7-30-2020

Formal Language Constraints in Deep Reinforcement Learning for Self-Driving Vehicles

Tyler Bienhoff
University of Nebraska - Lincoln, tbienhoff@gmail.com

FORMAL LANGUAGE CONSTRAINTS IN DEEP REINFORCEMENT LEARNING FOR SELF-DRIVING VEHICLES

by

Tyler Bienhoff

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Stephen Scott

Lincoln, Nebraska

August, 2020

FORMAL LANGUAGE CONSTRAINTS IN DEEP REINFORCEMENT LEARNING FOR SELF-DRIVING VEHICLES

Tyler Bienhoff, M.S.

University of Nebraska, 2020

Adviser: Stephen Scott

In recent years, self-driving vehicles have become a holy grail technology that, once fully developed, could radically change the daily behaviors of people and enhance safety. The complexities of controlling a car in a constantly changing environment are too immense to directly program how the vehicle should behave in each specific scenario. Thus, a common technique when developing autonomous vehicles is to use reinforcement learning, where vehicles can be trained in simulated and real-world environments to make proper decisions in a wide variety of scenarios. Reinforcement learning models, however, have uncertainties in how the vehicle acts, especially in a previously unseen situation that can lead to dangerous situations with humans onboard or nearby. To improve the safety of the agent, we propose formal language constraints that augment a standard reinforcement learning agent while being trained in a simulated self-driving environment. The constraints help the vehicle navigate turns and other situations by penalizing the agent when an action is chosen that could lead to a dangerous situation such as a collision. Empirically, we show that the agent, with these constraints, has a slight performance improvement as well as a significant decrease in collisions. Future work can expand upon the current constraints and evaluate using different reinforcement learning algorithms with constraints for training the self-driving agent.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The benefits of self-driving vehicles are enormous and could potentially change how millions of people move around in their daily lives. The average American spends 310 hours driving every year [13]; with autonomous vehicles, this is additional time each person could spend working or otherwise enriching their lives. When it comes to safety, 93% of vehicle accidents are a result of "human errors and deficiencies" [32]. Hence, there is substantial room for reduction in accidents with an automated system compared to a human driver since machines can have additional sensors, compute decisions faster, and do not become fatigued or distracted when driving.

An autonomous vehicle (AV) is one that operates with some level of automation. The Society of Automotive Engineers (SAE) defines six levels of automation for vehicles from level 0 with no automation all the way to level 5 [16]. At level 5, the vehicle is fully autonomous and can operate on all roadways in all circumstances. While lower levels of automation are currently available (e.g., basic cruise control or Tesla autopilot) and can substantially improve the safety of a vehicle when paired with a human driver [3], we are focusing on level 5, fully autonomous systems that operate without any human operator.

Researchers have been working on AVs for decades. In 1988, researchers at Carnegie Mellon built ALVINN [22], an AV that used a simple neural network to steer and navigate on a road. In the 2000s, the US Defense Advanced Research Projects Agency (DARPA) launched the DARPA Grand Challenge to spur development to create fully autonomous ground vehicles hosting several events over the years. In 2005, 5 of the 23 teams competing in the challenge finished the 212-kilometer off-road course with fully autonomous vehicles [2]. In 2007, 6 of 11 teams completed a 96-kilometer urban course obeying all traffic laws [4]. Despite these promising early results and many of the participants working on AVs after these challenges, self-driving cars have not yet arrived for the average consumer to use.

Nevertheless, major technology companies, startups, and car manufacturers continue to develop technology for AVs. Waymo, a sister company of Google, is one of the leading companies whose test vehicles travel an average of 13,000 miles between disengagements where a safety driver monitoring the vehicle must take control [5]. In April 2017, Waymo launched a limited self-driving taxi service in Phoenix, Arizona and even runs some vehicles without safety drivers or any other employees in the vehicle [28, 25]. While the service is still running, it is limited to several hundred preapproved participants and has not expanded beyond the small region in Phoenix.

Tesla is also developing and even selling self-driving technology to consumers for thousands of dollars which they say will one day make their vehicles fully autonomous. Currently, the software, called Autopilot, can operate near autonomously on highways, maintaining the position in the lane, slowing down or speeding up based on traffic, and even automatically changing lanes to pass slow vehicles and take highway exchanges [1]. However, as the vehicle is not fully autonomous, the driver still has to pay attention and be prepared at any time to take control of the vehicle. Furthermore, the car lacks the capability to drive in basic urban environments as it currently does not respond to stop signs, traffic lights, or other road signs. Elon Musk, the CEO of Tesla, claims that Teslas will be fully autonomous and able to drive without a human operator by the end of 2020 [17]. However, the deadline for this claim is not likely to be met with other companies and experts believing that AVs are still years away from becoming widespread [6, 18].

As researchers and companies continue to develop AV technology, they must deal with the extensive complexities of building a system that can sense, interpret, and safely and efficiently act in an ever-changing environment. Machine learning and reinforcement learning models have emerged as leading techniques to build a system that can train in simulated and real environments and teach itself how to properly behave when driving. The majority of the time, a well-trained model will make accurate choices but can sometimes make poor decisions, especially due to the stochastic properties of most algorithms. Since most machine learning models behave as a "black box," it is often not known why a specific decision is made and how to prevent such an action from occurring again in the future. A safe reinforcement learning algorithm attempts to eliminate these bad outcomes from the model. Thus, allowing an agent to freely learn how to solve the task while reducing the number of dangerous situations the agent encounters. One technique to increase the safety of agents is to impose outside constraints on the actions an agent can make to prevent the agent from reaching a dangerous state. The constraints can either limit the available actions to the agent or reduce the reward received by the agent when taking the action.

The following are the contributions presented in this work:

- We add a formal constraint framework to a baseline reinforcement learning agent in a simulated driving environment. Previous works have used constraint-like behaviors to assist self-driving agents, and this framework was previously introduced for reinforcement learning; however, we are the first to add a formal framework to a self-driving environment.

- We develop three constraints to improve the performance and safety of the agent in the environment: Strict Turn, Loose Turn, and Dithering constraints.
    - The *Strict Turn* constraint helps the agent navigate turns in intersections by penalizing the agent for steering in the wrong direction.
    - The *Loose Turn* constraint follows a similar idea as the Strict Turn constraint but is less restrictive in raising violations.
    - The *Dithering* constraint assists the agent when driving straight by raising violations when the vehicle is steering back and forth or dithering in the lane.
- The constraints are empirically evaluated in the simulated environment with various hyperparameters. Overall, the constraints reduce the number of collisions the agent is involved in while successfully completing the same amount of episodes. The Dithering constraint provides a small, but consistent improvement to the agent, while the Strict Turn constraint, in specific configurations, provides the largest reduction in collisions. The results provide a promising example of how the constraint framework can improve the performance and safety of the agent.

The remainder of this thesis is organized into the following chapters. Chapter 2 provides background information on basic machine learning and reinforcement learning techniques as well as the formal language constraint framework. Chapter 3 discusses previous work related to this thesis. Chapter 4 explains the experimental setup including the environment, task, and base agent along with the constraints tested in this study. Results and related discussion of the tested constraints are provided in Chapter 5. Finally, Chapter 6 provides a summary and suggestions for potential related future work.

Chapter 2

Background: Machine Learning and Safety Constraints

Machine earning (ML) is a subfield of artificial intelligence within computer science. The goal in machine learning is for a system to automatically learn to complete a task from experiences without being explicitly programmed to complete the task. Machine learning algorithms generalize previous information to make predictions for new data that has not been seen before. In his 1997 book about ML [19], Mitchell gives the following definition, "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks T, as measured by P improves with experience E."

Deep learning is a subfield within machine learning that has emerged over the past few years as advances in computing and big data have allowed ML models with millions of parameters to solve increasingly difficult tasks. Earlier models typically were much smaller since training larger networks was infeasible. Current state-of-the-art deep learning models can surpass human-level capabilities in specific tasks such as the ImageNet challenge where the goal is to classify millions of images into more than 21,000 classes [8].

This chapter provides a brief overview of the machine learning algorithms used in this paper. In Section 2.1, we introduce the supervised learning task within machine learning. Section 2.2 introduces neural networks, which have emerged as a very effective tool for solving a variety of supervised learning problems. Section 2.3 focuses on convolutional neural networks, a popular variant of neural networks often used for image processing. Section 2.4 introduces reinforcement learning and the actor-critic algorithm. Finally, Section 2.5 describes formal constraints with respect to reinforcement learning.

2.1 Supervised Learning

One common task within machine learning is supervised learning. Given some input-output pairs as training examples, the goal in supervised learning is to learn a function that maps the inputs to the outputs as accurately as possible. That is, we want to find a function F, such that $F(X) \approx Y$, where X represents the input set and Y represents the corresponding outputs also known as labels. The learned function can be used with unseen inputs to predict their corresponding labels. One example of supervised learning is classification where the goal is to classify or separate inputs into two or more categories. Hence, the function must learn to identify features in the inputs that separate examples with different labels.

2.1.1 Inputs and Outputs

Inputs and outputs can vary greatly depending on the problem. Some possible input types include images, text, and vectors of real or discrete values or any combination of these. For some machine learning algorithms, inputs must be encoded into a vector of real numbers. For example, an $n \times m$ RGB image can be represented with a $(3 \times n \times m)$-dimension vector of normalized real values with the red, green, and blue values of every pixel each corresponding to some value in the vector.

To encode a categorical attribute as a vector of real numbers, a technique called

one-hot encoding is used. Each possible value of the attribute is added as a binary variable. A value of "1" is given to the variable representing the current value of the attribute, and the variables representing the other possible values are given a value of "0." For example, if one of the attributes is color with possible values of "red," "green," and "blue," three binary variables are created. If a given training example has the color green, then the one-hot encoding would be $[0, 1, 0]$ with these variables corresponding to red, green, and blue, respectively.

Outputs are typically a real number or a vector of real numbers that can encode the desired label. Continuing the previous example with an image as the input, the goal might be to classify images depending on whether they contain a car, a truck, or an airplane. Hence, the output could be a vector of three real values with each value representing a category and how likely the image is to be in the category. The **softmax** function [11], defined in Equation 2.1, normalizes the output values and provides a probability distribution for which category the image belongs to based on the ML model based on the output values.

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{2.1}$$

The output value with the highest probability after applying the softmax function is the model's categorization of the image and also provides a confidence level of the model in the prediction. An output vector of $[0.40, 0.04, 0.06]$ would be converted to the following probability distribution $[80\%, 8\%, 12\%]$ with the model choosing the fist category to classify the input. Furthermore, the model has higher confidence with this prediction compared to, for example, another probability distribution with $[50\%, 30\%, 20\%]$ where the model still classifies the image in the first category but with less confidence. This covers just one of the many input and output encoding methods commonly used with machine learning algorithms.

2.2 Artificial Neural Networks

Artificial neural networks (ANNs) are a commonly used supervised learning algorithm. They were originally inspired by biological neural networks found in human and animal brains and involve connecting together thousands or millions of processing nodes called neurons. Each neuron takes some inputs, performs a simple computation, and produces an output that is usually passed as input to many other neurons where more simple computations occur. The computation involves weighting each input and computing the linear combination of these values. For example, a neuron with n inputs $(x_1, x_2, \ldots, x_n)$ and weights $(w_1, w_2, \ldots, w_n)$ would compute $z = w_1x_1 + w_2x_2 + \cdots + w_nx_n + b$, where b is the bias term that is adding to the linear combination. An activation function, such as the sign function or a step function, is applied to the linear combination to generate the output $y = g(z)$, where g is the activation function. For example, if the sign function is used, the output will be 1 if the linear combination z is positive, -1 if z is negative, and 0 if z is 0. Figure 2.1 illustrates a single neuron with three inputs.

The neurons are organized into layers with a typical neural network consisting of a single input layer, one or more layers of neurons known as hidden layers, and a single output layer. In a dense or fully connected ANN, all neurons directly receive a weighted input from every neuron in the previous layer. The network transmits signals from the input nodes through the hidden layers to the output nodes to process information with the input layer representing a single training sample and the output layer containing the predicted label. Figure 2.2 contains a simple neural network with three inputs, a single hidden layer, and one output. W_1 and W_2 are the weight

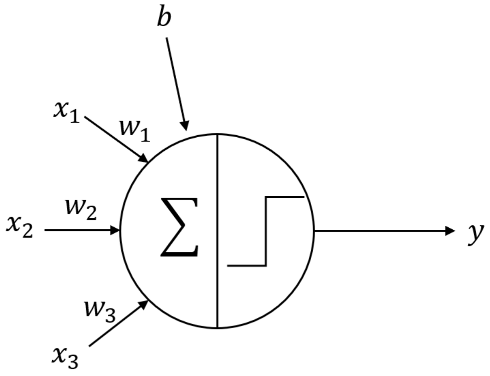


Figure 2.1: A single neuron within a neural network. This neuron has three inputs which are multiplied by the respective weights and added together with the bias to compute the linear combination. This value passes through an activation function that gives the output y.

matrices for the inputs to the hidden layer and output layer, respectively, whereas $\vec{b_1}$ and $\vec{b_2}$ and the bias vectors for the respective layers.

2.2.1 Gradient Descent & Backpropagation

Given a specific input, changing the values of the weight matrices and bias vectors changes the value of the output. Hence, these are the parameters of interest that must be optimized or "trained" for the neural network to make accurate predictions for the given task. One method to find optimal weight and bias values is to do a brute force parameter search. However, even using binary search techniques to speed this up, the large number of parameters in a moderately sized neural network (often in the millions) makes this search computationally infeasible. Hence, another technique called *stochastic gradient descent* is commonly used to find the optimal parameters for the neural network.

