Deep Reinforcement Learning in Self-Driving Car Systems

Independent Study Thesis

Presented in Partial Fulfillment of the Requirements
for the Degree Bachelor of Arts in the Department
of Mathematics and Computer Science at The
College of Wooster

by
Eric Michael Gabriel

The College of Wooster
2019

Advised by:

Dr. Nathan Fox

Abstract

This paper explains the attempted development of a deep reinforcement learning-based self-driving car system for a simulated, 3D environment. As a relatively new deep learning paradigm with a lot of potential, the interest in developing this system is to draw conclusions about the place for deep reinforcement learning in production-ready self-driving car systems. The deep reinforcement learning algorithm called double deep Q-learning, which uses a double deep Q-network with convolutional and simple recurrent layers, is used to steer the self-driving car. As such, the requisite foundational material, that of reinforcement learning and deep learning, are explored so as to make the developed system understandable. The results of the attempted development are surprisingly negative, and accordingly, the conclusions reflect this.

Acknowledgements

"John answered and said, A man can receive nothing, except it be given him from heaven." - John 3:27 (KJV)

Contents

List of Figures

Chapter 1

Autonomous Vehicles

An *autonomous vehicle* or a *self-driving car* is any vehicle that is at least partially driven by an automated system rather than by a human driver. The perks associated with giving vehicles autonomy are manyfold. In terms of practicality, increasing the average level of autonomy in the population of vehicles on the road should result in a reduced number of crashes, less severe crashes as measured by injuries and fatalities, and a reduced amount of air pollution [29]. On the consumer's end, autonomous vehicles appeal to everyone's inclination for leisure and convenience; drivers would not have to be 100% focused 100% of the time. Given these societal benefits and the supposed demand for autonomous vehicles, most major tech companies are working on self-driving car technology in some fashion.

Though a lot of progress has been made in the past 20 to 30 years, a fully autonomous car, one that is capable of driving itself in nearly all circumstances, has not yet been developed. Furthermore, obviously, not every vehicle operates at the same level of autonomy. Therefore, in discussing autonomous vehicles, it

is useful to measure a vehicle's level of autonomy by some set of standards. One of the most common such measurements is the one adopted by National Highway Transportation Safety Administration (NHTSA), which is an organization that helps regulate automobiles in America.

1.1 Levels of Autonomy

The NHTSA has adopted the Society of Automotive Engineers (SAE) International's classification system for autonomy in driving systems. This classification system features six levels and is typically split in half. The lower half is where the human does the majority of the driving, and the upper half is where the automated systems do the majority of the driving. These classifications can be seen in Figure 1.1 on page 2 and Figure 1.2 on page 3. While the classifications are useful, they are a little vague given that the levels somewhat blur together. Therefore, some examples are in order.

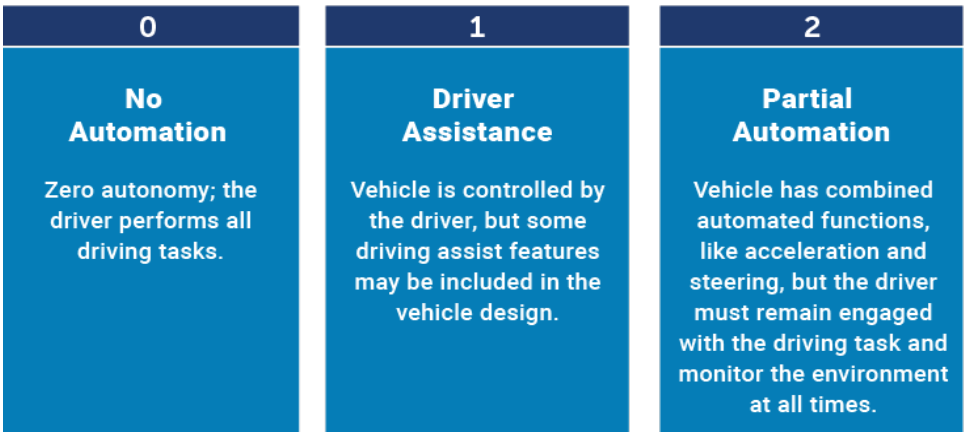

Figure 1.1: The first three levels of autonomy in the SAE International classification [30]

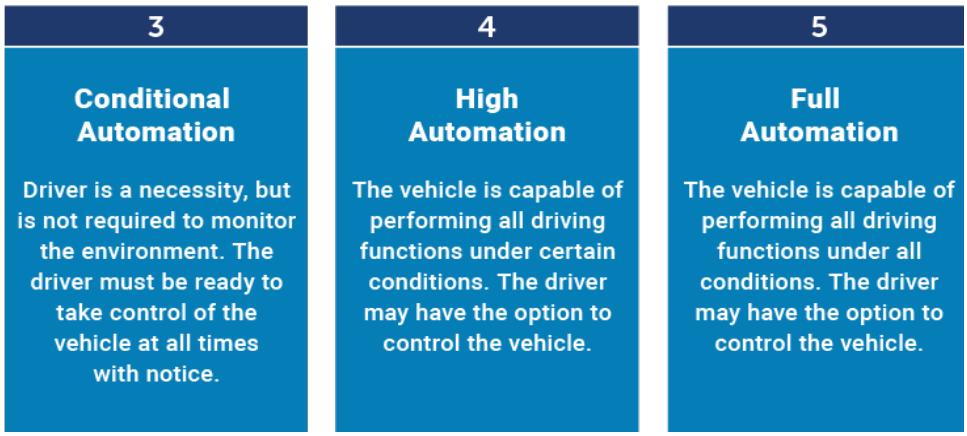

Figure 1.2: The last three levels of autonomy in the SAE International classification [30]

A car at Level 0 would not have any intelligent systems that make any driving maneuvers. Qualifying systems as "intelligent" is important because nearly all modern vehicles contain some sort of a computer that helps operate the vehicle. Therefore, Level 0 includes vehicles with nothing more than some embedded computers.

Level 1 is where a relatively simple autonomous system is present in the vehicle. Such a system is mostly limited to making adjustments in steering or acceleration; it cannot really drive the vehicle. Therefore, this includes vehicles with either adaptive cruise control, which applies the brakes when the car gets too close to the car ahead, or lane centering, which is supposed to perform course correction if a vehicle begins to drift over a line [4]. Having both adaptive cruise control and lane centering [4], i.e., having multiple intelligent systems, would classify a vehicle as Level 2.

Level 3 is where the automated system is actually capable of driving the vehicle beyond making small adjustments. At this level, the vehicle is fully

capable of driving in dense freeway traffic or at lower speeds [4]. While this is quite a step up from the previous levels, Level 3 has a significant downside: the human driver must always be ready take over if the automated system is unable to proceed [4]. In terms of practicality, requiring this amount vigilance from drivers makes this level not much better than Level 2.

Level 4, however, is actually useful. A Level 4 self-driving car can drive in freeway environments and in city environments. When the automated system is unsure how to proceed in either environment, it will bring the vehicle to a safe stop [4]. At that point, the human driver could retake control, or the car could just wait until the issue goes away.

At Level 5, a self-driving car would be considered "fully autonomous." At this level, the human "driver" could conceivably go to sleep and not have worry about anything (though, this will probably not be permissible for a long time for legal reasons at the very least). Like Level 4, a Level 5 car would safely stop itself if it encounters an unresolvable issue [4]. Encountering an unresolvable issue, however, would be much less likely at Level 5 than at Level 4 simply because Level 5 is so much more advanced. Level 5 would meet or exceed human driving capabilities.

Whether or not Level 5 is actually attainable is debatable, but many companies and research groups are certainly aiming for such an achievement. As one can imagine, there is not just one way of developing a Level 3, 4, or 5 vehicle; there are different approaches to consider.

1.2 How Do Autonomous Vehicles Do It?

In general, autonomous vehicle systems can take in data from infrared and conventional cameras, ultrasonic proximity sensors, light detection and ranging (LiDAR) sensors, a GPS, an accelerometer, a gyroscope, and the car's internal systems, e.g., whether or not the anti-lock braking system has been activated. From this mountain of data, an advanced autonomous vehicle system is supposed to output the appropriate steering angle and acceleration amount.

Being able to go from the inputs to the *correct* outputs is, understandably, the hard part. The means of bridging this gap can be thought of as a kind of spectrum. On the one end, there is the *end-to-end* approach, and on the other end, there is the hard-coded *modular pipeline* approach.

1.2.1 Modular Pipelines

A *modular pipeline* autonomous vehicle system is composed of different modules. These modules each serve a certain, well-defined purpose, and are intended to be utilized in some sequence. One possible modular pipeline is a three-stage pipeline: a perception-planning-control pipeline. The perception-planning-control pipeline, as the name implies, splits the autonomous vehicle system into perception, planning, and control stages.

The perception stage takes in the raw data, applies some preprocessing operations to that raw data, and then outputs the preprocessed data [12]. Preprocessing is usually done with *computer vision* techniques, which are just algorithms for extracting more information from images, and with statistical analysis. There are a few reasons why preprocessing the input can be good idea,

but the main one is that it makes the planning stage much easier.

The planning stage takes the preprocessed data from the perception stage, applies some algorithm(s) to the data, and tries to generate "near-term goal states that represent the desired position and orientation of the car in the near future" [12]. "Near-term goals" are local goals. They are things like the waypoints that make up a right-turn, where a waypoint is something like an (x, y, z) coordinate with a (yaw, pitch, roll) triple. They are not natural language commands like "make a right turn soon."

The continuous controller stage takes the near-term goal states from the planning stage and determines how much the current steering angle and acceleration amount need to be changed in order to achieve the near-term goal states by the next time-step [12]. Then, the continuous controller interfaces with the car in some manner to see that the instructions are carried out.

There are a few advantages to taking this approach when developing an autonomous vehicle system. Generally speaking, it usually is easier for developers to understand how a modular system operates, isolating issues should be fairly straightforward, and optimizations can be made for specific use cases, e.g., sliding on ice or figuring out who has the right-of-away at a 4-way stop sign intersection. There are, of course, disadvantages associated with this approach. Again, generally speaking, a general-purpose modular pipeline can be difficult to implement efficiently given its many sub-pieces and everything that could occur while driving. Additionally, failures in one stage of the pipeline could very easily "cascade" down the pipeline and produce an erroneous response.

1.2.2 End-to-end Systems

The *end-to-end* approach requires much less explanation than the modular pipeline approach. A pure end-to-end system approach simply takes the raw data as input, applies some sort of artificial intelligence to the raw data, and then outputs a steering angle and a positive or negative acceleration amount. This approach is called "end-to-end" because from one end of the system to the other, there are no human, hard-coded algorithms that dictate what must be done to the data or even how the data is related to the steering and acceleration amounts. The artificial intelligence does all of the heavy lifting.

The advantages of this approach are that such a system should be able to learn a general set of principles it could apply to a wide variety of driving scenarios, and the development of such a system should not require expert knowledge of autonomous vehicles. The disadvantages are that the computational power and time required to actually teach an AI to drive correctly would likely be substantial. Particularly, getting such a system to obey even simple traffic rules from the raw input data alone would be very difficult. Additionally, the AI would essentially be a *black-box*, implying that the developers would not be able to know exactly how the AI makes its decisions. This means that if something goes wrong, there would not be much that the developers could do to isolate the issue.

1.3 In-Between Approaches and This Paper

These two approaches to developing self-driving car systems are not mutually exclusive. In fact, they are often used in conjunction with one another to varying degrees; hence, there is a spectrum of approaches. The justification for synthesizing the two approaches is that each approach has its own disadvantages, and the other approach can, at least partially, compensate for those disadvantages.

This brings us to the function of this paper: to explain and document the attempted development of a simulated, Level 3 autonomous vehicle system for a simulated, 3D environment. This system is primarily end-to-end, though it does incorporate some modular aspects. The primarily end-to-end system uses the *deep reinforcement learning* algorithm called *double deep Q-learning* to teach a *double deep Q network* to drive a simulated car in a simulated environment solely from camera images and miscellaneous sensor data. The end goal of attempting to develop such a system is to extrapolate and draw a conclusion about the role that the relatively new field of deep reinforcement learning could have in self-driving cars, which is a mostly unexplored application.

Deep reinforcement learning, which is explained in Chapter 4, is a fairly complicated subject that requires an understanding of the fields of *reinforcement learning* and *deep learning*. As such, Chapter 2 and Chapter 3 respectively provide that requisite foundational material. With the conceptual and theoretical underpinnings established, this paper then presents and examines the aforementioned system and its development in Chapter 5. The results of this research and the conclusion are given in Chapter 6.

Chapter 2

Reinforcement Learning

This explanation of *reinforcement learning* begins with a high-level, minimally-mathematical description of reinforcement learning. The purpose of this is to make the subsequent explanation of the mathematical foundations of reinforcement learning clearer and less detached from the final goals: to explain the underlying math of reinforcement learning.

2.1 The Basic Idea

The reinforcement learning setting begins with an *agent* and an *environment* [2]. The environment is either a virtual world, e.g., a video game or simulation, or the real world, and the agent is some unit of software that takes in observations of the state of the environment and outputs some commands to control an object in the environment.

Initially, the agent does not know how to control its object, but the agent does know how to observe the state of the environment and what actions it can

take in the environment for any given observation. The reason for this is simply that the agent has not yet learned, for a given observation, which actions are good and which actions are bad; all actions have equal value at the beginning. Calculating the value of taking any action for any observation is more or less what the agent sets out to learn in the reinforcement learning setting.

Calculating the value of taking an action when given some observation is, perhaps surprisingly, not so straightforward. This is because in most real-world applications, the value of an action can not be known immediately after the action is carried out. That value, which is more or less a quantification of the consequences of the action in so far as it results in the agent controlling its object in a goal-oriented manner, can be delayed and/or spread out over time. Not only that, but assuming that the agent does not just take one action and quit, there is a sort of confounding effect between the different actions; it would not be entirely clear which action is the "root cause" of a reward or penalty. Calculating an obfuscated, positive or negative reward for taking an action under some observation is what reinforcement learning algorithms allow the agent to do.

The "learning" in "reinforcement learning" is learning by doing, learning from interactions with the environment. From the reinforcement learning agent's perspective, "interactions" are represented as *transitions* [27], which are tuples that discretize time [2] and consist of three principal parts. The first part of a transition is the observed state of the environment for the current time step. The second part of a transition is the action that the agent selects, given the current state. The third and last part of an interaction is the consequence of taking the selected action in this current state. The consequence is represented as a numerical reward, a real number.

The reward, unlike the current state of the environment or the next action to take, is given by the programmer of the software learning agent, or more specifically, by a function that outputs rewards that the programmer defines before learning begins. Reward functions vary from application to application, but all (good) reward functions will attempt to give the agent feedback on its actions insofar as those actions have achieved or worked towards achieving a goal that the programmer wants the agent to achieve. Such a function could give rewards only when the end goal is accomplished, or it could give rewards while there is progress being made towards the end goal.

How exactly an agent learns from the rewards of a reward function is through reinforcement. Reinforcement in the mathematical sense is an approximation of reinforcement in the psychological and biological senses, e.g., Pavlov's dog experiment [38]. For a given state, the actions that lead to positive rewards are reinforced, such that they will be more highly favored relative to other possible actions. The actions that lead to negative rewards will be discouraged, such that they will be less highly favored. The current degree to which an action for some observation is favored is represented by the agent's current estimate of the reward for that action and given observation.

What this all means is that successful reinforcement learning depends heavily on the agent's ability to correctly estimate the reward of any action for any given observation of the state of the environment. Once the agent has learned those estimates, it can, according to some defined policy for selecting actions, select the actions that will result in the best outcomes. Typically, after learning is finished, the policy for selecting actions is some type of greedy policy: Given an observation, select the action with the highest estimated reward. Thus,

the agent ultimately should learn to optimally control its object for any given state of the environment so as to achieve an end goal that is directly related to the reward function.

This is still a bit of a simplification; uncertainty has yet to be mentioned. Many real-world scenarios are not deterministic, and the agent seldom has complete control over every facet of the environment [2]. This means that one state does not necessarily follow another. If the next state cannot be known before it occurs, then the next reward, which is related to next state, also cannot be known before it is received. Therefore, the truly reward-maximizing action, whose reward has already been stated to be delayed into the future, cannot be estimated with 100% certainty. To go back to self-driving cars, this is tantamount to saying that the software agent that is driving the car cannot control what other drivers do, the road conditions, or the weather, and so, the driving agent will necessarily be somewhat unsure about which action is truly best.

Therefore the actual algorithms by which a reinforcement learning agent learns to control its object need to be able to learn to handle uncertainty as well as delayed rewards [5]. Many such algorithms have their mathematical foundations in the finite Markov decision process and temporal difference learning, the topics of the next sections.

