety demanded from autonomous vehicles, a black box approach is much less likely to be approved than a model that can be split into distinct and more provable components. This is especially true given how vulnerable modern deep learning techniques are to attacks [20, 1]. That said, end-to-end frameworks are significantly easier to implement, and as such act as a fantastic entry point into autonomous vehicle architectures. Using an end-to-end approach as a baseline, the model can later be modularized as needed to promote the desired functionality.

If a customized vehicle sensor arrangement is to be used, then reinforcement learning and/or imitation learning is all but required. Imitation learning is a great tool if a sufficiently skilled human operator is available to record a large number of driving episodes, as is the case of Tesla, which has a fleet of vehicles driven by human operators every day [21]. When an assumption is made that a simulator is to be used, which is all but required for those making their first autonomous vehicle, the utility of imitation learning is a bit less clear. Many simulators have fairly crude left/right and forward/back controls on the keyboard, which makes driving well fairly difficult even for humans. Reinforcement learning, assuming a strong reward function can be generated, allows the model to learn entirely on its own without having to devote the human capital.

Deep-Q networks (DQNs) [22] provide a strong discrete state space tool for predicting which action will maximize rewards at any given step. Since they use CNNs such as VGG13 [23], DQNs are able to quickly process high resolution sensory data. Dueling Deep-Q networks (DDQN) [24] expand on this idea by computing advantage functions for actions instead of just calculating pure action reward values. Due to DDQN's success with discrete spaces, such as Atari games, it is used as the model for this thesis.

When more computing resources are available, asynchronous advantage actor-critic

(A3C) [25] allows highly parallelized learning across multiple agents. A central critic function acts as a value estimator which learns to update the policies of independent agents, each of which can explore different parts of the state space. The best parts of each actor's learning are brought into the main algorithm, allowing rapid learning when a large number of actors are made available. This is the algorithm of choice in the Carla [4] paper, but due to limited resources for this thesis (all experiments run on a single computer with an NVIDIA 1080 Ti GPU), the benefits of A3C might not have made themselves apparent. This thesis seeks to present a simple architecture for someone entering the field for the first time, so no assumptions can be made about the availability of compute clusters.

Though DDQN is proven to be a powerful approach, it does not have any concept of time. DRQN [26] reconfigured baseline DQN to include a recurrent neural network (RNN) [27], which allows input and hidden layer results from previous steps to be factored into the current step's predictions. This technique allows an internal memory to be formed that can keep track of previous events, thus allowing more intelligent decisions to be made. Long-short term memory cells (LSTMs) [28] are another form of memory that can be inserted into neural networks. LSTMs allow the model to decide what features should be stored in memory, and when to leverage that memory. Though the addition of memory to models can produce more robust predictions, they are much more advanced algorithms and as such are not recommended for those first entering the field. That said, success has been made by NVIDIA [17] with an end-to-end model that uses LSTMs in a semi-unsupervised fashion to learn simple driving actions.

2.2.1 Behaviors

When determining whether or not to use end-to-end learning, it is important to characterize the desired agent behaviors in great detail. Agents that are only required to control steering on the highway, for example, can more easily be formulated as end-to-end problems. Given the fairly small action space and by extension behavioral requirements (in

this case the only behaviors are to follow the road and maintain a predefined speed), it is less difficult to visualize and thoroughly evaluate a model [1]. Agents responsible for full environmental navigation, on the other hand, do not have this same luxury. When a single model is balancing multiple behaviors and a vast array of inputs, it is nearly impossible to properly evaluate all edge cases, let alone develop a strong understanding of how the model actually works.

A level 5 autonomous vehicle has a vast array of behavioral requirements [29]. Beyond simple lane control, the agent must be able to understand and apply the rules laid out by road signs, avoid all obstacles, navigate in tight and poorly marked locations like parking garages, leverage navigational aids to travel between two arbitrary points, and obey all road rules such as traffic lights and stop signs. To attain full level 5 autonomy, the vehicle must have no human in-the-loop requirements whatsoever, even emergency takeover requirements would reduce the vehicle to level 4 autonomy.

Selecting behavioral requirements is a problem in and of itself, but no researcher entering the field should expect to address higher level behaviors with their first model. Tesla's approach with autopilot [5, 30] is an example of starting with a simpler approach first. Their original Autopilot was just a traffic aware cruise control based on a neural network, but was later improved to also support automatic lane change and highway navigation. To this point, this thesis seeks only to address a very simple behavioral space.

2.2.2 Exploration Techniques

Due to the usage of DDQN within this thesis, it is worth discussing modern literature around exploration vs exploitation, a key component of reinforcement learning.

One of the most commonly used exploration techniques, due to its ease of implementation, is ϵ-greedy exploration [31, 32]. Based on some probability, either the greedy best action is taken (exploitation) or a random action is taken (exploration). #Exploration [33] attempts to discretize the state space by creating a hash function that converts the state

space into some number of buckets. Frequency of state visitation within each bucket is counted, and an exploration function is developed that prioritizes less frequently visited buckets.

Bootstrapped DQN [34] attempts to look a few steps in the future to find states that demand exploration (known as *deep exploration*), then takes a series of necessary (or at least assumed necessary) actions to get to that point. Allowing exploration to take place across multiple time steps is very important, especially as behaviors become increasingly more complex. An autonomous vehicle operating at a high frame rate, for example, has less capacity to ”explore” the state space unless it can take several premeditated actions, such as turning left for several steps in a row. A variant of the Bellman equation that promotes uncertainty [35] is another example of this type of exploration which can look several steps into the future for exploration opportunities.

2.3 Vehicle Perceptors

Being able to observe and identify objects in the world is pivotal to an autonomous vehicle agent. Neural networks have taken the research on object identification by storm, largely starting with the AlexNet [36] paper, which introduced a CNN to identify what category of object is in a given image. VGG [23] extends the size of the network with a repeating convolution+pooling framework that slowly reduces down to a couple fully connected layers. Deeper models can be hard to train, however, an issue addressed by ResNet [37], which uses a residual function to pass gradients across layer groupings to reduce the likelihood that gradients will trend towards zero. With this technique, ResNet models could be deeper than traditional network architectures without sacrificing learning. Inception [38] changes this approach, suggesting a wider instead of deeper architecture where each layer would run multiple sizes of convolution and combine the results.

Many of these algorithms are trained against the ImageNet datasets [36] to allow clear comparisons. There are a vast array of CNN architectures to choose from these days, but

emulating VGG [23] is a fantastic starting place that allows a better understanding of the mechanics of convolution, which is why a VGG-style approach is used in this thesis.

When multiple objects are in the image, and each object needs to be individually categorized and located within the image, more advanced algorithms such as YOLO [39] must be used. Many of these algorithms rely on anchor points that propose regions, which often results in any given object being identified multiple times, or regions being proposed that contain multiple objects, only one of which ends up being identified.

A more robust way of identifying objects within an image is to fully categorize every pixel in a process known as semantic segmentation. An autonomous driving agent that knows exactly how the road is laid out before it, for example, does not need to bother processing the different colors of black and gray on the asphalt, and can instead focus on the more abstract road shape. By leveraging an encoder-decoder configuration, semantic segmentation can use pre-trained encoder models from object identification tasks to forgo a significant portion of the training effort, as with SegNet's [40] use of pre-trained VGG16 models [23]. SegNet maps the max pooling indices from the encoder over to the decoder, allowing the upsampling to more closely match the underlying features. The original Deeplab [2] paper was able to surpass SegNet's performance with the addition of atrous rates at the end of the pipeline, designed to expand the field of view when constructing the final predictions. The latest Deeplab v3 [3] offers one of the best semantic segmentation accuracy rates available, applying cascading atrous pyramids to allow contextual information to be more easily processed. SqueezeDet [41] is a semantic segmentation approach written explicitly for autonomous driving and tested on the popular KITTI [9] autonomous driving dataset, which attempted to have the CNN perform both image classification and boundary box detection.

Given the importance of being able to run image segmentation on autonomous vehicles, but with a very limited amount of resources (both power and chip capability due to cost), a new generation of segmentation algorithms is on the rise that makes small sacri-

fices to accuracy in exchange for large reductions in processing time and energy. Multinet [42], for example, compresses semantic segmentation and object identification into a single architecture. SqueezeNet [43] is very small DNN with the sole purpose of being nearly as efficient as modern CNNs (such as AlexNet [36]), but significantly smaller. A modified version of SqueezeNet [44] was even developed with the express purpose of semantic segmentation on embedded devices for autonomous vehicles. Compressed networks like these are important developments, but this thesis chooses to leverage Deeplab since their implementation is open source.

2.3.1 Advanced Vision

More advanced instance-aware segmentation algorithms [45, 46] can not only identify the category of a pixel, but can detect the differences between two instances of a given category. In this way, multiple vehicles within a scene can be treated independently as needed. Though these techniques are potentially very useful, they are not considered entry level algorithms and are thus reserved for future research.

For tasks that require an understanding of the agent's location within the world (as is the case with autonomous driving), simultaneous localization and mapping algorithms (SLAM algorithms) [47, 48] allow the agent to learn its position from nothing more than a series of images over time. The technique attempts to identify salient features within an image frame, then correlates the features to the next frame to map how the world has changed and, in theory, identify the agent's position within the world. If the agent then completes a full loop of some small section of the world, the algorithm is able to detect this and close the loop on the internal model, improving model accuracy. SLAM models have demonstrated the ability to run in real time [48], which is critical for autonomous vehicles. These algorithms are outside the scope of this thesis, however, as SLAM requires additional downstream processing to fully utilize the predicted mapping.

Others still have shown that the relative velocities of objects in an image can be cal-

culated, a growing field known as scene flow [49]. With visible object dynamics properly measured, agents would be more capable of making intelligent decisions about what objects pose a threat and how to react to them. Neither SLAM-style approaches nor scene flow is used within this thesis due to the complexities of downstream processing and data fusion.

2.3.2 Active Perceptors

Other sensors exist on the market beyond cameras. The most widely used secondary perceptor is LiDAR [51], a portmanteau of light and radar. Unlike vision, which is a passive sensor, LiDAR actively emits packets of lasers outside of the human visible light spectrum. These lasers reflects off of distant objects and are picked up by the sensor, which converts the travel time of the light into a distance measurement for that point. The LiDAR sensor quickly sweeps across a predefined angle space (such as -140 degrees to +140 degrees) in a flat line, generating a point cloud of distance measurements for each angle. A 3D version of LiDAR also exists, which adds a vertical component to the sweep. Though LiDAR is highly accurate, it is very expensive, and any form of active sensor has a chance to cause damage to the agent's surroundings in as-yet unseen ways. For these reasons, and due to

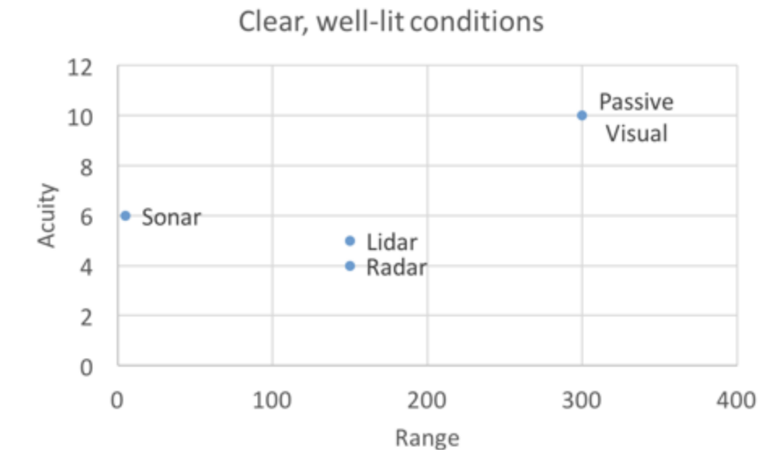

Figure 2.1: Comparisons between common vehicle sensors [50]. Passive visual is far superior in most situations, but is extremely difficult to parse.

the complexity of making use of point clouds, LiDAR is not used in this thesis (though it is supported within Carla [4]).

An alternative (or supplement) to LiDAR is radar [52, 50], which has fairly high acuity at a short to medium range. By acting on a different wavelength from light, radar has the advantage of being able to see well even in poor lighting and extreme weather conditions. Sonar is actually even better than radar, but only works at very short ranges. When designing an autonomous vehicle, it is incredibly important to understand the physics of each sensor, what wavelengths propagate well in any given environmental conditions, and how a combination of sensors pointing in the appropriate directions can yield a universally strong set of inputs for the underlying model.

Chapter 3

Background Material

Autonomous driving architectures require many individual components working in tandem. Multiple technologies from broad backgrounds are included in this pipeline, which requires a fairly wide set of background knowledge. To help the reader better understand the content of this thesis, this chapter provides a necessary baseline of knowledge on the following subjects used in the proposed architecture: reinforcement learning, Deep-Q Networks [22], convolutional networks, image segmentation, and Carla [4] (the simulator of choice for this thesis).

3.1 Reinforcement Learning

Reinforcement learning (RL) is conceptually based on Markov Decision Processes (MDP). Arulkumaran et. al. [32] has an excellent survey of modern reinforcement learning, and many of the concepts and understanding in this section are drawn in part from that paper.

A fundamental assumption behind an MDP is that at any time t, the world can be fully defined as a state s_t, which exists within a set of all possible states S. The agent can take an action a_t upon the world, which is drawn from a set of possible actions A. Transitions can then be mapped by way of $\tau(s_{t+1}|s_t, a_t)$, onto a distribution of possible states s_{t+1}. What is integral to the MDP assumption is that τ does not require knowledge of any previous states to determine the next world state, just the current state and the action taken.

The transition dynamics of the world typically are not known by the agent, so it must craft a policy π that forms a probability distribution of what state s_{t+1} is most likely to occur given a state s_t and an action a_t. When presented with a specific state, the policy π can then be used to determine what action to take in order to maximize the quality of state

s_{t+1} from the perspective of the agent. But how is this subsequent state quality measured?

State quality manifests itself by way of the reward function $R(s_t, a_t, s_{t+1})$, which provides a quantitative value to a transition that just took place. Assuming the goal of the agent is to maximize a reward, an optimal policy can be defined as $\pi^* = \arg\max_\pi \mathbb{E}[R|\pi]$. Thus at any given state, the optimal policy π^* can be used to calculate a probability distribution of actions. The action with the highest probability, as defined by π^*, can then be greedily taken every step to maximize short term reward. To factor in long term reward as well, a discount factor $\gamma \in [0,1]$ can be applied that will measure future rewards potential, such that $R = \sum_{t=0}^{T-1} \gamma^t r_{t+1}$.