Stochastic gradient descent is an optimization algorithm with the goal of finding the parameters (i.e., the weights and biases of the neural network) that minimize an

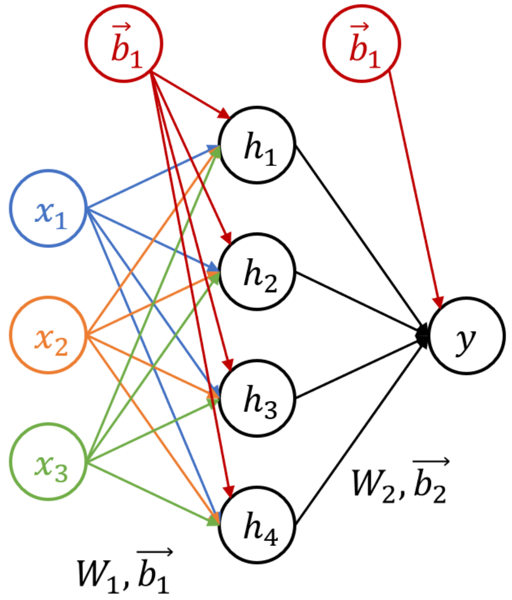


Figure 2.2: A simple neural network with three inputs (x_1, x_2, x_3), a single hidden layer $(h_1, ..., h_4)$, and a single output y. W_1 and $\vec{b_1}$ are the weight matrix and bias vector for the hidden layer. W_2 and $\vec{b_2}$ are the weight matrix and bias vector for the output layer.

error provided by a loss function. There are a variety of loss functions with the goal being to quantify how accurate the machine learning model is at labeling inputs to the model. One common loss function is the sum of squared errors given in equation 2.2. This is an appealing loss function as its gradient can be efficiently computed. W is the set of weight matrices to optimize, D is the set of training examples, $\vec{t_d}$ is the target output for training example d, and $\vec{y_d}$ is the predicted output of the model with weights W for training example d. The squared error between the target label and predicted label is summed for each training instance to give the error value for the given set of weights with the error function being lower when the predicted and target outputs are closer together.

$$E(W) = \frac{1}{2} \sum_{d \in D} (\vec{t_d} - \vec{y_d})^2 \tag{2.2}$$

Gradient descent search is used to minimize this function. We will illustrate gradient descent on a simple example before expanding to large neural networks with the full backpropagation algorithm. Suppose we have a single linear neuron that computes a weighted sum of its inputs without any activation function as shown in equation 2.3:

$$y = w_1x_1 + w_2x_2 + ... + w_nx_n + b = \vec{w} \cdot \vec{x} + b \tag{2.3}$$

With the simple, neuron the error function will be convex with a global minimum. For gradient descent search, the weight vector is initialized with random values, then repeatedly modified with small updates. Each update moves the weight vector in the direction of the steepest descent in order to minimize the error function as quickly as possible. This process continues until the global minimum value of the error function is reached, and hence, we have the optimal weight values.

The direction of steepest descent is found by calculating the derivative of the error function with respect to W and the weights are updated in the opposite direction of the gradient:

$$w_i \leftarrow w_i - \Delta w_i(t) \tag{2.4}$$

where

$$\Delta w_i(t) = \eta \frac{\partial E}{\partial w_i}. \tag{2.5}$$

Here w_i is each component of the weight matrix and η is the learning rate which is a positive constant that determines the step size in updating the gradient. The update is negative to move in the direction that decreases E. We use $\Delta w_i(t)$ to indicate the update provided at each step. Given that the learning rate is small enough and the error function is convex, it is guaranteed to converge to the global minimum using this update rule.

To use gradient descent on a neuron with an activation function, the function must be differentiable to find the update direction. Hence, the step function cannot be used and must be replaced by a differentiable function such as the sigmoid function, $\sigma(x) = \frac{1}{1+e^{-x}}$. The sigmoid function squashes its input which can be any real number to a range 0 to 1 that is monotonically increasing.

To scale up to a network with multiple neurons and hidden layers, gradient descent can be applied independently at each neuron to update the weights. However, only output nodes have a target value directly provided by the target label of the example that can be used in finding the gradient of the loss function. Nevertheless, backpropagation uses the chain rule to find the gradient with respect to each weight one layer at a time moving backward from the output layer through the hidden layers. The chain rule allows a calculation of how much each weight contributes to the various outputs, and hence, how much each weight should be updated based on the loss function for the current training example.

It is necessary to choose a loss function with a derivative that is efficient to calculate as there be thousands or millions of updates for each neuron while training the network. This is part of the reason why the sum of squared errors was chosen for the loss function.

Due to the popularity of the neural networks and backpropagation, there are many variations of gradient descent that can speed up the convergence of the weights and training time. Two popular variants involve adding momentum and using root mean square propagation (RMSProp) [30]. Adding momentum changes the weight update rule to use an exponentially weighted average of the past gradients. Hence, the gradient in equation 2.4 is replaced with the following value at iteration t:

$$\Delta w_i(t) \leftarrow \beta \Delta w_i(t-1) + (1-\beta)\frac{\partial E}{\partial w_i}(t) \tag{2.6}$$

Here, β is a constant between 0 and 1 called the momentum that determines how much the past gradients affect the current update of the weight. Adding momentum increases updates in the same direction between iterations and can dampen oscillations in weight updates.

RMSprop speeds up convergence by scaling the learning rate for a given weight by an exponentially weighted average of the magnitudes of gradients for that weight. The mean square is calculated for each weight i at iteration t:

$$m_i(t) \leftarrow \beta m_i(t-1) + (1-\beta)\left(\frac{\partial E}{\partial w_i}(t)\right)^2 \tag{2.7}$$

Here $m_i(0) = 0$ and β is a constant between 0 and 1. The current learning rate for each weight is divided by the square root of the value from equation 2.7: $\eta_i = \frac{\eta}{\sqrt{m_i(t)}}$. Similar to adding momentum, RMSprop tends to smooth oscillations and increases the rate of convergence compared to basic gradient descent.

Adam [14] is another variant of gradient optimization method which combines gradient descent with momentum and RMSprop. It uses first-order and second-order momentum when updating weights. Adam is widely used in current deep learning models as it has empirically been found to converge faster than other gradient algorithms.

2.2.2 Rectified Linear Unit

One of the early difficulties with training large neural networks is a problem known as vanishing gradient where the gradients of the loss function approach zero, making the network slower to train. When using an activation function such as the sigmoid function where a large input is squashed into a small output range, a large change in the input will result in a small change in the output and a small gradient as shown in Figure 2.3.

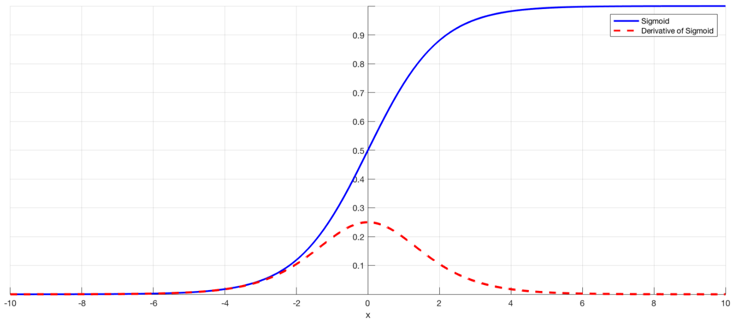


Figure 2.3: The sigmoid activation function (blue) and its derivative (red).

This problem becomes more prevalent as more and more layers are added to the network since the chain rule multiplies the gradients of each layer together to compute the updates of earlier layers. Hence, the update values decrease exponentially as they are propagating back through the network. This makes training the network inefficient as the weights in the initial layers of the network cannot be effectively tuned, but these layers are crucial for extracting relevant features from the input.

Without an activation function, ANNs would simply be a linear regression model and would not be able to learn models as simple as the XOR function. Thus, a nonlinear activation function is required for neural networks to learn complex models. Hence, the rectified linear unit (ReLU) was developed as an activation function for training deep neural networks where $g(z) = \max(0, z)$. The function replaces any negative value with zero and simply passes on any positive value as shown in Figure 2.4. The nonlinearity of the function around $z = 0$ allows deep neural networks to learn complex models without the gradient vanishing.

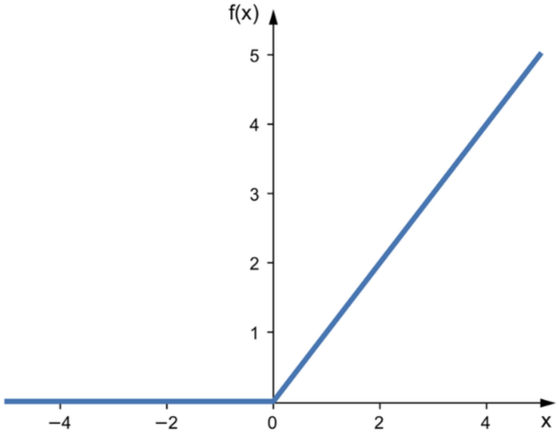


Figure 2.4: The rectified linear activation function.

2.3 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) have emerged as a popular variant of the standard artificial neural network, with particular success in tasks involving image processing. The neural networks discussed thus far only use fully connected layers where every node in a layer receives an input from every node in the previous layer. CNNs, on the other hand, use three different types of layers to construct the network: convolutional layers, pooling layers, and fully connected layers.

2.3.1 Convolutional Layer

The convolutional layer is what differentiates a CNN from a standard neural network. Instead of connecting all nodes from the previous layer to all nodes in the current layer each with a unique weight, learnable filters are applied over the output values from the previous layer. Each filter computes the dot product over a small receptive field of the input and repeats this operation to cover the entire input.

Since CNNs are often designed for image processing, most convolutional layers are applied to 2-dimensional data which will be the focus of this subsection. The kernel size gives the dimensions of each filter which are square or rectangular in shape. Figure 2.5 provides an example convolution of a 2×2 kernel over a 3×3 image. Each output value is a dot product between a part of the input and the given filter.

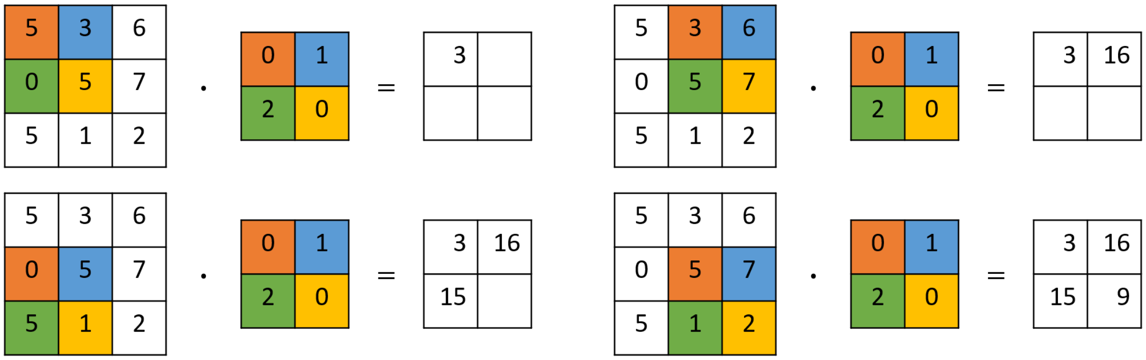


Figure 2.5: A 2×2 kernel over a 3×3 single channel image with stride 1.

The image in Figure 2.5 has a single channel meaning each pixel is represented by a single value. For example, this might be a grayscale image where the value represents how dark each pixel is. A standard RGB image has three channels where each pixel has a red, green, and blue value. Furthermore, each convolutional layer typically has many filters that apply to all channels of its input and each produces a separate output. Hence, the inputs and outputs of a convolutional layer will be three dimensions: $height \times width \times channels$.

The stride is how much the kernel shifts for each computation. In Figure 2.5, the stride is 1 since the filter shifts one value after each computation. When using a larger kernel, a stride of greater than 1 can be used to avoid repeated computations that may be very similar. As the stride increases, there are fewer computations and the output decreases proportionally.

An activation function, such as ReLU, is typically applied after each convolutional layer. This gives each layer a nonlinear characteristic so that complex models can be learned.

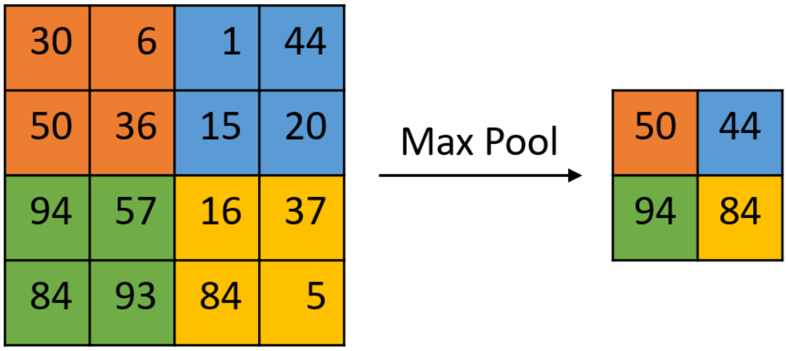
Since each output is a weighted sum of a subarea of the input, this introduces sparse connectivity into the network compared to a dense layer where an output can receive input from every neuron in the previous layer. Additionally, since the same filter is applied to different areas of the input, this introduces parameter sharing where filters trained in one part of the input can extract features from other parts of the input. These properties allow convolutional layers to use significantly fewer parameters than comparable fully connected layers and are what set apart CNNs from standard neural networks.

2.3.2 Pooling Layer

The pooling layer is the other unique layer utilized by CNNs. A pooling layer is a form of down-sampling used to reduce the size of the current input, and hence, reduce the number of parameters and computations in the network. Each channel of the input is partitioned into non-overlapping rectangles and each sub-region has a single output. The most common type of pooling is max-pooling where the maximum value from each area is the output. A typical pooling layer will split the input into 2×2 subregions, reducing the height and width by half and keeping the same number of channels. A max-pooling example is shown in Figure 2.6. Pooling layers are commonly used between convolutional layers in a CNN.