2.2 The Finite Markov Decision Process

The *finite Markov decision process* (finite MDP) is a mathematical formulation of the interaction process, not the learning process, between an

agent and its environment as described in the previous section. The MDP begins by considering three spaces: the finite *state space* $\mathcal{S}$, the finite *action space* $\mathcal{A}$, and the finite *reward space* $\mathcal{R}$. These spaces respectively represent the finite set of all possible states of the environment, the finite set of all possible actions that the agent could take, and the finite set of all possible rewards the agent can receive [38]. As such, the state at time step t is a random variable S_t from $\mathcal{S}$, the action taken at time step t is a random variable A_t from $\mathcal{A}$, and the reward from taking action A_t in state S_t is a random variable R_t from $\mathcal{R}$. Thus, starting with initial state S_0, the MDP sequence would be:

$$S_0, A_0, R_0, S_1, A_1, R_1, S_2, \ldots, S_j, A_j, R_j, S_{j+1},$$

where j is the number of elapsed time steps [38]. Note that the subscript on a reward variable indicates the time step with which a reward is associated, not the successive time step in which a reward was observed (taking any action advances the process by one time step) [2].

The actual process in the MDP, p, is defined probabilistically as [38]:

$$p(s', r|s, a) := \Pr(S_{t+1} = s', R_t = r|S_t = s, A_t = a),$$

for any state s and successive state s' in $\mathcal{S}$, any r in $\mathcal{R}$, and any a in $\mathcal{A}$. This is a discrete probability density function [24] that describes the probability distribution for s', the actual state after s, and r, the actual reward for doing a after observing s. In the context of a finite MDP, it helps to think of p as, more or less, a look-up table, rather than as a distribution with a small number of

parameters, e.g., the normal distribution.

As one might expect, a summation over the entire density function must necessarily equal 1:

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a) = 1,$$

for any s in $\mathcal{S}$ and any a in $\mathcal{A}$ [38]. The interpretation of this is that for any given state, regardless of which action the agent decides to take, some state will be received or "sampled" from $\mathcal{S}$ and some reward will be sampled from $\mathcal{R}$. The other noteworthy part of this probability distribution is that this distribution is contingent only upon the current time step's given state and selected action, not the states or actions from two or more time steps ago. This is what is referred to as the *Markov property* or *Markov assumption*, which is a property that the MDP assumes all states possess [24], hence the "Markov" in "Markov decision process."

On its own, p does not really tell that much about what decision the agent should or will make since it does not assign a value to being in any state. The real use for p is in calculating the cumulative future reward from the current state and onward. Since all future rewards are random variables in the MDP, the cumulative future reward can be defined as an expectation. This expected future cumulative reward can be calculated like so [24, 34]:

$$\mathbb{E}_\pi \left[\left(\sum_{k=t}^{T} R_k \right) | S_t = s \right] = \mathbb{E}_\pi [R_t + R_{t+1} + R_{t+2} + \ldots + R_T | S_t = s],$$

where T is the time step at which the MDP ends [38]. The subscript π indicates that the expected value of being in state s is to be calculated under the

assumption that the *policy* for action selection, called π, is always followed. An MDP policy maps a given state to $|\mathcal{A}|$ probabilities: probabilities for selecting each possible action, $\pi(a|s)$. Thus, this expected value is the expected cumulative reward from current state s and onward, assuming that the agent always picks actions using π.

There is something, however, that this expected value lacks. When the process does not terminate ($T = \infty$), the expected value could also be infinity. To avoid this case, a *discount rate*, γ in the interval [0, 1], can be introduced [38]:

$$\begin{aligned}\mathbb{E}_\pi\left[\left(\sum_{k=t}^{\infty}\gamma^{k-t}R_k\right)|S_t = s\right] &= \mathbb{E}_\pi\left[\gamma^{t-t}R_t + \gamma^{t-t+1}R_{t+1} + \gamma^{t-t+2}R_{t+2} + \ldots|S_t = s\right] \\ &= \mathbb{E}_\pi[R_t + \gamma^1 R_{t+1} + \gamma^2 R_{t+2} + \ldots|S_t = s].\end{aligned}$$

For all γ less than 1 and when R is bounded (in a closed interval), the expected value will be some real number [38]. The discount rate, for γ less than 1, also has the added benefit of preferring an immediate reward over a farther-off reward of equal value, with the extreme of $\gamma = 0$ where only the next reward matters. This is often desirable since the distant future is often less certain than the immediate future in an MDP, and there is usually some time limit on an MDP [2]. The programmer of the agent sets γ, and whether γ is closer to 0 or closer to 1 heavily depends on the task at hand.

This last expected value is referred to as the "value function," $v_\pi(s)$ and incorporates $p(s', r|s, a)$ as follows [38]:

$$
\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi\left[\left(\sum_{k=t}^{T}\gamma^{k-t}R_k\right)|S_t = s\right] \\
&= \mathbb{E}_\pi[R_t] + \mathbb{E}_\pi\left[\left(\sum_{k=t+1}^{T}\gamma^{k-t}R_k\right)|S_t = s\right] \\
&= \sum_a \pi(a|s)\sum_{s'}\sum_r p(s', r|s, a)\left(r + \gamma\mathbb{E}_\pi\left[\left(\sum_{k=t+1}^{T}\gamma^{k-t}R_k\right)|S_{t+1} = s'\right]\right) \\
&= \sum_a \pi(a|s)\sum_{s'}\sum_r p(s', r|s, a)(r + \gamma v_\pi(s')).
\end{aligned}
$$

Note the recursive nature of v_π in the last equation. In English, this last equation is an equation for the reward for being in state s and then taking actions according to policy π. This reward is the sum of the expected values of taking any action in s. Each action has probability $\pi(a|s)$ and a non-probabilisitcally-weighted value that is, for each possible next-most state s' and each possible r for doing a in s, the expected reward for the current possible action a plus the γ-discounted value of proceeding to s' after a.

The other noteworthy thing about this last equation is that it is often referred to as *Bellman's equation* [38]. In various algorithms that are collectively referred to as *model-based learning* algorithms, Bellman's equation can be used to solve for the optimal policy, π^*, which is a policy whose expected cumulative reward is greater than that of any other π. Recall that π is a probability distribution for a given s, so to learn an optimal π^* is to learn, for any given state, which action it most likely should take. Learning π^* is tantamount to learning the optimal state-value function [38], $v_{\pi^*}(s)$, which is the state-value function where $v_{\pi^*}(s)$ is greater than any other $v_\pi(s)$, for any s .

Model-based learning, though, tends to be computationally expensive and can even be unusable when the model, $p(s', r|s, a)$, is not known [2]. Therefore, instead of examining model-based learning, we will examine a *model-free learning* approach called *temporal difference learning*, which does not require any foreknowledge of $p(s', r|s, a)$.

2.3 Temporal Difference Learning

Temporal difference learning leverages the fact that learning π^* is tantamount to learning v_{π^*}. Also, rather than trying to outright solve for v_{π^*}, temporal difference learning methods attempt to estimate v_{π^*}. This involves, for each state s, each subsequent state s', and for each received reward r, continually re-estimating $v_\pi(s)$ by [13]:

$$v_\pi(s) := v_\pi(s) + \eta(r + \gamma v_\pi(s') - v_\pi(s)).$$

For now, this $v_\pi(s)$ can be thought of as a lookup table where the index is the state s and the value at $v_\pi(s)$ is, still, the discounted expected cumulative reward for being in s. Thus, $v_{\pi*}$ is learned by continually re-estimating each value in the lookup table.

Updating these lookup values works as follows. The temporal difference learning agent is taking some actions in some states and receiving some rewards. Before any given r is received, since $v_\pi(s)$ is the expected discounted future reward, $v_\pi(s)$ is a partial estimate of r. Once r is received, however, the actual r is preferred over the estimate of r; the actual r should be incorporated into

$v_\pi(s)$. The addition in the parentheses is a re-estimation of $v_\pi(s)$ that adds r to a re-estimate of the expected discounted future reward from s' and onward (the "$\gamma v_\pi(s') - v_\pi(s)$" part). The η, which is in $(0, 1)$, is used to weigh the relative importance of the re-estimate versus the old estimate. The whole equation, therefore, updates $v_\pi(s)$ partly by using the new estimate and partly by using the old estimate, which is a practice referred to as *bootstrapping* [13].

This update equation, though usable for reinforcement learning, is more so a general statement about what temporal difference learning algorithms are trying to do. Algorithms that follow this bootstrapping format are called temporal difference learning algorithms, though different temporal difference learning algorithms estimate v_{π^*} in slightly different ways. One such algorithm that is relevant to this paper is the Q-learning algorithm.

2.3.1 Q-learning

Q-learning is a temporal difference learning algorithm [15], and thereby a reinforcement learning algorithm, that, instead of bootstrapping $v_\pi(s)$, bootstraps $Q_\pi(s, a)$:

$$Q_\pi(s, a) := Q_\pi(s, a) + \eta(r + \gamma \max_{a'} Q_\pi(s', a') - Q_\pi(s, a)),$$

where "$\max_{a'} Q_\pi(s', a')$" means to compute $Q_\pi(s', a')$ where the next action, a', maximizes $Q_\pi(s', a')$. Like for $v_\pi(s)$, for now, $Q_\pi(s, a)$ can be thought of as a 2 dimensional lookup table where s is a row index and a is a column index. The value of $Q_\pi(s, a)$ for any given s and any given a is the expected discounted

future value for doing that a in that s [40]:

$$Q_\pi(s,a) \equiv \mathbb{E}\left[\left(\sum_{k=t}^{T} \gamma^{k-t} R_k\right) | S_t = s, A_t = a\right].$$

How exactly Q-learning is done can be seen in Algorithm 1.

Algorithm 1 The Q-learning algorithm [2].

```
1: Init. η
2: Init. Q to be an |S| by |A| table where each element is 0
3: for episode from 1 through M do
4:     Observe S_0
5:     Select some A_0 using π and S_0
6:     Do A_0
7:     for t from 1 through T do
8:         Receive R_{t-1}
9:         Observe S_t
10:        Q_π(S_{t-1}, A_{t-1}) := Q_π(S_{t-1}, A_{t-1}) +
             η(R_{t-1} + γ max_{A_t} Q_π(S_t, A_t) − Q_π(S_{t-1}, A_{t-1}))
11:        Select some A_t using π and S_t
12:        Do A_t
```

The first two lines in Algorithm 1 are just initializing important pieces of data. Note that Q_π is initialized to all zeros. This is done as a sort of null hypothesis; relative to each other, all actions are equally valuable. Alternatively, entries in Q could be initialized to some small random number, but that is ultimately up to the programmer of the algorithm. The outer for loop iterates through *episodes*. In the reinforcement learning context, an episode is just a term for a sequence of time steps from $t = 0$ to $t = T$. Almost always, there will be multiple episodes (hence the necessity of the loop) since different sequences of states and actions can be experienced in different episodes.

The next three lines before the inner for loop have the agent observe the

initial state S_0, and then select and do some action A_0 by using S_0 and π. One very commonly used π is what is referred to as an ϵ-*greedy policy*. Under this policy, the programmer sets ϵ to some real number in the interval [0, 1]. Any time an action has to be selected, like on lines 5 and 11 of Algorithm 1, a random number will be drawn from the interval [0, 1]. If this random number is less than ϵ, then a random action will be selected. If this random number is greater than or equal to ϵ, then the a that maximizes $Q(s, a)$ will be selected and executed. Thus, an ϵ-greedy π selects a random action with probability ϵ and selects the best expected action with probability $1 - \epsilon$. Though this is a popular policy, it is not the only policy for action selection, and ϵ-greedy is not actually a part of Q-learning. Therefore, there is no mention of it in Algorithm 1.

The inner for loop begins after the initial state has been observed and the initial action has been executed; it loops through each time step in the given episode. Note that the loop starts at 1 rather than 0. This is because taking action A_0 is assumed to take the remainder of time step 0, thereby advancing the clock from time step 0 to time step 1.

The first thing that is done in any time step after time step 0 is to receive the reward R_{t-1}. Still, the subscript on a reward indicates the time step to which a reward corresponds, not the one in which it is received. Second, having executed the previously selected action from the previous state, the agent observes the current state S_t, the state to which the agent advances after doing A_{t-1} in S_{t-1}.

The information we now have, a 4-tuple of the form (S_{t-1}, A_{t-1}, R_{t-1}, S_t), is sufficient to apply the Q-learning update function. This update is done on Line 10. After making this update, the only steps that remain are selecting an action to take for the current time step and then actually executing that selected

action. These last two steps operate analogously to the initial steps of the given episode operate.

Once the inner loop is done executing, which happens only when T is not infinity, the current episode is over, so the outer loop concludes too. If there are more episodes that the agent still has to go through, i.e., if there is more learning to do, then the above process will be repeated. Otherwise, the final episode must have just been completed, so Q-learning is done.

As the succinctness of Algorithm 1 may suggest, Q-learning is actually one of the more straightforward temporal difference learning algorithms. One manifestation of this is that a proof of Q-learning's mathematical convergence of Q_π to its optimal version, $Q_{\pi*}$, has been around since 1989. The theorem for convergence is given in Theorem 1.

Theorem 1. *[25] Given a finite Markov decision process with finite state space $\mathcal{S}$, finite action space $\mathcal{A}$, and process p, the Q-learning algorithm, given the update rule:*

$$Q_\pi(s,a) := Q_\pi(s,a) + \eta(r + \gamma \max_{a'} Q_\pi(s',a') - Q_\pi(s,a)),$$

converges with probability 1 to the optimal Q-function $Q_{\pi*}$ as long as:

$$\sum_t \eta_t = \infty, \qquad \textit{and} \qquad \sum_t \eta_t^2 < \infty.$$

In other words, since $0 < \eta < 1$, the two summations in Theorem 1 require "that all state-action pairs be visited infinitely often" [25].

2.4 Some Challenges in Reinforcement Learning

Applying temporal difference learning in the context of a Markov decision process is a very powerful way to do reinforcement learning. Contrary to the impression that this chapter may have imparted, however, reinforcement learning is not magic. There are plenty of challenges associated with reinforcement learning. Therefore, to conclude this explanation of reinforcement learning, a few of these issues will be mentioned.

2.4.1 Exploration versus Exploitation

The agent "explores" by selecting random actions, and the agent "exploits" by selecting actions that it currently believes are optimal. By taking random actions, the agent explores the consequences of the wide range of actions available to it. By taking the actions it believes to be optimal, the agent exploits those actions such that repeatedly taking them results in better and better estimates of the rewards for taking those actions. In this sense, exploration is a bit like a breadth-first search while exploitation is a bit like a depth-first search.

As one might expect, there is a tradeoff between exploration and exploitation. If the agent explores too much, then, while it will roughly know the value of many actions in a lot of states, it could spend a long time doing so, i.e., it would be more akin to using brute force. On the other hand, if the agent exploits too much, then, while the agent should have good estimates of the rewards for the sequences of actions it has exploited, it could very well be missing out on a better sequence of actions. In other words, it could be

exploiting a path that is suboptimal.

The exploration and exploitation tradeoff is typically managed by some programmer-defined parameter. One such parameter is ϵ, which was mentioned in the previous section. Selecting the correct constant ϵ, however, can be difficult. Instead, oftentimes, ϵ decays over the course of learning. A decaying ϵ would be start at 1 and then decay by a very small amount each time step until it asymptotically approaches 0.1, 0.5, or 0, for instance. This way, the agent could explore first and exploit later. Note, this does not completely eliminate the tradeoff issue, though it does alleviate it somewhat.

2.4.2 Picking a Good Discount Factor

Picking an appropriate discount factor γ is important since γ is more or less the degree to which short term rewards are favored over long term rewards. Selecting an appropriate γ can be tricky. If γ is set too close to 0, then the agent will be too short-sighted to accomplish its goal. If γ is set too close to 1, then, especially when the underlying MDP is highly non-deterministic, the agent could be too far-sighted to reliably select the best action for the current time step. Setting γ too close to 1 seems to be less of an issue than setting γ too close to 0, but either way, if γ is poorly chosen, then the agent may not able to successfully learn.

2.4.3 The Markov Property and Partial Observability

The Markov decision process assumes that the Markov property always holds, i.e., optimal action selection requires only the most recent state. For

many real-world applications where some amount of short term memory is necessary, possibly such as in driving [15], or where the agent's observations do not provide enough information, this may not be a valid assumption. When the "sensations received by the agent are only partial glimpses of the underlying system state" [15], the Markov decision process is said to be a *partially observable Markov decision process*. Throughout this chapter, we have implicitly assumed full observability, i.e., we have worked with a *fully observable Markov decision process*.

