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

Training and Controlling a Simulated Car with Deep Reinforcement Learning

Marcus Loo Vergara

Supervisor: Frank Lindseth

TDT4501 – Computer Science, Specialization Project

Fall 2018

Abstract

In this report we will investigate the current landscape of state-of-the-art reinforcement learning methods and we will evaluate the viability of these methods in the context of training and controlling a simulated car. Autonomous driving has gathered a lot of interest to researchers, governments, and private companies as of late as such technologies promises to solve several problems that are prominent in modern society. Examples of such problems include individuals spending a lot of their time in traffic due to congestion, and people and institutions having to financially support costly car accidents made by human error. Advancements in machine learning is what drives autonomous vehicle technology forward, and we have already seen several big actors in the automobile and artificial intelligence industries take advantage of this; having autonomous vehicles drive for several miles on public roads without incident. In this report, we will start by taking a look at the ways machine learning is being used in recent publications on self-driving cars – where we will be particularly looking at imitation learning. Later, we will cover fundamental reinforcement learning theory and subsequently dive deeper into general-purpose "deep" reinforcement learning methods that utilize deep learning to handle increasingly complex tasks. Finally, we will use our implementation of *Proximal Policy Optimization* – the general-purpose deep reinforcement learning method that is currently considered to be the best baseline in reinforcement learning research – to teach an agent to drive in a driving-like environment. Using this model, we will experiment with different kinds of neural network architectures and empirically compare and discuss advantages and disadvantages of each of these models.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

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

We have seen recent advancements in artificial intelligence and parallel computing power due to the recent successes in deep learning. These advancements has lead to an increase in interest for autonomous vehicles, and has made companies and researchers wanting to explore the viability of autonomous driving technologies in a real-world setting.

1.2 Autonomous Driving

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

to some target location B with limited to no intervention made to the control system of the vehicle by a human driver. The *Society of Automotive Engineers* has defined a set of levels of increasing automation when it comes to self-driving cars [Int14]. These levels are as follows:

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

Today's automobiles consist primarily of a mix of level 0 ("no automation") and level 1 ("hands on") autonomous vehicles. Some car manufacturers such as Tesla, have produced vehicles that are commercially available which have features that are considered level 2 ("hands off") automation. There are also a few examples of commercial and non-commercial vehicles featuring level 3 ("eyes off") automation, such as Audi's A8's "Traffic Jam Pilot;" capable of autonomously driving the car at speeds up to 60 km/h in traffic jams. There are no commercially available

level 4 ("mind off") or level 5 ("steering wheel optional") automobiles, however there is on-going development for such vehicles, and Waymo's self-driving car is an example of a self-driving car that has no steering wheel. Level 5 automation is naturally the ultimate goal in autonomous driving research and development.

Why is autonomous driving interesting?

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

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

Commuting Time and Emissions: Computers can also communicate with each other wirelessly, and plan their behaviours in unison to an extent humans are incapable of. An autonomous car on the road can know where its nearby autonomous cars are going and plan their trajectories accordingly. This would reduce commute time, CO2 emission and have overall benefits to society. If cars were more efficient, people might eventually not need to own cars at all, potentially cutting down on the environmental cost of producing cars and the financial expenses of

owning a vehicle.

1.3 Components of an Autonomous Car

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

Sensors, Compute and Control

What components are required to making a car drive intelligently? This part of the process should consider what components and sensors are necessary; how can we make the car energy efficient enough to drive on the road with limited battery capacity.

Mapping and Localization

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

Perception

This is about making sense of sensor input, and generating predictions of various types of properties for both static and dynamic objects in the environment, e.g. object location in 3D space, object trajectories, object types, etc. Cars, buses, trucks, bicycles and pedestrians are examples of dynamic objects that should be tracked, and static object may include traffic lights,

signs, lamp posts, etc. A perception module should be able to determine the location static object such as lane lines and signs, in addition to also understanding the meaning of various road signs and signals. Perception information can be extracted with traditional RGB cameras, infrared cameras, LiDAR laser scanners, etc.

Planning and Control

Planning is all about finding out how to manipulate the acceleration, break and steering to navigate a vehicle from A to B. To control we have to use the output from the sensors or potentially the output of a perception module to determine the control vector. Ideally, the vehicle should be able to drive without incident with other autonomous vehicles, normal vehicles, pedestrians, etc.

Simulation and Validation

It is vitally important that we have ways to test and verify our control and perception algorithms before deploying them onto a real vehicle. This is the purpose of creating simulators. Since we use simulators to verify our algorithms before deployment, it is important that a simulator emulates the physics and appearance of real-life as closely as possible.