2.3.3 Fully Connected Layer

The last few layers in a CNN are fully connected layers. The convolutional and pooling layers extract the initial features of interest from the input, whereas these final layers conduct the high-level reasoning and encode the output. An additional benefit of using dense layers at the end of a CNN is the ability to combine image data



30	6	1	44
50	36	15	20
94	57	16	37
84	93	84	5

50	44
94	84

Figure 2.6: A max-pooling example with a 2 × 2 filter.

processed by convolutional layers with other types of numeric inputs that cannot be processed by 2D convolutions.

2.4 Reinforcement Learning

The previous sections have been focused on supervised learning where the goal is to learn a general function to map a set of inputs to their respective outputs. Reinforcement learning (RL) is a separate paradigm within machine learning focused on goal-directed learning based on trial and error. Instead of using a provided set of input/output pairs to train the model, actions are taken that can lead to positive or negative rewards; the objective is to accumulate as much reward as possible.

The premise behind reinforcement learning comes from how humans and animals naturally learn. If I give my dog a treat every time he goes to the bathroom outside and scold the dog whenever he chews on my shoe, then my dog will learn which behaviors are good and which ones should be avoided. With RL the idea is to replicate this learning model with computers.

2.4.1 Key Concepts

RL problems are set up with an environment and an agent that interacts within this world. At each time step, the agent receives an observation of the current state of the environment and takes some action. The environment changes based on the action taken as well as possibly outside forces. The agent then receives some reward signal from the environment and the observation for the next step and the cycle continues. The goal of the agent is to maximize the cumulative reward known as return. RL algorithms attempt to learn the best action for the agent to take at each state to achieve this goal.

RL environments can be formalized as Markov Decision Processes (MDPs). MDPs obey the Markov property where transitions between states only depend on the current state and current action taken; previous actions and states have no effect. MDPs are represented by a 4-tuple $M = (S, A, R, P)$ where S is set of all states and A is the set of all actions. $R : S \times A \times S \to \mathbb{R}$ is the reward function with $r_t = R(s_t, a_t, s_{t+1})$. Finally, $P : S \times A \to [0, 1]$ is the transition probability function, where $P(s' \mid s, a)$ is the probability of transitioning to s' given you are in state s and take action a.

A state $s_t \in S$ is a complete description of the environment at some time whereas an observation o_t is a partial description of the state which is perceived by the agent. The state space is the set of all possible states in the environment, and the action space is the set of all actions the agent can take in the environment. The action space can be discrete with only a finite number of options, or it can be continuous where actions are real-valued vectors.

The policy is what the agent follows to decide what action to take given the current state. The policy can be deterministic where the agent will always take the same action given a certain state, or it can be stochastic where the agent will take an action based on some probability distribution. Stochastic policies with parameters θ are denoted by π:

$$a_t \sim \pi_\theta(\cdot \mid s_t). \tag{2.8}$$

A sequence of states and actions of the agent is a trajectory τ,

$$\tau = (s_0, a_0, s_1, a_1, ...). \tag{2.9}$$

The reward function R provides a reward value based on the current state-action or simply the current state,

$$r_t = R(s_t, a_t). \tag{2.10}$$

To calculate the reward for a trajectory, we will use the infinite-horizon discounted return allowing arbitrary length trajectories that converge. This sums up every reward received but discounts future reward, as intuitively it is better to receive reward earlier rather than later. The discount value γ is between 0 and 1:

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t. \tag{2.11}$$

The RL goal is to find the policy for the agent to follow that maximizes expected return. With a stochastic policy and environment transitions, the probability of a trajectory τ given policy π is,

$$P(\tau|\pi) = \rho_0(s_0) \prod_{t=0}^{T-1} P(s_{t+1}|s_t, a_t)\pi(a_t|s_t) \tag{2.12}$$

where $\rho_0(s_0)$ is the probability of starting in state s_0 and $P(s_{t+1}|s_t, a_t)$ is the probability of moving to state s_{t+1} given the current state and action. The expected return for the policy is:

$$J(\pi) = \int_{\tau} P(\tau|\pi)R(\tau) = \mathop{\mathbb{E}}_{\tau\sim\pi} R(\tau). \tag{2.13}$$

Hence, the objective is to find the optimal policy, denoted π^*, which maximizes $J(\pi)$.

It is often beneficial to quantify the value of being in a state. The value function $V^{\pi}(s)$ provides the expected return from starting in a given state s and following policy π,

$$V^{\pi}(s) = \mathop{\mathbb{E}}_{\tau\sim\pi} [R(\tau)|s_0 = s] \tag{2.14}$$

The optimal value function, $V^*(s)$, similarly gives the expected return if starting in state s and following an optimal policy.

Likewise, the action-value functions, $Q^{\pi}(s, a)$ and $Q^*(s, a)$, give the expected return if starting in state s and take action a and subsequently follow policy π and the optimal policy, respectively.

$$Q^{\pi}(s, a) = \mathop{\mathbb{E}}_{\tau\sim\pi} [R(\tau)|s_0 = s, a_0 = a] \tag{2.15}$$

While the Q function can provide an absolute value to taking an action in a state, it can be useful to know how this value compares to other actions. Hence, the advantage function quantifies how much better it is to take action a in state s compared to selecting an action according to the policy:

$$A^{\pi}(s, a) = Q^{\pi}(s, a) - V^{\pi}(s). \tag{2.16}$$

With simple RL problems, it is possible to enumerate all state-action pairs and, for instance, store every Q value in a table. This allows for algorithms such as Q-learning [33] where all Q values can be directly compared and the policy will simply choose the action with the largest value at the current state. After each action, the Q value is updated with the actual reward received and the value of the next state compared to the predicted value. However, in complex problems with large state spaces, it is infeasible to store and maintain every Q value. With the introduction of Deep Q-Network (DQN) [21], it is now common to use neural networks to approximate Q-values, the policy, and other functions for deep reinforcement learning.

2.4.2 Policy Optimization

As the popularity of RL has surged over the past few years, many algorithms have been developed to efficiently train large models. This section will focus on policy optimization methods that explicitly model and optimize a policy; that is, we will use gradient ascent on the expected return, J to update the policy parameters. This is different than Q-learning or action-value methods which learn the Q function from which the policy is derived.

Advantage actor critic (A2C) [29] has emerged one of the more popular deep RL algorithms. It is an actor-critic method where the actor determines the policy of the agent while the critic asses actions by modeling the value function (i.e., predicts the value of being in a given state). The policy and value functions are updated every t_{max} steps using the following gradient for the parameters θ of the policy:

$$\nabla_\theta \ln \pi_\theta(a_t|s_t) A_w(s_t, a_t). \tag{2.17}$$

Here, $A_w(s_t, a_t)$ is the advantage function that uses the critic value function with parameters w:

$$A(s_t, a_t, w) = r_{t+1} + \gamma V_w(s_{t+1}) - V_w(s_t). \tag{2.18}$$

While the policy and value function can have different parameters, typically they will share the same neural network but with separate output layers.

This RL algorithm was originally introduced as asynchronous advantage actor-critic (A3C) [20] with parallel actors running independently and updating a global value function. However, researchers at OpenAI have empirically shown that A2C is as good as the asynchronous version and is more cost-effective when training [34].

2.5 Safety Constraints

While the goal in RL is to maximize expected return, in some situations the safety of the agent can be particularly important. For instance, when dealing with an expensive robot it may better to sacrifice some reward in order to avoid damaging the device. With a stochastic environment and agent introducing uncertainties, even an optimal policy maximizing return may perform poorly in some cases. Hence, a subfield known as Safe RL has been introduced with algorithms and techniques to attempt to reduce these risks when training and/or evaluating models. There are two overarching groups of Safe RL algorithms: modifying the optimization criterion and changing the exploration process using external knowledge or a risk metric [10].

2.5.1 Optimization Criterion

As previously noted, maximizing expected return may result in a risky situation, so the idea with these algorithms is to modify the optimization criterion to consider risk when evaluating policies. One modification technique, worst case criterion, attempts to mitigate the effects of variability in the policy by scoring policies on their worst-case return. A second approach, risk-sensitive criterion, balances the return and risk where the sensitively to risk can be controlled. This balance can be a linear combination of return and risk where risk might be the variance of the return or the probability of reaching an error state. Finally, constrained criterion maximizes expected return while keeping utilities within certain bounds. Here constraints restrict which policies are allowed while the objective remains to find the allowed policy with the best return.

2.5.2 Exploration Process

During the training process, RL algorithms must balance exploration to gather knowledge of the task and exploitation to take advantage of current knowledge to gain the best return. Randomized exploration strategies help the agent explore the environment but can lead to irrelevant and sometimes risky states for the agent. Hence, these Safe RL strategies modify the exploration process to mitigate risky situations. Some strategies provide external knowledge to the agent typically from a teacher. Information from the teacher can be used to bootstrap the learning process to reduce the need for random exploration or examples from the teacher can replace the random exploration aspect of the learning algorithm. Alternatively, the teacher can provide advice to the agent during training to maximize return while remaining in safe states. Without external knowledge, the exploration process can be altered to take into account the safety value of an action where the safety value measures the randomness associated with taking an action. The utility of taking an action becomes a weighted sum between the safety value and the Q-value of the action to preference useful actions with smaller risks. For a thorough overview of these Safe RL techniques, consult [10].

2.5.3 Formal Language Constraints

Instead of modifying the optimization or exploration of the agent, [24] adds constraints to a base environment during training. This allows any RL algorithm to maximize performance in the base environment while learning to avoid safety constraint violations. These constraints, specified in formal language, work with any MDP system and allow for complex constraints that are computationally efficient. When selecting an action that would result in a violation, a hard constraint forces the agent to select a different action, and a soft constraint reduces the reward the agent receives at the current time step.

Formally, this framework modifies MDP $M = (S, A, R, P)$ into a "Formal Language Constrained MDP" (FLCMDP) $M' = (M, C_0, C_1, ...)$ where each constraint defines a set of trajectories that results in a constraint violation. Each constraint consists of a tuple of functions $C = (D_C, T_C, SAug_C, RShape_C, AShape_C)$, which modify the MDP into the constrained MDP. Here $D_C : Q_C \times \Sigma_C \rightarrow Q_C$ is a recognizer (e.g., DFA) that encodes the constraint with Q_C states over alphabet Σ_C. Reaching an accepting state means the constraint has been violated. $T_C : S \times A \rightarrow \Sigma_C$ is a translation function that converts the current state and action in the MDP into an input token for the recognizer. $SAug_C : Q_C \times S \rightarrow S'$ augments the MDP state with information about the state of the recognizer to provide an updated MDP state. For soft constraints, $RShape_C : Q_C \times \mathbb{R} \rightarrow \mathbb{R}$ performs reward shaping. For hard constraints, $AShape_C : Q_C \times A \rightarrow A'$ restricts the actions an agent can make to avoid a constraint violation.

At each step t, the MDP passes the current state and action to the translation layer T_C which provides a token to the recognizer D_C. The recognizer provides reward shaping $RShape_C$ for the current step to the MDP (if using a soft constraint) and computes the allowed action set $AShape_C$ for time step $t + 1$ (if using a hard constraint). Additionally, the recognizer provides the state augmentation $SAug_C$ to the MDP for the next step. This process is illustrated in Figure 2.7.

When using soft constraints, reward shaping is performed at training time when a violation occurs. There are two methods for implementing reward shaping. In sparse

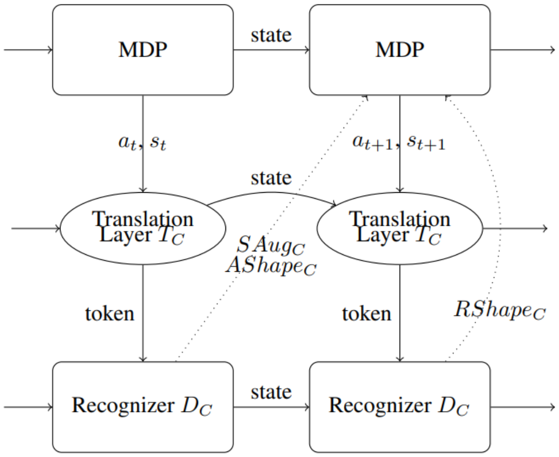


Figure 2.7: The formal language constraint framework. The translation layer creates the token for the recognizer which provides reward shaping, state augmentation, and restricts the available actions. Source: [23]

reward shaping, the environment reward signal is only modified when a constraint is violated, otherwise the reward value is passed through to the agent. The other technique is dense reward shaping which applies shaping at every time step. The shaping value is determined by a potential function $\Phi_C(q_t)$ that determines how close the current recognizer state $q_t \in Q_C$ is to a violation. One method of calculating this value is by determining the proportion of visits to the current recognizer state ultimately lead to a violation as follows:

$$\Phi_C(q_t) = viols_C(q_t)/visits_C(q_t). \tag{2.19}$$

Here, $viols_C(q_t)$ counts the number of times constraint C has been violated after visiting q_t, $visits_C$ is the total number of times q_t has been visited.

With this framework, the MDP state observations can be augmented with the state of the recognizer. This provides the agent with additional information to help it avoid constraint violations. There are different ways to encode the recognizer information. One common variant uses a one-hot encoding of the DFA state at time

t as the state augmentation. The agent is not provided with direct knowledge of the DFA but can infer information from the encoding to determining which states are preferred and which ones should be avoided to reduce violations and reward shaping.

Chapter 3

Related Work

There has been a recent push to apply RL algorithms to solve a variety of problems including developing AVs. However, it is important for self-driving to have a high level of safety that is not always guaranteed with RL. Here, we look at some of the prominent recent work developing RL algorithms for AVs and specifically papers that introduce constraints or other methods to improve the safety of the agents.

In 2017, Dosovitskiy et al. [9] released the initial version of the CARLA simulator for developing and testing self-driving vehicles with active development and feature updates still happening every couple of months. The paper introduces a basic driving task for agents to complete that has become a standard benchmark for other works developing agents in the CARLA simulator. They provide three baseline agents for completing this task. The first is a modular pipeline that has separate subsystems for visual perception, planning, and control of the vehicle. The second approach uses end-to-end imitation learning from a human driver in the environment. The final method uses a simple RL A3C RL agent to complete the task.