"The most recent state" can be relaxed somewhat. In some situations, the most recent state can be defined to be information from the n previous time steps, rather than just the most recent time step. A state defined as such would still technically satisfy the Markov property. However, it would be unclear how many previous time steps' information should be included, and it would also be unclear if that number should be constant. Furthermore, even in the best-case, doing so effectively increases the number of states, which can increase the learning period.

2.4.4 Having Many Possible States

Whenever the number of states is very large or the states themselves are high dimensional data structures, reinforcement learning algorithms that use lookup tables can quickly become intractable in terms of both time and space. In Q-learning for instance, the Q lookup table is $|\mathcal{S}| \times |\mathcal{A}|$, so the Q lookup table, when there are a lot of states, would be large and unwieldy.

This last issue turns out to be rather significant and rather common, so it is

revisited and partially "solved" in Chapter 4, the chapter on *deep reinforcement learning*. An explanation of deep reinforcement learning, however, requires an understanding of not just reinforcement learning, but also deep learning. Therefore, the next chapter covers just that: deep learning with deep neural networks.

Chapter 3

Deep Learning with Deep Neural Networks

From a certain level of abstraction, the brain is just a system of interconnected brain cells called *neurons*. In a simplistic sense, the network of neurons revolves around one neuron sending an electrical and mechanical signal to another connected neuron [14]. When the receiver neuron receives the signal, it will either "activate" and forward that signal to other neurons or it will not activate and not forward that signal elsewhere. Whether a neuron activates or does not activate (and the degree to which it activates) are thought to be influenced by the strength of the connection, the strength of the sent signal, and the internal chemistry of the neuron. Individually, these low level pieces are fairly well understood, yet, how the network of neurons gives rise to general cognition is, by and large, a mystery.

Nevertheless, this low-level understanding was enough to inspire Warren McCullock and Walter Pitts in 1943 and Frank Rosenblatt in 1957 to formulate

the artificial neurons respectively called the *McCullock-Pitts artificial neuron* and the *Rosenblatt perceptron*, which are crude mathematical approximations of a biological neuron [33]. The perceptron became the foundation for the first artificial neural network called the *multi-layer perceptron*, which also is a crude mathematical approximation of its biological counterpart: the biological neural network. The multi-layer perceptron has many extensions and offshoots, but ultimately, from the multi-layer perceptron, we arrive at today's *deep neural networks* which allow for *deep learning*. Thus, to explain deep learning, we begin with an examination of the Rosenblatt perceptron.

3.1 The Rosenblatt Perceptron

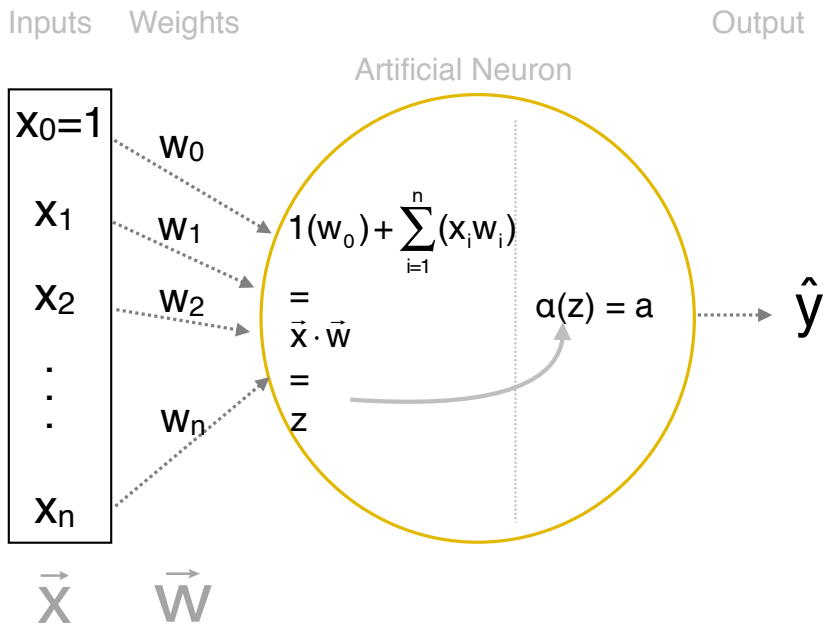

Figure 3.1: The Rosenblatt perceptron [33]

To understand the math depicted in the Rosenblatt perceptron in Figure 3.1,

the feed-forward process will be examined first.

3.1.1 The Feed-Forward Process

The feed-forward process is the name for the process by which an artificial neuron takes an input and produces an output.

The input to an artificial neuron is an $n+1$ length row vector, $\vec{x} = [x_0 = 1, x_1, x_2, \ldots, x_n]$, where each of the $n+1$ components of $\vec{x}$ is a real number. In Figure 3.1, $\vec{x}$ looks like a column vector, but by convention and to be able to build off of the math presented in this section, $\vec{x}$ will be referred to as a row vector. Components x_1 through x_n will come from some row in a data set that has (at least) n columns. The first component of $\vec{x}$, x_0, does not come from the data set; rather, it is always 1. This is because x_0 functions as the "bias" input, or in linear regression terms, the y-intercept.

The first step in the feed-forward process is to compute the dot product of the input vector by the weight vector: $\vec{x} \cdot \vec{w}$, where the weight vector contains $n+1$ elements: $\vec{w} = [w_0, w_1, w_2, \ldots, w_n]$. Each element of $\vec{w}$ is a real number and corresponds to one of the elements from the inputs vector. The dot product of two vectors that each contain only real numbers will produce a real number, which in this case is called the *net-input*, z: $\vec{x} \cdot \vec{w} = z$.

Referring back to the initial explanation of biological neural networks, the weights represent the strength of the connection to the neuron while the inputs represent the signals coming from other (un-pictured) neurons. Mathematically, the weights in $\vec{w}$ are somewhat analogous to the coefficients in a linear regression model, which are learned from a data set. An algorithm for learning $\vec{w}$ from a

data set will be presented later; for now, it will be assumed $\vec{w}$ is simply given.

The second step of the feed-forward process is to take z and feed it into an *activation function*, α, to get the *activation value*, $\alpha(z)$. The activation function is analogous to the internal dynamics of the biological neuron that, for a given input, determine whether or not the neuron will activate. There are many different activation functions in use in today's neural networks, but for the Rosenblatt perceptron, the one utilized was the Heaviside unit step function [33]:

$$\alpha(s) = \begin{cases} 1 & z \geq 0 \\ -1 & z < 0. \end{cases}$$

The third step is to forward the activation value $\alpha(z)$, whether it is 1 or -1. In Figure 3.1, since there are no successive neurons like there typically would be in the brain, outputting the activation value concludes the feed-forward process. This is to say, $\hat{y}$ is $\alpha(z)$.

The purpose of the feed-forward process, as the Heaviside activation function sort of indicates, is for any given input vector, to produce an output of -1 or 1. The outputs may seem arbitrary, but they mean something. Almost always, the 1 corresponds to some class or category in the data set, call it "class A," while -1 corresponds to the complement of class A. The complement of class A can consist of a single other class (class B), or it can consist of multiple other classes (classes B, C, D, and so on). Thus, the purpose of the artificial neuron's feed-forward process is, for any given $\vec{x}$ from the data set, to assign some class to it.

This assignment, however, is really only a prediction of the class to which $\vec{x}$

belongs, hence $\hat{y}$ rather than y in Figure 3.1. Making good predictions is not something that the perceptron can do automatically; rather, it has to learn to do so. How the perceptron learns to do so is via its learning rule: *the perceptron learning rule*.

3.1.2 The Perceptron Learning Rule

Whether the Rosenblatt perceptron outputs a 1 or a -1 depends on the values of $\vec{x}$, the weights in $\vec{w}$, and the activation function α. Since $\vec{x}$ is exogenous to the artificial neuron and we have defined α to be the Heaviside unit step function, all that the neuron can alter are the weights in $\vec{w}$. Therefore, the perceptron learning rule is the rule by which the perceptron learns a $\vec{w}$ that correctly maps some $\vec{x}$ to its actual class, y.

Learning $\vec{w}$ requires observing, for all $\vec{x}$ in some data set, which $\vec{x}$'s map to 1 and which $\vec{x}$'s map to -1. The idea is that by observing and learning how the historical inputs are related to their respective historical outputs, the artificial neuron will learn some $\vec{w}$ to correctly map any $\vec{x}$ to its correct y. Practically speaking, the perceptron would (ideally) be capable of classifying an $\vec{x}$ that does not have a known y, e.g., predicting a future outcome.

The perceptron learning rule begins by setting the weights to some small random numbers. Then, for each $\vec{x}$ from the given data set, the artificial neuron will output which class it believes $\vec{x}$ falls into, called $\hat{y}$. This value, $\hat{y}$, is produced by the feed-forward process. The actual class into which $\vec{x}$ falls, called y, must be known for learning to work. This is because the learning algorithm compares $\hat{y}$ to y: $\hat{y}$ is y or $\hat{y}$ is not y, i.e., the perceptron's guess was correct or incorrect.

In the case that the prediction was correct ($\hat{y}$ is y) no adjustment to the weights need be made. However, if the prediction is incorrect ($\hat{y}$ is not y), then the weights should be updated so as to not make the same incorrect guess again.

The approach that the perceptron learning rule takes is to update each weight in $\vec{w}$ individually. The ith weight in $\vec{w}$, w_i, is updated by [34]:

$$w_i := w_i + \eta(y - \hat{y})x_i,$$

where x_i is the ith element of $\vec{x}$ and η is a real number in (0, 1]. The ingenuity of this update equation lies in the fact that it updates the weight not only by looking at the correctness of the prediction, but also by examining this weight's input, x_i: "the weight update is proportional to the value of x_i" [33]. As for η, this variable is the roughly the same as η from the reinforcement learning chapter, Chapter 2. It is the learning rate, which in this context works to limit the degree to which any given error can affect the weights of the network.

This explanation of the feed-forward process and the perceptron learning rule is pretty much all that there is to the Rosenblatt perceptron. By itself, the Rosenblatt perceptron is not much different from or powerful than logistic regression. The Rosenblatt perceptron really becomes interesting and powerful when it is used in conjunction with other Rosenblatt perceptrons.

3.2 The Multi-layer Perceptron

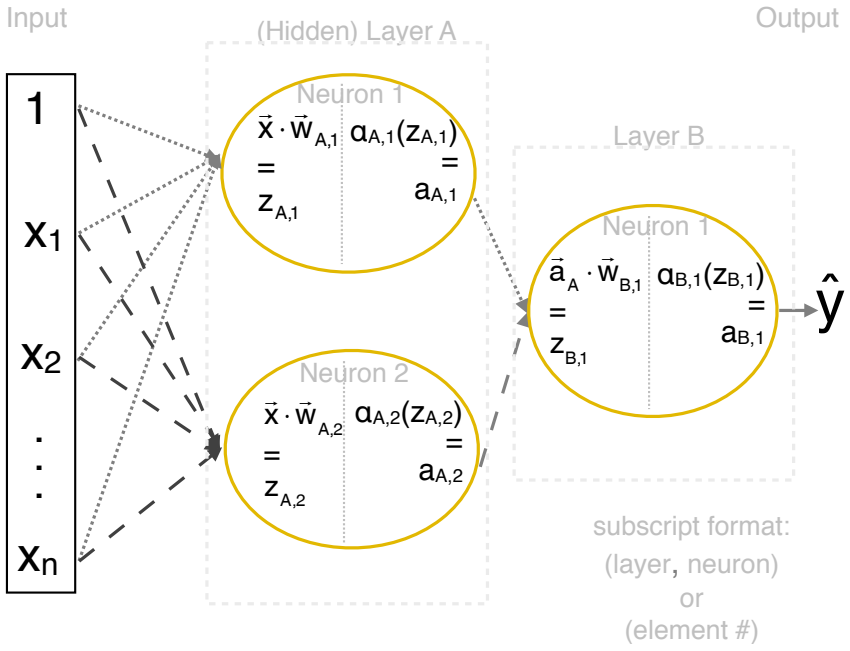

Figure 3.2: A multi-layer perceptron with one hidden layer [33]

Figure 3.2 is a multi-layer perceptron, which is essentially a bunch of Figure 3.1's with more arrows. As is true of the Rosenblatt perceptron, the multi-layer perceptron has a feed-forward process and learning process. The feed-forward process is also used in the learning process, so we begin there.

3.2.1 Making Predictions via the Feedforward Process

The multi-layer perceptron's feed-forward process begins by feeding the input vector, $\vec{x}$, into the first *layer* of the network (layer A), where a *layer* is a set of artificial neurons that receive the same input vector, which in this case is $\vec{x}$. Layer A, at least in the case of Figure 3.2, is a special kind of layer called a

hidden layer, which is defined as a layer whose output is never directly observed (hence the "hidden" qualifier).

The linear algebra that occurs in Layer A of a multi-layer perceptron, from the perspective an artificial neuron, is unchanged from the Rosenblatt perceptron discussion:

$$\vec{x} \cdot \vec{w}_{A,j} = z_{A,j},$$

where the subscript "A, j" means: "layer A, jth neuron in the layer." Since there are two artificial neurons, the first step in the feed-forward process for Figure 3.2 is to compute $z_{A,1}$ and $z_{A,2}$:

$$\vec{x} \cdot \vec{w}_{A,1} = z_{A,1},$$

$$\vec{x} \cdot \vec{w}_{A,2} = z_{A,2}.$$

This generalizes to: compute as many dot products as there are neurons in the given layer.

Typically, though, when discussing the linear algebra operations involved in a multi-layer perceptron layer, one matrix, rather than a bunch of individual weight vectors, is used. This is simply because *way* more than 2 neurons are usually used in a layer; it is not uncommon to use a few hundred to a few thousand artificial neurons in a given layer. Therefore, the computation of the activation values of Layer A (and all other layers from here on out) will use a weight matrix like so:

$$\vec{x}W_A = \vec{z}_A,$$

where $\vec{z}_A = [z_{A,1}, z_{A,2}]$. Here, W_A is a $j \times (n + 1)$ matrix, where j is the number

of neurons in the immediate layer and $n+1$ is the length of $\vec{x}$:

$$W_A = \begin{bmatrix} \vec{w}_{A,1} \\ \hline \vec{w}_{A,2} \\ \hline \vdots \\ \hline \vec{w}_{A,j} \end{bmatrix}.$$

Note that since $\vec{x}$ is $1 \times (n+1)$ and W_A^T is $(n+1) \times j$. Applying this all to Figure 3.2:

$$\vec{x}W_A^T = \vec{x} \begin{bmatrix} \vec{w}_{A,1} \\ \hline \vec{w}_{A,2} \end{bmatrix}^T = \vec{x} \begin{bmatrix} w_{A,1,0}, w_{A,1,1}, \ldots, w_{A,1,n} \\ w_{A,2,0}, w_{A,2,1}, \ldots, w_{A,2,n} \end{bmatrix}^T = \vec{z}_A.$$

The second step is to compute the activation value at each artificial neuron in Layer A: $\alpha(\vec{z_A}) = \vec{a_A}$, where α is applied to each component of z_A to produce $\vec{a_A}$. In the Rosenblatt perceptron, α was the Heaviside step function; however, in most multi-layer perceptrons, some other activation function is used. The reasons for this will be provide later, so for the purposes of this explanation, we will just state that α is the sigmoid (logistic) function [33]:

$$\alpha(z) = \frac{1}{1+e^{-z}},$$

which can be seen in Figure 3.3. Note the limits of the sigmoid function as z goes to positive or negative infinity:

$$\lim_{z \to +\infty} \frac{1}{1+e^{-z}} = 1,$$

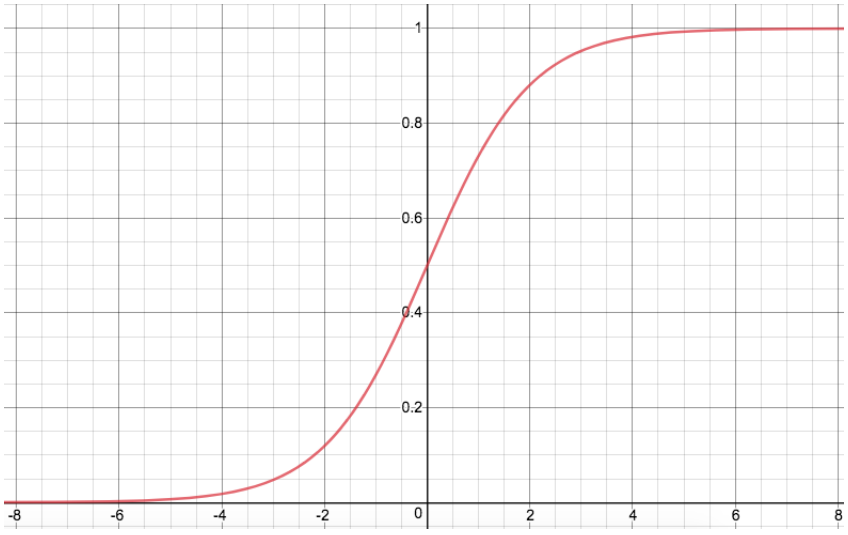

Figure 3.3: The sigmoid (logistic) function that is sometimes used as the activation function in multi-layer perceptrons [8]

and

$$\lim_{z \to -\infty} \frac{1}{1 + e^{-z}} = 0.$$

Therefore, not only is the output value are no longer -1 or 1, but the output value is now any real number in (0, 1). This indicates that a multi-layer perceptron with a sigmoid activation function can be used in both classification and regression tasks.