1.4 Other Topics in Autonomous Driving

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

Safety and Security

Assuring the safety of an autonomous system is important for deployment of autonomous systems. Simulators are useful in validation out methods, however deep learning based agents

are often hard to interpret. In the case of an accident, we need to have ways to understand why the agent failed. So safety is, among other things, about building interpretability into our systems.

Collaborating Vehicles

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

Ride Sharing

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

Privacy and Ethics

Maintaining the privacy of the users and the surrounding people is also important to consider. When every vehicle is equipped with out and inward-facing cameras and LiDARS, people can easily be tracked and identified; something that could violate existing privacy laws. Additionally, autonomous vehicles may have to make ethical decisions in certain situations. The most commonly posed scenario is that in which the car finds itself in a dangerous scenario, and has to decide whether to sacrifice a innocent bystander vs. the owner of the vehicle.

1.5 Objective

In this report, we want to investigate state-of-the-art methods in reinforcement learning – with the objective of utilizing this theory to devise reinforcement learning models that are able to

safely and reliably maneuver by themselves inside simulators, and potentially on public roads with various driving scenarios. In pursuit of this objective, we set out to answer the following questions:

- What are the principles and methods behind deep reinforcement learning, and are state-of-the-art reinforcement learning methods able to learn how to drive a car in a controlled and reliable manner?
- How influential is the internal neural network model of these methods to the training speed and performance, and are there alternative models have desireable properties over the tried and true models used in these works?

Chapter 2

Background

2.1 Machine Learning

2.1.1 Introduction

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

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

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

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

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

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

2.1.2 Artificial Neural Networks

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

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

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

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

2.1.3 Deep Learning

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

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

Convolutional Neural Networks

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

2.2 Imitation Learning for Self-Driving Vehicles

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

2.2.1 Background

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

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

2.2.2 End-to-End Imitation Learning

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

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

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

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

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

2.2.3 Conditional Imitation Learning

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

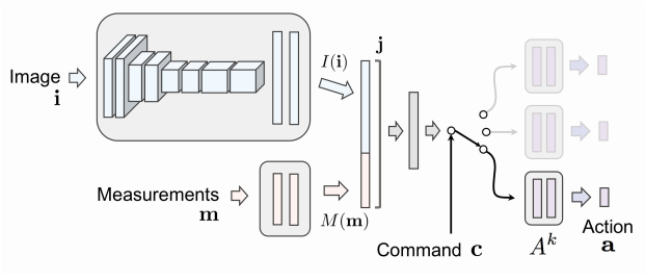


Figure 2.1: Codevilla's *et al.* conditional imitation learning model. Notice how the fully-connected layers in each A^k branch are conditioned on the input command c.

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

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

2.3 Reinforcement Learning

2.3.1 Introduction

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

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

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

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

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

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

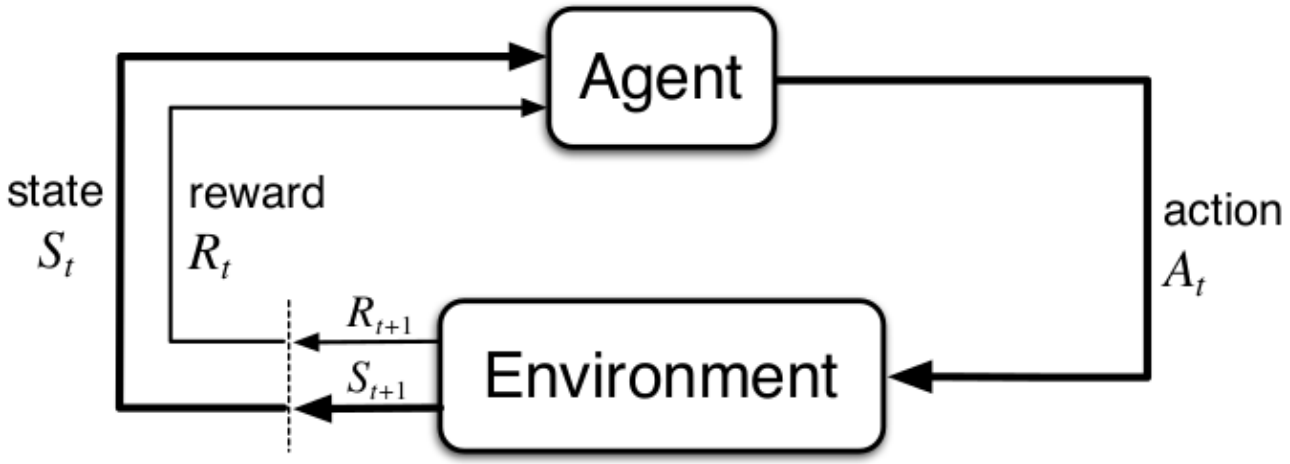


Figure 2.2: Reinforcement learning loop. Starting at time-step t, the agent observes the state s_t and reward r_t. Whenever an agent takes an action, a_t, the environment returns a new state, s_{t+1} and a scalar reward value, r_{t+1}, representing the state and reward in time-step $t+1$.