The goal of a reinforcement learning algorithm is to find the optimal policy. A common policy solving algorithm is known as the value function, which uses a state-value function $V^\pi(s) = \mathbb{E}[R|s,\pi]$ to measure the rewards potential of any given state. An optimal state-value function is then defined as $V^*(s) = \max_\pi V^\pi(s) \quad \forall s \in S$. Given an optimal state-value function, the agent can greedily choose actions that will transition into subsequent states which have the maximum state-value.

Unfortunately, the transition function is not available to RL agents. To circumvent this issue, a new function known as the state-action-value, or quality function [53], is defined: $Q^\pi(s,a) = \mathbb{E}[R|s,a,\pi]$. Though similar to the state-value function, the quality function calculates the expected reward for the transition from s given action a, plus all subsequent rewards collected by following policy π at state s_{t+1} onward.

The state-value function can then be redefined as $V^\pi(s) = \max_a Q^\pi(s,a)$. If the optimal Q^π is known, then at each state a greedy selection of action $a = \arg\max_a Q^\pi(s,a)$ will produce the highest reward without having to know the reward function or the state transition mapping. Learning Q^π can be made simple by leveraging the recursive Bellman equation [54]:

$$Q^\pi(s_t, a_t) = \mathbb{E}_{s_{t+1}}[r_{t+1} + \gamma Q^\pi(s_{t+1}, \pi(s_{t+1}))]$$

Instead of recursively running the Bellman equation variant of the Q function, an ar-

bitrary starting Q^{π} can be trained against real life data with the state-action-reward-state-action (SARSA) algorithm [55]:

$$Q^{\pi}(s_t, a_t) \leftarrow Q^{\pi}(s_t, a_t) + \alpha\delta$$

where α is the learning rate, $\delta = Y - Q^{\pi}(s_t, a_t)$ is the temporal difference (TD) error and Y is the target defined as $Y = r_t + \gamma Q^{\pi}(s_{t+1}, a_{t+1})$. A near-optimal policy can be achieved by minimizing the TD error δ, which allows actions to be taken greedily each step, as future rewards earning potential becomes baked in to the policy by way of γ.

Updating the Q function each step based on an action taken by the policy is called on-policy learning. Q-learning, on the other hand, uses off-policy training, in which the action a taken each update is not necessarily the action predicted by the policy π. When learning off-policy, exploration vs exploitation becomes a crucial balance - will the agent seek the highest rewards, or will it explore for yet unseen greener pastures? A simple exploration vs exploitation algorithm is ε-greedy, which will take the greedy action with some probability, otherwise it will take a random action.

3.2 Deep-Q Network

Learning a Q-function for an extremely large state space can require an impossible amount of compute power. A simple Atari game, for example, has a state space as defined by its memory capacity of $18 \times 256^{3\times210\times160}$ [22]. An autonomous vehicle, on the other hand, exists in an effectively infinite state space, making storage of all possible states for the Q-function impossible. Deep-Q Networks (DQN) [22] leverage a neural network as the Q function, compressing the full world state into that which is made available by the observation space. At each state, observation data is fed into the DQN, and a prediction of the action which will produce the best future-looking reward is returned. This thesis leverages a DQN to make predictions against its observations, so it is important to understand the

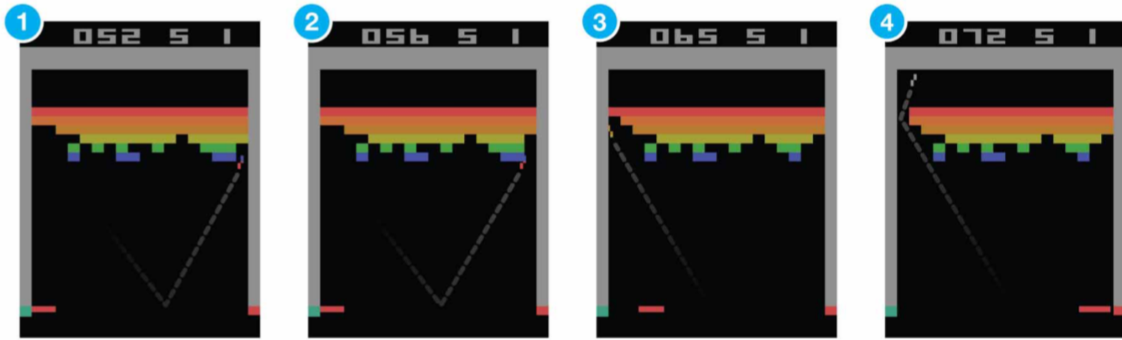

Figure 3.1: DQN learning to play Atari games [22]. Getting the ball into the corner so that it bounces around on top is a winning strategy.

underlying mechanics of a Deep-Q Network.

A major point of success for DQN is its capacity to compress the state space. DQN was first demonstrated against an Atari 2600, which has a number of possible states defined by its RAM capacity: a massive Q table size of $18 \times 256^{3\times210\times160}$. Instead of mapping all of those states, most which will never be visited in a given game, DQN instead uses the observation space as inputs to the network: 210×160 pixel data. DQN can then extrapolate the internal state value by way of the observation to produce an action prediction.

By using stochastic gradient descent during training, all neurons have a chance to be updated at any given training event. This procedure means that training against a specific state has a high likelihood of changing the way the network perceives other states. In this way, the model is capable of making modest predictions against states it has not yet observed. The corollary to this learning benefit, however, is the possibility of catastrophic forgetting [56, 57], where a neural network is vulnerable to forgetting a set of features if they are not seen across too many training cycles. Balancing these two forces is at the heart of neural network model design and training.

3.2.1 Queues

When using on-policy training, each step is likely to be very similar to the previous step, which can cause catastrophic forgetting. In a worst case scenario, the network will never actually converge as it oscillates between behavioral modes. Adding a cyclical buffer of observations and resultant rewards can help alleviate this issue, a technique known as experience replay [22, 58]. When training occurs, a batch of recently visited states is randomly sampled from the buffer, increasing the likelihood of data independence. Batching has the added advantage of allowing multiple states to be trained against at the same time, reducing overall training time requirements.

It is often the case, however, that a significant amount of the reward comes from a small percent of the transitions taken. Prioritized experience replay [59] converts the naive cyclical queue into a priority queue, where experiences that produce large errors can be prioritized over those that are predicted perfectly. Priority values within the queue are assigned to $p_t = |\delta_t| + \varepsilon$, where δ_t is the model error at the state, and ε is a very small constant to ensure the priority values are never equal to 0.

A probability can then be established over the priority values: $P(i) = \frac{p_i^a}{\sum_k p_k^a}$, where a is a hyperparameter that establishes a balance between pure uniform selection and always selecting the highest values, and the divisor is the normalization of all priority values in the queue. To correct the potential bias induced by some states being trained against more often than others, importance sampling (IS) [59] can be used to reduce learning rates for frequently visited states. Weight updates are thus reduced by $(\frac{1}{N}\frac{1}{P(i)})^b$, where N is the size of the buffer, and b is a hyperparameter that determines to what degree IS should be utilized. Typically at the start of training $b = 0$, but is annealed to $b = 1$ by the end of training when the model is beginning to converge, as natural occurrence rates are important to consider during convergence.

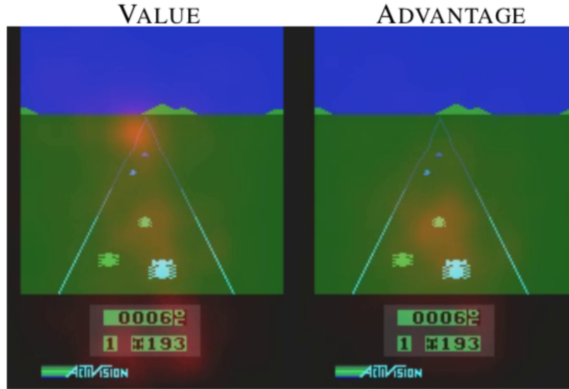

Figure 3.2: With DDQN, the state function seeks out high value states, while the action function seeks out best case short term actions, as visualized in the DDQN paper [24].

3.2.2 Dueling DQN

The Q function used by DQN can actually be split into two functions; a state value function and an advantage function. Dueling DQN [24] takes this approach, rewriting $Q^{\pi}(s_t,a_t) = A(s_t,a_t) + V(s_t)$, where $V(s_t)$ is the state value function that maps a specific state to a reward potential, and $A(s_t,a_t)$ is an advantage function that estimates a relative reward for taking action a_t verses other actions at that state. This decoupling allows the model to develop two different layers of understanding: which states tend to be more valuable, and actions that can lead towards those more beneficial states. Dueling DQN is implemented by making an duplicate copy of the fully connected layers in the model; one for the state value function and one for the advantage function.

Neural networks, however, require training through back-propagation. As a result, there may be more than one solution for V and S when Q is provided. Subtracting the mean advantage from all possible actions will result in the chosen advantage becoming 0, allowing the equation to be trained with a neural network. Thus the two paths can be combined using the formula:

$$Q^{\pi}(s_t,a_t) = V(s_t) + (A(s_t,a_t) + \frac{1}{A}\sum_{a'} A(s_t,a'))$$

As a result, DDQN is able to learn more quickly about highly valuable states, along

with short term actions that can lead to those states, which is why it was chosen for the proposed system architecture in this thesis.

3.3 Convolutional Neural Networks

A major beneficial point of DQN is that it allows a convolutional neural network to exist as the central part of the model. Convolutional neural networks were inspired by the complexities of image processing with neural networks, the first of which was LeNet [60]. Images pose a different set of problems than other data, as the pixel values are very highly dependent. Shifting an image right by 1 pixel, for example, does not change the meaning behind the image, but with a standard neural network it would result in all of the values becoming mapped completely differently [61].

Standard neural networks are just a series of fully connected layers and activation functions. As the width of the network grows too large, however, the scale can become unruly. A fairly small image of size $160x160$ pixels, for example, would be flattened down to an array of 25,600 neurons. Even if this layer were immediately followed by a layer with only 1024 neurons, this configuration would require over 26 million parameters. Beyond just the number of parameters, all neuron pairs are completely independent, so an image will be processed completely differently if it is shifted to the right by 1 pixel. Convolutional layers within neural networks address both of these issues.

Instead of fully connecting one layer to another layer, a function is executed that resembles the convolution operator. Convolution is a process in which two functions are compared for similarity. Imagine two finite waveforms, A and B, overlapping. Waveform A is fixed and waveform B can be moved from all the way on the left side of waveform A to the right side. At each possible index (assumed discrete), the overlapping area of the two wavesforms is measured as a scalar value. This process repeats for all indices, yielding a vector of cross-correlations between the two waveforms. What is most powerful about this operator is that if waveform B is a specific pattern, a matching pattern can be detected at

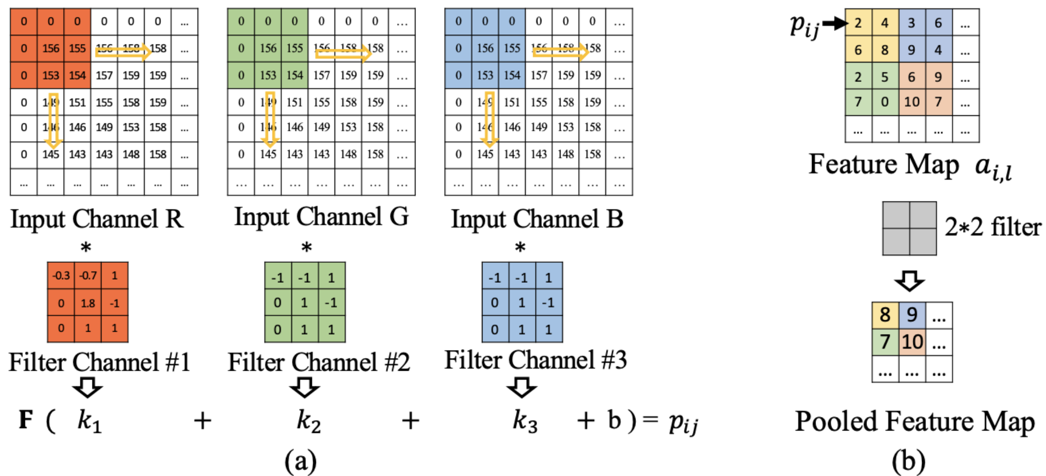

Figure 3.3: An example of a 3*x*3 convolution layer followed by a 2*x*2 max pooling layer [19].

any and all points along waveform A.

Convolutional neural networks leverage convolution to try and identify a learned set of features within an input image. First the features are searched for in the top left corner, after which a scan is performed all the way across the row, then down to the next row, eventually ending in the bottom right corner of the image. By using this scan technique, feature identification becomes translationally invariant.

To better understand how CNNs actually work on a technical level, it is best to start with a simple two dimensional matrix as an input. Given a 6*x*6 matrix, the features must be found. Features are identified with a kernel, a matrix of a small size (often 3*x*3) that is overlaid on top of the input data. Element-wise multiplication is performed on the kernel and the subset of data, then the resultant matrix is summed to produce a single value. The kernel then ”slides” over to the right by one pixel and the process repeats. Upon reaching the right side of the data, the kernel resets to the left side but moves down by 1 pixel. When the process completes, an 4*x*4 matrix will be produced.

The reduction in size of 2 is because the kernel remains entirely inside the original image. To retain the same size as the input, padding can be used whereby regions outside

of the input data are set to a value of 0, allowing the kernel to be centered on the top left pixel of the image. The size can also be changed by way of stride, which refers to how many pixels the kernel shifts each step.

Given the prevalence of RGB image data formats, most image data volume are 3 dimensional matrices. In these scenarios, the kernel is actually also a three dimensional matrix (in this case now $3x3x3$ pixels). But the convolution operation, assuming no padding, will only yield a matrix of size $1x4x4$ since the entire depth of the image is summed into a single value each step. To produce larger data volumes, multiple kernels can be used, each producing a new layer. If 32 kernels of size $3x3x3$ were used, for example, the result would be a $32x4x4$ data volume. As with neural networks, each convolution operation is typically followed by a non linear operation such as rectified linear units (ReLU) [62], allowing these convolution layers to be chained together to produce a non-linear chain of feature extractors.

Already, these layer types use smaller networks than baseline neural networks. In the original example, an image of size $3x160x160$ is used as input, and 32 kernels of size $3x3x3$ are used for convolution. The number of parameters required is only 864! But in this case, the output data volume is now of size $32x158x158$. Getting this down to a smaller size so that it can be passed into a fully connected layer would require increasing the stride to reduce the density of kernel operations. This approach, however, is limited by the size of the kernel being used, as a stride larger than the kernel size would result in unused data. Even so, just using a kernel size of 2 or more means that features might be missed.