The third step in this layer is to forward the activation values of Layer A. For the Rosenblatt perceptron, forwarding the activation value (a real number) means outputting that activation value as the prediction: $\hat{y}$. For a hidden layer in a multi-layer perceptron, forwarding the activation values (a vector of real numbers) means feeding those activation values into the next layer. The next layer can either be an output layer, which is a layer whose activation value(s) will be the output of the whole network of artificial neurons, or another hidden

layer. In Figure 3.2's case, the next layer is an output layer.

The math involved in either case is a repeat of the first, second, and third steps, but with $\vec{a}_A$ in place of $\vec{x}$, and B in place of A. For the sake of clarity, for Figure 3.2, this would be:

$$z_B = \vec{a}_A W_B^T,$$

and

$$a_B = \alpha(z_B),$$

where $W_B = \left[w_{B,1,0}, w_{B,1,1}, \ldots, w_{B,1,n}\right]$. Back at the third step, we forward the activation value, which is just a real number since there is only 1 neuron in layer B. Layer B in Figure 3.2 is not another hidden layer; rather, it is an output layer, meaning a_B is $\hat{y}$, the output of the whole multi-layer perceptron. Thus, the whole feed-forward process with α as the sigmoid function from Figure 3.2 can be summarized in a single line:

$$\hat{y} = \alpha(\alpha(\vec{x}\ W_A^T)\ W_B^T).$$

3.2.2 Learning via Backpropagation

By far, the most popular way a multi-layer perceptron learns its weight matrices is via a type of *gradient descent* called *backpropagation* [34]. A *gradient* is a vector-function $\nabla f(x_1, x_2, \ldots, x_v)$ where each component of $\nabla f(x_1, x_2, \ldots, x_v)$ is the partial derivative of f with respect to one its v variables [11]:

$$\nabla f(x_1, x_2, \ldots, x_v) = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_v}\right].$$

For instance, the gradient of $f(x, y) = z = x^2 + y^2$, denoted $\nabla f(x, y)$, is: $\left[\frac{\partial f}{\partial x} = 2x, \frac{\partial f}{\partial y} = 2y\right]$.

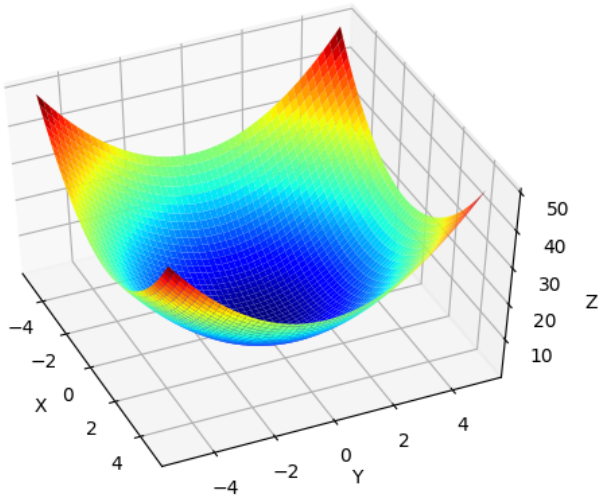

Figure 3.4: A graph of the function $f(x, y) = x^2 + y^2$

Gradients have the interesting property that at any coordinates in the function's domain, they point in the direction of greatest increase for the given function [11]. Applying this to the $f(x, y)$ function, suppose we are at $(x = 1, y = 1)$. To increase f the most, we want to travel in the direction of vector: $\nabla f(1, 1) = [2, 2]$. Visually, this makes sense since at $(1, 1)$, the direction of greatest increase is towards the nearest red corner in Figure 3.4.

The red corners in Figure 3.4, however, are not the actual ends of the function. They are just where the illustration ends; the actual function, f, goes up forever in the positive z direction. Therefore, though $[2, 2]$ is in the direction of greatest increase local to $(1, 1)$, moving along $[2, 2]$ from $(1, 1)$ will definitely

not produce a point that is the global maximum of the function, which does not exist. Instead, it is just a "step," where a step is just a movement from one point to another, from $(1, 1)$ to: $[1, 1] + [2, 2] = (3, 3)$.

Nevertheless, repeated steps can be made in the direction that the gradient prescribes to increase f the most, i.e., to step up the steepest part of the local hill. Making such repeated steps is referred to as *gradient ascent*. To perform gradient descent, steps are still made, but instead, they are steps in the opposite direction. In the case of our function f at $(1, 1)$, a gradient descent step would move from $(1, 1)$ in the direction of the vector $-\nabla f(1, 1) = [-2, -2]$. Again, visually, this makes sense since it would be stepping towards the dark blue center, which is the lowest local point.

Note, however, that this gradient descent step at $(1, 1)$ will overstep the local (and global) minimum since: $[1, 1] + [-2, -2] = [-1, -1]$. To compensate for this propensity to overstep, oftentimes, the gradient will be multiplied by some scalar in (0, 1). This gets at the big idea for gradient descent: repeated small steps in the direction of greatest decrease can be used to minimize a function.

In the case of a neural network, the function to minimize via gradient descent is the loss function, denoted as $\mathcal{L}$, which is the function that measures the error of the prediction(s). Multi-layer perceptrons can use many different loss functions like the mean absolute error, categorical cross-entropy, or mean squared error. For the purposes of this explanation, however, we will assume that $\mathcal{L} = \frac{1}{2}(\hat{y} - y)^2$.

Practically speaking, to understand what it means to "minimize the loss function via gradient descent," it must first be stated that the objective is not to find the minimum of the function. We do not need calculus to know that the

minimum of $\mathcal{L}$ is 0. We do, however, "need" calculus to find the *parameters* of the loss function that produce that minimum. To find those parameters of the loss function is to minimize the loss function.

At face value, there do not seem to be any parameters in $\mathcal{L}$. The parameters, however, are hiding behind the $\hat{y}$; they are the weights in the weight matrices. Recall that the inputs $\vec{x}$, the activation function α, and the target y are all exogenous. The weights of the neural network, however, are not, and they also affect the value $\hat{y}$. This makes them "parameters" that can be found via gradient descent.

Gradient descent finds the weights of the neural network iteratively, via the update equation [32]:

$$W_l := W_l - \eta \frac{\partial \mathcal{L}}{\partial W_l},$$

where l represents the layer to which the given weights belong and η is the learning rate in $(0, 1]$. Note that this is more or less the equation that was used to compute the destination coordinates when traveling form the initial coordinates of $(1, 1)$ in the opposite direction of the gradient. This update equation is deceptively simple. Though it can be written on one line, the math involved is by no means trivial. After all, taking the partial derivative of the loss function with respect to a weight matrix sounds completely nonsensical at first (at least, to the author).

This is where backpropagation comes into play. Backpropagation repeatedly applies the chain rule to simplify $\frac{\partial \mathcal{L}}{\partial W_l}$ into a series of linear algebra operations. An explanation of exactly how backpropagation applies the chain rule involves matrix calculus and/or some confusing subscripts. Therefore, instead of walking

through the application of the chain rule, the important results will just be presented with the understanding that the chain rule was used to arrive at them.

First, for any weight matrix W_l, the partial derivative of the loss function $\mathcal{L}$ with respect to W_l is [32]:

$$\frac{\partial \mathcal{L}}{\partial W_l} = \delta_l \vec{a}_{l-1},$$

where l is a layer's identifier. For the purposes of generalizability, this equation renames $\vec{x}$ to $\vec{a}_0$. Also, given that letters have been used to identify layers, $l-1$ will be taken to mean the previous layer before this layer, e.g., $B-1=A$.

As for δ_l, there are two cases to consider. First, when l is the output layer, δ_l is [32]:

$$\delta_{output} = (\hat{y} - y)\alpha'(\vec{a}_{\text{output}-1} W_{\text{output}}^T).$$

Second, when l is a hidden layer, δ_l is [32]:

$$\delta_l = W_l \delta_{l+1} \cdot \alpha'(\vec{a}_{l-1} W_l^T)^T.$$

This all means that the update equations are really either

$$W_{\text{output}} := W_{\text{output}} - \eta \left((\hat{y} - y)\alpha'(\vec{a}_{\text{output}-1} W_{\text{output}}^T) \right) \vec{a}_{\text{output}-1},$$

or

$$W_l := W_l - \eta \left(W_l \delta_{l+1} \circ \alpha'(\vec{a}_{l-1} W_l^T)^T \right) \vec{a}_{l-1},$$

where "$\circ$" indicates component-wise multiplication.

These equations somewhat explain why the algorithm is called "backpropagation" in the first place: It propagates the loss backwards (from the

outputs to the inputs) in order to update the weights of the neural network [33]. Furthermore, updating one layer's weights takes into account updates that have already been made to other layers towards the back of the network. This follows from the δ_{l+1} term being used in the computation of δ_l.

What these equations do not explain is why they are in a generalized form (for any l). After all, Figure 3.2 only has one hidden layer, so the update equations would just be:

$$W_B := W_B - \eta \left((\hat{y} - y)\alpha'(\vec{a}_A W_B^T) \right) \vec{a}_A,$$

and

$$W_A := W_A - \eta \left(W_A \delta_B \circ \alpha'(\vec{a}_0 W_A^T)^T \right) \vec{a}_0.$$

Really, the utility of having a generalized form is that the backpropagation algorithm can now be applied to neural networks with more than one hidden layer.

3.3 Deep Learning with Deep Neural Networks

A neural network that has more than one hidden layer is called a *deep neural network* [33]. The *training*, which is a term for the process by which a neural network learns its weights, of a deep neural network is referred to as *deep learning*. In many contexts, having multiple hidden layers turns out to be very useful and powerful, so it is worth conceptualizing why having multiple neurons, and thereafter, multiple hidden layers, is useful at all.

3.3.1 Multiple Nodes and Multiple Layers

Having multiple artificial neurons in a hidden layer is almost always preferable to having a single neuron (the Rosenblatt perceptron) because of the Universal Approximation Theorem (Theorem 2).

Theorem 2 (Universal Approximation Theorem)**.** *[9]*

Let $X \subseteq \mathbb{R}^m$ be compact, and let α be an arbitrary activation function. Denote the space of continuous functions on X by $C(X)$. Then, for any function f in $C(X)$ and for any $\epsilon > 0$, there exists some natural number n where for:

$$f_n(x_1, x_2, \ldots, x_m) = \sum_{q=1}^{n} w_q \alpha \left(\sum_{r=1}^{m} a_{q,r} x_r \right),$$

f_n is able to uniformly approximate a function f within an arbitrary margin ϵ:

$$|f(X) - f_n(X)| < \epsilon.$$

A neural network is actually a *function approximator*, which means that, for a data set with some input space X and some output space $f(X)$, the neural network is actually an estimate of the function f that perfectly maps (achieves a loss of 0) all inputs to their respective outputs. The theorem places restrictions, though, on X and f. For X, it must be a *compact subset* of $\mathbb{R}^m$. For a simple description of "compact," the Heine-Borel Theorem (Theorem 3) can be applied.

Theorem 3 (Heine-Borel Theorem)**.** *[7] Let $A \subseteq \mathbb{R}^m$. A is compact if and only if A is closed and bounded.*

As for f, it must be continuous on X. Therefore, the Universal Approximation Theorem says [9]: A neural network with n artificial neurons and

just one hidden layer can approximate any continuous function on a compact input space to any arbitrary degree of accuracy, ϵ.

The Universal Approximation Theorem is powerful, but not as powerful as it sounds. For one, there is no upper bound placed on n. This allows for the number of neurons required to approximate some f to a desired degree of accuracy to be so large that such a neural network would be computationally intractable [9]. Second, the theorem requires that f be a continuous function, which is not a condition that can always be met.

Deep neural networks are able to somewhat address the first and second issues via their additional hidden layers. Typically, having more layers is more efficient than having more neurons. One example of this was when multi-layer perceptrons were trained to approximate n-degree *sparse polynomials*, which are polynomials with few non-zero coefficients, for any n input variables. In this specific case at least, the authors showed that a multi-layer perceptron with k hidden layers would required $2^{\sqrt[k]{n}}$ total artificial neurons [39], i.e., approximating a sparse polynomial with a single hidden layer ($k = 1$) requires more neurons than with any deep neural network. It should be mentioned that neural networks are not regularly used to compute sparse polynomials, so this formula, $2^{\sqrt[k]{n}}$, probably does not generalize well. Nevertheless, the formula illustrates the hypothesized efficiency gains that usually accompany additional hidden layers.

A precise mathematical explanation as to why adding more layers improves performance and efficiency has yet to be discovered. The prevailing hypothesis, though, is that having more layers provides the neural network with a greater capacity to approximate functions that are more hierarchical (not continuous) in nature [3]. In this sense, each layer detects some set of patterns at one level of

the hierarchy that are important in producing the correct outputs. Then, when those detected patterns are forwarded to the next layer, the next layer will not have to re-detect those patterns. Instead, it can take what the last layer detected and draw higher-level conclusions. Then, this layer forwards what it learned from what its prior layer learned, which forwards that to the next hidden layer, and so on.

The typical examples of hierarchical pattern building through repeated hidden layers are focused on facial recognition. Suppose that a deep neural network is being used to classify a person as male or female, solely by looking at a photo of the person. The first hidden layer could, say, identify edges and curves, the second hidden layer could identify individual facial features (noses, eyes, mouths, etc.) by considering the extracted edges and curves, the third hidden layer could identify how the facial features are related by considering those individual facial features (eyes are close together, the nose is crooked, etc.), and a fourth hidden layer could identify the gender of the face based on those relations between facial features from the third hidden layer. Given that this is just a made-up example, the "division-of-labor" for hierarchical pattern recognition need not be so clean or quick; more layers could be needed.

Now, there is, more or less, a limit on how many hidden layers there can be in a neural network. As one might expect, adding more hidden layers will not always help since a given hierarchy could already be fully captured by the existing hidden layers. Applying this point to our facial recognition example, faces, presumably, can really only be decomposed into a hierarchy of some unknown but finite depth. In addition to this, there is what is referred to as the *vanishing gradient problem*. For the sake of organization, an explanation of this

problem is withheld until Section 3.5.1.

Instead, a digression will be made to examine special kinds of neural networks that are typically deep neural networks: *convolutional neural networks* and *recurrent neural networks*. Each of these is used in this paper's self-driving car software as detailed in Chapter 5, so they are worth mentioning.

3.4 Some Other Neural Networks

3.4.1 Convolutional Neural Networks

A *convolutional neural network* uses *convolutional layers* to consider spatial information inherent in some input vector or, more commonly, some input matrix. Most often, the input is an image, so we will speak in terms of images, with the understanding that the same concepts generalize to other input vectors and matrices with information spatially encoded.

Convolutional layers operate by sliding small, fixed-sized *windows* across the image in some small, fixed-sized strides until that window has seen every pixel in the image. A window in this case, is a matrix with real number "weights," which can be learned via backpropagation, and the number of windows in a layer is the number of neurons in that layer. What is being learned, conceptually speaking, is the set of weights that correspond to some useful pattern, some feature. Each time the window moves, a *convolution operation* is performed [33]. For convolutional neural networks, the convolution operation is a component-wise multiplication of the weights of the window with the pixel values over which the window is currently hovering, followed by the summation of the result of that

multiplication [37]. Therefore, a single convolution produces a real number, which can be seen in Figure 3.5. All of the convolutions for a given neuron produce a new "image," called a *feature map*, with the patterns represented by the neuron's window identified across the "image." Thus, with n neurons, each with their own window, the end-product of a convolutional layer is a "convolved" image with a depth of n, which can be seen in Figure 3.6.