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

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

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

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

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

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

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

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

Exploration vs. exploitation

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

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

Fully observable vs. partially observable environment

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

Discrete vs. continuous state and action spaces

Depending on the environment, the state and action spaces may be discrete or continuous. Chess is an example of a game with discrete state and action spaces; there are a countable number of possible states and actions at any given time. Autonomous driving with a proximity sensor is an example of an environment with continuous state and action spaces; the state and actions are real-numbered values, *e.g.* proximity values are in meters for the state and the steering angle in degrees for the action.

Model-based vs. model-free

Model-based methods are methods that attempt to create a model of how the environment works. If an agent is in state s and takes action a, it will get a new state s' and reward r. Intuitively, these observations could be used to approximate $P(s'|s,a)$ and $r(s,a)$; that is, the

model of the environment. After a model has been approximated, we could use a MDP-solving algorithm such as Bellman's value iteration algorithm or Howard's policy iteration algorithm to derive a policy.

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

Deterministic vs. stochastic policy

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

On-policy vs. off-policy

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

Monte-Carlo vs. temporal difference

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

Reinforcement learning – Justification

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

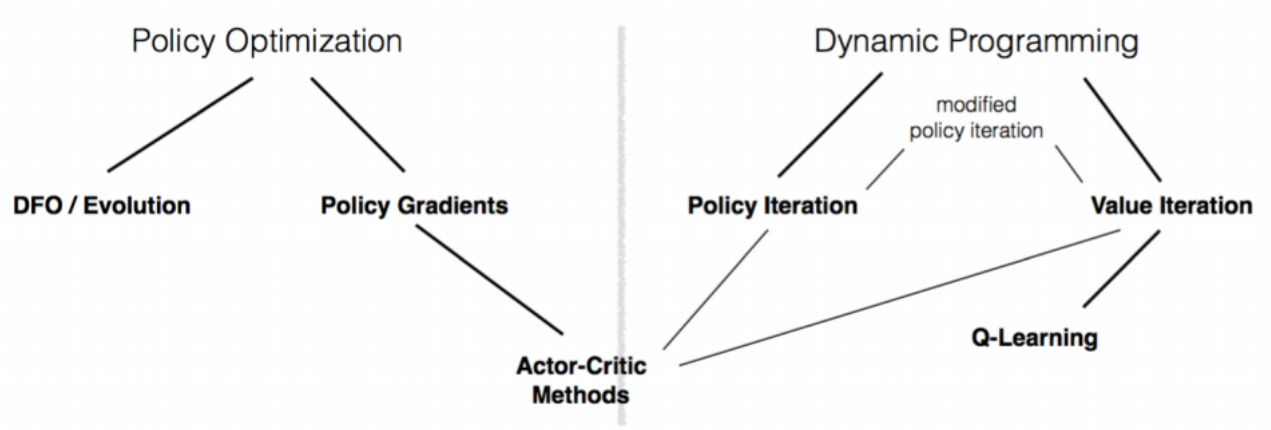


Figure 2.3: Classes of reinforcement learning algorithms. Illustration borrowed from presentation by Peter Abbeel of UC Berkeley.

2.3.2 Reinforcement Learning Algorithms

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

Simplest Approach – Brute Force Search

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

Policy Gradient

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

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

Likelihood Policy Gradient

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

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

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

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

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

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

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

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

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

sampling m trajectories under policy π_θ:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Q-learning

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

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

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

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

$Q(s, a)$ is from how it should be given what we know about the next state and the reward we received this time-step. This is the *temporal difference* from state s to s'; also known as 1-step temporal difference. $0 < \alpha < 1$ is a scalar that determines how quickly the values in the Q-table should be updated based on the temporal difference. When this equation is applied over several iterations, the Q-values in the Q-table will converge values that accurately predicts the return from taking an action in a given state.

Algorithm 1 Q-learning algorithm

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

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

2.3.3 Deep Reinforcement Learning

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

Deep Q-Learning

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

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

Algorithm 2 Deep Q-learning with experience replay

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

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

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

Algorithm 2 shares several similarities to the original Q-learning algorithm. We are still working with an ϵ-greedy policy to encourage exploration, and we are still calculating 1-step temporal differences. We also require discrete action-spaces, however the state-space can be continuous or high-dimensional. The method scales linearly with the state-space, as opposed to Q-learning,

which scales exponentially.