The solution to this problem is the max pool operation, which replaces the convolution kernel element-wise multiplication operation with a simple maximum operator. In this way, the features with the highest prevalence are retained, and the weaker ones are dropped. A standard max pool kernel is of size $2x2$ and uses a stride of 2, allowing the data to effectively be chopped in half across each dimension (except the depth, which remains constant). If applied to the previous output volume of size $32x158x158$, the result would be

a data volume of size $32x79x79$.

As can be imagined, a series of convolution and pooling layers can simultaneously identify salient information, while also reducing the data volume down to a more manageable size, all while keeping the number of parameters to a fairly small number. Often CNNs will converge to a small size before switching to fully connected layers. Most importantly, all operations along the way support gradient descent, meaning that the parameters within the kernels can actually be learned by the network.

3.4 Carla

A properly constructed DDQN model with an underlying CNN is able to make decisions about its observations, but observations for autonomous vehicles carry risk and expense. This thesis makes use of a simulator to allow the agent to take actions on the world and observe the produced state spaces. Specifically, Carla [4] is used due to its high level of polish and its ability to produce ground truth semantic segmentation data.

Carla [4] is an urban town simulator based on Unreal Engine 4 [14] that provides a platform for autonomous driving research. Two town maps are made available with the open-source code, each with a different layout and asset collection. A vehicle agent can then be controlled within the simulation, allowing it to drive around and observe the map. Pedestrians and other vehicles can be added and controlled, and they all observe actual traffic patterns and speed limits.

Carla uses an observe, act, update control loop. An observation is provided to the agent, and the simulation will pause while waiting for an action command from the agent. Once the command is received, the simulation will progress forward by a predefined number of milliseconds based on the provided framerate. Once the simulation update has completed, the loop begins again.

Figure 3.4: Carla allows simulations of densely populated urban landscapes [4]. Two weather patterns are displayed here, showcasing the diversity of available scenes.

3.4.1 Simulator Outputs

Carla is capable of providing a vast amount of information to the agent each update. Basic information is always provided, such as the vehicle's speed, but advanced information can be queried by way of cameras that are attached to the vehicle.

Many valuable statistics are made available by Carla each update, but the most important for this thesis are speed, inlane percent, and collision measurement. The vehicle's velocity is provided in km/hr. The inlane percent measures how much of the vehicle's area is within the proper lane, with a value of 100% being desireable. If the vehicle enters the opposite lane or starts driving off road, the metric will transition to 0%. The collision measurement is 0 by default, but whenever a collision takes place, it increases as a function of the vehicle's speed at the time of collision, thus coarsely measuring the energy of the impact.

Agents can register cameras onto their vehicle, which provide significantly more information than the basic statistics. Cameras are registered onto the vehicle with a specific attachment point relative to the center of the vehicle, a facing orientation (default is straight ahead), a resolution (number of desired pixels in width and height), and a field of view (default of 90 degrees). In addition, cameras are registered as one of three types: standard image - red, green, blue (RGB) data, semantic segmentation, and pixel depth.

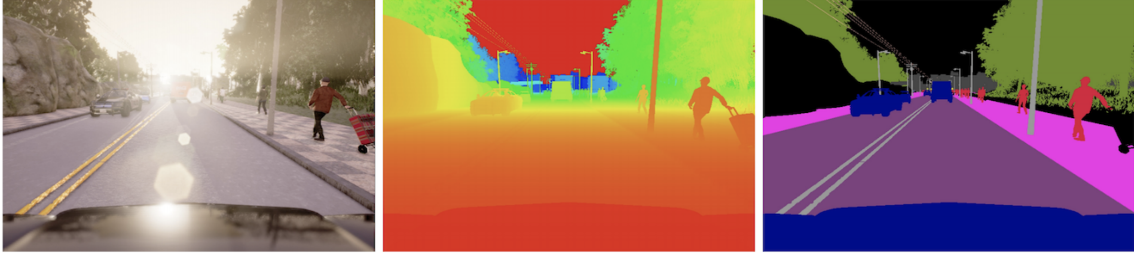

Figure 3.5: Image Carla cameras from their paper [4]. From left to right: a normal RGB camera, a depth camera, and a semantic segmentation camera.

The RGB camera is the most straightforward, acting as a standard camera. A data volume of $3 \times w \times h$ is sent to the agent every frame, where w is the resolution width, and h is the resolution height. Data points within the volume are integers ranging from 0 to 255. Carla has an image quality parameter that determines how much time the simulator will spend attempting to make photo-realistic images (thanks to the underlying Unreal Engine 4 [14]). When the value is set to low, scenes render more quickly (allowing for faster simulations), but images passed into the RGB camera are more flat and do not look as realistic. When the quality metric is set to high, the simulation will spend more time rendering, but the RGB camera will have access to very high quality images.

The semantic segmentation camera acts as a perfect semantic segmentation of the RGB camera, converting each observed pixel into one of 13 unique categories: unlabeled, building, fence, other, pedestrian, pole, road line, road, sidewalk, vegetation, car, wall, and traffic sign. This data can thus be used directly by the agent or used as ground truth for training semantic segmentation models. Using the semantic segmentation camera directly by an agent is acceptable for training or demonstrative purposes, but no expectation can be made that this data will be readily available in a real life scenario. Data is produced by this camera in a volume of size $1 \times w \times h$, where each data point is an integer value between 0 and 12 that is mapped to the categories.

Similar to the semantic segmentation camera, the depth camera converts each pixel in the RGB cameara into a scalar depth value, representative of the distance from the camera

to the first encountered obstacle. Distance is represented in kilometers, and values beyond 1km are clipped to be only 1km. Thus a data volume of $1 \times w \times h$ is produced, where each pixel is a float value between 0 and 1. As with semantic segmentation, depth information is not expected to be readily available in real life scenarios, though there are techniques that address this problem, such as SLAM [47, 48] and creating estimates from 3D Lidar [63].

3.4.2 Simulator Inputs

Five commands are made available to agents: steering angle, throttle, brake, hand brake, and toggling reverse. The steering angle expects a value between -1.0 and 1.0, representing fully turning the wheel left or right. Throttle and brake both require values between 0.0 and 1.0, representing no application and full application on the respective pedal. The hand brake and reverse toggle are either 0 or 1 depending on if they are enabled.

Chapter 4

System Architecture

Can a simple end-to-end autonomous vehicle agent learn to drive at fairly slow speeds while remaining in lane and not colliding with objects? In an attempt to answer this question, a straightforward end-to-end model architecture is proposed. This architecture is explained in depth in this chapter, followed by explanations on how the architecture is trained, and how it is evaluated.

The above question can be framed as a primary hypothesis: Can an end-to-end rein-

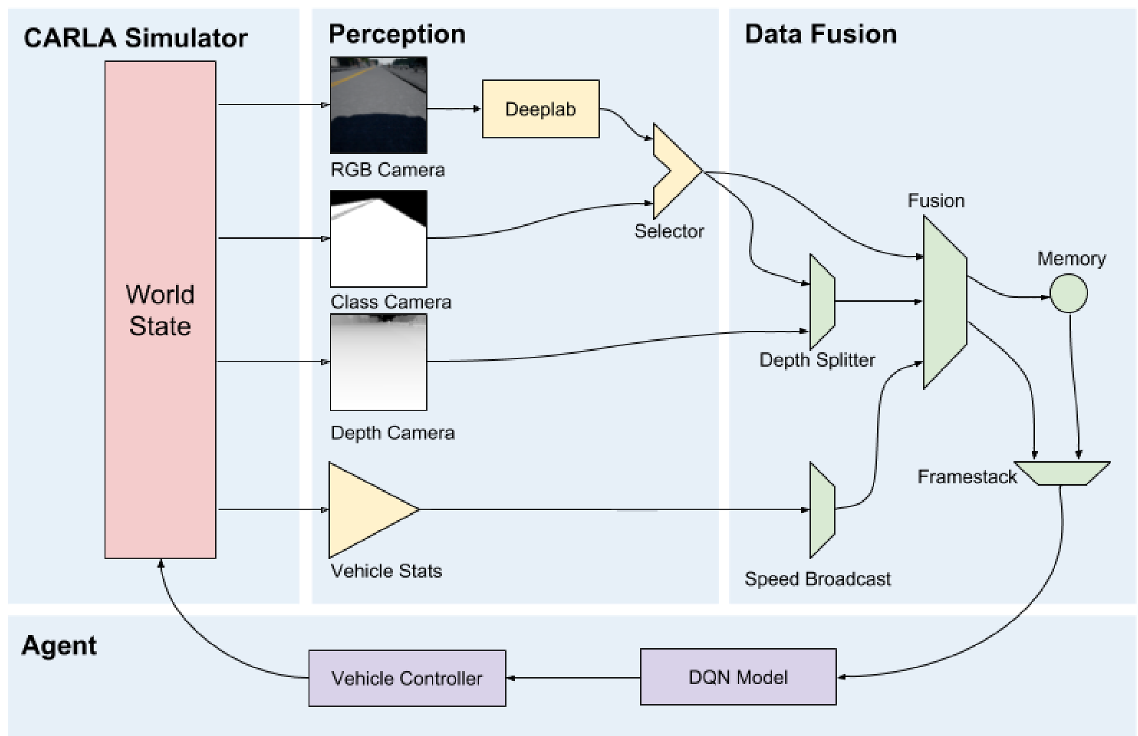

Figure 4.1: Agent Architecture. Perceptors observe the world state. A selector determines if the ground truth image segmentation data (class data) is used or if RGB camera data should be converted into class data by way of Deeplab [3]. The class data is then used to split the depth data into multiple layers. The data is then fused with the vehicle speed and passed into a DQN model [22], which outputs a desired action. Finally, a controller smooths out the actions and sends them to the actuators. The cycle repeats each frame as the world state changes due to actions taken.

forcement learning architecture be trained within a simulator to drive at a constant speed of 25km/hr such that it avoids static obstacles and remains within lane? No requirements exist for navigation, localization, other vehicle avoidance, pedestrian avoidance, local speed limit awareness, or even stop sign and traffic light awareness. The architecture could, however, be expanded piece by piece to include other requirements with future research.

The architecture is designed as three major components: perception, data fusion, and decision making. Perceptors are responsible for observing the environment (all sensors), data fusion is responsible for combining said data into a usable form, and the decision making component is responsible for converting the fused data into vehicle commands (steering, throttle, and braking). Finally, the architecture is designed to be configurable, allowing different experiments to be fired off with only configuration file changes.

4.1 Perception

The perception component of the architecture is responsible for collecting all necessary observation data from the environment. Depending upon the architecture configuration being executed, one or more of the following data sets is needed from Carla [4] by way of perceptors: RGB image data, semantic segmentation data, depth data, and the vehicle's operating statistics (such as current speed).

4.1.1 Vehicle Cameras

The RGB images, semantic segmentation, and depth data are all received from Carla by way of cameras placed on the vehicle within the simulation. The three cameras are all located at the same location on the vehicle, allowing them to have input streams that are aligned on a pixel-coordinate basis. The cameras are located at the very front of the vehicle (on the grill), providing sufficient coverage of the road and any potential upcoming obstacles. During sharp turns it is common for the vehicle to have little to no view of the road beneath the vehicle, so a 40 degree down pitch on the camera is applied to help provide

better road visibility. A standard field of view of 90 degrees is used for all cameras, and the resolution of each camera is 160x160 pixels.

Semantic segmentation data is received from Carla in the form of an integer between 0 and 12. Since this agent is not learning to navigate, stop at lights or stop signs, or interact with vehicles and pedestrians, several of the classes are not useful for the purposes of this architecture. As a result, the number of classes is reduced to 3: roads, road lines, and miscellaneous. All classes other than roads and road lines are converted to the miscellaneous class.

RGB image data is collected at the highest possible rendering quality, and an assumption is made by this architecture that depth map data is available, allowing it to be used.

In addition to the visual inputs, the perceptor receives basic statistics about the vehicle. The vehicle's current speed is used by the model, and several other statistics are plugged into the rewards calculation, including: percent of the vehicle occupying the wrong side of the road (other lane), the percent of the vehicle that is off road, and a measurement representing energy of collisions.

4.1.2 Semantic Segmentation Perceptor

The architecture can be configured with one of two perceptor configurations: receive ground truth semantic segmentation, or receive RGB data and use that to predict semantic segmentation.

When using the second configuration, the ground-truth semantic segmentation camera is replaced with an RGB camera, and an embedded model that generates semantic segmentation predictions. This arrangement more accurately reflects reality, as there does not currently exist a camera that can produce raw semantic segmentation data in the same way cameras can produce raw RGB image data. Instead, cameras must ingest RGB data and make predictions on what category each pixel belongs to.

For this architecture, Deeplab v3 [3] functions as the semantic segmentation predic-

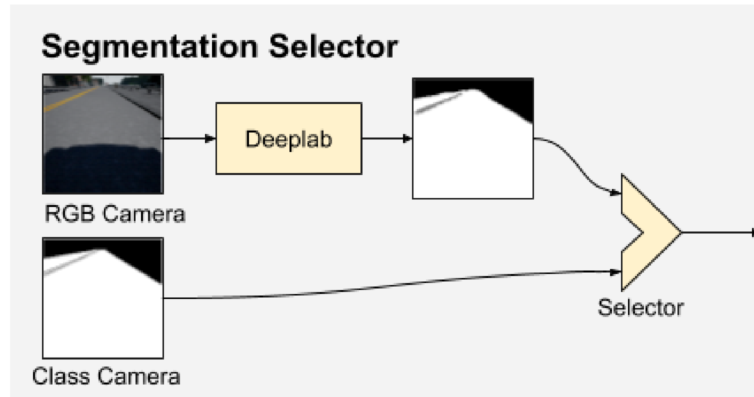

Figure 4.2: The semantic segmentation selector is responsible for either using ground truth segmentation data provided by Carla [4], or using passing RGB camera data into Deeplab v3 [3] for segmentation.

tor. A custom trained Deeplab model is used by this perceptor, the training of which is discussed in the Training chapter of this thesis. When deployed, the Deeplab perceptor outputs a stream of predictions with the same three categories as the ground-truth semantic segmentation perceptor (road, road lines, and miscellaneous), allowing the two to be interchangeable.

4.2 Data Fusion

Raw observation data must then be fused together and potentially transformed further before it can be handed off to the underlying model in the form of a single data volume. Multiple transformation and fusion configurations are made available, and each experiment is responsible for selecting the desired arrangement. The transformations are: depth splitting, appending speed, and frame stacking [22].

4.2.1 Depth Splitting

If depth and semantic segmentation data are directly passed into the internal CNN model as input layers, the model will have to spend resources learning the relationships between the two layers. Depth splitting (DS) is a novel technique that attempts to combine

the continuous depth information with categorical class information to produce more useful data to the CNN, allowing it to free up resources that would have otherwise been spent on learning those relationships.