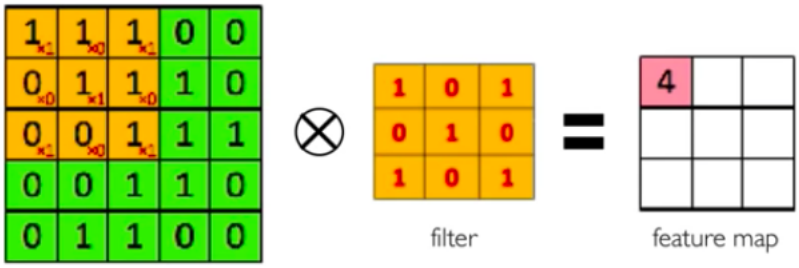

Figure 3.5: The first convolution operation performed by a neuron in the convolutional layer [37]

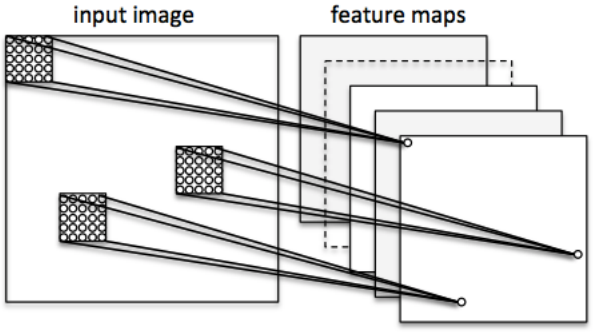

Figure 3.6: The result of all of the convolutions across the whole input image [33]

What is missing from Figure 3.5 and Figure 3.6 is that each real number in the result of the convolutions is the net-input, z, to the activation function for that convolutional layer. Convolutional neural networks almost exclusively use

the *rectified linear unit* (ReLU) as the activation function, which is defined as [37]:

$$\text{ReLU}(z) = \max(z, 0).$$

Usually, multiple convolution layers are stacked in a row, so the output of one convolution layer can be fed into the next convolution layer. The same principles and operations from above are applied. When the final convolutional layer outputs its feature maps, a reshaping step that puts all feature map into one dimension usually occurs. A reshaping step, which in this case is referred to as "flattening," usually occurs because a multi-layer perceptron usually receives those feature maps, and a multi-layer perceptron can only process 1-dimensional data. The back-end multi-layer perceptron works just like a regular multi-layer perceptron and can be used to make predictions about the inputs [33].

This might beg the question: If the feature maps have to be flattened in the end and a multi-layer perceptron is still used, then why not just feed them into a multi-layer perceptron at the start? In addition to identifying spatial relationships in the input data, the purpose of a convolutional layer is to, more or less, reduce the size of the inputs. Roughly, by reducing the input size and simultaneously highlighting the important features, convolutional layers tend to make the multi-layer perceptron's job much easier. For deep convolutional neural networks (essentially all convolutional neural networks are deep), this is especially true given that they effectively hold a monopoly on the image-processing neural network "market." This includes use cases like detecting objects in an image, identifying objects in an image, and segmenting an image into the objects thereof.

3.4.2 Recurrent Neural Networks

Convolutional neural networks are to spatial data as *recurrent neural networks* are to temporal data. These handle temporal data primarily by "remembering" what they have just seen, i.e., they have "short term memory." As such, they are well suited to make predictions on any kind of data where having knowledge of what happened previously increases predictive power. This includes tasks like language translation, speech recognition, and weather forecasting.

There are many different recurrent neural networks such as the gated recurrent unit, the long short-term memory network, and the neural Turing machine, but the one that we will examine is simple, literally: the *simple recurrent neural network*. A simple recurrent neural network is a multi-layer perceptron, but the activation values of at least one of the hidden layers from the previous feed-forward process are given as an endogenous inputs (for that selfsame hidden layer) alongside the exogenous inputs from the next feed-forward process. Figure 3.7 illustrates this.

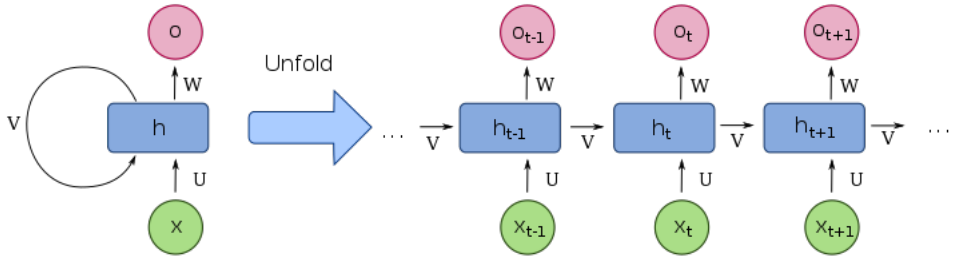

Figure 3.7: An simple recurrent network with one hidden layer [10]

The other thing that Figure 3.7 illustrates is the concept of *unfolding*

through time. Unfolding a simple recurrent neural network is, essentially, performing a trace of how the network produces an input, sequentially. Graphically, the purpose of showing the unfolding process is to show that a simple recurrent neural network's memory is very short. At worst, it is one time step long, and at best, it is a few time steps long, with the memory getting further masked the farther back in time one goes.

3.5 Challenges

Just as reinforcement learning has its list of challenges, so too does deep learning. The challenges of deep learning are not quite as substantial, but they are still worth mentioning.

3.5.1 Vanishing Gradients

A serious issue that constrains the depth of a deep neural network is the *vanishing gradient* problem. In the vanishing gradient problem, the partial derivatives in the update equation "vanish" to 0. If the partial derivative goes to 0 (near 0, really), then that means that the weights at the earliest layers will be very slowly updated. Overall, this can substantially slow learning.

The culpable party here is usually the activation functions. The derivative of some activation functions such as the sigmoid activation function are small and approach 0 rather quickly as one deviates from 0, as Figure 3.8 indicates. This implies that, since each δ_l is multiplied by the derivative of the activation function, repeatedly multiplying small derivatives can quickly produce a near-zero δ_l. The problem is especially bad for recurrent neural networks

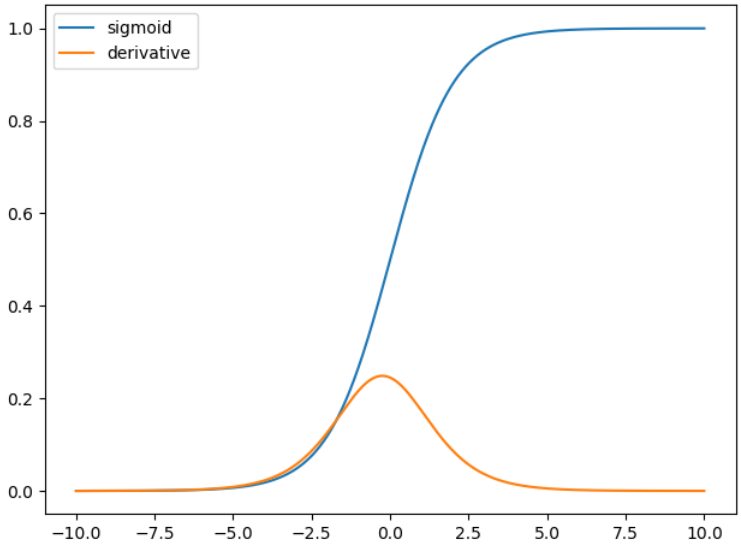

Figure 3.8: The sigmoid activation function and its derivative [18]

because of the additional time component [33]. Additionally, as the backpropagation weight update equation suggests, the vanishing gradient problem would be compounded further if the learning rate is too low.

3.5.2 Local Minima

For some function and at some point in that function's domain, the step that a gradient descent algorithm takes is a step in the direction of greatest decrease that is local to that point. This is to say, the steps are in the direction of a local minimum; they are not necessarily steps in the direction of a global minimum. This opens the door to the possibility that the neural network's weights could be updated such that they get stuck in a local minimum. Getting stuck in a local minimum can happen at any learning rate, though, it is much more likely to occur when the learning rate is "too low," which is a problem-specific threshold.

Fortunately, this is not as big of an issue as it was previously. Various

improvements have been made to the gradient descent update equation that can mostly prevent the weights from getting stuck in a local minimum. One improvement is to have the learning rate decay over the course training [20]. This way, the learning rate can start out high, overstep the local minima, and take smaller steps once it gets close to a global minimum. Another improvement is to incorporate a *momentum* factor into the update equation [20]. This momentum works just like the momentum that we experience in our everyday lives. As such, if the weights are about to get stuck in a local minimum, the momentum from previous steps could carry the weights out of that local minimum. These are two relatively basic improvements, but they do generally, albeit not always, result in more frequent convergence of the weights to the global minimum (or a better local minimum) of the loss function.

3.5.3 Hyperparameters

The *hyperparameters* are the parameters of the neural network that the programmer, rather than backpropagation, sets. Hyperparameters would be things like: the activation functions in each layer, the number of neurons in each layer, the number of hidden layers, the loss function, the learning rate, and so on. Picking good hyperparameters is more of an art than a science; it often requires expertise to get them right quickly. Expertise, though, is oftentimes still not enough, so a *grid search* of the hyperparameters is often performed. A grid search is essentially a brute-force, "try every possible desirable combination of hyperparameters" approach to finding good hyperparameters. As with most brute force algorithms, this can take a lot of time, especially if the network is

really deep and/or if the data set on which the network is learning happens to be very large.

3.5.4 Number of Weights

The total number of weights in the neural network is a function of the input size, the number neurons in each layer, and the number of layers in the network. Accordingly, wider networks (more neurons per layer) and deeper networks will often have more weights to learn. Having more weights to learn means that the neural network has to spend more time training.

Additionally, when the network is sufficiently large, the feed-forward time can become an issue as well. This is pertinent to real-time systems that rely on deep neural networks, which in the case of a self-driving car, could include a deep convolutional neural network for object detection.

3.5.5 A Black Box

A *black box* is a machine for which one knows the inputs and the outputs but not the inner mechanisms that transformed those inputs into those outputs. Virtually all practical neural networks are black boxes. Now, this is not to say that neural networks are un-implementable; rather, it is saying that there is no known way to produce a human-understandable explanation as to how a neural network makes its decisions. All that there is to interpret is the weight matrices, which collectively contain millions of real numbers. Relating this back to self-driving cars, this is bad because if the car makes a mistake, there is virtually no way to make sense of the weight matrices to understand what the network

was "thinking." There is a lot of research going on in this area, but so far, there has not been much progress.

Chapter 4

Deep Reinforcement Learning

Deep reinforcement learning uses deep learning to do reinforcement learning. To understand why throwing deep learning at a reinforcement learning problem can be a good idea, the premise of reinforcement learning must be reexamined.

Reinforcement learning algorithms take some representation of the state of the environment as an input to a policy function. In some scenarios, the state is a high dimensional data structure. Having a high dimensional data structure implies that there many possible states to consider. This has important implications for tabular reinforcement learning algorithms like Q-learning, which uses a state-action lookup table: The table can become extremely large and sparse [38]. The largeness comes from the fact that, for Q-learning at least, the dimensions of the table are the number of states by the number of actions. The sparseness comes from the fact that, given so many states and actions, it will likely take a long time to take enough actions in enough states to have a useful, generalizable table [38]. To understand the extent to which a large number of states affects reinforcement learning, we will frame the reinforcement learning

process as a variant of the coupon collector's problem.

4.1 The Coupon Collector's Problem

The coupon collector's problem is as follows. There is an eccentric coupon collector who wants to collect all n kinds coupons in existence. Assuming n is finite, if each of the n kinds of coupons is equally likely to be collected at any given time, how many coupons must the coupon collector collect in order to obtain each of the n distinct coupons?

This equally likely condition is tantamount to sampling with replacement, so this question does not have a solution. After all, it is theoretically possible to sample the same coupon infinitely many times in a row. Therefore, an actually solvable problem is often posed instead: what is the *expected* number of coupons that must be obtained in order to obtain all n kinds of equally likely coupons? This problem is solvable because it only requires calculating the expected value of the number of total samples: $\mathbb{E}[S]$, where S is the total number of samples. This expected value has the closed-form as in Proposition 1 and is proven thereafter. The given proof uses the concepts of a *Bernoulli trial* and a *geometric distribution*, which are defined in Definition 1 and Definition 2.

Definition 1. *A **Bernoulli trial** is a random variable with two possible outcomes, each with a known probability, p.*

Definition 2. *A **geometric distribution** is a distribution describing the number of independent, identical Bernoulli trials required to get the first desired outcome, e.g., a success.*

Proposition 1. *[1]* $\mathbb{E}[S] = n \cdot H_n$*, where* $H_n = \sum_{i=1}^{n} \frac{1}{i}$*, the* n^{th} *harmonic number.*

Proof. [1] Let each element in a population of size n be unique and equally likely to occur. Let S be a random variable for the total number of samples needed to sample, with replacement, all n elements from the population. Furthermore, let S_i be the number of samples between unique sample $i-1$ and unique sample i. This means that $S = S_1 + S_2 + \ldots + S_n = \sum_{i=1}^{n} s_i$. Therefore, $\mathbb{E}[S] = \mathbb{E}[\sum_{i=1}^{n} S_i]$.

Note that S_i is a *discrete* random variable since it is some natural number that is determined by a random process (random sampling). In abstract terms, S_i is the number of Bernoulli trials that resulted in a failure plus the one success, where a Bernoulli trial is defined in Definition 1.

Let each Bernoulli trial have two outcomes, success or failure, and let the probability of success be p_i for any natural number i. While p_i and p_{i-1} for S_{i+1} depend on each other and will not be the same, the sampling process that determines S_i is independent of that of S_{i+1}. This implies each Bernoulli trial that determines the value of S_i is independent. Therefore, S_i is a random variable that follows a geometric distribution [1] (Definition 2) with probability of success p_i.

Returning to $\mathbb{E}[S] = \mathbb{E}[\sum_{i=1}^{n} S_i]$, the *linearity of expectations* can be applied: $\mathbb{E}[S] = \mathbb{E}[S_1] + \mathbb{E}[S_2] + \ldots + \mathbb{E}[S_n]$. Since each S_i is geometrically distributed, the known expected value formula for a geometric distribution, $\mathbb{E}[S_i] = \frac{1}{p_i}$, can be applied to this sum of expected values: $\mathbb{E}[S] = \mathbb{E}[\sum_{i=1}^{n} S_i] = \frac{1}{p_1} + \frac{1}{p_2} + \ldots + \frac{1}{p_n}$.

Next, a closed-form solution for p_i will be derived where, still, i is any natural number less than or equal to n. Note that p_i necessarily depends on the population size n as well as $i-1$, the number of unique elements sampled so far. Given that every element in the population always has an equal probability of being sampled, it must be true that p_i is something like:

$\frac{\text{number of unique elements not sampled}}{\text{size of population of unique elements}}$. The denominator of that fraction is n. The numerator must be $n - (i - 1)$. Thus, $p_i = \frac{n-(i-1)}{n}$.

Applying this important result to the sum of expected values:

$$\begin{aligned}
\mathbb{E}[S] &= \frac{1}{p_1} + \frac{1}{p_2} + \ldots + \frac{1}{p_n} \\
&= \frac{n}{n-(1-1)} + \frac{n}{n-(2-1)} + \ldots + \frac{n}{n-(n-1)} \\
&= n \cdot \left(\frac{1}{n-(1-1)} + \frac{1}{n-(2-1)} + \ldots + \frac{1}{n-(n-1)} \right) \\
&= n \cdot \left(\frac{1}{n} + \frac{1}{n-1} + \ldots + \frac{1}{1} \right)
\end{aligned}$$

Given that *harmonic number* n is defined as $H_n = \sum_{i=1}^{n} \frac{1}{i} = \frac{1}{1} + \frac{1}{2} + \ldots + \frac{1}{n}$:

$$\mathbb{E}[S] = n \cdot H_n.$$

Thus, the expected number of samples, $\mathbb{E}[S]$, to sample all n elements with replacement from this population is equal to $n \cdot H_n$.

□

4.2 Applying the Coupon Collector's Problem

The closed-form solution to the coupon collector's problem is useful because the coupon collector's problem and reinforcement learning applications that satisfy the Markov assumption are somewhat analogous. In explicit terms, the set of unique coupons can be thought of as the set of unique state-action pairs. Sampling from those unique coupons can be also be thought of as sampling from

the state-action pairs, i.e., noting rewards for state-action pairs. Note, however, that these two problems are clearly not perfectly analogous. Strictly speaking, it is not true that each state is equally likely or that "sampled" state-action pairs are actually independent in reinforcement learning settings. Nevertheless, these parallels are sufficient to use this closed-form solution to obtain a rough estimate of how long tabular reinforcement learning would take when the state is a high dimensional data structure.

Therefore, for the sake of example, suppose that Q-learning is being used to play some arbitrary video game, say, some Temple Run-esque game. Temple Run is a mobile game in which players run through what is essentially a never-ending obstacle course. The objective is to run for as long as possible without running into an obstacle and without running off of the path. The frequency of obstacles increases over the course of the game, so the difficulty increases as well.