Chen et al. [7] improve the imitation learning method by first creating an agent with privileged information from the environment. Once trained, this agent acts as a teacher to the final agent which does not receive privileged information. The first agent is able to quickly learn how to act in the environment and is able to provide the final agent with an endless supply of training examples to imitate. This two-stage imitation learning method is able to train agents that perform significantly better on the CARLA benchmark compared with baseline agents.

On the other hand, Liang et al. [15] present an RL agent to complete the CARLA benchmark. The agent model is warmed up through imitation learning from human demonstrations to limit random exploration at the beginning and fine-tuned via RL to improve the generalization of the agent. Additionally, the authors significantly modify the reward function, including a constant negative reward when the agent chooses a poor steering action such as turning the wrong direction or turning too sharply when the vehicle should be driving straight in the lane.

More recently, Toromanoff et al. [31] developed an RL agent for CARLA without using any imitation learning or pretraining. Their reward module is made of three components: desired speed, desired position, and desired rotation. The reward value from each component depends both on the agent's actions and the current state of the environment. For example, to receive the maximum reward with the desired speed component, the vehicle should match the ideal speed value determined by the current state. The ideal speed decreases as the vehicle approaches a slow vehicle or a red light and increases when there are no obstacles in front of the vehicle. Similarly, the desired rotation of the vehicle and the associated reward value is different depending on if the vehicle is driving straight in the lane or completing a turn. This dynamic reward signal helps give the agent a significant performance improvement compared to other RL agents in CARLA.

Prior to the release of CARLA, Shalev-Shwartz et al. [27] developed a safe RL technique for AVs using hard constraints. They show that an RL agent cannot sufficiently penalize rare events such as collisions without causing a reward variance problem that reduces the performance of the agent in normal, accident-free trajectories. Hence, they argue for hard constraints that are provided outside of the learning framework and limit the allowable trajectories to ensure functional safety of the agent. They test the hard constraints in a simple, multi-agent driving environment and present very limited results.

Hu et al. [12] use hard constraints, which they refer to as a masking mechanism, to prevent the agent from choosing unsafe actions. Specifically, they use masks to limit the acceleration of the vehicle, prevent the agent from exceeding the speed limit, and keep the agent a safe distance from the vehicle in front. The agents are tested in traffic merging scenarios. Saxena et al. [26] use these same tests for their agents but do not use the masking mechanism to restrict actions. However, they specifically mention that their current model cannot guarantee collision-free driving, and in the future, they want to add an *overseer* that determines if a chosen action is safe to execute using hard and/or soft constraints.

Chapter 4

Experimental Setup

This chapter details the RL environment and experiments performed in this thesis. Section 4.1 introduces CARLA, a simulator for autonomous driving research that is used as the environment for all of the experiments. Section 4.2 describes the task in CARLA for the agent to solve. Section 4.3 details the RL algorithm which trains the agent including the training hyperparameters and the CNN that approximates the policy for the agent. Finally, in Section 4.4 we detail three soft constraints which shape the reward function to improve the final performance of the agent in the environment

4.1 CARLA Environment

CARLA [1] [9] is an open-source, high-fidelity simulator developed for training and validating autonomous vehicles in a variety of settings. It is being developed as a general-purpose environment to resemble real-life driving. There are several distinct cities in CARLA including urban layouts, rural towns, and multi-lane freeways. Traffic intersections include stop signs, traffic lights, roundabouts, and freeway on/off ramps. Available vehicles include a range of cars, trucks, motorcycles, and bicycles along with pedestrians walking along the sidewalk and crossing the street. The agent

[1] http://carla.org/

vehicle can be equipped with a variety of sensors including RGB cameras, LIDAR, RADAR, and GPS. Sensors can be placed anywhere on the vehicle and each sensor can be configured depending on the need (e.g., a long-range camera for detecting far away objects in the current layer versus a wide-angle camera for detecting nearby obstacles). Additionally, there are sensors for detecting collisions, and when the vehicle leaves the current lane. CARLA provides different weather and lighting configurations: the position of the sun, cloud cover, precipitation rate, and wind level among other options can be changed to create different situations. Some preset weather situations include heavy rain, sunset which adds camera flare, and even after rain conditions with wet roads but no precipitation. Nighttime driving weather is available with street and car lights illuminating the roadways

CARLA uses a server-client model where the server maintains the world including global settings such as the weather. Vehicles must use a client to connect to the server and interact with the world. Multiple clients can connect to the server at the same time, and each client can control multiple vehicles allowing for multi-agent systems to be developed. The Traffic Manager is a client in charge of spawning and controlling numerous vehicles to maintain traffic in the simulated city. Each of these vehicles has an autopilot system that uses privileged information from the server to perform a near-optimal driving policy. Hence, the Traffic Manager can help prepare the environment for testing similar to a real road where other vehicles can have a significant impact on the situation. An agent vehicle through a client can connect to the environment; every time step each sensor on the agent's vehicle receives data from the world, and the agent must output driving commands including throttle, braking, and steering values to control the vehicle. The throttle and brake commands are real values between 0 and 1, which represent pressing the throttle and brake pedal, respectively. The steering value is a real number between -1 and 1 representing the steering wheel angle. This assortment of configurations makes CARLA an ideal testbed for training and testing self-driving agents.

4.2 CARLA Task

For training and evaluating our agent, we will use the same driving experiment and conditions as presented in the original CARLA paper [9]. The basic task is goal-directed navigation where the agent is initialized at some point in the town and must drive to a given destination. The agent is given directions to the destination by receiving a command to turn left, turn right, or go straight through each intersection as would be provided by a navigation system. There are six weather conditions divided into two groups. The training weather set includes clear day, clear sunset, daytime rain, and daytime after rain. The test weather set, not used during training, includes cloudy daytime and soft rain at sunset. There are 100 different driving scenarios, each with unique starting and endpoints within the city. The tasks have varying difficulties with some having one or zero turns to reach the destination while others require the vehicle to navigate multiple turns and intersections. The number of other vehicles and pedestrians also varies depending on the task. Combining each driving scenario with the 6 weather conditions means there are 600 unique configurations for training and evaluating the agent.

If the agent reaches the destination within a certain amount of time, the episode is considered a success. If the agent vehicle has a collision with an object such as another vehicle, a pedestrian, a street sign, or a building, then the episode ends and is classified as a collision. The time limit for the vehicle to reach the destination is set to the time it takes to follow the optimal path to the destination at a speed of 10 km/h. If the agent does not have a collision but fails to reach the destination in the allotted time limit, the episode is classified as a timeout.

4.3 RL Agent

This section describes our agent which is based on the RL agent from [9]. The vehicle for our experiments is configured with a single forward-facing RGB camera. The camera resolution is 800×600 and has a $100°$ field of view. Other inputs to the agent are measurements of the state of the vehicle including the current velocity, acceleration, and position in the world, along with a one-hot encoding of the high-level navigation command: a $TURN\_RIGHT$, $TURN\_LEFT$, or $GO\_STRAIGHT$ command is given when approaching and driving through an intersection; otherwise, the agent receives the $LANE\_FOLLOW$ command. These inputs are concatenated into a vector of 18 values, known as the measurement vector.

While the CARLA server expects real-valued commands for controlling the vehicles throttle, braking, and steering, we use a discrete action set with a finite number of possible commands. We use 7 steering values and 7 throttle/braking values. The available steering values are $\{-1, -0.5, -0.25, 0, 0.25, 0.5, 1\}$. Here, -1 corresponds to a sharp left turn, -0.5 to a moderate left turn, and -0.25 to a slight left turn while 1, 0.5, and 0.25 correspond to a sharp right, moderate right, and slight right steering angles, respectively. We consider throttle and braking to be mutually exclusive (as is typically the case when driving), so we use the following tuples where throttle and braking are the first and second values, respectively: $[1, 0], [0.5, 0], [0.25, 0], [0, 0], [0, 0.25], [0, 0.5], [0, 1]$. Values in the first half of the list with a positive throttle imply the car is accelerating, and tuples with a positive braking value imply the car is slowing down. The middle value $[0, 0]$ allows the vehicle to coast; hence, this value is commanding how the vehicle should accelerate. Thus, our agent must use the inputs to predict the optimal steering and acceleration values for the vehicle.

We use the Advantage Actor-Critic (A2C) algorithm for training the agent. The actor and critic functions share the same core network and parameters but with separate output layers. The core network uses a CNN to initially process the input image and a fully connected neural network to process the remaining inputs. The outputs of these layers are concatenated and further processed by additional dense layers. The architecture is illustrated in Figure 4.1 and described in more detail below.

At each time step, the camera provides an 800×600 pixel image with three channels, one for the red, green, and blue values. Before being fed to the CNN, the image is downscaled to an $84 \times 84 \times 3$ image. This substantially reduces the number of parameters that must be trained in the neural network. The first convolutional layer has 32 filters with a kernel size of 8 and a stride of 4. This reduces the image to $20 \times 20 \times 32$. The second layer has 64 filters with a kernel size of 4 and a stride of 2, bringing the output to $9 \times 9 \times 64$. The third and final convolutional layer has 32 filters with a kernel size of 3 and stride of 1 bringing the output to $7 \times 7 \times 32$. This output is flattened into a one-dimensional vector with 1568 values and processed through a fully connected layer with 512 outputs. No pooling layers are used in this CNN.

The real value inputs in the measurement vector are processed through a 3-layer dense neural network with 128, 256, and 512 neurons for the first, second, and third layers, respectively. The 512 outputs from the CNN are concatenated with the 512 outputs from this network and fed through three additional fully connected layers. The first layer has 1024 neurons while the second and third layers each have 512 nodes. ReLU activation is applied to each convolutional and fully connected layer.

For the policy, the actor function appends an additional dense layer to the core

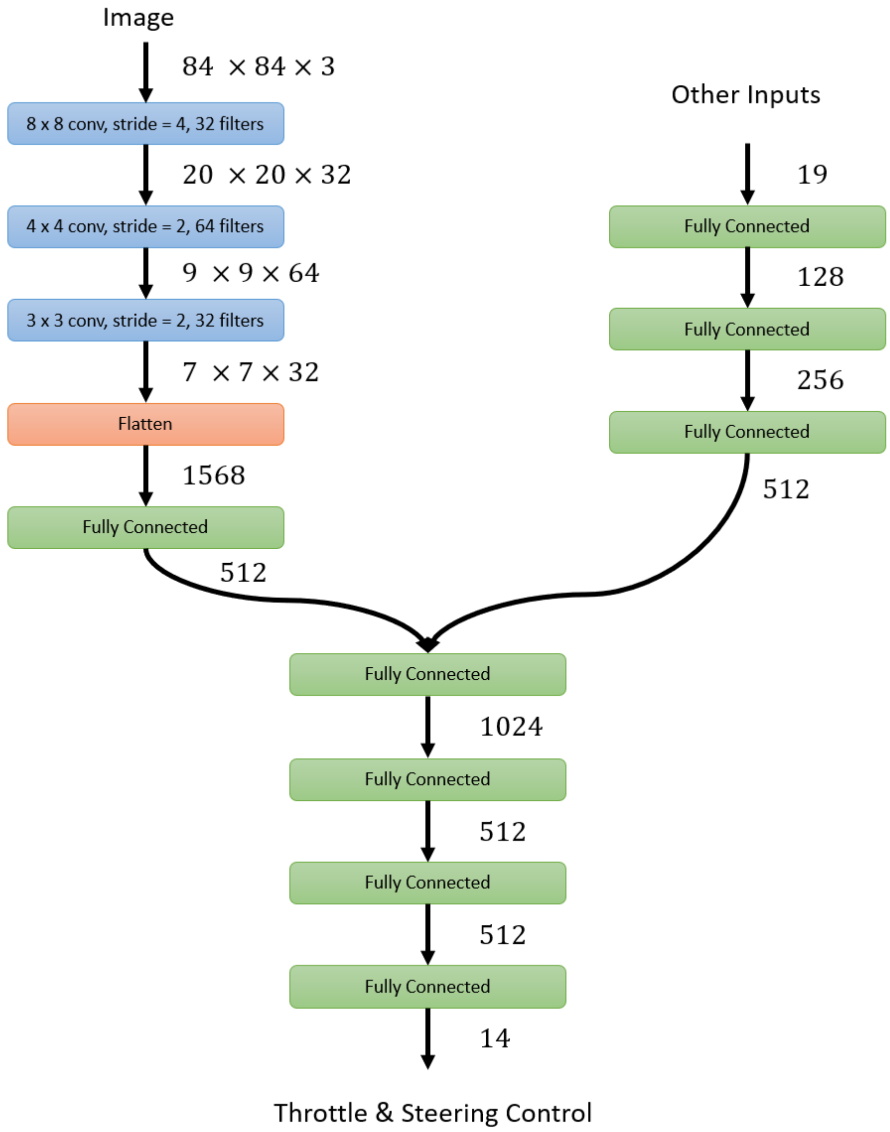


Figure 4.1: The architecture for the neural network that approximates the policy of the RL agent.

network with a softmax output, and the action is chosen randomly from this distribution. The critic function adds a linear layer with a single output for the value function estimate of the state.

The neural network uses RMSProp as the optimizer with a learning rate of 0.0007. We use 20-step rollouts for updating the actor and critic functions. Each model is trained for 2 million environment steps and evaluated after 1 million times steps and once fully trained.

A major difference between this agent and the RL agent from [9] is that we use A2C trained out for 2 million steps while [9] uses A3C with 10 asynchronous, parallel actor threads each trained for 1 million steps. We use A2C to significantly reduce the hardware requirements necessary for training a single model. As shown in Section 5.1, the agents have similar performance despite our agent having a single actor thread and only 20% of the total training time compared to the other agent.