Depth maps and semantic segmentation data serve very different purposes. Depth maps provide a continuous understanding of the world, as data contained in adjacent pixels tends to flow more smoothly. Though it is sometimes possible to make out the outlines of objects using only depth information, it is not always the case. Road lines, for example, are typically painted onto roads, meaning they can never be identified using only depth maps.

On the other hand, semantic segmentation data is useful in that it can provide hard edges between any two regions of differing categories, such as a road and a road line. But it falls short in that a farther away vehicle should be treated very differently than close up vehicle. Though the model can learn this to a degree, CNNs are translationally invariant,

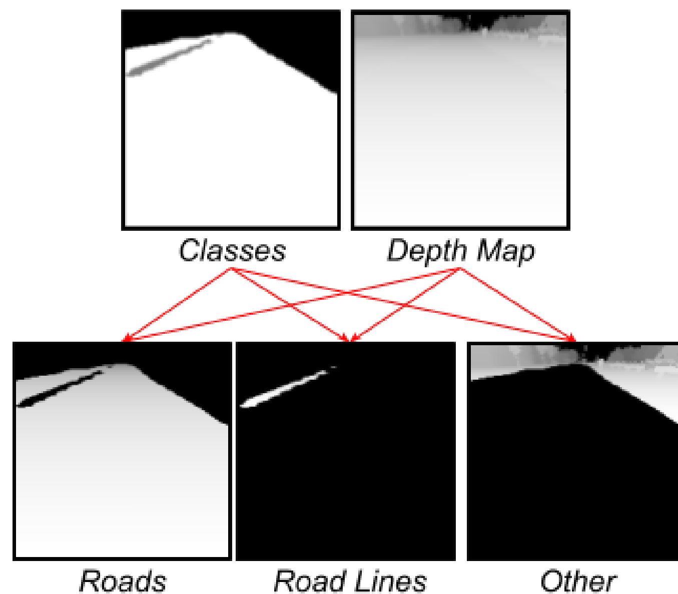

Figure 4.3: A breakout of how depth splitting works. The depth values are extracted from a single layer. Each value is placed into a single new layer depending on the category of that pixel. In this example, the trees in the top of the depth map can be seen clearly in the converted *other* layer, but are absent from the converted *roads* and *road lines* layers. With these split layers, the CNN is now in theory more easily able to parse depth and segmentation features.

not scalar invariant.

Depth splitting attempts to handle some of this fusion ahead of time, freeing up the CNN's limited kernel parameters for feature selection that can more easily be processed by the fully connected layers. DS does this by producing a small number of layers that each contain depth information about only a single category (such as roads for one layer). By splitting the data up in this way, the model can learn to process and understand the fading depth and hard depth edges of a single category at the onset.

As with baseline depth information, the DS volume will be valued from 0.0 to 1.0, representing extremely close objects (0km) all the way to very distant objects (1km or farther). The new data volume is allocated with a default value of 1.0 for each cell, which allows the difference between a close up object of a given category and any adjacent pixels contained in other categories to be as large as possible. Data is then mapped into the new DS:

$$F_{c,x,y} = \begin{cases} D_{x,y} & S_{x,y} = c \\ 1.0 & otherwise \end{cases} \quad \forall x \in X, \forall y \in Y, \forall c \in C$$

where F is the fused data volume, D is observed depth data, S is the observed semantic segmentation data, X is the set of horizontal pixels, Y is the set of vertical pixels, and C is the set of semantic segmentation categories.

A normalized version (ranging from 0.0 to 1.0) of the semantic segmentation is then appended to the data volume, allowing the model to process class information directly in addition to the previously attached class-separated depth maps.

4.2.2 Append Speed

The append speed transformation will append a new layer onto the data volume which contains the vehicle's speed. This speed layer is just a scalar value broadcast out to a matrix; a normalized version of the speed in km/hr.

Appending the speed as a CNN layer should allow the vehicle's speed information to

propagate through the CNN modeling, thus allowing the model to learn important speed information throughout the entirety of the kernel layers instead of just in the fully connected layers at the end of the model. Dedicating an entire layer has a few implications:

- The size of the final observation object is artificially enlarged, reducing the maximum number of steps that can be stored in the prioritized replay queue with the same allocated memory footprint (though in theory the data could be stored as a scalar, and the additional layer can be created on demand before training).
- An additional layer exists going into the CNN, which increases the depth of the first layer's kernel by 1, slightly increasing the total model parameters, memory footprint, and processing time. The parameters and processing times of subsequent layers are not effected.
- Speed data can be combined with DS data (or raw depth + segmentation data) throughout the entirety of the CNN.
- When combined with frame stacking, the CNN component of the model can learn speed and steering change over time more easily than having to analyze differences in DS data.

4.2.3 Frame Stacking

DQN has no concept of memory - every step is technically independent from the prior step. This stands in stark contrast to RNNs [27] which have a working memory of previous steps, or even long-short term memory units (LSTMs) [28] which can dynamically select what information should be remembered and for how long. To help mitigate this issue with DQN, especially in a task such as driving where the concept of velocity is critical, multiple subsequent observations can be sent to the model each step, a concept known as frame stack [22].

Frame stacking is a fairly naive but effective way of solving the memory problem, providing a very small window into the past observation(s). When applying frame stacking, multiple steps worth of observations are given to the decision making model at once, allowing them to be processed together.

As with append speed, frame stack increases the size of the data entering the replay queue, reducing the number of frames that can be stored with a given amount of memory by an amount proportional to the frame stack size. The first kernel in the CNN also must have additional parameters to process the initial data volume.

4.3 Decision Making

With the observation data in hand, reinforcement learning is leveraged to determine what action yields the best future-looking rewards. Specifically, a dueling deep-Q network with prioritized experience replay [59, 24].

The Q function model used in this architecture is 12 layers deep (16 when counting max pools), and is structured in a manner loosely similar to a VGG13 network [23]. There are three convolution ”blocks” in a row, where each block contains three convolution layers followed by a max pool operation. All convolutions utilize 3x3 kernels, a stride of 1, and ”SAME” padding. All max pool layers utilize 2x2 kernels, a stride of 2, and ”SAME” padding. All convolutions and fully connected layers use ReLU [62] as an activation function unless otherwise specified.

The observation data volume is first passed through a max pool, then through three similar convolution blocks (A, B, and C). The data volume is then flattened and passed to two identical structures: the action value and the state value based on the DDQN structure. The action and state values are combined to produce the final output array.

The three convolution blocks are ordered one directly after the other. Block A uses a filter depth of 64, block B uses a filter depth of 128, and block C uses a filter depth of 256. After the convolutions, the produced data volume is flattened. Data is then passed into two

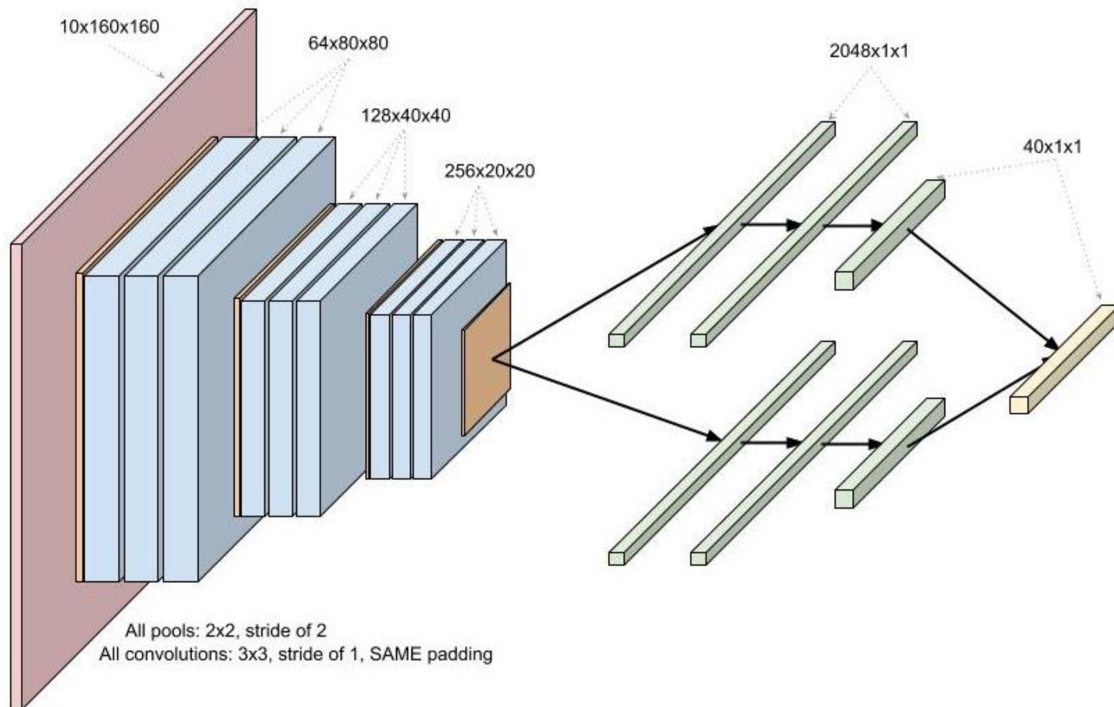

Figure 4.4: Architecture of the Q network [53]. Max pool and convolutional layers identify features in the input data volume, which are then passed through two identical fully connected networks (the dueling DQN configuration). The final output is a vector of all 40 possible actions, the highest index of which is the action the model wishes to take.

identical streams: the action stream and the state stream. Each stream has a dropout layer of 0.7 keep probability, a fully connected layer with depth of 2048, another dropout layer of 0.7 keep probability, another fully connected layer with depth 2048, and a final fully connected layer with depth 40 (one for each possible action). The final fully connected layer has no activation function.

The output array of the action stream is then reduced by its own mean and added to the output array of the state stream to produce the final Q function value. An argmax function is then used to determine the action index which is estimated to produce the highest future reward. During training, the selected index is occasionally replaced with a random index due to the exploration function in ε-greedy fashion. The selected index is then converted into the discrete steering and throttle command value.

The total number of parameters in the network is 115.35M, in large part because of the first fully connected layer after the convolution being duplicated twice for DDQN (each requires 52.4M parameters). The network is given 5GB of memory on the GPU while training, and takes up 1.9GB when saved to disc.

4.4 Controller

Actions that can be sent to Carla each frame are defined as continuous values, but this architecture discretizes them so as to use the discrete DDQN model. Thus actions are defined as the Cartesian product of 8 possible speeds (-1.0, -0.5, -0.25, 0, 0.25, 0.5, 0.75, and 1.0) and 5 possible turning angles (-0.5, -0.25, 0. 0.25, 0.5), yielding 40 actions to choose from each step. Since Carla expects independent throttle and brake values, the negative throttle values from DQN are converted into brake values. The handbrake and reverse actions are not utilized by this architecture.

Given the discrete nature of this action space, the vehicle is prone to jerky driving as it transitions between steering angles. When compounded with the occasional poor driving instruction (such as a hard left turn when the vehicle should in fact be going straight), the vehicle has a hard time driving without zig-zagging. To address this issue, and help reduce the potential damage caused by a single step's bad action command, an exponentially weighted moving average (EWMA) [64] is used to smooth out the controls:

$$c_t = \beta * c_{t-1} + (1-\beta) * a_t$$

Where c_t is the command that will be sent to the vehicle, c_{t-1} is the last command sent to the vehicle, a_t is the action requested by DQN, and β is the percent to which the old command is favored over the new command. Higher values of β will produce smoother, but slower movement, which can be dangerous if the vehicle can not turn fast enough to take turns. Lower values, on the other hand, can produce unrealistic driving conditions,

where the vehicle is able to change from a full left turn to a full right turn in only a few milliseconds.

Movement commands are then bundled up and sent off to the CARLA server, which has paused the simulation while waiting for a response. Once the packet is received, CARLA updates the world state based on the provided vehicle control request. A new set of camera observations are then sent back to the agent, repeating the cycle. A framerate of 50 frames per second is used by this architecture, yielding an observation and an action every 20 milliseconds.

In theory a proportional-integral-derivative (PID) controller could be used instead of a simple EWMA controller. A properly tuned PID controller could provide the agent a higher degree of control over the actions while still retaining the delayed inputs, since PID controllers can steady out at the control speed faster than an EWMA controller.

Chapter 5

Hypothesis and Training

With the vehicle architecture in place, models can be trained. To understand exactly *what* models need to be trained, however, a set of secondary hypotheses are formulated which will help guide the hyperparameters and ideally produce the best possible model:

- Does probabilistic exploration yield better models over linear decay exploration?
- Does depth splitting yield better models over keeping depth as one layer?
- Does a model train better when using a ground truth segmentation perceptor, or Deeplab v3?
- How well does a trained model function if its segmentation perceptor is swapped out?

This chapter will discuss in depth the Deeplab model (training and evaluation), how probabilistic exploration works, the DQN reward function, and the training setup to support the evaluation of the above hypotheses.

5.1 The Deeplab Model

To leverage Deeplab v3 [3] during the experiments, a model is first trained based off of CARLA ground truth RGB image data and the associated per-pixel image segmentation data.

Before any of the experiments are run in full, a handful of test vehicle models are trained to make sure the vehicle architecture is sufficient. One such model is selected to act as a collector to train the Deeplab model. This methodology is used so as to separate the

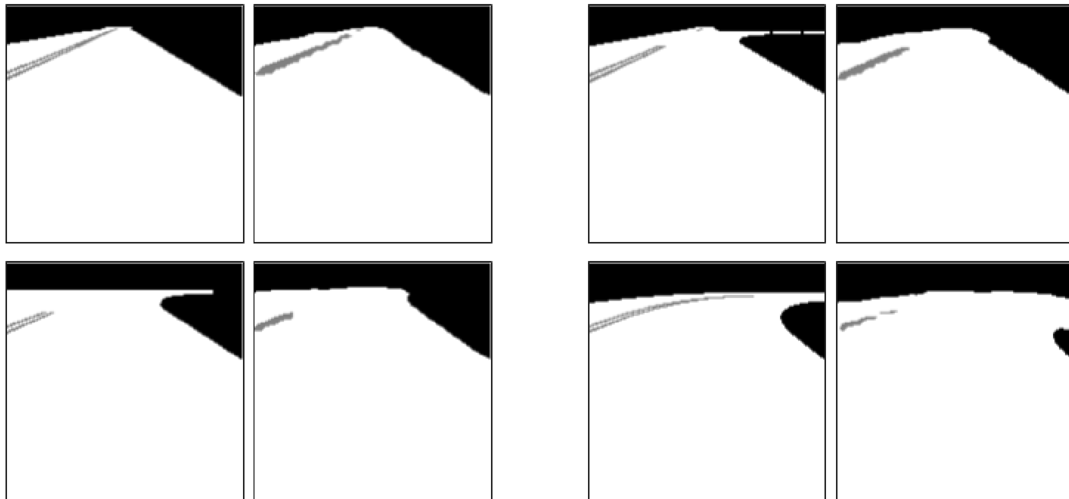

Figure 5.1: Four examples of ground-truth class data (left image) alongside the Deeplab v3 predictions (right image) for that same data. Straightaways are predicted fairly well, while curves and T-junctions tend to have higher errors. Road lines are predicted surprisingly well, but in turns can have significant missing portions (as seen in the bottom right example).