In this example, the state of the environment that is fed into the policy will be a frame from a mobile device's screen. Suppose the mobile device displays frames at 1,280 pixels by 720 pixels, in grayscale, and at 30 frames per second (fps). This means that there are $1,280 \text{ pixels} \cdot 720 \text{ pixels} = 921,600$ pixels per frame. Note that assuming that pixels are represented as 8-bit integers, a common size for grayscale images, and that each integer represents a different shade of gray, then there are $(2^8)^{(1280 \cdot 720)} = 8.95 \cdot 10^{2,219,433}$ unique 1,280 pixel by 720 pixel grayscale frames. Certainly, only a *very* small subset of all possible 1,280 pixel by 720 pixel grayscale frames will actually occur in this Temple Run-esque game. There are no well-known estimates of the order of such a subset, so arbitrarily, suppose there are 1,000,000 unique frames that will occur in this hypothetical game. For reference, at our stated 30 frames per second, if

each of these unique frames occurred one after another, it would take about $\frac{\text{1,000,000 unique frames}}{\text{30 fps}} = 33,333$ seconds, approximately 9.3 hours, to get through them all.

As for the action space, suppose our Temple Run-esque game has 7 possible actions: go straight, lean right, lean left, turn right, turn left, jump, and baseball-slide. At this point, we have enough information to estimate the size of the state-action pair table. Assuming that the reward value contained in each of the table's cells is a 4 byte (32-bit) floating point number, the rewards would take up (1,000,000 unique frames · 7 possible actions per frame) · 4 bytes per reward = 28,000,000 bytes, which is equivalent to 28 megabytes (MB). On today's computers, this is a minuscule amount of memory; the real memory issues arise from storing the states/frames in the table. Doing so would require 1,000,000 unique frames · 921,000 pixels per frame · 1 byte per pixel = 921,600,000,000 bytes, which is 921.6 gigabytes (GB).

Even if one could avoid storing the states by, say, using a perfect hash function to map images to a unique row index, i.e., even if memory is not an issue, then there would still be the issue of training time. An absolute lower bound on the training time was just stated above: 9.3 hours. This is a very naive estimate. To get a better estimate of the time required to fill in this table, we will employ the closed-form solution to the coupon collector problem. Letting each n equal the number of state-action pairs in the table, we see that: (1,000,000 unique frames · 7 possible actions) $H_{(1,000,000 \cdot 7)} = 7,000,000 \cdot 16.3386364 = 114,370,454.8$ frames. Getting through that many frames at 30 frames per second would take $\frac{\text{114,370,454.8 frames}}{\text{30 fps}} \approx 3,812,349$ seconds, which is about 44 days.

These figures, a 921.628 GB table and 44 days of training, are rough estimates that were made with some questionable assumptions. For one, the number of frames in the games, 1,000,000, was arbitrarily assumed. Given that this value is used in the calculation of the table size and the training time, increasing (decreasing) it would increase (decrease) the table size and training time. Second, frames in video games are not "sampled" independently; generally, successive frames are related. Nevertheless, these estimates are arguably more realistic than the naive estimates of 9.3 hours of training with a 28 MB table. Therefore, these figures show that having a large number of states negatively affects tabular reinforcement learning to a great extent.

Given this intractability of the tabular approach to reinforcement learning when dealing with many possible states, one wonders if reinforcement learning can still be used for larger, more complex problems. Fortunately, there is a solution to the intractably large table and training time problem: not using a table. Instead, a function approximator that approximates the value function can be used.

In the discrete case, the approximated value function, like the state-action table, would map a state and an action to a reward value. The same principle behind action selection from the tabular setting still applies: in a given state, the reward values are compared for each possible action, and the action with the highest reward value is taken. The difference now, though, is that neither the states nor the reward values are stored in memory. States are just inputs to the approximated policy function, and reward values are, instead, computed by the approximated policy function.

The optimal value function, which is really what an approximator is trying

to approximate in this case, can be complicated. The approximation of complicated functions, as we know from Chapter 3, is something that deep neural networks generally excel at. Therefore, deep reinforcement learning uses a deep neural network, as opposed to a state-action table, that approximates the optimal value function.

While deep learning and reinforcement learning have each been around for some time, deep reinforcement learning as described above has really only been around for past 10 to 15 years. Despite that, there is a decent variety of continuous and discrete algorithms. Rather than listing off a bunch of deep reinforcement learning algorithms, we will move on to examine a particularly popular discrete deep reinforcement learning algorithm: deep Q-learning via a deep Q-network.

4.3 Deep Q-learning with Deep Q-networks

As the name suggests, *deep Q-learning* uses deep learning to do Q-learning, which is the reinforcement learning algorithm that was examined in Chapter 2. In the 2015 article in which deep Q-learning was first presented, a *deep Q-network* was the deep neural network used. A deep Q-network is described as "a deep convolutional neural network [being used] to approximate the optimal action-value function:

$$Q^*(s,a) = \max_{\pi} \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ...|s_t = s, a_t = a, \pi], "$$

where, still, π is the behavior policy, r_i is the reward associated with time step i, and s is a state [27].

That quotation is a bit of an oversimplification since it seems to suggest that the deep Q-network is approximating the exact same optimal value function listed above. In other words, it makes it sound like the deep Q-network takes the same inputs as the action-value function, a state *and* an action, and outputs the same output, a single Q-value for the respective action and state. Neither of these, however, is true of deep Q-networks. The deep Q-network does not take an action as an input, only the state. Additionally, the network outputs multiple Q-values each time a state is fed forward through the network. Each of the Q-values that the network outputs is the Q-value of a particular action in the action space.

To make this more explicit, we return to the Temple Run-esque video game example. Recall that the action space of this example is the set: $A = \{$go straight, lean right, lean left, turn right, turn left, jump, baseball slide$\}$, which has order $|A| = 7$. Therefore, when a grayscale image is fed forward through the deep Q-network in this example, the output would some 7-dimensional vector of Q-values: $[q_1, q_2, \ldots, q_n]$. Which Q-value in that vector corresponds to which action is somewhat arbitrary, but the logical mapping is: q_1 is the Q-value of "go straight" (action 1 from A), q_2 is the Q-value of "lean right" (action 2 from A), and so on. Selecting an action from the output of the deep Q-network follows the imperative from tabular Q-learning; the corresponding action of the greatest Q-value in that output vector will be selected.

At first, this approach might seem needlessly complicated, at least relative to the alternative of feeding a state *and* an action into a deep neural network and

getting a single number, a single Q-value, as the output. This alternative approach already exists, and it is called *neural fitted Q-iteration* [28]. The reason why the deep Q-network approach is preferable to that of the neural fitted Q-iteration is that, holding all else equal, the deep Q-network will be more efficient [27]. The reason for this is that for deep Q-networks, the state only has to be fed forward once in order to get all of the Q-values that one needs for a given state. In neural fitted Q-iteration, the state would have the be fed through $|A|$ times, one time for each action in the action space. The gains from this efficiency can be "reinvested" to make the deep Q-network deeper (more layers) and/or wider (more neurons per layer). Ultimately, this should lead to a more powerful network [27].

The neural network architecture is only one of the many unconventional things about deep Q-learning. There also is the whole learning algorithm, which consists of multiple unconventional yet useful methods.

4.3.1 The Learning Algorithm

The deep Q-network and its learning algorithm are recent innovations [27], with its two explanatory papers coming in 2013 and 2015. The ultimate goal of these papers was to come up with a single kind of neural network that would be applicable to a wide variety of scenarios. More specifically, they wanted to create a neural network that was suitable for *general artificial intelligence*, which is a level of AI that is akin to general human cognition [26]. The network that resulted from those efforts is the deep Q-network. To refine their ideas for the deep Q-network and as a sort of "proof of concept," the authors trained their

deep Q-network to play Atari games. As the title suggests, via its learning algorithm, their deep Q-network achieved or exceeded human level performance in about half of the games it was trained to play [27]. Fortunately, in addition to the proof of concept, the paper provides algorithmic pseudocode for deep Q-learning. This algorithm is presented in Algorithm 2. It closely follows the original algorithmic pseudocode, though Algorithm 2 is a bit more verbose.

Algorithm 2 The deep Q-learning algorithm: training the deep Q-network [27].

Init. replay memory D with capacity for N transition tuples
Init. action-value approximating deep Q-network, called Q, with random weights: θ
Init. *target* action-value deep Q-network, called Q^-, with random weights: $\theta^- = \theta$
for *episode* from 1 through M **do**
    for t from 1 through T **do**
        Request state s_t
        Preprocess s_t with function Φ to get preprocessed state $\Phi(s_t) = \phi_t$
        Get a random number r in $[0, 1]$
        if $r < \epsilon$ **then**
            Randomly select an action a_t
        else
            Ask Q which action, a_t, to take: $a_t = \arg\max_a Q(\phi_t, a|\theta)$
        Execute action a_t, and observe reward r_t and state s_{t+1}
        Preprocess s_{t+1}: $\phi_{t+1} = \Phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
        Sample a random batch of transitions, B, from D
        Init. empty list: *targets*
        for $transition_j$ in B **do**
            if r_j in $transition_j$ is last reward before episode's end **then**
                Set $y_j = r_j$
            else
                Set $y_j = r_j + \gamma \max_{a'} Q^-(\phi_{j+1}, a'|\theta^-)$
            Append y_j to *targets*
        Batch gradient descent step for Q w.r.t θ with *targets* and loss:
        $(y_j - Q(\phi_j, a_j|\theta))^2$, for each transition in B and j from 1 through $|B|$
        Every C steps, set $\theta^- = \theta$
    Decrease ϵ

The very first line of Algorithm 2 initializes what is referred to as *replay memory*. Replay memory, D, stores the N most recent *transitions*. Transitions are defined as quadruples of the form $(\phi_t, a_t, r_t, \phi_{t+1})$, where ϕ_t and ϕ_{t+1} are preprocessed states for one time step and its consecutive time step respectively, a_t is an action, and r_t is a reward. The transitions in replay memory are used in

experience replay. Experience replay (Line 16 through Line 25) is the approach deep Q-learning takes to training the deep Q-network, Q. A high level description of experience replay is that it trains Q on some random batch of transitions [27] from D.

The reason experience replay and replay memory were incorporated into deep Q-learning is that they "reduced the variance of the updates because [then,] successive updates were not correlated with one another" [38]. "Correlated updates" refers to the concept that one update to the weights of a neural network partially determines what the next update to the weights will be. Generally, correlated updates are the result of correlated states. This is to say that correlated updates are likely to occur in time series data sets and occasionally in unshuffled data sets.

Correlated updates would be especially bad for deep Q-learning because small updates to a deep Q-network can significantly change the policy thereof [27]. To make this a bit more explicit, consider the alternative approach of training the deep Q-network each time a transition rolls in. While this would eliminate the need for replay memory, it would very likely result in the deep Q-network updating it weights to reflect the most recent sequence of transitions. It would be *overfitting* that sequence at the expense of generalizability. This means that when some unexpected change occurs in the sequence, the deep Q-network would not be able to handle that change. In the time steps immediately thereafter, the updates to the weights will be substantial so as to compensate for the overfitting that occurred. Multiple sudden and dramatic changes to weights of the network qualify the network as "unstable." Also note that, overall, this will increase the variance of the weight updates. Furthermore,

if that unexpected change does not come soon enough, then the deep Q-network could conceivably never recover and diverge altogether [38].

As a slight digression, this issue is exaggerated when the inputs to the deep Q-network are *stacked*. Stacking inputs, which was briefly mentioned at the end of Chapter 2, refers to the practice of feeding not just the most recent state as input into the network but also some number of states before that as well. Stacking would almost certainly compound the instability from training as the transitions roll in since it would be easier for the network to pick-up on the correlation between inputs. Despite this risk, stacking can still be useful in deep reinforcement learning. The authors of the original deep Q paper, for instance, in training their deep Q-network to play Atari games, stacked the 4 most recent frames [27]. This worked seemingly because of the stability that arises from training via experience replay rather than as the transitions roll in.

Moving on to Lines 2 and 3 of Algorithm 2, though it seems strange, these lines actually indicate that there are two neural networks at work in deep Q-learning. The two networks are both deep Q-networks, but they each are used differently. The deep Q-network on Line 2, Q, is used to select actions and is trained via experience replay. The deep Q-network on Line 3, Q^{-}, is solely used to generate the targets for the gradient descent step in experience replay (Line 24). It is never actually trained; rather, the weights of Q^{-} are replaced with those of Q every C time steps (Line 26). For perspective, C was 10,000 frames in the Atari experiment from the original paper [27]. This approach provides stability to the deep Q-network. Conceptually speaking, this is because improvements to the weights of Q are improvements *relative* to the weights of Q from n time steps ago rather than relative to the most recent time step, for

integer n greater than 1. More details on how each of these networks is used are provided in the discussion of the inner for-loop.

Next is the outer for-loop, which iterates through M episodes, where an "episode" is still just a set of T times steps. Therefore, the inner for-loop iterates through time steps 1 through T in the given episode. The first two operations in the body are to get state s_t and apply some preprocessing function, Φ, to s_t to get preprocessed state ϕ_t. Note, however, that preprocessing is not always necessary. Seeing as how Φ depends on the specific application, $\Phi(s_t) = \phi_t = s_t$ can also be acceptable.

Next is the if-else block in which an action is selected according to the ϵ-greedy action selection from Chapter 2. Random actions will therefore be selected with probability ϵ and deliberate actions will be selected with probability $1 - \epsilon$. Deliberate action selection, in this case, involves taking ϕ_t, feeding it forward through Q, and then selecting the action with highest Q-value from the output of Q. Usually, ϵ decays to 0.1 or 0 so that random actions are taken less and less frequently as training progresses. This decay occurs after each episode concludes, on Line 27. The basic idea is to let the deep Q-network explore at first, then learn from the exploration, and then learn from that which was learned of the exploration as well as from the exploration, and so on. Regardless, whether by randomness or by Q, we now have an action to execute, a_t.

On Line 13, action a_t is executed, and reward r_t and state s_{t+1} are noted. In other words, we note the result of a_t in ϕ_t. Lines 14 and 15 indicate that s_{t+1} is preprocessed into ϕ_{t+1} so that, now with ϕ_t, a_t, r_t, and ϕ_{t+1} all known, the transition quadruple for this time step, $(\phi_t, a_t, r_t, \phi_{t+1})$, can be added to replay

memory, D.

Everything before Line 16 can be view as the data collection steps. From Line 16 downward is where the actual training, updating the weights, occurs. As was mentioned earlier, these lines are also the lines of the whole experience replay process. Training via experience replay starts by sampling a batch of transitions, B, from D. For reference, $|B| = 32$ in the original deep Q paper. Given this small size, B is sometimes referred to as a *mini-batch*.

Next, in the for-loop beginning on Line 18, the j transitions in B is used to calculate the jth target, y_j, on which Q will be trained. There are two possibilities for y_j. First, when this transition is the last transition before the episode terminates, $y_j = r_j$ where r_j is the reward in $transition_j$. The second case is when this transition is not the last transition from the last time step in the episode. In this case, $y_j = r_j + \gamma \max_{a'} Q^-(\phi_{j+1}, a'|\theta^-)$., where a' is any possible action in the action space. Note that $\gamma \max_{a'} Q^-(\phi_{j+1}, a'|\theta^-)$ is just the discounted (by γ) estimated maximal reward for ϕ_{j+1} according to Q^-. Also note that this equality is the mostly the same as the update equation from the explanation of tabular Q-learning in Chapter 2. The difference now, though, is that Q^- is a neural network. As such, the same rules apply. Whether y_j is r_j or the output of Q^-, y_j is appended to a list called $targets$. Therefore, once this for-loop finishes executing, $targets$ will contain the target for each transition in B.

The point of calculating all of these targets is to execute Lines 24 and 25: update the weights of Q and θ, via batch gradient descent, with respect to θ. As per Chapter 3, there exist multiple batch gradient descent algorithms. All of them, however, take the gradient of a loss function with respect to the weights of

the neural network. In this case, the loss function is the same as from tabular Q-learning: $(y_j - Q(\phi_j, a_j|\theta))^2$. Thus, the weights between layers, including the weights between the last hidden layer and the multi-neuron output layer, are each updated via back-propagation. The final step in any given time step of any given episode is therefore: copy the weights of Q into Q^- if C time steps have passed since last doing so. Now, the current time step is finished. From here, the next time step, whether in this episode or the next, could begin or training could be over.

With this all in mind, we can state a better definition of deep Q-learning with deep Q-networks than that which was stated at the beginning of this section. Deep Q-learning aims to approximate the optimal Q-learning policy via a deep Q-network. A deep Q-network is a deep neural network that takes the state as an input and outputs as many Q-values as there are actions in the action space. The process by which the deep Q-network is trained is called experience replay. The first part of experience replay is computing the targets for a random batch of transitions (the quadruples) from replay memory by using old weights of the deep Q-network. The second part of experience replay is doing a batch gradient descent step on the tabular Q-learning loss function. Though there is no formal proof, given enough time and the right parameters, the deep Q-network should stabilize, ideally at or near the optimal policy it is trying to approximate.