The reward function is a weighted sum of four terms:

$$r_t = -(d_t - d_{t-1}) + 0.18(v_t - v_{t-1}) - 0.00002(c_t - c_{t-1}) - 2(l_t - l_{t-1}). \qquad (4.1)$$

Here d is the distance remaining to the goal in meters, v is the velocity of the vehicle, c is the collision damage, and l is percentage of the vehicle outside of its lane. The agent will receive positive reward as long as it progresses towards the destination while staying within the lane. However, any collision will result in a significant negative return for the episode.

4.4 Constraints

In this section, formal language constraints are proposed with the goal of improving the safety of the baseline agent. The constraints use the framework described in Section 2.5.

The recognizer of each constraint is implemented as a deterministic finite automaton (DFA). The translation function encodes the current state and action into a simple token for the recognizer; each implementation is dependent on the corresponding constraint and DFA. Every step the translation function provides a token to the

DFA. An accepting step in the DFA implies the constraint is violated at the current step.

We use augmentations to modify the MDP state observations received from the environment with information from the state of the constraint recognizer. This is done by appending a one-hot encoding of the DFA state to the measurements vector before it is processed by the actor and critic networks, hence allowing the agent to learn if the constraint is likely to be violated in an upcoming step.

Each constraint is implemented as a soft constraint with sparse reward shaping. Violations do not restrict the available actions but instead, reduce the reward received to discourage actions that lead to violations. With sparse reward shaping, the reward signal is only modified when a violation occurs; otherwise, the reward from the environment is directly given to the agent. We use a constant reward shaping signal where the reward is reduced by a constant value of r_C for each violation regardless of the positive or negative reward from the environment:

$$r'_t = r_t - r_C * F(q_t),$$

where r'_t is the shaped reward given to the agent at step t and r_t is the reward from the environment. r_C is the constant shaping value for constraint C and $F(q_t)$ is one of the current DFA state q_t is and accepting state and 0 otherwise. We test with reward shaping values $r_c \in \{0.1, 0.5, 1, 2, 10\}$.

This remainder of this section describes three safety constraints considered in this work: Strict Turn, Loose Turn, and Dither.

4.4.1 Strict Turn Constraint

Analyzing the baseline agent during training, it is clear the model has a difficult time navigating through intersections and specifically making turns. The vehicle will often leave the roadway when turning, often leading to a collision. Hence, the focus of the first constraint, known as Strict Turn, is to help the agent safely and successfully navigate intersections without a collision.

The basic idea of the constraint is that if the agent is receiving the $TURN\_LEFT$ command for an upcoming or current intersection and chooses an action that steers the vehicle to the right, then the constraint is violated. Similarly, if the navigation command is $TURN\_RIGHT$ and the agent selects an action to steer left, a violation occurs as well. During the early stages of training when actions are seemingly random, this constraint helps guide the agent into turning the correct direction. As training progresses, the constraint helps fine-tune the turning sequence by penalizing the agent for oversteering and having to take corrective actions. A violation occurs every time step the agent is turning in the incorrect direction, hence the name Strict Turn.

The translation function outputs a 1 if the current navigation command is $TURN\_LEFT$ ($TURN\_RIGHT$) and the agent selects an action to steer to the right (left). Otherwise, the translation function outputs a 0 including any time the agent receives a $GO\_STRAIGHT$ or $FOLLOW\_LANE$ command, and when the agent selects a command to turn in the correct direction or steer straight. The translation function is provided in Table 4.1.

This constraint is very simple to implement as a DFA with only two states $[q^{(0)}, q^{(1)}]$ and two input tokens in the alphabet $\Sigma = \{0, 1\}$. State $q^{(0)}$ is the starting state and $q^{(1)}$ is the accepting state. Regardless of the current state, an input of 0 transitions to $q^{(0)}$ and an input of 1 transitions to $q^{(1)}$. The DFA for Strict Turn is

Table 4.1: The translation function for the Strict Turn constraint.

Navigation Command	Steering Action	DFA Token
TURN_RIGHT	Left (negative) Steering Actions	1
TURN_RIGHT	Right & Straight (nonnegative) Steering Actions	0
TURN_LEFT	Right (positive) Steering Actions	1
TURN_LEFT	Left & Straight (nonpositive) Steering Actions	0
GO_STRAIGHT	All Actions	0
FOLLOW_LANE	All Actions	0

depicted in Figure 4.2.

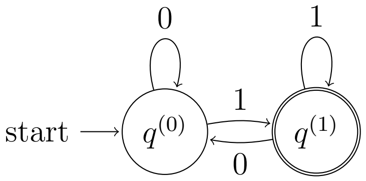


Figure 4.2: The DFA for the Strict Turn constraint.

4.4.2 Loose Turn Constraint

While the Strict Turn constraint discourages the agent from making poor decisions when navigating intersections, it can be overeager in finding violations and penalizing the agent. For instance, the same violation and reward shaping signal is triggered whether the agent turns sharply in the wrong direction or makes a small correction to stay in the lane as the agent enters or exits the intersection. Hence, the Loose Turn constraint triggers a violation for a sharp turn in the wrong direction, but only penalizes a slight turn if it happens for consecutive steps.

For this constraint, the DFA contains 5 states $Q = \{q^{(-2)}, q^{(-1)}, q^{(0)}, q^{(1)}, q^{(2)}\}$, where $q^{(0)}$ is the initial state and $q^{(-2)}$ and $q^{(2)}$ are the accepting states. The alphabet contains 5 tokens $\Sigma = \{-2, -1, 0, 1, 2\}$. The translation function produces a negative token when the agent selects a left steering action but the navigation command is

$TURN\_RIGHT$. When a the navigation command is $TURN\_LEFT$ but a right steering action is selected, the translation function gives a positive token. A larger (absolute) steering angle $\{-1, 1, -0.5, 0.5\}$ results in a token of -2 or 2 while a slight turning angle $\{-0.25, 0.25\}$ outputs a -1 or 1. The full translation function is provided in Table 4.2.

Table 4.2: The translation function for the Loose Turn constraint.

Navigation Command	Steering Action	DFA Token
$TURN\_RIGHT$	-1	-2
$TURN\_RIGHT$	-0.5	-2
$TURN\_RIGHT$	-0.25	-1
$TURN\_RIGHT$	Nonnegative Steering Actions	0
$TURN\_LEFT$	0.25	1
$TURN\_LEFT$	0.5	2
$TURN\_LEFT$	1	2
$TURN\_LEFT$	Nonpositive Steering Actions	0
$GO\_STRAIGHT$	All Actions	0
$FOLLOW\_LANE$	All Actions	0

The DFA begins in $q^{(0)}$ and transitions to the state corresponding to the received token. For example, token -1 would transition the DFA to state $q^{(-1)}$. A -2 or 2 from a significant incorrect turn triggers a violation. Further, if the DFA is in $q^{(-1)}$ or $q^{(1)}$ and receives a nonzero reward of the same sign (e.g., in $q^{(-1)}$ and receives token -1 or -2), then the DFA transitions to the $q^{(-2)}$ or $q^{(2)}$ accepting state. Any other token (including all tokens in the accepting state) resets the DFA by transitioning back to $q^{(0)}$.

After completing a turn, the agent will receive the $FOLLOW\_LANE$ command until it approaches the next intersection. Thus, it is not realistic to receive the $TURN\_RIGHT$ and $TURN\_LEFT$ commands on consecutive steps; therefore, any pair of consecutive tokens will not be of opposite signs (i.e., at least one token is 0 or they are both either positive or negative). Because of this, tokens for states of the opposite sign are not explicitly expressed in the DFA. If for some reason this situation occurred, the DFA would simply reset to $q^{(0)}$. The DFA for this constraint is provided in Figure 4.3 and the full DFA transition function is provided in Table 4.3.

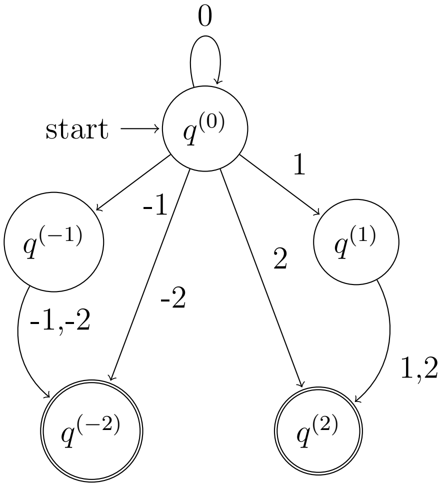


Figure 4.3: The DFA for the Loose Turn constraint. Any transitions not shown go to $q^{(0)}$.

Table 4.3: The transition table of the DFA for the Loose Turn constraint.

	-2	-1	0	1	2
$q^{(-2)}$	$q^{(0)}$	$q^{(0)}$	$q^{(0)}$	$q^{(0)}$	$q^{(0)}$
$q^{(-1)}$	$q^{(-2)}$	$q^{(-2)}$	$q^{(0)}$	$q^{(0)}$	$q^{(0)}$
$q^{(0)}$	$q^{(-2)}$	$q^{(-1)}$	$q^{(0)}$	$q^{(1)}$	$q^{(2)}$
$q^{(1)}$	$q^{(0)}$	$q^{(0)}$	$q^{(0)}$	$q^{(2)}$	$q^{(2)}$
$q^{(2)}$	$q^{(0)}$	$q^{(0)}$	$q^{(0)}$	$q^{(0)}$	$q^{(0)}$

4.4.3 Dithering Constraint

While the previous two constraints help the agent navigate turns, the majority of the episodes are spent driving straight within the lane. The action set allows the agent to jerk the wheel left and right multiple times per second creating an unpleasant and potentially dangerous situation. The goal of this constraint is to help guide the agent within the lane and avoid jerky or dithering movements. Specifically, a violation occurs if the steering command changes from one side to the other by four or more points on consecutive commands. Furthermore, a string of consecutive commands that switch sides (e.g., left, then right, then left steering commands) accumulate and can also lead to a violation.

To implement this constraint, the translation function maps the 7 steering values to tokens that are consecutive integers from -3 to 3, where -3 is the sharp left steering angle. Larger integers correspond to steering angles which are further to the right, so 0 is no turn and 3 is the sharp right steering angle. The absolute difference between two tokens is the point spread of the actions where a spread of at least four is a triggers a violation. For instance, if one action is sharp right ($1 \rightarrow 3$) and a second action is moderate left ($-0.5 \rightarrow -2$), the point spread is $|3-(-2)| = 5$. The translation function is provided in Table 4.4.

Table 4.4: The translation function for the Dithering constraint.

Steering Action	DFA Token
-1	-3
-0.5	-2
-0.25	-1
0.0	0
0.25	1
0.5	2
1	3

The DFA for the recognizer of this constraint is implemented with 9 states $Q = \{q^{(-4)}, q^{(-3)}, ..., q^{(4)}\}$ where $q^{(0)}$ is the initial state and $q^{(-4)}$ and $q^{(4)}$ are the accepting states. The states correspond roughly to the steering angle of the previous command. If the current state is $q^{(0)}$ or the state has the same sign as the input token, then the DFA will transition to the state corresponding to the input signal. For example, if the current state is $q^{(2)}$ and the input token is 1 since 2 and 1 are both positive the DFA will transition to $q^{(1)}$. This situation happens when the steering angle is the same or

Table 4.5: The transition table of the DFA for the Dithering constraint.

	-3	-2	-1	0	1	2	3
$q^{(-4)}$	$q^{(-3)}$	$q^{(-2)}$	$q^{(-1)}$	$q^{(0)}$	$q^{(4)}$	$q^{(4)}$	$q^{(4)}$
$q^{(-3)}$	$q^{(-3)}$	$q^{(-2)}$	$q^{(-1)}$	$q^{(0)}$	$q^{(4)}$	$q^{(4)}$	$q^{(4)}$
$q^{(-2)}$	$q^{(-3)}$	$q^{(-2)}$	$q^{(4)}$	$q^{(0)}$	$q^{(3)}$	$q^{(4)}$	$q^{(4)}$
$q^{(-1)}$	$q^{(-3)}$	$q^{(-2)}$	$q^{(-1)}$	$q^{(0)}$	$q^{(2)}$	$q^{(3)}$	$q^{(4)}$
$q^{(0)}$	$q^{(-3)}$	$q^{(-2)}$	$q^{(-1)}$	$q^{(0)}$	$q^{(1)}$	$q^{(2)}$	$q^{(3)}$
$q^{(1)}$	$q^{(-4)}$	$q^{(-3)}$	$q^{(-2)}$	$q^{(0)}$	$q^{(1)}$	$q^{(2)}$	$q^{(3)}$
$q^{(2)}$	$q^{(-4)}$	$q^{(-4)}$	$q^{(-3)}$	$q^{(0)}$	$q^{(1)}$	$q^{(2)}$	$q^{(3)}$
$q^{(3)}$	$q^{(-4)}$	$q^{(-4)}$	$q^{(-4)}$	$q^{(0)}$	$q^{(1)}$	$q^{(2)}$	$q^{(3)}$
$q^{(4)}$	$q^{(-4)}$	$q^{(-4)}$	$q^{(-4)}$	$q^{(-0)}$	$q^{(1)}$	$q^{(2)}$	$q^{(3)}$

one is zero. On the other hand, if the state and token are of different signs, then the next state is calculated by subtracting the current state from the input token and bounding the difference by ±4. For instance, if the current state is $q^{(-3)}$ and the input token is 2 with opposite signs, then the difference is $-3 - 2 = -5$ which is bounded at -4. Thus, the next state is $q^{(-4)}$, an accepting state. The full DFA transition list is in Table 4.5.

Chapter 5

Results & Discussion

This chapter provides results for implementing the RL agents introduced in Chapter 4 for the CARLA environment. We first provide results for an unconstrained baseline agent, followed by experiments with each formal constraint evaluated separately. The models are trained for two million time steps and tested at the end of training as well as halfway through training for additional comparisons. Each constraint is trained with **five** reward shaping values and both with and without constraint state augmentations. Each experiment is trained and tested using 10 random seeds. Hence, there are 100 ($5 \times 2 \times 10$) (i.e., shaping values × with/without state augmentation × seeds = 100) models trained and 200 checkpoints evaluated for each constraint, as each of the models is evaluated at two different points.. Overall, with three constraints and the baseline agent (10 seeds), we train 310 models and provide results on 620 checkpoints.