Deeplab collected data from bias towards any specific model used within the experimentation proper, amplified by the fact that the experiments are ultimately run with a different set of hyperparameters and a slightly different CNN architecture for the Q network.

The selected agent is used to collect data on both the train town and the test town. The agent used for data collection was not perfect, occasionally driving into the other lane or colliding with objects. For the purposes of training Deeplab, this behavior is necessary. Since Deeplab is used to train a new model in one of the experiments, it must have a modest accuracy when it comes to abnormal situations, like driving in the wrong lane. Without adequate exposure to those states, Deeplab would have poor performance in non-ideal states, which would put the DQN models attempting to train in said environments at a disadvantage.

A total of 1.3 million steps worth of RGB image and ground truth class segmentation pairs were collected on the CARLA train map, and an additional 256,000 steps on the CARLA test map. At the time the data was collected, four image classes were being used:

road, road lines, sidewalks, and miscellaneous. The Deeplab model was then trained using the Xception65 [65] model with atrous rates of 6, 12, 18 [3] and an output stride of 16. A learning rate of 1e-6 was used with decay factor of 0.01, and the model was trained off of a Deeplab v3 Cityscapes pretrained model. A batch size of 4 was used, and one epoch was run. The trained model has an mIoU [66] of 75.75, which is not terribly below the results in the Deeplab v3 paper of 81.3. For reference, mIoU is the mean intersection-over-union, which is the most widely used measurement tool for semantic segmentation [66].

It was later decided to bring the image classes down to 3 by merging the *sidewalks* category into the *other* category, so another batch of images was collected and a new Deeplab model was trained using the same hyperparameters. Interestingly enough, this new model only achieved an mIoU of 64.38. As a result, the model trained on 4 classes is used, and after each prediction the pixels predicting sidewalks are converted into pixels predicting other. It is possible that the first model training attempt was a fluke, or that the data collected the second time was poor, but it is worth pursing a follow-on study to see if there is an advantage to training models in a higher dimensional space, only to reduce the dimensionality during production.

5.1.1 Probabilistic Exploration

Reinforcement learning agents require exploration to attain new experiences, and thus expand their learning. Coupled with the problem of catastrophic forgetting with neural networks, it is pivotal to balance highly rewarding scenarios with disastrous ones. A vehicle can, for example, learn to overfit to more idyllic situations if the exploration value is too low during convergence. When unfortunate situations present themselves in evaluation, the model is prone to have forgotten how to recover properly, which can lead to stagnation or failure.

Further, the meaning behind an exploration percentile can be largely lost when applied to high enough frame rates. If an agent is processing the world at 50 frames per second, any

given ”exploratory” action taken will only last for about 20 milliseconds. With a late-in-training exploration rate of 10%, these random actions function more as perturbations than as legitimate behavioral exploration, as a poor decision one frame can be easily corrected by a few subsequent corrective frames, thus yielding little to no actual exploration.

This architecture introduces the novel concept of *probabilistic exploration* (PE). Instead of steadily decreasing exploration rates based on time, a new exploration rate is selected pseudo-randomly every N steps. That single exploration rate is maintained for N steps, at which point another exploration value is selected and the process repeats. A probability distribution is provided as a hyperparameter that maps a probability to an exploration value. In its simplest form, the distribution is discretized into a small number of buckets. Since these buckets can be anywhere from 0% to 100% exploration rates, the agent is able to get the best of both worlds: it can occasionally have low exploration, allowing it to bypass potentially difficult obstacles that require careful actions, and it can occasionally have high exploration, allowing it to fail those same obstacles to provide additional learning potential.

Another way to conceptualize this approach is an agent which must cross a bridge to find a treasure chest. Given a low exploration, a well trained agent can cross the bridge with ease. Once it has crossed the bridge and reached the other side, an uncharted region, it benefits more from high exploration, allowing it to fully understand what opportunities exist. PE allows this scenario to take place, where as linear explore does not.

When training, the architecture allows selecting which configuration to utilize: linear exploration or probabilistic exploration.

5.2 The Reward Function

Among the RGB, pixel depth, and class segmentation observations read from CARLA, a host of additional information is provided which is used to determine the reward for the step. The previous step’s generated observation data volume is then combined with the action taken and the resultant reward, then placed into the prioritized replay queue [59].

The primary objective of the agent is to drive as close to 25km/hr as possible. Moving at the desired speed is only valuable if the vehicle is remaining in lane, so the speed reward is 0 if the vehicle is in any way out of the lane. In addition, not moving at all carries a small penalty to prevent stagnation. Thus the reward for driving is:

$$R_{speed} = \begin{cases} 0.0 & inlane \neq 100\% \\ 1 - \frac{|speed-25|}{25} & speed \geq 1 \\ \frac{speed-1}{10} & otherwise \end{cases}$$

The vehicle's driving speed across each step is then tracked with a Bayesian filter (gamma of 0.99). If at any point after the 600th step in an episode the tracked speed is less than 1.8km/hr, the episode is terminated. Since CARLA does not report slow speed collisions, this is important in terminating episodes where the vehicle has crashed into an object and is unable to move. Some agents also learn to stop in situations that would have resulted in offroading. This termination condition prevents running unnecessary cycles against those agents, and keeps their final reward values more accurate.

In addition to receiving no speed reward when out of lane, the agent is given a flat penalty when in the other lane or when offroading to further incentivize staying in the correct lane.

$$P_{otherlane} = \begin{cases} 0.3 & otherlane \\ 0.0 & otherwise \end{cases}$$

$$P_{offroad} = \begin{cases} 0.5 & offroad \\ 0.0 & otherwise \end{cases}$$

If at any point the vehicle is more than 50% offroad, the episode terminates. No episode termination is applied when the vehicle is in the other lane, as this state is much easier to recover from during an episode, while offroading can often put the agent in states from

which recovery is impossible or nearly impossible.

When the vehicle transitions from fully in-lane to partially out of lane, an exiting lane penalty of 0.1 is applied. This penalty helps make it more clear to the agent exactly when the out of lane event takes place, allowing it to learn more rapidly to avoid that situation in the future. Thus the exiting lane penalty is:

$$P_{exitingLane} = \begin{cases} 0.1 & exitingLane \\ 0.0 & otherwise \end{cases}$$

Collisions are, as expected, unacceptable events. There is no reason to assume that the vehicle should be able to recover from a collision in any meaningful way, so any collision detected results in an immediate termination of the event. In addition, all collisions produce an immediate penalty:

$$P_{collision} = \begin{cases} 100 & collision \\ 0 & otherwise \end{cases}$$

Thus the reward for each step is calculated as:

$$R_{step} = R_{speed} - (P_{otherlane} + P_{offroad} + P_{exitingLane} + P_{collision})$$

5.3 Training Configuration

Tensorflow [67] is used for training, and the open source Tensorflow baselines ”deepq” project is used to build the DDQN model (though the custom CNN model is built manually using tensorflow layers) [24].

An Adam optimizer [68] is used with a gamma of of 0.995, grad norm clipping of 10, and no parameter noise. A prioritized experience replay queue [59] is leveraged with an alpha of 0.6, a buffer size of 30,000, a beta0 of 0.4, and an eps of 1e-6.

Model training happens in two discrete sections: the first section leverages a learning

rate of 1e-4 and trains every 6 steps for 1e6 steps, and the second section has a learning rate of 1e-5 and trains every 18 steps for 2e6 steps. Training less frequently in the second section allows the agent to collect a wider variety of experiences between each training event, in theory promoting richer training queue.

For both the first and second sections, probabilistic exploration N is set to 200 steps. The first training section has buckets: 10% chance of 0% explore, 20% chance of 15% explore, 40% chance of 30% explore, 20% chance of 45% explore, and 10% chance of 60% explore. The second training section has buckets: 20% chance of 0% explore, 40% chance of 10% explore, 20% chance of 20% explore, 10% chance of 30% explore, and 10% chance of 40% explore.

When using linear decay explore, the first training section decays from 100% to 10% over time 0% to 10%. The second training section decays from 10% to 5% over time 0% to 100%.

A batch size of 32 frames is pulled from the queue each training step. No training starts until 30,000 steps, and the DQN target network is updated every 3,000 steps. Each episode runs for up to 3000 steps (representative of 60 seconds of real time driving due to the simulator running at 50 steps per second), prefaced by a 100 step deadzone in which no actions take place, allowing the episode to fully load. No early stopping is leveraged.

As with all neural networks, hyperparameter selection was a difficult problem to address. The fact that multiple controllers are ultimately trained on the same hyperparameters only aggravated the problem. As a result, dozens of attempts were made to find "good-enough" hyperparameters that did the models justice. During the hyperparameter exploration phase, the learning rate, training timesteps, exploration size, and train frequency were the primary hyperparameters that were modified.

5.4 Implementation

All of the code is written in Python3 [69], using Tensorflow [67] for the CNN model, Tensorflow's baselines [67] for the DDQN model, numpy [70, 71] for vector mathematics (namely data fusion), PIL [72] for image processing and recording, and Deeplab v3 [3] for training the deeplab model. OpenAI's gym [73] format was used to wrap the training, and a modified version of the Carla Gym code from project Ray [74] is used for control of the agent. The code is publicly available at https://gitlab.com/grant.fennessy/rl-carla

Chapter 6

Experiments and Results

Evaluation of the hypotheses requires training multiple models from scratch, each with slightly different configurations. Trained models are then evaluated, with key benchmarks recorded to allow comparisons between the models. All hypotheses are mapped to architecture configurations and evaluation techniques. In this chapter, the evaluation techniques, analytic metrics, and the results of each experiment are discussed. The hypothesis to experiment mappings are:

Does probabilistic exploration yield better models over linear decay exploration? A model is trained with probabilistic exploration, and another is trained with linear decay exploration. Both use depth splitting and ground truth perception. Evaluation results are compared.

Does depth splitting yield better models over keeping depth as one layer? A model is trained with depth splitting, and another is trained without depth splitting. Both use probabilistic exploration and ground truth perception. Evaluation results are compared.

Does a model train better when using a ground truth perceptor, or a realistic one? A model is trained with a ground truth semantic segmentation perceptor, and another is trained with Deeplab v3 for the semantic segmentation perceptor. Both use probabilstic exploration and depth splitting. Evaluation results are compared.

How well does a trained model function if a perceptor is swapped out? The model trained with the ground truth semantic segmentation perceptor is reconfigured to use a Deeplab v3 perceptor and differences in evaluation results are recorded. Likewise, the model trained trained with Deeplab v3 semantic segmentation perceptor is reconfigured to use the ground truth perceptor and differences in evaluation results are recorded. The evaluation differences between the two reconfigurations are compared to see which produced

more favorable results.

Core Hypothesis: Can a simple end-to-end autonomous vehicle agent learn to drive at fairly slow speeds while remaining in lane and not colliding with objects? The best model from all experiments will be evaluated in terms of behavior quality. An final ”extra” third training round is performed against it, evaluation is performed, and behaviors are observed.

6.1 Evaluation

All experiments are measured by running the experimental model on the test town within CARLA, which has a different layout, different buildings, and new foliage when compared to the train town. Unlike training, where vehicle start location and the weather pattern are chosen at random at the start of each episode, evaluation has a known set of 240 episodes with preset conditions that are run for each experiment. The set of 240 episodes is comprised of 20 start locations and 12 weather conditions; 6 weather conditions that existed during training and 6 weather conditions that did not. Episodes run until reaching 3,000 steps or until a termination condition is reached (same termination conditions as training). Thus the maximum number of steps during evaluation is 720,000.

Experiments are run 3 times each to allow a mean and standard deviation to be developed. Runs are completely independent and start from scratch.

Evaluation of the secondary hypotheses demands an understanding as to whether or not the model has improved in quality. Given that the model is a reinforcement learning agent, the best metric for cross-model evaluation is the reward function. The primary hypothesis, on the other hand, is independent of the reward function, but instead demands certain behaviors are met. Since the primary hypothesis is ultimately evaluated on the best model, all model evaluations will record both the reward metric and behavioral metrics. The metrics are as follows:

1. **Reward**: The sum of all rewards from all steps divided by the maximum possible

reward of 720,000 (due to the maximum per-step reward of 1.0). A value of 100% would imply an agent that drives at exactly 25km/hr on all steps.

2. **Drive %**: The drive % measures the number of steps that took place during evaluation divided by 720,000. A value of 100% implies that the agent never had an early termination (due to stagnation, offroading, or collision), while a lower value implies some degree of failures.

3. **Km**: Total kilometers driven across all steps in the evaluation. This is ultimately a function of mean speed and drive %.

4. **Km/Hr**: Mean speed taken across all steps in the evaluation. Target speed is 25km/hr.

5. **Km/OOL**: How many kilometers are driven on average between each out of lane (OOL) infraction. An OOL infraction occurs any time the vehicle exits the lane in any way. A 2 second timer (100 steps) is kicked off after the infraction is detected, during which time no additional infractions can occur. Once the timer completes, a new infraction occurs if the vehicle is still in any way out of lane. Wrapping up the out of lane infractions into these events helps to filter out instances where the vehicle just barely nudges out of the lane several times in rapid succession. Ideally this value is infinite if no OOL instances occur.

6. **Km/Collision**: How many kilometers are driven on average between each collision with an object in the environment. Ideally this value is infinite if no collisions occur.

The primary metric for evaluating model train capability is the total accumulated reward (normalized by dividing into the total possible reward). Though the other metrics, such as the number of kilometers driven per collision, are of paramount importance for an autonomous driving agent, they act as more of a representation of the reward function quality than how well the model is trained. As a result, the reward value is leveraged for

evaluating model quality during the given experiments, and the remaining metrics are used to evaluate if the reward function is mapping the desired behaviors.

Training models for the experiments typically take around 37 hours each (68 hours when using Deeplab), and evaluating models takes around 6 hours each (11 hours when using Deeplab). Running all of the experiments requires a total of about 29 days worth of simulation.

6.2 Depth Splitting

The following hypothesis is to be evaluated: Splitting the depth map by class will produce better models than using depth as a single layer.