4.4 Double Deep Q-learning with Double Deep Q-networks

Deep Q-learning has a known problem in that it tends to overestimate the Q-values in both the tabular and function approximator cases [40]. To understand why overestimation would be bad, consider Theorem 4.

Theorem 4. *[40] For a given state, let all true/optimal Q-values be equal, and let the estimated Q-values, as a whole, be unbiased estimates of the optimal Q-values, i.e., the sum of their errors is 0. Let C be the mean squared error of the optimal Q-values. Under these conditions, the largest estimated Q-value is greater than or equal to: $Q^* + \sqrt{\frac{C}{m-1}}$, where m is the size of the action space and Q^* is the optimal Q-value.*

Having unbiased Q-values as a whole does not mean that each estimated Q-value is unbiased since the error of one Q-value estimate could conceivably cancel out the error of another Q-value estimate. Therefore, the interpretation of Theorem 4 is that "even if the Q-value estimates are on average correct, estimation errors of any source can drive the estimates up and away from the true optimal values" [40].

The root cause of this error is the $\max_{a'}$ operator in the calculation of the targets: $y_j = r_j + \gamma \max_{a'} Q^-(\phi_{j+1}, a'|\theta^-)$. This $\max_{a'}$ operator introduces error because the same network is used to "select and to evaluate an action" [40]. This statement seems to be referring to possibility that the action that Q^- selects need not be the same action that Q selects since Q could learn a different policy than that of Q^-. At least some of the compounding error would arise from this discrepancy.

Fortunately, there exists an improved version of deep Q-learning that is able to substantially mitigate this overestimation issue: *double deep Q-learning*. Double deep Q-learning is exactly the same as deep Q-learning in all but one respect, so its learning algorithm need not be stated. The one change is that non-terminal targets are calculated as:
$y_j = r_j + \gamma Q^-(\phi_{j+1}, \arg\max_{a'} Q(s_{j+1}, a'|\theta_t)|\theta^-)$. Under this equation, the action for which Q^- computes the discounted estimated reward value will definitely be the same action as that selected by Q. Therefore, this potential source of error is hereby eliminated.

As a proof of concept that this alteration remedies the overestimation issue, the authors of the paper in which double deep Q-learning was proposed tested the double deep Q-network on Atari games. Firstly, just as was true of the deep Q-network, the double deep Q-network met or exceeded human level performance in about half of the games it was trained to play. On top of this, the double deep Q-network roughly met or exceed the performance of the deep Q-network in most of the games. Thus, the double deep Q-network, with its slight alteration, actually seems to be an improvement upon the deep Q-network.

With all of this understanding of deep reinforcement learning and its background, we are ready to proceed to the experiment section of this paper. We are ready to discuss the aforementioned attempt at developing a double deep Q learning-based autonomous vehicle system for a simulated car in a simulated environment.

Chapter 5

A Deep Reinforcement Learning Self-Driving Car

Though the Atari-playing deep Q-network started off from randomness, it eventually learned to play half of the Atari games at or beyond a human level. That this was accomplished by only taking in images of the screen and by receiving reward signals is rather impressive. Additionally, the fact that the data collection process was entirely automated is impressive as well. After all, no humans were used to identify optimal behavior or to assign a value to the actions that the deep Q-network was selecting.

These qualities of requiring very little explicit programming, performing at or beyond a human level, and not requiring human input for data labeling would, if they can be transferred over, be very appealing to self-driving car developers. Furthermore, if it is trained sufficiently well, a deep neural network-based driving system should be able to, as opposed to a set of human-created if-else statements, generalize from its past experiences and be

able to react to novel scenarios. All of these suggest that deep reinforcement learning could be used to make a solid self-driving car system.

There certainly would, however, be some significant challenges with using deep reinforcement learning as the mainstay of a self-driving car system. For one, coming up with a reward function that reflects proper driving behavior and all of the rules of the road may be impossible. Also, there is the issue that reinforcement learning is "learning by doing," so due to obvious safety and legal hazards, learning in the real-world is infeasible. Therefore, developing a deep reinforcement learning-based self-driving car system would necessarily involve learning in a computer simulation.

This brings us to the endgame of this paper: the attempted development of a double deep Q-network-based, self-driving car system for a computer simulation to the end that it can be used to draw conclusions about deep reinforcement learning's place in self-driving car systems.

5.1 Microsoft's AirSim

Since a deep reinforcement learning self-driving car must learn in a simulation, this project uses a simulation. The simulation of choice is AirSim, an open source drone and car simulator by Microsoft. AirSim provides a 3D urban environment with pedestrians, other vehicles, traffic signals and signs, speed limits, various weather conditions, and different times of day. Additionally, AirSim comes with a complete Python API for automating all interactions with the simulation. As Figure 5.1 suggests, the graphics of the simulation are described as "near photo-realistic," though, that comes with the caveat that one

has sufficiently powerful hardware [36].

Figure 5.1: Some images of the AirSim simulation [19]

On the more technical side, AirSim operates under the client-server model. AirSim is the server, which is is run locally. The client, which is where the deep reinforcement learning code runs, is implemented by the user and makes calls via the AirSim API to interact with the simulation. There is some overhead associated with passing messages, such as sending requests to the server and sending images from the server, but overall, this approach is supposed to be better as it allows for multi-processing.

5.2 The Driving Agent

A double deep Q-network is trained to steer a simulated car in AirSim at a low speed (less than 20 miles per hour). A high-level depiction of the architecture of this neural network can be seen in Figure 5.2. To get into the finer details of the network in Figure 5.2, we first examine the inputs. Even

more detail about the hyperparameters and other settings of the agent can be found in Appendix A.

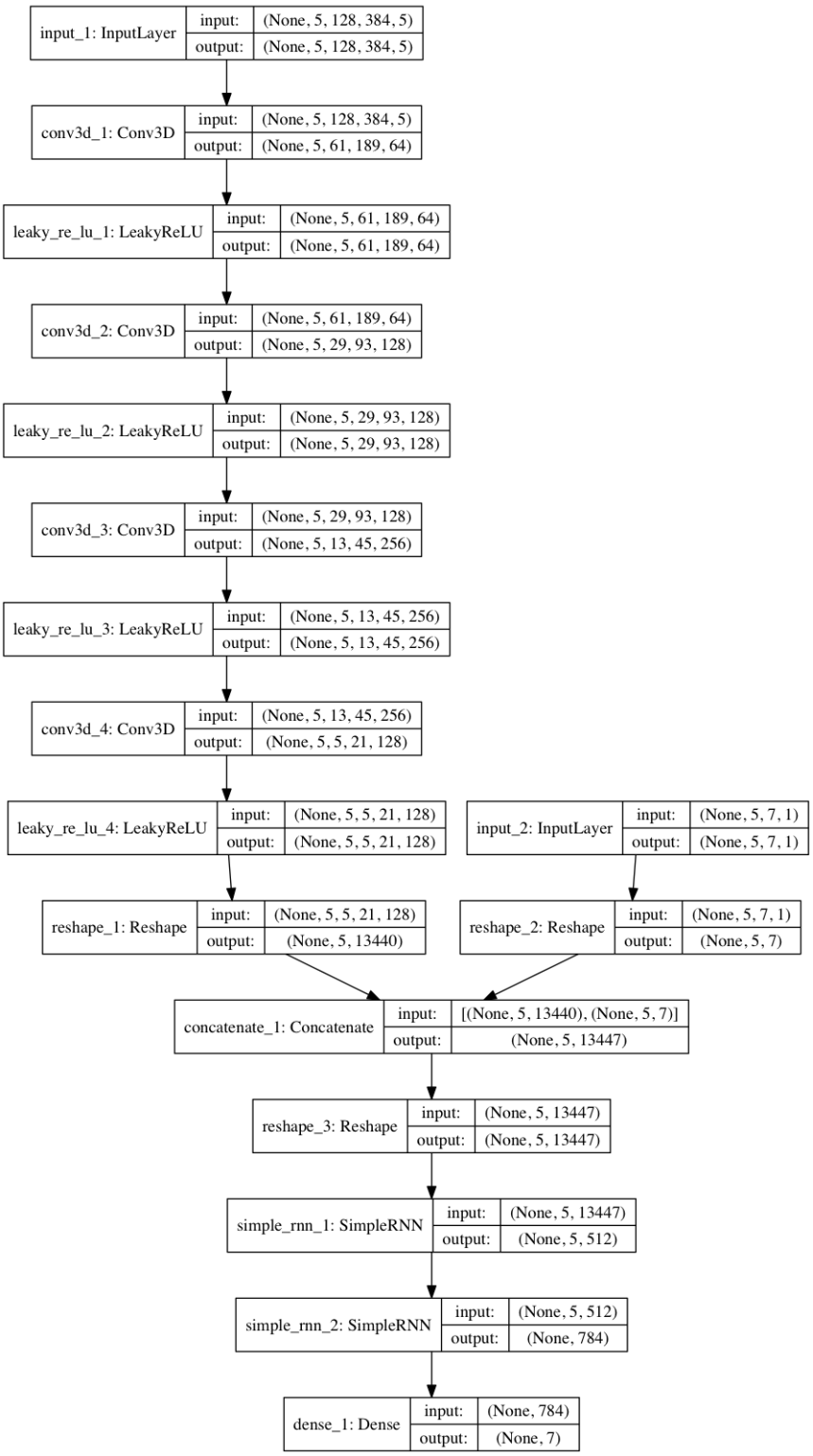

Figure 5.2: The high-level architecture of the neural network used to drive the simulated car

5.2.1 The Inputs

The double deep Q-network takes in states of the car's environment. All states consist of two kinds of data: an image from the front-facing camera and some miscellaneous sensor data. The camera image takes the left input path while the miscellaneous sensor data takes the right input path. Each time step's miscellaneous sensor data is contained in a vector of length seven. The first element of this vector is the vehicle's current Manhattan (city-block) distance from the destination in meters. The second element is the compass direction of the car, which is in radians. The third element is the bearing of the vehicle relative to the destination endpoint, which is in radians. The fourth element is the current steering angle, which is some number in [-1, 1]. The fifth element is the current speed in meters per second. The sixth and seventh elements are the absolute difference from the car's current location to the destination for the x and y coordinates, each of which is in meters.

As for each time step's camera image, there are two main parts: the red, green, blue (RGB) camera image and the concatenated pixel coordinates. A sample RGB camera image can be seen in Figure 5.3. This image is actually what the double deep Q-network sees; it is at the same resolution (128 by 384), field of view, and angle.

Figure 5.3: The camera image input to the DDQN driving agent.

Regarding the "concatenated coordinates," this refers to the last two channels (after the three RGB channels). As per [22], convolutional neural networks struggle to map pixels in an input image to pixels in an output image. The cause of this failure is said to be convolutional neural networks' *translation invariance*. This property basically says that convolutional neural networks tend to be better at recognizing if features are present in an image than at recognizing where those features are present. This is expected to be a problem since it matters whether an object is on the left or right side of the simulated vehicle. A solution to that problem is to append two channels: one with the x coordinates of each pixel and one with the y coordinates of each pixel [22]. Thus, overall, a single camera image for any given time step has 5 channels.

These dual inputs to the double deep Q-network are stacked such that the five most recent time steps qualify as the "current state." The underlying Markov decision process that the driving agent engages in is assumed to be a partially observable one. This is something that others have posited [21, 35] and intuition suggests. Therefore, the dual inputs are stacked.

5.2.2 The Network Layers

The two pieces of the input stack/state enter the network individually. This is because it does not really make sense to send the stack of miscellaneous sensor data vectors through convolutional layers. As such, only a state's stack of camera images passes through the four convolutional layers.

The activation for each of the convolutional layers is the *leaky rectified linear unit* (LeakyReLU) function, which, for each neuron's net input z, is defined as [23]:

$$\text{LeakyReLU}(z) = \begin{cases} z & z \geq 0 \\ cz & z < 0. \end{cases}$$

Note that the difference between LeakyReLU and ReLU (from Chapter 3) is the $z < 0$ case. Making the output of this case be cz, where c is some constant in $(0, 1)$, rather than 0 means that the derivative in this case would be non-zero. Having a non-zero derivative is desirable because it should prevent the vanishing gradient problem from occurring.

The output of the final convolutional layer is reshaped along its spatial dimensions such that temporality (the stack of length five) is preserved. This is to say that, for each time step, the multiple feature maps are flattened into a single vector, so there are five of those vectors in the stack. Aside from allowing for the stack of flattened feature maps to be concatenated to the stack of miscellaneous data vectors, having a sequence (the input stack of length five) allows for simple recurrent layers to be used. Having simple recurrent layers rather than, say, regular perceptron layers is supposed to partially address the assumed partial observability inherent in the driving decision process [21, 35].

5.2.3 The Outputs

The final simple recurrent layer forwards its activation values to the output layer. The output layer of the double deep Q-network in Figure 5.2 has seven perceptrons. There is one neuron for each steering angle available to the driving agent since each neuron's output is the Q-value for that steering angle. The steering angles available to the driving agent are: -1.0, -0.667, -0.333, 0.0, 0.333, 0.667, and 1.0, where -1.0 rotates the steering wheel completely in the counter-clockwise direction and 1.0 completely in the clockwise direction.

5.2.4 Training for the Task at Hand

The driving agent is tasked with only steering the car along one specific road; it is not tasked with controlling the car's speed. Part of the road on which the car drives can be seen in Figure 5.3. The road itself is about 115 meters long, so the goal for the driving agent is to reach the other end of the road without colliding with anything and by staying on the road. It is a flat, two-lane road with no other cars, no pedestrians, and no side-roads, so all that the driving agent has to learn is to keep itself on the road.

The reward function for this driving agent considers whether the agent is on the road, whether the agent is on the curb, and how far (using Manhattan distance) the agent has travelled towards its destination. Explicitly, using some

information from the simulation, the reward is calculated as:

$$r = \begin{cases} -1.0 & \text{if not on road and if not on curb} \\ -0.1 & \text{if not on road but if on curb} \\ \frac{\text{distance travelled towards destination}}{115} & \text{if on road and not on curb} \end{cases}$$

If the vehicle is off the road for more than six time steps, the episode will terminate so that a new episode can begin. Each time an episode starts, the car is reset to the exact same location.

There are even more training parameters that influence training, such as the gradient descent's learning rate η and the discount rate γ. Instead of listing-off all of that information here, it is presented in a table in Appendix A.

5.2.5 Results

The results of training under this framework, as well as some concluding remarks, can be found in the next chapter, Chapter 6.

Chapter 6

Conclusion

The given neural network did not produce any substantial, positive results; it was not able to steer the vehicle for more than 20 feet. These negative results tell a lot about the field of deep reinforcement learning. To show why this is the case, as well as to provide some reasons for the failure of this project, an overview of the research process is in order.

6.1 The Research Process

All of the author's ideas for the neural network were, at some point, implemented. After many iterations, the author arrived at the most complicated neural network. It had a third input, a depth image wherein each pixel in the image is the distance from the car to the surface of the object to which that pixel corresponds. It also used a more advanced convolutional layer referred to as a *convolutional long short-term memory* layer, which essentially is capable of identifying spatial and temporal relationships in the selfsame layer. After that network failed to achieve any substantial positive results, the model was

simplified to its current state, the one presented in the previous chapter.

Regarding the actual task that the network had to learn, it was repeatedly simplified over the course of this project. The first, foolhardy goal was to develop an agent that could start from any location and be able to drive aimlessly, indefinitely, so long as it could collisions. When the neural network failed to accomplish that objective, the task was simplified down to being able to drive aimlessly from a single starting point. All that the agent had to do was avoid collisions and stay on the road, albeit indefinitely. This, too, failed. The author supposed that the indefinite aimlessness was the issue, so a single destination was introduced along with some navigation information, hence the miscellaneous data inputs to the aforementioned neural network. Though the distance from the starting point to the destination was reduced multiple times (from 500 meters down to about 100 meters), the car was still unable to avoid collisions or make it to the destination.

The reason why introducing a medium-range destination failed was assumed to be the result of starting the car on the sidewalk rather than the road. While laughable, doing so was considered a necessity due to instability in the simulation, i.e., the simulation often crashed if the car respawned into or on top of another car whenever the next episode would begin. Therefore, a different starting point, one that was actually on the road, was chosen. This change yielded a slight improvement in results, yet, the agent still could not avoid collisions, avoid sidewalks, or drive to the destination, which required a single, right turn.

Still optimistic, the author attributed this failure to fact that the car had to go through an intersection and that the car could therefore not figure out which

direction it should go. To remedy this, another starting point, the final starting point for this project, was chosen: a two-lane road with no side-roads, no cars, and no pedestrians. All that the car would have to do is learn to keep itself on the road. Once again, though, the agent failed to learn to keep the car on the road for even 10% of the way to the destination, which was 100 meters away.