The CARLA paper [9], from which our baseline agent is derived, only provides the percentage of tasks successfully completed by the agent whereas we will compare the success, timeout, and collision rates of our various models. While the main goal is to successfully finish as many tasks as possible, it is also important to avoid collisions, which are significantly worse than safely failing to reach the destination in the allotted time. In the remainder of this chapter, we present the results from these experiments and discuss the outcomes.

5.1 Baseline Agent

The agent described in Section 4.3 is used as our baseline agent. To ensure the results are similar to previous works, we briefly compare our baseline results against the results from the RL agent in [9], which we will call CARLA RL. Table 5.1 contains the average success, timeout, and collision rate of our baseline agent after 1 million and 2 million time steps compared to the success rate of the CARLA RL agent, which is trained asynchronously for 10 million time steps. A higher success rate is better while a lower collision rate is better. Doubling the training time slightly increases the success rate but also increases the collision rate by the same amount; however, the differences are too small (0.7) to draw conclusions about the improvement of the agent. Nevertheless, the success rate of our agent is within half a percentage point of the CARLA RL agent after both evaluations, so we will use these two checkpoints when comparing against agents with constraints.

Table 5.1: The results of our baseline agent after 1 million and 2 million time steps averaged over 10 seeds compared to the CARLA RL agent from [9], which only reports the success rate. Higher success rates are better while lower collision rates are better.

	Baseline 1 million	Baseline 2 million	CARLA RL [9]
Success	26.1 (±1.3)	26.8 (±0.3)	26.5
Timeout	17.1 (±1.8)	15.7 (±6.4)	n/a
Collision	56.8 (±1.7)	57.5 (±6.4)	n/a

5.2 Strict Turn Constraint

We first add the Strict Turn constraint to the baseline agent. Table 5.2 reports the evaluation results of the agent after 1 million time steps without using state augmentation, and Table 5.3 provides the results after 1 million steps with state augmentation. Each value represents the percentage of episodes that end in a success, timeout, or collision, along with the standard deviation for each value. Figure 5.1 gives a visual representation of the various collision rates. For this constraint, the success rate for each shaping value never decreases compared to the baseline, but also never increases by more than one point. Effectively, the success rate remains about the same with little variation.

The collision rate, however, does substantially decrease, especially for reward shape values of 1 and 2. A lower collision rate correlates to a larger timeout rate meaning the agent is not completing more tasks but is at least avoiding costly colli-

Table 5.2: Test results when using the Strict Turn constraint without augmentation after 1 million time steps. “Suc.” is the success rate, “Time.” is the timeout rate, and “Col.” is the collision rate.

		Reward Shape Value				
	Baseline	0.1	0.5	1.0	2.0	10.0
Suc.	26.1 ± 1.3	27.0 ± 0.0	26.9 ± 0.3	27.0 ± 0.0	27.1 ± 0.3	26.4 ± 2.0
Time.	17.1 ± 1.8	12.4 ± 8.4	19.2 ± 13.1	21.1 ± 9.2	24.7 ± 11.3	19.1 ± 14.3
Col.	56.8 ± 1.7	60.6 ± 8.4	53.9 ± 13.2	51.9 ± 9.2	48.2 ± 11.1	54.5 ± 13.3

Table 5.3: Test results when using the Strict Turn constraint with augmentation after 1 million time steps.

		Reward Shape Value				
	Baseline	0.1	0.5	1.0	2.0	10.0
Suc.	26.1 ± 1.3	27.0 ± 0.0	27.0 ± 0.0	26.9 ± 0.2	27.0 ± 0.0	26.8 ± 0.6
Time.	17.1 ± 1.8	18.5 ± 9.5	18.3 ± 14.9	27.1 ± 10.4	19.9 ± 10.1	17.8 ± 13.5
Col.	56.8 ± 1.7	54.5 ± 9.5	54.7 ± 14.9	46.0 ± 10.3	53.1 ± 10.1	55.4 ± 13.3

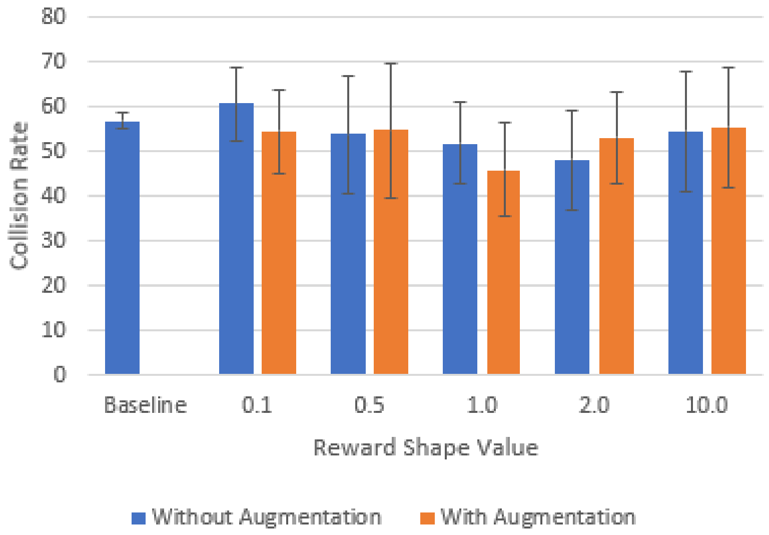


Figure 5.1: Collision rate and its standard deviation after 1 million time steps when using the Strict Turn constraint. A lower collision rate is better.

sions. Hence, we will mostly look at the collision rates of the agents. With augmentation and a reward shape value of 1, the collision rate decreases by over 10 points which is a nearly 20% decrease in collisions. The more the shaping value differs from this ideal value, the collision rate tends to increase towards the baseline collision rate. The collision rate for the agent without augmentation and a shaping value of 0.1 is the only value that is worse than the baseline rate. One explanation for this is the constraint affects the learning process of the agent, but the shaping value is too small to effectively guide the agent into taking safe actions.

The baseline agent has a small standard deviation of the collision rate compared to the agents with constraints. This implies the baseline agent has more consistent results whereas the constrained agents can sometimes perform very well and sometimes have significantly more collisions with the same constraint settings.

The evaluation results of the agent after 2 million time steps without using state augmentation are reported in Table 5.4, and Table 5.5 provides the results after 2 million steps with state augmentation. A visual comparison of the collision rates

Table 5.4: Test results when using the Strict Turn constraint without augmentation after 2 million time steps.

		Reward Shape Value				
	Baseline	0.1	0.5	1.0	2.0	10.0
Suc.	26.8 ± 0.4	26.9 ± 0.2	27.1 ± 0.3	26.6 ± 1.0	26.6 ± 1.4	25.5 ± 2.4
Time.	15.7 ± 6.4	19.4 ± 7.4	16.2 ± 7.2	20.8 ± 6.5	24.5 ± 6.6	30.1 ± 14.5
Col.	57.5 ± 6.4	53.7 ± 7.4	56.7 ± 7.3	52.6 ± 6.6	48.9 ± 5.7	44.4 ± 13.4

Table 5.5: Test results when using the Strict Turn constraint with augmentation after 2 million time steps.

		Reward Shape Value				
	Baseline	0.1	0.5	1.0	2.0	10.0
Suc.	26.8 ± 0.4	26.5 ± 1.6	26.7 ± 1.3	26.6 ± 0.8	26.9 ± 0.2	26.6 ± 1.2
Time.	15.7 ± 6.4	15.7 ± 7.8	19.7 ± 6.4	19.4 ± 7.7	20.8 ± 9.9	18.2 ± 7.7
Col.	57.5 ± 6.4	57.8 ± 7.3	53.6 ± 6.0	54.0 ± 7.1	52.3 ± 9.8	55.2 ± 7.9

is shown in Figure 5.2. The agents with augmentation show a similar pattern to the previous results with the best collision rates having a shaping value near 2, and the collision rate increasing for shaping values further away. Without augmentation, however, the smallest collision rate is for the reward shaping value of 10, but this also results in a 1 to 1.5 point decrease in the success rate. Additionally, the shaping value of 0.5 has the highest collision rate even compared to the agents with a 0.1 shaping signal, which seems to be an anomaly. However, this is likely a result of the agents with 0.1 slightly overperforming and the agents with a 0.5 signal underperforming by a few points compared to what is expected.

As with the earlier results, the standard deviations of the constrained agents are large compared to the baseline agent. However, many of these values are close to half the standard deviation after 1 million time steps, meaning the variance has decreased with additional training. The agent with reward shape 10 maintains a larger variance, especially without augmentation. This is likely a result of the large shaping value,

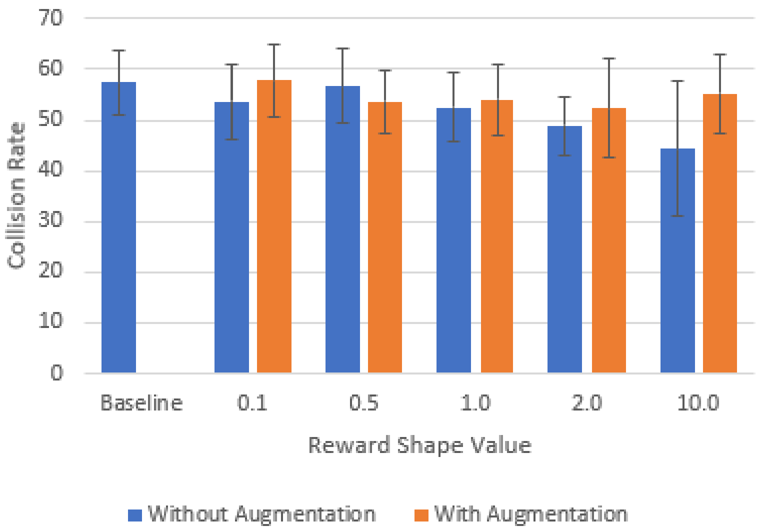


Figure 5.2: Collision rate and its standard deviation after 2 million time steps when using the Strict Turn constraint. A lower collision rate is better.

which can either significantly help or hurt the agent while training depending on when and how often violations occur.

We compare the collision rates of the agent with and without state augmentation in Table 5.6. These values are calculated by subtracting the collision rates with state augmentation in from the collision rates without state augmentation. Hence, a positive value means the collision rate decreases for the given reward shaping value when state augmentation is added to the agent. State augmentation improves the agent for some values but decreases the performance of the agent for other values. For example, with a shaping signal of 0.1, augmentation improves the agent for the first half of training but leads to worse performance after fully trained. On the other hand, with a reward shape value of 10, the performance is only slightly worse halfway through but significantly worse after 2 million steps. There does not seem to be an emerging pattern as to when state augmentation helps or hurts training.

Similarly, Table 5.7 compares collision rates of the agent after 1 million and 2

Table 5.6: The difference in collision rates with and without state augmentation for the Strict Turn constraint. A positive value means the collision rate is lower when the agent used state augmentation compared to not using state augmentation.

	Reward Shape Value				
Time Steps	0.1	0.5	1.0	2.0	10.0
1 million	6.1	−0.8	5.9	−4.9	−0.9
2 million	−4.1	3.1	−1.4	−3.4	−10.8

million training steps. The table shows the difference between these values with a positive number meaning collisions decreased after additional training. Half of the values are near zero, implying that the additional training time does not significantly change these models. Two values (no augmentation 0.1 and 10) show a large improvement. Two values show a small decline in performance (augmentation 0.1 and no augmentation 0.5 while one value shows a significant decrease in performance (with augmentation 1.0). This variety of results means that additional training can either improve or diminish the performance of the agent.

Table 5.7: The difference in collision rates from tests run halfway through and after fully training using the Strict Turn constraint. A positive value means the collision rate decreases after fully trained compared to testing halfway through training.

		Reward Shape Value				
	Baseline	0.1	0.5	1.0	2.0	10.0
Without Augmentation	−0.7	6.9	−2.8	−0.7	−0.7	10.1
With Augmentation	−0.7	−3.3	1.1	−8.0	0.8	0.2

Based on this data, the Strict Turn constraint provides an improvement to the baseline agent when comparing results after 1 million and 2 million time steps. The constraint, both with and without state augmentation, consistently reduces the collision rate without affecting the success rate of the agent, especially with reward shaping values of 1 and 2. These values provide a large enough shaping value to assist the agent in avoiding violations while allowing the agent enough freedom to successfully completed the given task. With this constraint, additional training time does not necessarily lead to improved performance or fewer collisions.

5.3 Loose Turn Constraint

In this section, we look at the results for the agent with the Loose Turn constraint. Table 5.8 contains the evaluation of the agent after 1 million steps without using state augmentation. Looking at the results, the success rate increases slightly by about one point each reward shape value except 10 where the success rate decreases by 1.2 points. The collision rates vary considerably with a shaping value of 1 having the worst collision rate with an almost 5 point increase in collisions over the baseline agents. The agents with a shaping value of 0.5 show the most improvement by 3.3 points while the remaining shaping signals (0.1, 2, and 10) show little to no improvement compared to the baseline.

Table 5.8: Test results when using the Loose Turn constraint without augmentation after 1 million time steps.

		Reward Shape Value				
	Baseline	0.1	0.5	1.0	2.0	10.0
Suc.	26.1 ± 1.3	26.8 ± 0.4	27.0 ± 0.0	27.1 ± 0.3	27.0 ± 0.1	24.9 ± 4.3
Time.	17.1 ± 1.8	17.4 ± 2.7	19.6 ± 8.3	11.5 ± 9.1	17.0 ± 7.8	18.4 ± 15.2
Col.	56.8 ± 1.7	55.8 ± 2.7	53.5 ± 8.3	61.4 ± 9.0	56.1 ± 7.8	56.7 ± 12.4

Table 5.9: Test results when using the Loose Turn constraint with augmentation after 1 million time steps.