To test this hypothesis, two experiments need to be run: an experiment that uses depth splitting and one that does not. When using depth splitting, the 1x160x160 layer is expanded out into a 3x160x160 data volume. When combined with the image segmentation data layer and the speed layer, and factoring in a 2x frame stack, depth splitting has a

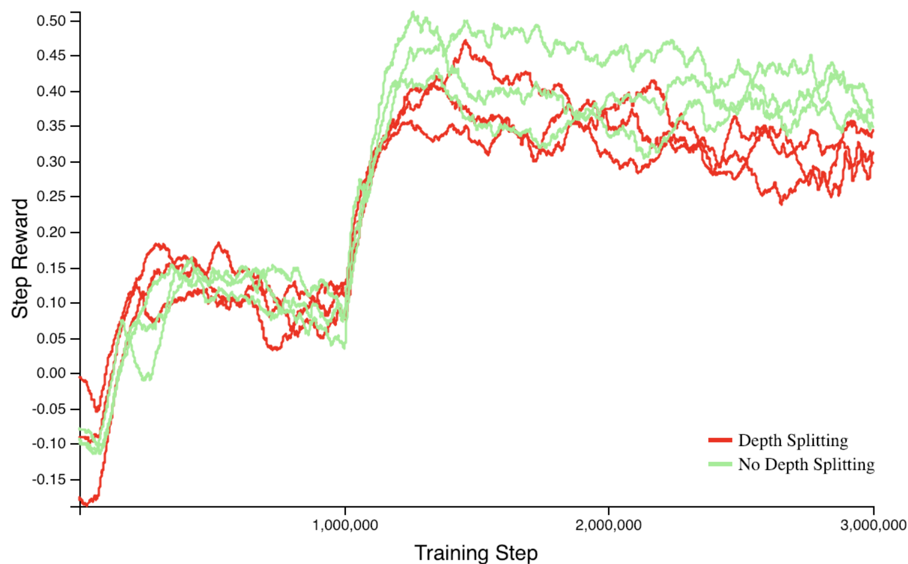

Figure 6.1: Rewards earned per step throughout training. At 1m steps the second part of training commences, thus the jump. No depth splitting appears to offer a higher rewards per step during training.

Run	Reward %	Drive %	Km	Km/Hr	Km/OOL	Km/Collision
1	0.1	21.4	01.8	02.1	0.01	0.18
2	12.3	**66.4**	13.4	05.0	0.03	**13.4**
3	**24.1**	63.2	**25.6**	**10.1**	**0.16**	1.35
Mean	12.2	50.3	13.6	5.7	0.07	4.98

Table 6.1: Results of no depth splitting.

Run	Reward %	Drive %	Km	Km/Hr	Km/OOL	Km/Collision
1	**58.4**	**78.9**	**66.3**	**21.0**	0.05	**2.07**
2	38.4	71.5	39.9	14.0	0.23	0.67
3	15.5	52.6	19.8	9.4	**0.35**	0.28
Mean	37.4	67.7	42.0	14.8	0.21	1.01

Table 6.2: Results of depth splitting.

fused data volume of size 10x160x160. Not using depth splitting instead passes the depth map layer directly into fusion, resulting in a final data volume of size 6x160x160. Aside from this difference, all other hyperparameters and architectural components are identical between the two. Both experiments use probabilistic exploration and ground truth image segmentation data.

Purely from the reward curves, it appears that not using depth splitting is a preferential strategy. Once the second batch of training kicks in, the models training without depth splitting reaches a higher running reward value faster, and tends to remain higher for much longer.

The evaluation results, however, tell a completely different story. Even though the no depth splitting models are confident in their capacity during training, they have a hard time performing well in the test environment. In terms of total reward attained, the depth splitting experiments are significantly better than just passing the depth directly into the model. Without depth splitting, the model only attains a mean reward percentage of $\mu = 12.2$ with standard deviation of $\sigma = 12.0$. Depth splitting, on the other hand, attains values of $\mu = 37.4$ and $\sigma = 21.5$.

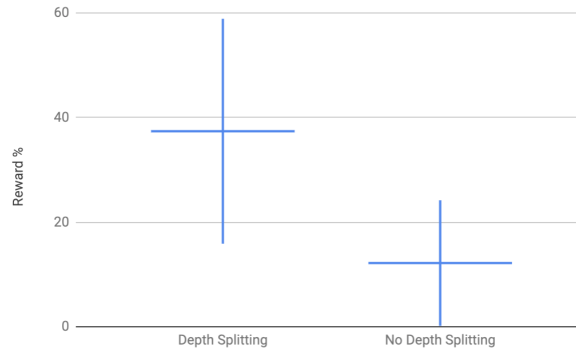

Figure 6.2: Mean and first standard deviation comparison between the two experiments; depth splitting and no depth splitting. Depth splitting presents a clear advantage over no depth splitting.

Comparing the training reward curves with the evaluation results tells a different story. Given the lower observation space of providing depth as a single layer (no depth splitting), it makes sense that the model would be able to learn faster, and thus converge and a local minima fairly quickly. The model appears to overfit on this limited supply of data, performing poorly on yet unseen data.

Without depth splitting, the vehicle has a propensity to travel so slow that the associated behaviors are difficult to properly evaluate. Realistically the model has a hard time training, and would need additional training before attaining the higher reward values.

Thus the hypothesis can be confirmed with confidence: in this architecture, splitting out the depth map into multiple layers based on pixel semantic segmentation produces significantly better models than just passing the depth map as a single layer into the convolutional neural network. This makes sense, after all, as all other layers of a CNN typically operate on depths of 64, 128, 256, or layers, yet often an image with only 3 layers is passed in at the start. In fact, these results imply that further layer splitting could in fact be beneficial to the model, allowing it to more rapidly pick out important observation features.

6.3 Exploration Style

The following hypothesis is to be evaluated: Using probabilistic exploration will produce better models than traditional linear decay exploration.

Determining whether or not this hypothesis is correct requires running two sets of experiments, much like the previous hypothesis. A set of 3 models are trained using linear exploration, and another set of 3 models are trained using probabilistic exploration. Since the previous depth splitting experiment set used probabilistic exploration, that data is used for the probabilistic exploration as all hyperparameters and configuration options between the two are virtually identical.

During learning, the linear exploration models appear to be a good deal better than those trained on probabilistic exploration. It is worth pointing out, however, that models trained on probabilistic exploration have a higher propensity to crash if they end up with a high exploration rate for N steps. This higher crash rate during training can yield fairly substantial negative rewards, since crashing applies a 100 point penalty.

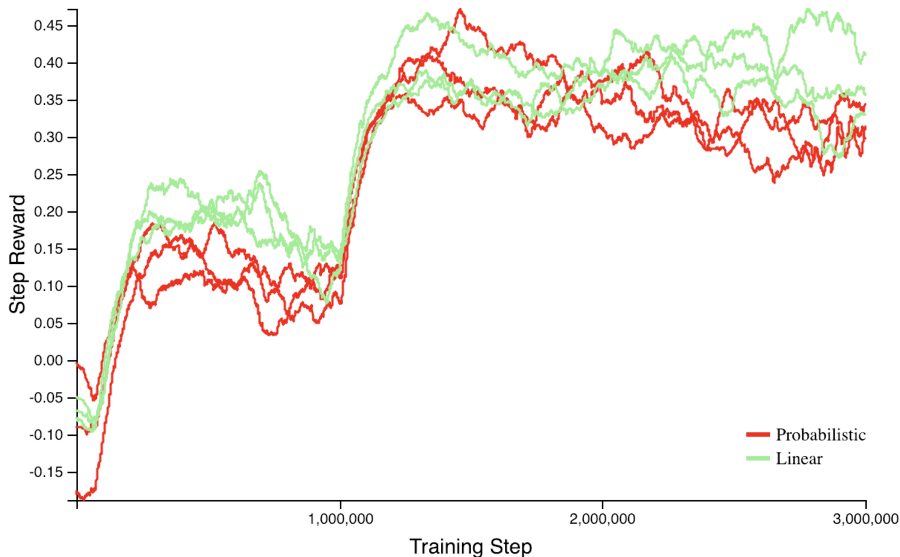

Figure 6.3: Rewards earned per step throughout training. At 1m steps the second part of training commences, thus the jump. Linear exploration appears to offer higher rewards per step during training.

Run	Reward %	Drive %	Km	Km/Hr	Km/OOL	Km/Collision
1	**30.6**	87.0	32.0	9.2	0.12	3.20
2	20.9	**90.6**	24.9	6.8	0.01	24.9
3	29.7	80.6	**32.1**	**10.0**	0.04	2.68
Mean	27.1	86.1	29.7	8.7	0.06	10.27

Table 6.3: Results of linear exploration.

Run	Reward %	Drive %	Km	Km/Hr	Km/OOL	Km/Collision
1	**58.4**	**78.9**	**66.3**	**21.0**	0.05	**2.07**
2	38.4	71.5	39.9	14.0	0.23	0.67
3	15.5	52.6	19.8	9.4	**0.35**	0.28
Mean	37.4	67.7	42.0	14.8	0.21	1.01

Table 6.4: Results of probabilistic exploration. *Depth splitting experiment data.*

The linear exploration evaluation attains moderate results. All three experiments yield models that are fairly equal in quality, producing a standard deviation of $\sigma = 5.36$ and a mean of $\mu = 27.1$. On the other hand, the probabilistic exploration experiments yield a wider range of model qualities. The standard deviation is much higher at $\sigma = 21.5$, but this included higher mean of $\mu = 37.4$. These experiments present an advantage to the use of probabilistic exploration over linear exploration.

It is worth noting that the probabilistic explorer models tend to drive at much slower

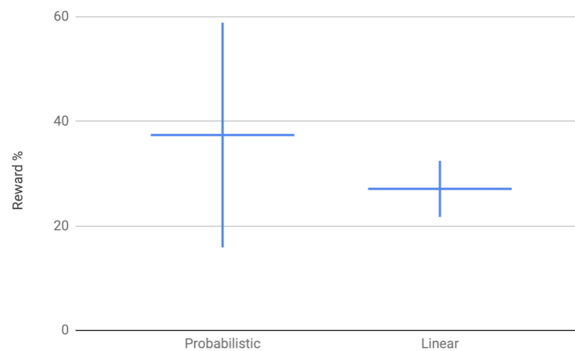

Figure 6.4: Mean and first standard deviation comparison between the two experiments; probabilistic and linear exploration. Probabilistic exploration presents a clear advantage over linear exploration.

speeds than the linear explorer models. This is paired with higher drive percentiles and lower collision rates. Though none of the models attain the high reward percentile desired, implying additional training might be required (or a larger model in the case of underfitting), it appears that the probabilistic exploration models are actually in a more advanced stage of training than the block exploration models. The reason for this is that the difficulty level for the vehicles to stay in lane increases exponentially as the vehicle speed increases.

When driving at 10km/hr, a vehicle can provide a few erroneous turn commands in a row, but still have time to recover before exiting the lane and potentially colliding with an object. At higher speeds, however, even a few frames in a row of incorrect inputs (such as a hard right during a left turn) can quickly send the vehicle on an irrecoverable course. The probabilistic models having a generally higher speed implies that they have more success at high speeds, while the linear exploration models either stagnate at lower speeds or at least are not ”confident” enough in the higher speed rewards to drive any faster.

These results indicate that the hypothesis is in fact met: probabilistic models, at least in this set of experiments, had an increased probability of producing a higher quality model. The likely cause of this improvement is from the dynamic exploration values alternating between very high exploration behavior and high exploitation behaviors. The vehicle can occasionally drive with low exploration, allowing it to test how well it actually understands the world and train against those deltas. The vehicle can also drive at high exploration, allowing it to experiment with driving faster, accidentally driving out of lane, and experience colliding with objects. Without this high exploration late in the cycle, the model risks being either too conservative or catastrophically forgetting important behaviors.

6.4 Deeplab Perception

The following hypothesis is to be evaluated: Does a model train better when using a ground truth perceptor, or a realistic one?

The models trained with Deeplab have a reward-per-step advantage of about 66%. In

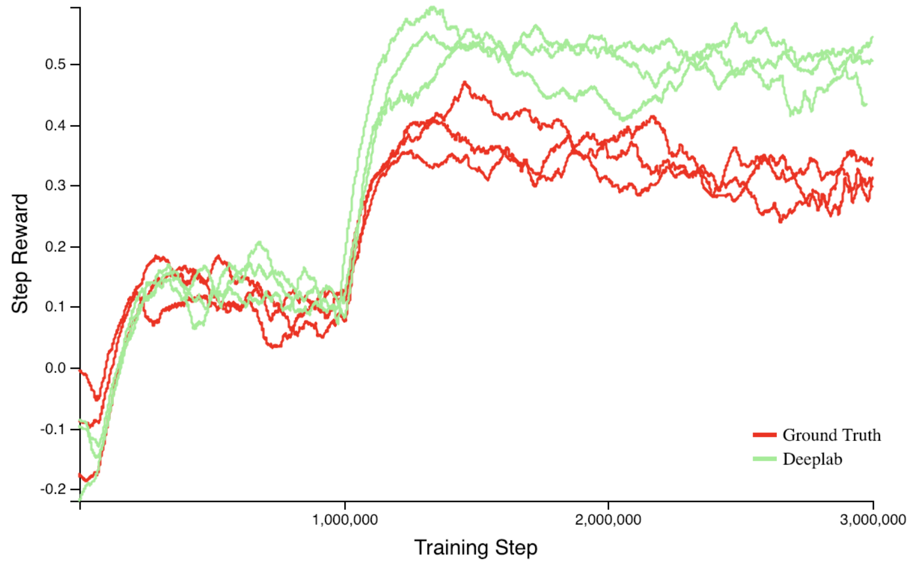

Figure 6.5: Rewards earned per step throughout training. At 1m steps the second part of training commences, thus the jump. During training, Deeplab offers substantially higher rewards each step.

fact, a few thousand iterations into the second round of training (starting at the 1 millionth step), the worst Deeplab model's reward-per-step was never surpassed by even the best ground truth model's reward-per-step.

During evaluation, however, models trained with ground-truth segmentation attained a total reward of 37.4 in comparison to the 28.7 total reward attained by models trained with Deeplab. That said, training against Deeplab takes around twice as long due to the time cost of running Deeplab predictions every frame.

From a behavior perspective, ground truth segmentation models tend to have higher speeds and slightly more unsafe driving characteristics when compared to the Deeplab models. As discussed previously, an increase in speed can yield a higher likelihood of collisions while training is taking place. Since the Deeplab models are driving slower, it is not surprising that the safety behaviors are better, but it might also be a signal that the models are less mature since they have not yet developed the confidence to drive faster.