Though great simplifications were made regarding the objective of the agent and though the most powerful neural network that the author could develop was deployed against the most basic of tasks, no substantial positive results can be reported. This is very unexpected, so some sort of an explanation is in order.

6.2 Reasons for Failure

As one would suspect, there are many reasons for this abject failure. Before any other reasons are given, though, it should be clearly stated that the author was unacquainted with deep reinforcement learning at start of this project. Therefore, it is perfectly valid to attribute some of this failure to the author's lack of experience and ability.

Aside from this reason, in no particular order, firstly, the number of time steps required for the neural network to learn to steer itself on even the most simplified of roads is likely much greater than the number of time steps it was allotted. As has been mentioned, there were many other neural networks that were trained to steer the vehicle. As such, though it sounds low, this final network was trained for no more than 250,000 time steps. The deep Q network that learned to play Atari games, however, required approximately 10 million frames/time steps for each game it learned to play.

This disparity was not due to impatience on the part of the author; it was due to the second reason for failure: the simulation. The simulation was simply too computationally expensive, and therefore too slow, for the purposes of this project. The simulation's internal clock could be sped up, but due to either a bug in the simulation or insufficient computer hardware, only other vehicles would be sped up when the clock speed was increased. The one that the agent was controlling would not be sped up.

Not only could the simulation not be run at a speed greater than that of real life, but usually, especially for the more complicated neural networks that were tried, the simulation time would effectively be *slower* than real-time. For the last neural network that was tried, training for 1 simulation second, which is about 4 time steps, required approximately 3 to 4 real-time seconds. To make this more explicit, if the driving agent trains at the more conservative rate of 1 simulation second per 3 real life seconds, then to reach 10 million time steps like the Atari player did, this deep Q network would need to train for: $\frac{10{,}000{,}000 \text{ time steps}}{1 \text{ training period}} \cdot \frac{3 \text{ real-time seconds}}{4 \text{ time steps}} \cdot \frac{1 \text{ hour}}{3600 \text{ real-time seconds}} \cdot \frac{1 \text{ day}}{24 \text{ hours}} = 87$ days. This ridiculous figure could even be an underestimate of the actual time required for such a network to learn to steer a car. It is not necessarily true that, since the Atari deep Q network required 10 million time steps/frames to be able to play an Atari game, so too must this deep Q network work through 10 million frames so as to be able to drive the simulated car.

Related to this second source of failure is the third source of failure: extensive prototyping times. Mostly, this is not referring to time it took to convert new ideas into Python code; rather, it is mostly referring to the time it takes to evaluate the consequences of those changes. The consequences of

changing the reward function, the inputs, the neural network, or the hyperparameters of the agent could not be readily determined due to excessive training times. Generally, it would be at least one whole day before the agent would have trained long enough so that the consequences of a change could be determined.

A fourth cause of failure is the difficulty associated with designing a good reward function. Deep reinforcement learning algorithms are obviously incapable of reading the minds of the programmer; they only know what the programmer's reward function tells them. Therefore, if the reward function is not truly aligned with what the programmer wants (and does not want) the agent to do, then the agent could very well behave unexpectedly. One manifestation of this was whenever the author designed a reward function that was based on two things: not colliding with anything and the distance travelled since the last time step. Several times, the deep reinforcement learning agent took advantage of this and simply drove in circles, which would allow it to receive a reward indefinitely, essentially. This issue was eventually resolved, but it illustrates the fickleness of reinforcement learning algorithms and their reward functions. Therefore, it could be the case that even the final reward function was not sufficiently suggestive.

A fifth and last source of failure is randomness. Deep reinforcement learning in general is fickle and can fail simply due to a bad random seed. In some circumstances, this is true of up to 25-30% of trained models [16, 17]. Putting aside the time that would be wasted on training dead-end networks, if this random failure occurred in any of the many training iterations, then the author could have mistakenly concluded that some beneficial change was a detrimental change. Furthermore, conceivably, this could lead to an overly complicated

neural network, which could definitely have been said for some of the unmentioned neural networks that were tried over the course of this project.

6.3 Future Work

While these issues completely derailed this project, they do not seem to be altogether insurmountable. For one, the proposed neural network architecture seems to be on the right track since it has the necessary components for capturing temporal and spatial relationships. It could be the case that a much deeper and/or wider deep Q network would be successful at learning to drive the simulated car. Deeper and wider networks, however, require more memory and computational power. Plus, deeper and wider neural networks often require more data, which in the reinforcement learning context equates to more training time. This means that this avenue would require much more powerful hardware and a much better simulation.

The simulation, AirSim, leaves a lot of room for improvement. The clock speed issue needs to be resolved, and a less computationally and memory intensive simulated environment needs to be created. Also, requesting multiple images from the simulation needs to be done much faster since, at least on the machine used for this project, doing so currently takes about 0.08 to 0.25 seconds. These improvements are crucial since the rapid prototyping and randomness issues will more or less fade away once training times are drastically reduced. If these improvements cannot be made, however, perhaps a better simulation will come along, one that is more suited to the demands and constraints of deep reinforcement learning.

As for the issue of finding a good reward function, the optimistic researcher would likely say that this issue may soon cease to exist. A breakthrough in *inverse reinforcement learning*, which aims to learn a reward function through observing human behavior [38], could be right around the corner. For this project, being able to generate a reward function simply by driving the simulated car would be a boon.

6.4 Deep Reinforcement Learning's Place with Self-Driving Cars

While the aforementioned changes and breakthroughs would probably be enough for this project to eventually yield positive results, this project very strongly suggests that deep reinforcement learning will never be used to drive a production-ready self-driving car.

The biggest hang-up seems to be that a deep reinforcement learning self-driving car would be heavily coupled to a simulation. In order for simulation performance to transfer over to real-world performance, the simulation needs to be as realistic as possible. Not only would the development of such a simulation require a lot of time and money, but also, as a simulation's realism increases, so too does its computational power requirements. All of this seems to indicate that research into applying deep reinforcement learning to self-driving cars will be extremely limited due to the financial and temporal costs of conducting such research.

On top of this, no matter how realistic the simulation is, it is still only an approximation of reality. An agent that can successfully drive a vehicle in a

simulation will likely not be able to drive a vehicle in reality just as successfully. While this disparity would partially be offset by the simulations's capacity to simulate rare events, it does not completely offset it. Interestingly, one proposed solution is to convert unrealistic simulation images into more realistic images via convolutional neural networks, e.g., the "realistic translation network" [31], but currently, such neural networks perform rather poorly. This all seems to indicate that an agent would have to do some degree of training on the road, which likely will never be legally permissible given that reinforcement learning is, as previous chapters mentioned, "learning by doing."

There is also the issue of designing a sufficiently good reward function. In the case of the Atari video game agent, the reward function was obvious (+1 if the agent wins and -1 if the agent loses) and the consequences of getting it wrong were minor to non-existent. In the case of a self-driving car agent, a reward function that would produce an agent that makes the correct and ethical choices at or beyond a human level is not at all obvious or may not exist, and the consequences of getting it wrong would be great, both in terms of money and human lives. In the opinion of the author, this point stands even if inverse reinforcement learning progresses beyond its current state.

Lastly, there are the relative costs of pursuing a deep reinforcement learning self-driving car agent. After all, the question is not "would a deep reinforcement learning self-driving car agent be difficult to develop?" Instead, it is "would a deep reinforcement learning self-driving car agent be more difficult to develop than other approaches?" Now, there are a lot of ways that difficulty can be measured, but most of them boil down to time and money. Given what deep reinforcement learning is currently capable of, the challenges the author

encountered over the course of this project, and the proofs-of-concept that other approaches already have, e.g., *imitation learning* wherein a neural network learns to "imitate" human behavior by directly observing human behavior [6], it seems reasonable to say that a deep reinforcement learning approach would be more expensive than most end-to-end and modular approaches. This is true in terms of both time and money.

This conclusion takes a very pessimistic view towards deep reinforcement learning's place with self-driving cars. This pessimistic view, however, should not be construed as a statement on deep reinforcement learning as a whole. It is simply the conclusion of this author that deep reinforcement learning has its own domain of problems, a domain that does not include production-ready self-driving cars, and that the field is still, relatively, in its infancy.

Appendix A

The Driving Agent's Hyperparameters and Other Settings

The double deep Q-network presented in Chapter 5 has many hyperparameters and other details that, for the sake of conciseness, were not mentioned. Those parameters, like the number of neurons per layer, the convolutional window sizes, and the learning rate, are listed here.

Name	**Value**
input stack length	5
image input size	$(5, 128, 384, 5)$
miscellaneous sensor data input size	$(5, 7, 1)$
convolutional Layer 1, number of neurons	64
convolutional Layer 2, number of neurons	128

convolutional Layer 3, number of neurons	256
convolutional Layer 4, number of neurons	128
convolutional Layer 1, window size	$(1, 8, 8)$
convolutional Layer 2, window size	$(1, 4, 4)$
convolutional Layer 3, window size	$(1, 4, 4)$
convolutional Layer 4, window size	$(1, 4, 4)$
convolutional Layer 1, strides size	$(1, 2, 2)$
convolutional Layer 2, strides size	$(1, 2, 2)$
convolutional Layer 3, strides size	$(1, 2, 2)$
convolutional Layer 4, strides size	$(1, 2, 2)$
convolutional activation function	LeakyReLU with $c = 0.3$
simple recurrent Layer 1, number of neurons	512
simple recurrent Layer 2, number of neurons	784
simple recurrent activation function	sigmoid
output layer, number of neurons	7
experience replay memory	sequential memory
sequential memory capacity (recent states)	4,000
policy	linearly annealed with epsilon greedy action selection
max ϵ	1.0
min ϵ	0.05
number of steps to reach min	70,000

discount rate, γ	0.972
number of total training steps	85,000
training frequency (steps)	16
warm up steps	256
training batch size	8
target network, Q^-, update frequency (steps)	9,000
optimizer	stochastic gradient descent
initial learning rate, η,	0.00000001
decay factor	0.0000000000018
time between decisions (seconds)	0.3

Table A.1: The hyperparameters and other settings of the driving agent

Bibliography

[1] Ahmad, N., Sinha, S., Gnegel, F., Boushehri, S., Chakaroglu, A., and Mensah, E. The coupon collector's problem and generalizations. *MathMods* (2014).

[2] Alpaydin, E. *Introduction to Machine Learning*, 2 ed. The MIT Press, 2010.

[3] Alpaydin, E. *Machine Learning: The New AI*. The MIT Press, 2016.

[4] AutoAlliance. Levels of automation. `https://autoalliance.org/wp-content/uploads/2017/07/Automated-Vehicles-Levels-of-Automation.pdf`, July 2018.

[5] Ballard, D. *An Introduction to Natural Computation*. The MIT Press, 1997.

[6] Bojarski, Testa, Dworakowski, Firner, Flepp, Goyal, Jackel, Monfort, Muller, Zhang, Zhang, Zhao, and Zieba. End to end learning for self-driving cars. *CoRR* (2016).

[7] Bosse, and Appen. Appendix: Proof of the Heine-Borel theorem. https://c.ymcdn.com/sites/www.amatyc.org/resource/resmgr/educator_feb_2018/Bosse-appen.pdf, 2018.

[8] Calculator, D. G. Desmos graphic calculator. https://www.desmos.com/calculator/v26hrvkoem, 2019.

[9] Csaji, B. Approximation with artificial neural networks. Master's thesis, Etovos Lorand University, 2001.

[10] Deloche, F. Recurrent neural network unfold. https://en.wikipedia.org/wiki/File:Recurrent_neural_network_unfold.svg, 2017.

[11] Department of Mathematics - Oregon State University. The gradient and directional derivative. https://math.oregonstate.edu/home/programs/undergrad/CalculusQuestStudyGuides/vcalc/grad/grad.html, 1996.

[12] Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., and Koltun, V. Carla: An open urban driving simulator. *1st Conference on Robot Learning (CoRL 2017)* (2017).

[13] Eden, T., Knittel, A., and Uffelen, R. Reinforcement learning - temporal difference learning. https://www.cse.unsw.edu.au/~cs9417ml/RL1/tdlearning.html, 2001.

[14] El Hady, A., and Machta, B. B. Mechanical surface waves accompany action potential propagation. *Nature Communications 6* (2015).

[15] Hausknecht, M., and Stone, P. Deep recurrent Q-learning for partially observable MDPs. *Association for the Advancement of Artificial Intelligence* (2017).

[16] Houthooft, R., Chen, X., Duan, Y., Schulman, J., Turck, F. D., and Abbeel, P. Curiosity-driven exploration in deep reinforcement learning via Bayesian neural networks. *CoRR abs/1605.09674* (2016).

[17] Irpan, A. Deep reinforcement learning doesn't work yet. `https://www.alexirpan.com/2018/02/14/rl-hard.html`, 2018.

[18] JahKnows. Role derivative of sigmoid function in neural networks. `https://i.stack.imgur.com/inMoa.png`, 2018.

[19] Kapoor, A., and Shah, S. Microsoft extends AirSim to include autonomous car research. `https://www.microsoft.com/en-us/research/blog/autonomous-car-research`, 2017.

[20] Karpathy, A. Cs231n convolutional neural networks for visual recognition. `http://cs231n.github.io/neural-networks-3/`, 2017.

[21] Kendall, A., Hawke, J., Janz, D., Mazur, P., Reda, D., Allen, J.-M., Lam, V.-D., Bewley, A., and Shah, A. Learning to drive in a day. *ArXiv e-prints* (2018).

[22] Liu, R., Lehman, J., Molino, P., Such, F., Frank, E., Sergeev, A., and Yosinski, J. An intriguing failing of convolutional neural networks and the coordconv solution. *32nd Conference on Neural Information Processing Systems* (2018).

[23] Maas, A., Hannun, A., and Ng, A. Rectifier nonlinearities improve neural network acoustic models. *Proceedings of the 30 th International Conference on Machine Learning* (2013).

[24] Marina, L., and Sandu, A. Deep reinforcement learning for autonomous vehicles - state of the art. *Bulletin of the Transilvania University of Braov* (2017).

[25] Melo, F. Convergence of Q-learning: A simple proof. `http://users.isr.ist.utl.pt/~mtjspaan/readingGroup/ProofQlearning.pdf`, 2000.

[26] Minh, V. Lecture 3 deep RL bootcamp: Deep Q networks. `https://www.youtube.com/watch?v=fevMOp5TDQs`, 2017.

[27] Minh, V., Kavukcuoglu, Silver, Rusu, Veness, Bellemare, Graves, Riedmiller, Fidjeland, Ostrovski, Petersen, Beattie, Sadik, Antonoglou, King, Kumaran, Wiestra, Legg, and Hassabis. Human-level control through deep reinforcement learning. *Nature* (February 2015).

[28] Mousavi, S., Schukat, M., and Howley, E. Deep reinforcement learning: An overview. *Proceedings of SAI Intelligent Systems Conference* (2018).

[29] NHTSA. Federal automated vehicles policy. Tech. rep., U.S. Department of Transportation, National Highway Traffic Safety Administration, 2016.

[30] NHTSA. Automated vehicles for safety. `https://www.nhtsa.gov/technology-innovation/automated-vehicles-safety`, January 2018.

[31] Pan, M., You, Y., Wang, Z., and Lu, C. Virtual to real reinforcement learning for autonomous driving. *CoRR* (2017).

[32] Praja, S. A derivation of backpropagation in matrix form. `https://sudeepraja.github.io/Neural/`, 2017.

[33] Raschka, S., and Mirjalili, V. *Python Machine Learning*, 2 ed. Packt Publishing, 2017.

[34] Russell, S., and Norvig, P. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

[35] Sallab, A., Abdou, M., Perot, E., and Yogamani, S. End-to-end deep reinforcement learning for lane keeping assist. *30th Conference on Neural Information Processing Systems* (2016).

[36] Shah, S., Dey, D., Lovett, C., and Kapoor, A. AirSim: High-fidelity visual and physical simulation for autonomous vehicles. *Springer Proceedings in Advanced Robotics* (2017).

[37] Soleimany, A. MIT 6.S191 (2018): Convolutional neural networks. `https://www.youtube.com/watch?v=NVH8EYPHi30`, 2018.

[38] Sutton, and Barto. *Reinforcement Learning: An Introduction*, 2 ed. The MIT Press, 2018.

[39] Tegmark, M., and Rolnick, D. The power of deeper networks for expressing natural functions. *Journal of Statistical Physics* (April 2018).

[40] Van-Hasselt, H., Guez, A., and Silver, D. Deep reinforcement learning with double Q-learning. *Association for the Advancement of Artificial Intelligence* (December 2015).