		Reward Shape Value				
	Baseline	0.1	0.5	1.0	2.0	10.0
Suc.	26.1 ± 1.3	26.3 ± 1.3	27.0 ± 0.0	24.5 ± 7.9	26.6 ± 1.0	26.2 ± 1.2
Time.	17.1 ± 1.8	17.1 ± 6.8	18.4 ± 7.9	21.7 ± 26.9	16.6 ± 13.2	21.1 ± 10.6
Col.	56.8 ± 1.7	56.6 ± 6.5	54.7 ± 7.9	53.8 ± 19.5	56.8 ± 13.1	52.7 ± 10.7

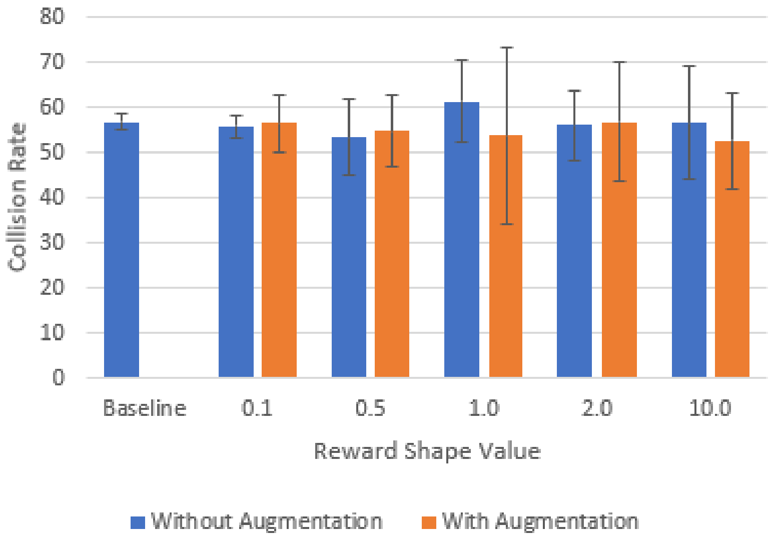


Figure 5.3: Collision rate and its standard deviation after 1 million time steps when using the Loose Turn constraint.

Table 5.9 gives the results after 1 million steps with state augmentation. The collision rates with and without state augmentation are also compared in Figure 5.3. These results show an overall improvement over the results without augmentation. The success rates are similar to or slightly above the baseline value with the exception of shaping value 1 have a 1.6 point decrease in successes. The collision rates for shaping values of 0.5, 1, and 10 show a slight improvement while the collision rates for 0.1 and 2 are approximately the same as the baseline rate.

The models with a reward shape value of 1 seem to be an anomaly with widely varying results. Without state augmentation, four of the seeds have collision rates exceeding 70%, and averaging the seeds gives the highest collision rate of any test. One of the seeds with state augmentation learned to basically take little or no actions with 94% of episodes ending in timeouts, 2% ending in success, and the remaining 4% resulting in a collision. This seed causes a large standard deviation and the reduces success rate shown in the results. Removing this data point, the results become very

Table 5.10: Test results when using the Loose Turn constraint without augmentation after 2 million time steps.

		Reward Shape Value				
	Baseline	0.1	0.5	1.0	2.0	10.0
Suc.	26.8 ± 0.4	26.6 ± 1.4	26.5 ± 1.1	26.3 ± 1.4	26.9 ± 0.2	25.6 ± 2.2
Time.	15.7 ± 6.4	16.7 ± 3.3	17.9 ± 8.5	21.6 ± 11.6	19.8 ± 5.2	22.3 ± 4.4
Col.	57.5 ± 6.4	56.8 ± 3.2	55.6 ± 7.7	52.1 ± 11.5	53.3 ± 5.1	52.1 ± 5.0

Table 5.11: Test results when using the Loose Turn constraint with augmentation after 2 million time steps.

		Reward Shape Value				
	Baseline	0.1	0.5	1.0	2.0	10.0
Suc.	26.8 ± 0.4	26.7 ± 0.9	27.0 ± 0.2	26.7 ± 1.0	26.9 ± 0.4	26.1 ± 1.2
Time.	15.7 ± 6.4	18.3 ± 3.3	18.2 ± 7.8	15.9 ± 8.8	21.0 ± 4.9	23.5 ± 5.1
Col.	57.5 ± 6.4	55.0 ± 2.9	54.9 ± 7.7	57.4 ± 8.8	52.2 ± 4.6	50.5 ± 4.4

similar to results without state augmentation with a good success rate but a collision rate exceeding the baseline value.

Overall, no pattern emerges for which shaping value presents the best results at this point in training. The results are less consistent and show the Loose Turn constraint providing little improvement over the baseline agent after 1 million training steps.

Results after 2 million time steps without state augmentation are provided in Table 5.10. The success rates are comparable to the baseline for each shaping value except 10 where the success rate drops by more than one point. The collision rates decrease as the shaping value increases. The collision rates for shaping values 0.1 and 0.5 only have a slight 1–2 point reduction while the collision rates for the larger shaping values have a 4–5 point decrease. Hence, the best shaping value for this configuration is 1 or 2 since they have the lowest collision rates while keeping a good success rate.

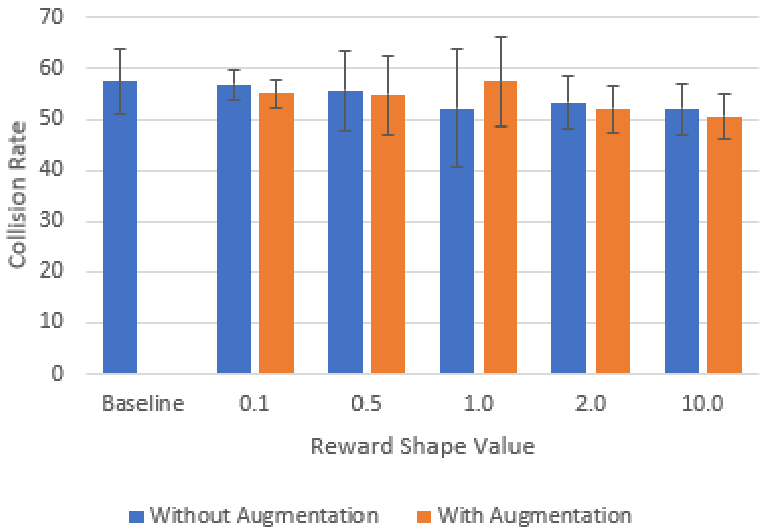


Figure 5.4: Collision rate and its standard deviation after 2 million time steps when using the Loose Turn constraint.

Table 5.11 contains the results with state augmentation after 2 million time steps. Figure 5.4 contains a chart comparing the collision results with and without state augmentation. Like the other results, the success rates are similar to the baseline value with the exception of reward shape 10 having a slight, almost 1 point reduction. For a shaping value of 1, the collision rate remains the same as the baseline whereas 0.1 and 0.5 have a small 2 point improvement. The larger shaping values 2 and 10 again have the best collision rates with a 5 and 7 point reduction in collisions over the baseline results, respectively. Again, the shaping value of 2 emerges as the best value with a low collision rate and a high success rate compared to the other models.

Similar to the Strict Turn constraint, the agents with the Loose Turn constraint have significant variance in performance and specifically collision rate compared to the baseline agent. This is regardless of training time and the inclusion or exclusion of state augmentation.

Table 5.12 compares the collision rate of agents trained with and without state augmentation. A positive value means adding state augmentation improved the agent

Table 5.12: The difference in collision rates with and without state augmentation for the Loose Turn constraint. A positive value means the collision rate is lower when the agent used state augmentation compared to not using state augmentation.

	Reward Shape Value				
Time Steps	0.1	0.5	1.0	2.0	10.0
1 million	−0.8	−1.2	7.6	−0.7	4.0
2 million	1.8	0.7	−5.3	1.1	1.6

by decreasing the collision rate. Overall, there is not a significant difference in the collision rates for most shaping values when state augmentation is added. Using a reward shape value of 10, adding state augmentation shows a 4-point decrease in collisions halfway through training but a smaller reduction after 2 million steps. The results with a shaping value of 1 show the biggest change; state augmentation provides a significant reduction in collision rate after 1 million time steps and a significant increase once fully trained. From these results, there is no clear trend showing that adding state augmentation improves or diminishes the performance of the agent.

We compare the collision rates after 1 million and 2 million training steps for the Loose Turn constraint in Table 5.13. For the larger shaping values (2 and 10), the additional training time leads to a decrease in the collision rate. For the smaller shaping values (0.1 and 0.5), further training does not have a significant effect on the collision rate, similar to the baseline agents. The shaping value of 1 has unique results where the agents without augmentation show a large 9 point improvement whereas the agents with augmentation have more collisions after additional training. This is from the abnormal results with a shaping signal of 1 encountered with the Loose Turn constraint.

While the Loose Turn constraint does not seem to improve the agent after 1 million time steps, the results show that the constraint can reduce the collision rate after additional training. The shaping value of 2 emerges as the ideal value that

Table 5.13: The difference in collision rates from tests run halfway through and after fully training using the Loose Turn constraint. A positive value means the collision rate decreases after fully trained compared to testing halfway through training.

		Reward Shape Value				
	Baseline	0.1	0.5	1.0	2.0	10.0
Without Augmentation	−0.7	−1.0	−2.1	9.3	2.8	4.6
With Augmentation	−0.7	1.6	−0.2	−3.6	4.6	2.2

reduces the reward enough for the agent to learn to avoid violations and collisions without hindering the success rate of the agent which happens with a shaping signal of 10. Due to the nature of the constraint, it is not violated as often as the Strict Turn constraint, so it seems to need additional training time, out to 2 million time steps, for the constraint to show a significant reduction in the collision rate over the baseline agent.

5.4 Dithering Constraint

The final constraint tested is the Dithering constraint. Table 5.14 contains the results of the agent after 1 million time steps without using state augmentation. For each model, the success rate increases over the baseline agent by 0.5–1 point with the larger increase resulting from larger shaping values. Compared to the baseline agent, the collision rate increases slightly for a shaping value of 10, remains approximately the same for three shaping values (0.1, 0.5, and 2), and decreases for a reward shape value of 1. With previous constraints, a shaping value of 10 would typically result in one of the lower collision rates but also decreases the success rate, whereas the opposite is true for this comparison with those agents having a high collision and high success rate. Here, a shaping signal of 1 emerges as the best choice with a 4.4 point decrease in collisions and the highest success rate.

Table 5.14: Test results when using the Dithering constraint without augmentation after 1 million time steps.

		Reward Shape Value				
	Baseline	0.1	0.5	1.0	2.0	10.0
Suc.	26.1 ± 1.3	26.7 ± 0.9	26.6 ± 0.8	27.1 ± 0.2	27.0 ± 0.0	27.1 ± 0.3
Time.	17.1 ± 1.8	16.2 ± 2.0	17.1 ± 2.4	20.6 ± 11.2	16.5 ± 6.1	14.2 ± 5.3
Col.	56.8 ± 1.7	57.1 ± 1.9	56.3 ± 2.5	52.4 ± 11.2	56.5 ± 6.1	58.7 ± 5.1

Table 5.15: Test results when using the Dithering constraint with augmentation after 1 million time steps.

		Reward Shape Value				
	Baseline	0.1	0.5	1.0	2.0	10.0
Suc.	26.1 ± 1.3	26.8 ± 0.4	27.2 ± 0.4	27.0 ± 0.0	26.9 ± 0.2	26.8 ± 0.5
Time.	17.1 ± 1.8	18.4 ± 2.3	16.8 ± 5.5	18.5 ± 8.3	18.7 ± 4.6	18.4 ± 5.7
Col.	56.8 ± 1.7	54.8 ± 2.4	56.1 ± 5.5	54.6 ± 8.3	54.4 ± 4.6	54.8 ± 5.7

Table 5.15 presents the evaluation of the agent after 1 million time steps using state augmentation. Across the board, there is a small but consistent improvement in the constrained agents' performance compared to the baseline models. The success rate for each agent is approximately a one point improvement over the baseline success rate. Further, each shaping value reduces the collision rate by 2–2.4 points with the exception of shaping value 0.5 having only a slight reduction. Since we have similar results for nearly all shaping values, the improvement is likely a result of the additional information the agent receives from the state augmentation and not from the reduction in the reward when violations occur. This claim is backed up from the results without state augmentation where the agents show little improvement over the baseline.

The variance of the results with this constraint is significantly less compared with the previous results. This is especially true for the agents with smaller shaping signals with the standard deviations being similar to the baseline values. A reward

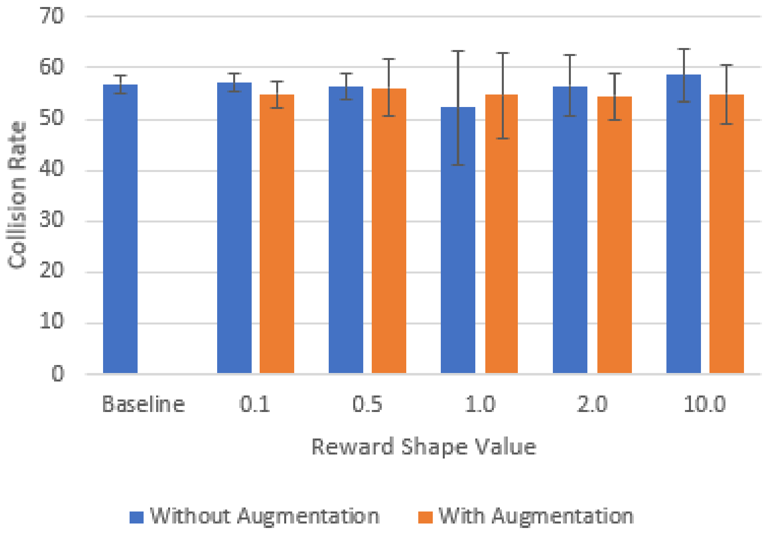


Figure 5.5: Collision rate and its standard deviation after 1 million time steps when using the Loose Turn constraint.

shape value of 1 has the largest variance in results both with and without state augmentation. Figure 5.5 graphs the collision rate for each shaping value with and without state augmentation for a visual comparison of the results.