The high outliers show that training these models is tenuous, and models should be

Run	Reward %	Drive %	Km	Km/Hr	Km/OOL	Km/Collision
1	**58.4**	**78.9**	**66.3**	**21.0**	0.05	**2.07**
2	38.4	71.5	39.9	14.0	0.23	0.67
3	15.5	52.6	19.8	9.4	**0.35**	0.28
Mean	37.4	67.7	42.0	14.8	0.21	1.01

Table 6.5: Results of ground truth model with ground truth evaluation. *Depth splitting experiment data.*

Run	Reward %	Drive %	Km	Km/Hr	Km/OOL	Km/Collision
1	38.6	**89.3**	41.3	11.6	0.04	13.8
2	4.4	72.4	7.8	2.7	0.01	0.1
3	**43.1**	85.0	**46.6**	**13.7**	**0.06**	**23.3**
Mean	28.7	82.2	31.9	9.3	0.04	12.4

Table 6.6: Results of Deeplab model with Deeplab evaluation

evaluated regularly and additional training should be induced when results are insufficient. These results also speak to the issues with deep learning at large - a new architecture can produce fantastic results, beating the state of the art, but then fail to attain those same results if trained again from scratch. For any deep learning model of even modest complexity, multiple independent experiments need to be run to highlight the robustness of the

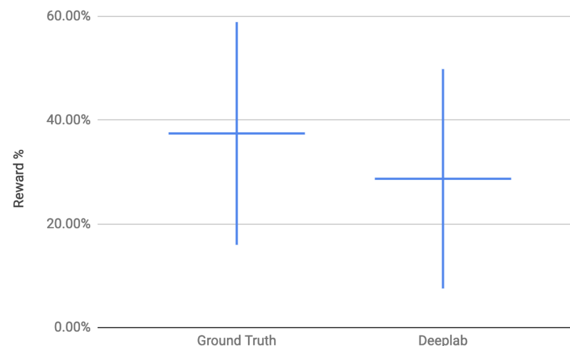

Figure 6.6: Mean and first standard deviation comparison between the two experiments; ground truth segmentation and Deeplab segmentation. While ground truth seems to have better results, the outliers are significant enough that the higher mean can not be said to have statistical significance.

methodology.

6.5 Post-Train Perception Adjustment

The following hypothesis is to be evaluated: Changing the perceptor on a fully trained model will reduce model quality.

If a model is overfitting against its input data, changes to that input data (which is a function of the perceptors) are theoretically more likely to have a deleterious effect on model quality. Evaluation of this hypothesis requires taking a trained model, switching out the perceptor, then re-running the exact same evaluation to see what metrics change.

With only a few minor exceptions with behavioral metrics, changing out the semantic segmentation perceptor on a model appears to yield reduced model quality. Even when the perceptor quality is improved, in this case from Deeplab predictions to ground truth, model performance is likely to take a small hit. This might seem counter-intuitive, as improving the perceptors should in theory produce better model results.

All of the models originally trained on Deeplab end up driving farther once they are switched over to ground truth. This increased driving distance is generally paired with a slight decrease in model speed, which implies the model might be driving more cautiously and thus less likely to produce an early termination. It is also possible that the uncertain perception produces less stagnation, thus bumping up total drive time. When paired with the dramatically increased km per collision rate, however, it becomes fairly clear that these models are far more likely to second guess themselves and err on the side of caution.

Switching from ground truth to Deeplab yields opposite results. Drive duration tends to decrease, and almost all behaviors suffer as a result of the reduced perceptor quality. This is most likely because the model is either overfitting to the high quality data, or fitting to very minute features that only existed due to perfect segmentation. Given the relatively short training time, it is highly unlikely that overfitting is the culprit.

What is most interesting with these results is the trend lines suggesting that higher

Run	Reward %	Drive %	Km	Km/Hr	Km/OOL	Km/Collision
1	-13.5	-12.0	-18.8	-3.3	+0.04	-1.12
2	-6.3	-13.6	-5.9	+0.7	-0.17	-0.11
3	+3.1	+16.9	+2.8	-1.3	-0.24	+0.06
Mean	-5.5	-2.9	-7.3	-1.3	-0.12	-0.39

Table 6.7: Results of models originally trained on ground truth semantic segmentation being converted to use a Deeplab perceptor.

Run	Reward %	Drive %	Km	Km/Hr	Km/OOL	Km/Collision
1	-0.35	+6.3	+3.9	+0.3	-0.02	-2.5
2	+1.11	+2.5	-0.9	-0.4	+0.01	+1.0
3	-5.21	+1.1	-3.1	-1.1	-0.03	+20.2
Mean	-1.48	+3.3	-0.0	-0.4	-0.01	+5.3

Table 6.8: Results of models originally trained on Deeplab being converted to use a ground truth semantic segmentation perceptor.

quality models will be more likely to suffer a reduction in model quality during the switch, while lower quality models are more inclined to have increased model quality. This is likely due to an increase in caution by the agent. While at high speeds, an increase in caution might extend the drive duration due to fewer accidents, but the overall attained reward might still be lower. When at lower speeds, however, increased drive distance attained from cautious driving is more likely to overcome lost speed.

Training an entire Deeplab-based model takes about 70 hours, while those with ground truth segmentation only take about 35 hours. Models trained with ground truth that are retrofitted to have Deeplab models actually outperform the Deeplab-based models by about 3.21% reward. These results could mean that models should be trained with the highest quality data possible, even if that data is of a higher quality than the data available during deployment. That said, it might be worth conducting further research by training a model with 80%-20% ground truth and Deeplab segmentation, drawing from a uniform distribution each frame. This could produce better results, as the non-perfect data may help reinforce higher level and thus more generic features, while the higher quality data is

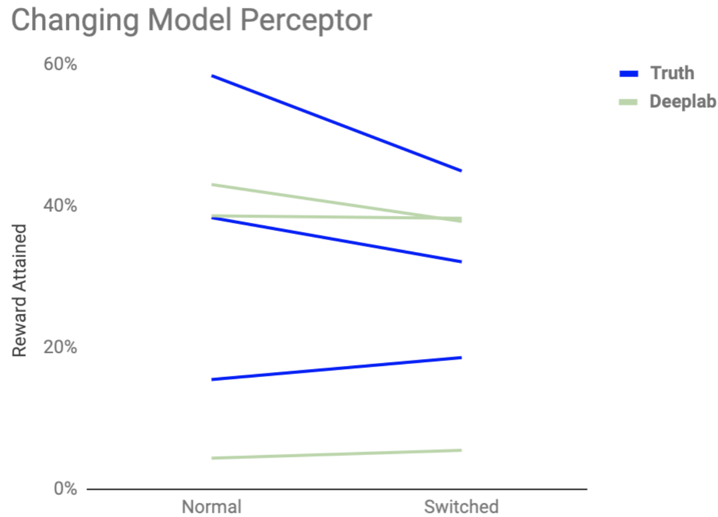

Figure 6.7: Chart showing the change in evaluation performance when a model trained with one perceptor is switched to another perceptor. Left column contains the results of models evaluated with the same perceptors as they are trained with. The right column shows the model evaluation results once the perceptor has been flipped either from ground truth to Deeplab, or vice versa. The blue (dark) lines are models originally trained with ground truth segmentation, and the green (light) lines are originally trained with Deeplab. Switching from ground truth to Deeplab appears to produce more significant changes. Better performing models tend to become worse when switched, while worse models tend to improve slightly when switched.

both faster to train against and allows the model to pick up on some of the nuances of the observation space.

6.6 Model Behavior Success

The best model is the first run of the depth splitting experiment, which attains a reward of 58.4% of the maximum. To properly evaluate the behaviors, each desired behavior must be compared to its resultant behavior.

The best model is able to attain a fairly high average speed of 21km/hr in comparison to the desired 25km/hr. With these higher speeds, however, comes more rigid requirements for lane-keeping and collision avoidance. Small perturbations in model predictions that are acceptable at lower speeds can cause a collision while driving at a faster speed. What is

Version	Reward %	Drive %	Km	Km/Hr	Km/OOL	Km/Collision
Best	58.4	78.9	**66.3**	**21.0**	0.05	2.07
Extra	57.7	**94.8**	59.5	15.7	0.15	**No collisions**
Extra+Deeplab	**59.5**	92.3	60.7	16.4	**0.19**	15.2

Table 6.9: Adding additional training yields a significant improvement in collision rate and episode termination rate, but the travel speed is somewhat lower. Both reward percentiles are about equal, with the additional training yielded a small reduction. Evaluating on Deeplab actually yields slightly improved reward values and behaviors (with the exception of collision rate).

obvious from these results is that the agent, which has an out of lane event roughly every 52 meters, has trouble staying perfectly in lane. The collision rate of approximately one collision every 480 meters is also unsatisfactory.

With such a short training time of only about 275,000 actual train steps taking place, however, it is not clear that the agent's learning capacity has been fully saturated. As a result, one final experiment is run for 3 million steps, a learning rate of 1e-6, a train frequency of 18 steps, and exploration blocks with sizes: 60% chance of 0% exploration, 20% chance of 10% exploration, 10% chance of 20% exploration, and 10% chance of 30% exploration.

From a purely reward driven perspective, the additional training actually lowers the reward slightly, which implies the architecture as configured has a hard time passing that level of reward. What is interesting, however, is that the behaviors from the additional training are significantly better (with the exception of the average speed). Across the 59.5km travelled during evaluation, not a single collision took place, and the out of lane rate is only about 35% that of the original model.

The model is then evaluated on Deeplab, and the results are actually a touch better in most categories. The speed is actually higher than the ground truth segmentation. A small number of collisions took place, but the out of lane rate improved over the ground truth segmentation model. Even though the total drive rate is a few percentage points lower, the

Behavior	Desired	Best	Extra	Extra+Deeplab
Speed in km/hr	*25.0*	**21.0**	15.7	16.4
OOL per km	*0.0*	19.1	6.7	**5.2**
Collisions per km	*0.0*	0.5	**0.0**	0.1

Table 6.10: Behaviors of the best model, along with behaviors of the best model with additional training. The average speed decreases, but there were no collisions and a reduced out of lane instance rate. When further evaluated on Deeplab, the speed and out of lane rate is slightly improved, but there were a few collisions.

accumulated reward ended up being the best of all of the models by a percentage point.

The speed of 16km/hr is very likely a result of the bucketization of the actions. The model learns that on straightaways it can hold forward with +0.5 throttle and a steering of 0.0, yielding a speed of about 16.26km/hr. For most of the evaluation, this is the exact behavior observed by the model that received additional training.

The behaviors on the Deeplab evaluation does not adhere as closely to the +0.5 throttle consistency, regularly poking above or falling below. This is likely due to the irregularities in segmentation imposed by the switch causing confusion on the model. The fact that the switch from ground truth to Deeplab actually yields slightly better results in most categories, however, implies that fairly dramatic shifts in upstream perceptors can actually improve downstream results.

Given the results, it can be believed that additional training will bring the out of lane rate even lower. To maintain a speed of 25km/hr, the model would have to alternate between +0.5 throttle and +0.75 throttle at a perfect rate. Alternatively, additional buckets could be added to the model output, allowing a higher degree of fidelity by the model when selecting the desired speed.

Chapter 7

Conclusion

This thesis has presented an autonomous vehicle agent that proved able to drive at slow speeds through a small simulated town. Necessary background material was discussed, allowing the reader to fully comprehend the components required to make the vehicle operate. The architecture was discussed in full, along with how the agent was trained, and all of the necessary evaluations the model underwent to provide appropriate metrics for model comparison. In total 20 experiments were run over the course of 29 days.

The architecture as designed in this thesis is fairly simple, but delivers a fairly high degree of success. The model proved capable of learning to drive close to the desired speed, and the final model had no collisions during a 60km driving segment (though it had regular out of lane events which are of course unsafe). This model was then retrofitted with a Deeplab v3 [3] and re-evaluated to yield even better results.

A novel technique dubbed probabilistic exploration was introduced, which proved to yield better model results given the same training window. By expanding the simple ε-greedy model to alternate between lower and higher exploration rates in large step intervals, the agent was able to produce better training data for the underlying neural network. This yielded better performance and a lower likelihood of overfitting.

Depth splitting was also introduced, a novel approach to handling inputs for CNNs. By splitting out the contents of one layer into multiple semantically meaningful layers, the CNN was better able to learn features that had a direct impact on prediction quality. Given the relatively small number of convolution layers in the network, any improvements in input data formulation are shown to have significant downstream impacts.

One of the most interesting observations of the experiments was that the models that attained higher rewards during training tended to have worse results during evaluation. Be-

yond this, there is no obvious correlation between training step reward and the evaluation results. The reward curves for each configuration trended together fairly closely, even if their results were significantly different. As far as experiments in this thesis were concerned, the reward earned per step was an inconsequential value.

It was shown that a model can be trained on a ground truth perceptor, then evaluated (put into the field) with a non-perfect perceptor. A comparative study was made between models trained fully on ground truth segmentation then switched to Deeplab, and models fully trained on Deeplab. Interestingly, models trained on ground truth first and then evaluated on Deeplab had higher degrees of success than those trained entirely on Deeplab. It is important to note that those results might not hold for all configurations. With this architecture, at least, training with ground truth segmentation was significantly faster and yielded better results, implying that non-perfect segmenters should not be integrated into the training pipeline, even if they will be used in production. In fact the best model of this study was the final ground-truth trained model that was converted to use Deeplab.

7.1 Future Research

Though this architecture proved successful, it is just an introductory example. There is plenty of room to expand this model with the hopes of improving the task, or even expanding the task requirements to include pedestrian avoidance, vehicle avoidance, or even light navigation.

One of the most obvious weaknesses of the existing model is the discretization of the state space. Increasing the number of buckets may well improve model performance, but switching wholesale into continuous model configurations [75] also has the potential to smooth out the model results.

DDQN [24] is a strong modeling framework, but other options exist that could allow increased parallelism during train time. A3C [25], as used by the Carla paper [4], allows multiple agents to train in parallel, thus providing a tool to induce a very high amount of

learning with the same total amount of time commitment (though extra compute resources are required). DQN [22] also suffers from the lack of memory, a problem that is only lightly addressed with the frame stacking [22] technique. Switching over to an DRQN [26] model, or adding in LSTMs [28] to store important information could produce significantly better performance.

Another simple component that has room for improvement is the controller, which currently is just an exponentially weighted moving average. This component could be swapped out for a more robust PID controller, which can be configured to smooth prediction data out before sending it to the agent for action, but be more responsive and allow the model to fully reach the desired speed (due to the integral component of PID).

During training of Deeplab v3 [3], it was noticed that training on a higher output dimension and then reducing the output dimensions actually yielded superior results over just training with a lower number of outputs. This may have been a fluke, but it could be worth exploring further. An experiment might have 10 superclasses, each comprised of 10 subclasses. One instance would train on the 100 total classes, then run prediction and pair down to the 10 superclasses before performing accuracy metrics. The second instance would just train on the 10 superclasses, then run prediction to see how the results compare to the prior instance.