The results of training the models for 2 million time steps without state augmentation are presented in Table 5.16. The success rates are all effectively the same with the results of the constrained agents differing by no more than 0.2 points from the baseline value. The collision results are more interesting with a shaping value of 1 having a similar value to the baseline and the collision rate decreasing as the shaping signal is increased or decreased. This is the opposite of most other results where a shaping value of 1 is typically one of the better values and the performance decreasing with larger or smaller values. However, the collision rates are very similar for these shaping values (excluding 1), differing by at most half a point while having about a 3 point decrease from the baseline value. Hence, the reward shape value of 1 is more of the outlier with the remaining values each showing consistent improvement over the baseline agent.

Table 5.16: Test results when using the Dithering constraint without augmentation after 2 million time steps.

	Reward Shape Value					
	Baseline	0.1	0.5	1.0	2.0	10.0
Suc.	26.8 ± 0.4	26.7 ± 0.8	26.8 ± 0.6	27.0 ± 0.1	26.7 ± 1.0	27.0 ± 0.1
Time.	15.7 ± 6.4	18.8 ± 6.5	18.4 ± 3.7	15.8 ± 5.2	18.7 ± 2.6	18.7 ± 2.6
Col.	57.5 ± 6.4	54.5 ± 6.6	54.8 ± 3.6	57.2 ± 5.2	54.7 ± 2.6	54.3 ± 2.6

Table 5.17: Test results when using the Dithering constraint with augmentation after 2 million time steps.

	Reward Shape Value					
	Baseline	0.1	0.5	1.0	2.0	10.0
Suc.	26.8 ± 0.4	26.5 ± 1.3	26.7 ± 1.0	27.1 ± 0.5	26.7 ± 1.0	26.8 ± 0.5
Time.	15.7 ± 6.4	17.6 ± 5.4	18.2 ± 4.2	16.9 ± 5.7	20.0 ± 4.1	16.2 ± 6.2
Col.	57.5 ± 6.4	55.9 ± 5.0	55.1 ± 4.4	56.1 ± 5.7	53.3 ± 4.4	57.0 ± 6.1

Table 5.17 contains the results after 2 million time steps with state augmentation. Similar to the previous results, the success rates are effectively the same with values differing by at most 0.3 points from the baseline. The three smallest shaping values (0.1, 0.5, and 1) each show a small 1.5–2 point reduction in collisions. A shaping value of 2 has the largest decrease in collisions by over 4 points while a shaping value of 10 has almost no improvement.

In contrast to previous constraints, the standard deviation of almost every model is less than the standard deviation of the baseline after 2 million training steps, especially for the shaping values in the middle. Hence, this constraint seems to make the agent more consistent between runs whereas previous constraints would often cause a large variance in performance with different seeds. Figure 5.6 displays the collision rate and standard deviation for each shaping value with and without state augmentation.

A comparison of the collision rates of the agents trained with and without state

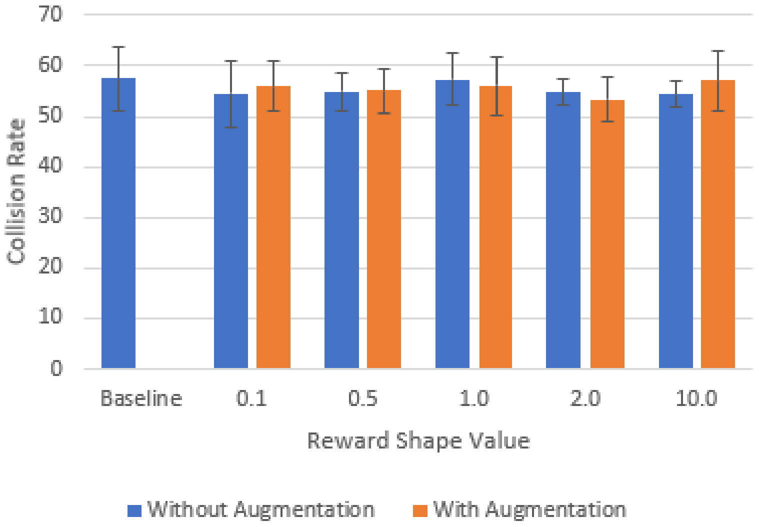


Figure 5.6: Collision rate and its standard deviation after 2 million time steps when using the Loose Turn constraint.

augmentation is provided in Table 5.18. After 1 million time steps, state augmentation provides a 2–4 point decrease in collisions for three of the shaping values and provides no significant change for shaping value 0.5. However, the collision rate worsens for shaping value 1 by 2.2 points. This is because the collision rate for this shaping value without augmentation is the best of all agents with the Dithering constraint, whereas the value with augmentation is more consistent without other values. After 2 million training steps, there is less difference between the agents with and without state augmentation with all but one differing by less than 1.5 points. The outlier is shaping value 10 which has a 2.7 point increase in collisions when state augmentation is added to the agent.

Table 5.19 contains a comparison of the collision rates after 1 and 2 million training steps are completed. Without augmentation, the additional training time improves every model except for shaping value 1, which is again an outlier. These agents performed well halfway with the best results overall in this constraint, but after 2 million time steps, the agents have the highest collision rate of any fully trained

Table 5.18: The difference in collision rates with and without state augmentation for the Dithering constraint. A positive value means the collision rate is lower when the agent used state augmentation compared to not using state augmentation.

	Reward Shape Value				
Time Steps	0.1	0.5	1.0	2.0	10.0
1 million	2.3	0.2	−2.2	2.1	3.9
2 million	−1.4	−0.3	1.1	1.4	−2.7

Table 5.19: The difference in collision rates from tests run halfway through and after fully training using the Dithering constraint. A positive value means the collision rate decreases after fully trained compared to testing halfway through training.

		Reward Shape Value				
	Baseline	0.1	0.5	1.0	2.0	10.0
Without Augmentation	−0.7	2.6	1.5	−4.8	1.8	4.4
With Augmentation	−0.7	−1.1	1.0	−1.5	1.1	−2.2

results. With state augmentation, there is no significant improvement in the agents with the additional training time. This is surprising but is likely a result of the additional information provided to the agent with state augmentation causing the improvement over the baseline agent. The constrained models are able to quickly learn to reduce constraint violations and collisions with the state augmentation as opposed to learning the rule from reward shaping, so additional training time does not significantly improve the agents.

The Dithering constraint seems to provide a small but consistent performance improvement to the agents, especially with state augmentation. Previous constraints showed larger reductions in the collision rate but only with some shaping values whereas other choices would result in little or no improvement in the agent. With few exceptions, the Dithering constraint provided a 2–3 point reduction in collisions while keeping or slightly improving the success rate of the agent. The best result is from the agents with a reward shape value of 1, no state augmentation, and only 1 million training steps, but adding augmentation provides a more consistent improvement in the agent regardless of the shaping signal.

Empirically, we found the ideal reward shaping values to be around 1–2 which corresponds to roughly 2–5% of the total return for a successful episode. This allows constraint violations to have a noticeable effect on the final reward sum, while multiple violations can occur without completely eliminating other positive rewards. We recommend testing several values before settling on an ideal shaping value for a constraint.

State augmentation does not provide a benefit to the Strict Turn collision as the DFA is very simple with only two states: either the starting state or the accepting state. Therefore, using augmentation does not really provide any additional benefit to the agent over reward shaping. However, for the Dithering constraint, because the DFA is more complex with 9 states, state augmentation provides the agent with additional information that the agent can use to guide the current action choices.

Chapter 6

Conclusion & Future Work

In this study, we imposed formal language constraints on an RL agent to improve its safety as it autonomously navigates a simulated driving environment. The constraints can significantly reduce the number of collisions involving the agent vehicle. While an autonomous car will not be allowed to drive on roads until nearly all dangerous behaviors have been eliminated, the relatively simple constraints introduced in this work can play a part in tuning RL algorithms toward safer actions.

Future work can explore new and more complicated constraints to facilitate the learning and safety of the agent. Possible future constraints include penalizing the agent for leaving the current lane, moving too close to another vehicle or pedestrian, or for a traffic violation such as running a stop sign. Constraints can also be learned based on violation feedback from the agent. Furthermore, new and/or old constraints can be applied to the same agent to see if the combination of constraints can further enhance the safety of the agent and reduce the number of collisions. This can be especially beneficial when constraints apply to different parts of the driving task. For example, employing a constraint focused on navigating turns within intersections and a separate constraint for helping the agent to maintain its position in the lane. Additional future work can look into different baseline agents to verify if constraints are universal and can be applied to various RL algorithms, or if constraints must be tuned specifically for each algorithm. It would also be interesting to compare how well the safety constraints perform depending on the performance of the baseline agent. That is, would a strong baseline agent negate the need for the constraints, or would the constraints provide an additional performance benefit to the already high functioning base agent.

Bibliography

[1] Autopilot. `https://www.tesla.com/autopilot`. Accessed: 2020-07-22.

[2] Darpa grand challenge 2005. `https://archive.darpa.mil/grandchallenge05/gcorg/index.html`. Accessed: 2020-07-22.

[3] Tesla vehicle safety report. `https://www.tesla.com/VehicleSafetyReport`. Accessed: 2020-07-22.

[4] Urban challenge. `http://www.grandchallenge.org/`. Accessed: 2020-07-22.

[5] Disengagement reports - california dmv. `https://www.dmv.ca.gov/portal/vehicle-industry-services/autonomous-vehicles/disengagement-reports/`, Jun 2020. Accessed: 2020-07-22.

[6] R. Brooks. Predictions scorecard, 2020 january 01. *Rodney Brooks - Robots, AI, and Other Stuff*, Jan 2020.

[7] D. Chen, B. Zhou, V. Koltun, and P. Krähenbühl. Learning by cheating. In L. P. Kaelbling, D. Kragic, and K. Sugiura, editors, *3rd Annual Conference on Robot Learning, CoRL 2019, Osaka, Japan, October 30 - November 1, 2019, Proceedings*, volume 100 of *Proceedings of Machine Learning Research*, pages 66–75. PMLR, 2019.

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

[9] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun. CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, pages 1–16, 2017.

[10] J. Garcıa and F. Fernández. A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research*, 16(1):1437–1480, 2015.

[11] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[12] Y. Hu, A. Nakhaei, M. Tomizuka, and K. Fujimura. Interaction-aware decision making with adaptive strategies under merging scenarios. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2019, Macau, SAR, China, November 3-8, 2019*, pages 151–158. IEEE, 2019.

[13] W. Kim, V. Anorve, and B. C. Tefft. American driving survey, 2014 - 2017. *AAA Foundation for Traffic Safety*, 2019.

[14] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In Y. Bengio and Y. LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[15] X. Liang, T. Wang, L. Yang, and E. P. Xing. CIRL: controllable imitative reinforcement learning for vision-based self-driving. In V. Ferrari, M. Hebert, C. Sminchisescu, and Y. Weiss, editors, *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings,*

Part VII, volume 11211 of *Lecture Notes in Computer Science*, pages 604–620. Springer, 2018.

[16] M. Lynberg. Automated vehicles for safety, Jun 2020.

[17] A. Marshall. Elon musk promises a really truly self-driving tesla in 2020. *Wired*, Feb 2019.

[18] A. Marshall. Ford taps the brakes on the arrival of self-driving cars. *Wired*, Apr 2019.

[19] T. M. Mitchell. *Machine learning.* McGraw Hill series in computer science. McGraw-Hill, 1997.

[20] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In M. Balcan and K. Q. Weinberger, editors, *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 1928–1937. JMLR.org, 2016.

[21] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[22] D. Pomerleau. ALVINN: an autonomous land vehicle in a neural network. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 1, [NIPS Conference, Denver, Colorado, USA, 1988]*, pages 305–313. Morgan Kaufmann, 1988.

[23] E. Quint, D. Xu, H. Dogan, Z. Hakguder, S. Scott, and M. B. Dwyer. Formal language constraints for markov decision processes. In *NeurIPS 2019 Workshop on Safety and Robustness in Decision Making*, 2019.

[24] E. Quint, D. Xu, S. Flint, T. Bienhoff, S. Scott, and M. B. Dwyer. Formal language constraints for markov decision processes. In *NeurIPS*, Under Review.

[25] R. Randazzo. Waymo to start driverless ride sharing in phoenix area this year. *The Arizona Republic*, Jan 2018.

[26] D. M. Saxena, S. Bae, A. Nakhaei, K. Fujimura, and M. Likhachev. Driving in dense traffic with model-free reinforcement learning. *CoRR*, abs/1909.06710, 2019.

[27] S. Shalev-Shwartz, S. Shammah, and A. Shashua. Safe, multi-agent, reinforcement learning for autonomous driving. *CoRR*, abs/1610.03295, 2016.

[28] D. Streitfeld. Waymo to offer phoenix area access to self-driving cars. *The New York Times*, Apr 2017.

[29] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction.* A Bradford Book, Cambridge, MA, USA, 2018.

[30] T. Tieleman and G. Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

[31] M. Toromanoff, É. Wirbel, and F. Moutarde. End-to-end model-free reinforcement learning for urban driving using implicit affordances. *CoRR*, abs/1911.10868, 2019.

[32] J. R. Treat, N. S. Tumbas, S. T. Mcdonald, D. Shinar, R. D. Hume, R. E. Mayer, R. L. Stansifer, and N. J. Castellan. Tri-level study of the causes of traffic accidents: Final report. volume i: Casual factor tabulations and assessments. May 1979.

[33] C. J. C. H. Watkins and P. Dayan. Technical note q-learning. *Mach. Learn.*, 8:279–292, 1992.

[34] Y. Wu, E. Mansimov, R. B. Grosse, S. Liao, and J. Ba. Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. In I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pages 5279–5288, 2017.