While probabilistic exploration does appear promising, it can also be combined with more advanced exploration techniques like bootstrapped exploration [34]. A more robust probabilistic exploration technique would draw from a distribution of exploration styles, selecting a new style every N steps. Much in the same way that ensemble collections of models can have superior performance over a single model, this could allow many proven exploration methods to be compiled and each have their chance to influence model performance.

Depth splitting also has a plethora of extended research opportunities. What is clear is that selecting what information is provided to a CNN is not the only relevant decision, but

also *how* that information is presented. Almost all image detection algorithms use RGB input layers, so examples of pre-input information modifications include: adding a black and white layer, adding a layer with very high contrast, adding a few copies of the base layers that have slight modifications to scale, etc. A better understanding of exactly what information CNNs excel at processing could yield significant improvements on almost all CNNs in operation.

Training on ground truth perceptors then switching to non-perfect perceptors for evaluation/production demands further study. Though the results presented in this thesis are a strong indication that non-perfect perceptors are preferred for the training pipeline, a wider study should be performed on more model types and configurations. Research can also be made into adding a ”tail” to training, where the first 90% of the training is performed on the ground truth perceptors, and the final 10% is performed on the non-perfect perceptors that will actually be used, allowing the model to more fully adapt and in theory produce higher quality predictions. If perceptors are later improved or swapped out, only the last 10% of training would need to be re-run instead of the entirety of training.

7.2 Final Comments

Autonomous vehicle agents are one of, if not the, greatest research challenges of this generation. To properly engage in research, one must understand modern convolutional neural networks, object detection algorithms, segmentation algorithms, reinforcement learning or supervised learning techniques, data fusion, and vehicle simulators, and that is just a drop in the bucket. Production level algorithms require researchers to know all of the pros and cons of each sensor type, why to use any given sensors, and how to fuse their data together. Localization algorithms are also incredibly complex, but have the potential to dramatically improve a vehicle’s understanding of the world.

And it can not be forgotten that any vehicle that may enter production must consider safety to be paramount. These vehicles are tasked with making transportation safer than

human drivers, so they must be able to live up to that measure. This is difficult to properly measure in and of itself, and requires an incredibly thorough understanding of all components in the architecture to have any hope of proving.

Progress has been accelerating lately, so there is hope yet that autonomous vehicles will one day rule the roads. Though this thesis just scratches the surface, it may well help bring new talent into the field, further accelerating the trend towards full transportation autonomy.

BIBLIOGRAPHY

[1] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. DeepTest: Automated Testing of Deep-Neural-Network-driven Autonomous Cars. *arXiv:1708.08559 [cs]*, August 2017. arXiv: 1708.08559.

[2] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L. Yuille. DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs. *arXiv:1606.00915 [cs]*, June 2016. arXiv: 1606.00915.

[3] Liang-Chieh Chen, George Papandreou, Florian Schroff, and Hartwig Adam. Rethinking Atrous Convolution for Semantic Image Segmentation. *arXiv:1706.05587 [cs]*, June 2017. arXiv: 1706.05587.

[4] 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.

[5] Electric Cars, Solar Panels & Clean Energy Storage | Tesla.

[6] Uber - Earn Money by Driving or Get a Ride Now.

[7] Waymo.

[8] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The Cityscapes Dataset for Semantic Urban Scene Understanding. *arXiv:1604.01685 [cs]*, April 2016. arXiv: 1604.01685.

[9] A. Geiger, P. Lenz, and R. Urtasun. Are we ready for autonomous driving? The

KITTI vision benchmark suite. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3354–3361, June 2012.

[10] Bernhard Wymann, Christos Dimitrakakis, Andrew D. Sumner, Eric Espi, and Christophe Guionneau. TORCS : The open racing car simulator. 2015.

[11] Craig Quiter and Maik Ernst. *deepdrive/deepdrive: 2.0*. March 2018.

[12] Xinlei Pan, Yurong You, Ziyan Wang, and Cewu Lu. Virtual to Real Reinforcement Learning for Autonomous Driving. April 2017.

[13] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. Image-to-Image Translation with Conditional Adversarial Networks. *arXiv:1611.07004 [cs]*, November 2016. arXiv: 1611.07004.

[14] Epic Games. Unreal Engine 4.

[15] Matthew Johnson-Roberson, Charles Barto, Rounak Mehta, Sharath Nittur Sridhar, Karl Rosaen, and Ram Vasudevan. Driving in the Matrix: Can Virtual Worlds Replace Human-Generated Annotations for Real World Tasks? *arXiv:1610.01983 [cs]*, October 2016. arXiv: 1610.01983.

[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. *arXiv:1604.07316 [cs]*, April 2016. arXiv: 1604.07316.

[17] Huazhe Xu, Yang Gao, Fisher Yu, and Trevor Darrell. End-to-end Learning of Driving Models from Large-scale Video Datasets. *arXiv:1612.01079 [cs]*, December 2016. arXiv: 1612.01079.

[18] Lex Fridman, Jack Terwilliger, and Benedikt Jenik. DeepTraffic: Crowdsourced

Hyperparameter Tuning of Deep Reinforcement Learning Systems for Multi-Agent Dense Traffic Navigation. *arXiv:1801.02805 [cs]*, January 2018. arXiv: 1801.02805.

[19] Zhuwei Qin, Fuxun Yu, Chenchen Liu, and Xiang Chen. How convolutional neural network see the world - A survey of convolutional neural network visualization methods. *arXiv:1804.11191 [cs]*, April 2018. arXiv: 1804.11191.

[20] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and Harnessing Adversarial Examples. *arXiv:1412.6572 [cs, stat]*, December 2014. arXiv: 1412.6572.

[21] Lex Fridman, Daniel E. Brown, Michael Glazer, William Angell, Spencer Dodd, Benedikt Jenik, Jack Terwilliger, Julia Kindelsberger, Li Ding, Sean Seaman, Hillary Abraham, Alea Mehler, Andrew Sipperley, Anthony Pettinato, Bobbie Seppelt, Linda Angell, Bruce Mehler, and Bryan Reimer. MIT Autonomous Vehicle Technology Study: Large-Scale Deep Learning Based Analysis of Driver Behavior and Interaction with Automation. *arXiv:1711.06976 [cs]*, November 2017. arXiv: 1711.06976.

[22] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

[23] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv:1409.1556 [cs]*, September 2014. arXiv: 1409.1556.

[24] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and

Nando de Freitas. Dueling Network Architectures for Deep Reinforcement Learning. *arXiv:1511.06581 [cs]*, November 2015. arXiv: 1511.06581.

[25] Volodymyr Mnih, Adri Puigdomnech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. *arXiv:1602.01783 [cs]*, February 2016. arXiv: 1602.01783.

[26] Matthew Hausknecht and Peter Stone. Deep Recurrent Q-Learning for Partially Observable MDPs. *arXiv:1507.06527 [cs]*, July 2015. arXiv: 1507.06527.

[27] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, March 1994.

[28] Sepp Hochreiter and Jrgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, November 1997.

[29] On-Road Automated Driving (ORAD) committee. Taxonomy and Definitions for Terms Related to On-Road Motor Vehicle Automated Driving Systems. Technical report, SAE International.

[30] Introducing Navigate on Autopilot, October 2018.

[31] Richard S. Sutton and Andrew G. Barto. Reinforcement Learning: An Introduction. *IEEE Transactions on Neural Networks*, 16:285–286, 1998.

[32] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. A Brief Survey of Deep Reinforcement Learning. *IEEE Signal Processing Magazine*, 34(6):26–38, November 2017. arXiv: 1708.05866.

[33] Haoran Tang, Rein Houthooft, Davis Foote, Adam Stooke, Xi Chen, Yan Duan, John Schulman, Filip De Turck, and Pieter Abbeel. #Exploration: A Study of Count-Based

Exploration for Deep Reinforcement Learning. *arXiv:1611.04717 [cs]*, November 2016. arXiv: 1611.04717.

[34] Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. Deep Exploration via Bootstrapped DQN. *arXiv:1602.04621 [cs, stat]*, February 2016. arXiv: 1602.04621.

[35] Brendan O'Donoghue, Ian Osband, Remi Munos, and Volodymyr Mnih. The Uncertainty Bellman Equation and Exploration. *arXiv:1709.05380 [cs, math, stat]*, September 2017. arXiv: 1709.05380.

[36] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[37] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *arXiv:1512.03385 [cs]*, December 2015. arXiv: 1512.03385.

[38] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going Deeper with Convolutions. *arXiv:1409.4842 [cs]*, September 2014. arXiv: 1409.4842.

[39] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You Only Look Once: Unified, Real-Time Object Detection. *arXiv:1506.02640 [cs]*, June 2015. arXiv: 1506.02640.

[40] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation. *arXiv:1511.00561 [cs]*, November 2015. arXiv: 1511.00561.

[41] Bichen Wu, Alvin Wan, Forrest Iandola, Peter H. Jin, and Kurt Keutzer. SqueezeDet: Unified, Small, Low Power Fully Convolutional Neural Networks for Real-Time Object Detection for Autonomous Driving. *arXiv:1612.01051 [cs]*, December 2016. arXiv: 1612.01051.

[42] Marvin Teichmann, Michael Weber, Marius Zoellner, Roberto Cipolla, and Raquel Urtasun. MultiNet: Real-time Joint Semantic Reasoning for Autonomous Driving. *arXiv:1612.07695 [cs]*, December 2016. arXiv: 1612.07695.

[43] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5mb model size. *arXiv:1602.07360 [cs]*, February 2016. arXiv: 1602.07360.

[44] Michael Treml, Jos Arjona-Medina, Thomas Unterthiner, Rupesh Durgesh, Felix Friedmann, Peter Schuberth, Andreas Mayr, Martin Heusel, Markus Hofmarcher, Michael Widrich, Bernhard Nessler, and Sepp Hochreiter. Speeding up Semantic Segmentation for Autonomous Driving. October 2016.

[45] Jifeng Dai, Kaiming He, and Jian Sun. Instance-aware Semantic Segmentation via Multi-task Network Cascades. *arXiv:1512.04412 [cs]*, December 2015. arXiv: 1512.04412.

[46] Pedro O. Pinheiro, Ronan Collobert, and Piotr Dollar. Learning to Segment Object Candidates. *arXiv:1506.06204 [cs]*, June 2015. arXiv: 1506.06204.

[47] Sebastian Thrun and John J. Leonard. Simultaneous Localization and Mapping. In Bruno Siciliano and Oussama Khatib, editors, *Springer Handbook of Robotics*, pages 871–889. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[48] Raul Mur-Artal, J. M. M. Montiel, and Juan D. Tardos. ORB-SLAM: a Versatile and

Accurate Monocular SLAM System. *IEEE Transactions on Robotics*, 31(5):1147–1163, October 2015. arXiv: 1502.00956.

[49] M. Menze and A. Geiger. Object scene flow for autonomous vehicles. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3061–3070, June 2015.

[50] Michael Barnard. Tesla & Google Disagree About LIDAR – Which Is Right?, July 2016.

[51] Bo Li, Tianlei Zhang, and Tian Xia. Vehicle Detection from 3d Lidar Using Fully Convolutional Network. *arXiv:1608.07916 [cs]*, August 2016. arXiv: 1608.07916.

[52] Ben Popper. The billion dollar widget steering the driverless car industry, October 2017.

[53] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992.

[54] Richard Bellman. On the Theory of Dynamic Programming. *Proceedings of the National Academy of Sciences*, 38(8):716–719, August 1952.

[55] G. A. Rummery and M. Niranjan. On-Line Q-Learning Using Connectionist Systems. Technical report, 1994.

[56] RM French. Catastrophic forgetting in connectionist networks. *Trends in Cognitive Sciences*, 3(4):128–135, April 1999.

[57] Roger Ratcliff. Connectionist models of recognition memory: Constraints imposed by learning and forgetting functions. *Psychological Review*, 97(2):285–308, 1990.

[58] J. L. McClelland, B. L. McNaughton, and R. C. O'Reilly. Why there are complementary learning systems in the hippocampus and neocortex: insights from the successes

and failures of connectionist models of learning and memory. *Psychological Review*, 102(3):419–457, July 1995.

[59] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized Experience Replay. *arXiv:1511.05952 [cs]*, November 2015. arXiv: 1511.05952.

[60] 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.

[61] Weibo Liu, Zidong Wang, Xiaohui Liu, Nianyin Zeng, Yurong Liu, and Fuad E. Alsaadi. A survey of deep neural network architectures and their applications. *Neurocomputing*, 234:11–26, April 2017.

[62] Richard H. R. Hahnloser, Rahul Sarpeshkar, Misha A. Mahowald, Rodney J. Douglas, and H. Sebastian Seung. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature*, 405(6789):947–951, June 2000.

[63] Y. He, L. Chen, J. Chen, and M. Li. A novel way to organize 3d LiDAR point cloud as 2d depth map height map and surface normal map. In *2015 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pages 1383–1388, December 2015.

[64] J. Stuart Hunter. The Exponentially Weighted Moving Average. *Journal of Quality Technology*, 18(4):203–210, October 1986.

[65] Franois Chollet. Xception: Deep Learning with Depthwise Separable Convolutions. *arXiv:1610.02357 [cs]*, October 2016. arXiv: 1610.02357.

[66] Md Atiqur Rahman and Yang Wang. Optimizing Intersection-Over-Union in Deep Neural Networks for Image Segmentation. In George Bebis, Richard Boyle, Bahram Parvin, Darko Koracin, Fatih Porikli, Sandra Skaff, Alireza Entezari, Jianyuan Min, Daisuke Iwai, Amela Sadagic, Carlos Scheidegger, and Tobias Isenberg, editors, *Ad-*

vances in Visual Computing, Lecture Notes in Computer Science, pages 234–244. Springer International Publishing, 2016.

[67] Martn Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Man, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Vigas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015.

[68] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv:1412.6980 [cs]*, December 2014. arXiv: 1412.6980.

[69] Python.

[70] Eric Jones, Travis Oliphant, Pearu Peterson, and others. *SciPy: Open source scientific tools for Python*. 2001.

[71] Travis E. Oliphant. *Guide to NumPy*. CreateSpace Independent Publishing Platform, USA, 2nd edition, 2015.

[72] The friendly PIL fork (Python Imaging Library). Contribute to python-pillow/Pillow development by creating an account on GitHub, February 2019. original-date: 2012-07-24T21:38:39Z.

[73] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. *OpenAI Gym*. 2016.

[74] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A Distributed Framework for Emerging AI Applications. *arXiv:1712.05889 [cs, stat]*, December 2017. arXiv: 1712.05889.

[75] 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. *arXiv:1509.02971 [cs, stat]*, September 2015. arXiv: 1509.02971.