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

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

In their experiments, they found it to be necessary to do prepossessing of the input images, denoted as $\phi(s)$ in Algorithm 2. This prepossessing step transforms the 210×160 128-bits images into 84×84 greyscale images, reducing the computation load of backpropagation. More importantly, the preprocessing also stacks the 4 previous frames, turning the state-space into $84 \times 84 \times 4$. The authors found this modification to be crucial, as the network could not learn to predict motion otherwise. If you can only see a single image of the game of *Pong*, it is impossible to predict what the best action is, because a single frame does not encode the motion of the ball.

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

Asynchronous Advantage Actor Critic

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

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

When compared to earlier deep learning-based actor-critic methods such as Deep Deterministic Policy Gradient [LHP$^+$15], Mnih *et al.* credits most of the success of A3C to the "asynchronous" part of the algorithm. Instead of maintaining a replay memory of previous experiences, A3C works by running multiple environments simultaneously. Each agent has its own copy of the network parameters, and they calculate their respective gradients over several time-steps asynchronously. These gradients are periodically applied to a global copy of the network parameters, and synchronized across the parallel agents. The benefit of training with parallel agents rather

than replay memory is that all the samples we train on will use a recent version of the policy, meaning we are more likely to train samples that are probable under our current policy, thus improving the speed of training. *A2C* is the synchronous variant of A3C, and OpenAI has shown that it has equal to or better convergence properties than A3C [Ope17]. In A2C we run the environments synchronously to gather the samples we use in a minibatch. An advantage of this is that we can utilize the GPU for performing batched updates with large batch sizes. Most things that we will be referring to regarding A3C will also hold for A2C.

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

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

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

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

Conceptually, advantage represents how much better the agent did compared to a *baseline* expectation, as discussed in Section 2.3.2. There are several ways to calculate this advantage, and in A3C, the advantage is calculated by how much better the agent did compared to what the critic expected, as we can see in Equation 2.18. $R_t(\tau)$ is the *return* of the agent when following trajectory τ, and $V(s_t;\theta_v)$ is the critic's prediction, or the baseline. Note that for the return value of a trajectory to be accurate, we need to consider the trajectory as $t \to \infty$.

However, because we are doing TD-learning, we instead bootstrap the last reward value with a discounted prediction of the future return given the last state, $\gamma^k V(s_{t+k};\theta_v)$. This is reasonable, as $V(s_t;\theta_v) \approx R_t(\tau)$. The purpose of training with advantage rather than return, is that it will normalize the reward signal during training. This reduces the variance and stabilizes the training process, as the gradient descent steps are now zero-mean [Wil92].

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

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

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

Proximal Policy Optimization and Trust Region Policy Optimization

Proximal Policy Optimization (PPO) by Schulman 2017 *et al.* [SWD+17] at OpenAI is currently considered the best baseline for reinforcement learning research. It has much better convergence properties than previous reinforcement learning approaches, due to a clever combination of clipping the policy loss, and calculating the loss in terms of a probability ratio instead of

optimizing the policy's log likelihood directly. PPO is an actor-critic method similar to A3C, that combines *trust region optimization* with gradient decent to stabilize training; creating a loss function that guarantees that the agent's policy will improve monotonically. PPO attempts to improve on *Trust Region Policy Optimization* [SLM+15] (TRPO) by Schulman *et al.*, which formulates an objective function that constraints the update step within some pessimistic lower-bound called a *trust region*.

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

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

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

reformulating the above objective as an unconstrained loss function:

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

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

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

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

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

Where ϵ is a hyperparameter that determines how much the new policy can diverge, per update, from the old policy in the direction of improved policies; in other words, the size of the trust region (see Figure 2.4.) The min in the objective computes minimum of the unclipped and clipped objective. The unclipped objective is the regular $L^{IS}_{\theta_{old}}$ loss, and the clipped objective clips the probability ratio r_t to the interval $[1-\epsilon, 1+\epsilon]$ to ensure conservative changes. The min term makes it so that whenever the new policy is advantageous, that is, if $A > 0$ && $r_t(\theta) > 1$ or $A < 0$ && $r_t(\theta) < 1$, we constraint our updates by the clipped objective. Otherwise, if the new policy is detrimental, that is, $A < 0$ && $r_t(\theta) > 1$ or $A > 0$ && $r_t(\theta) < 1$, we will push the

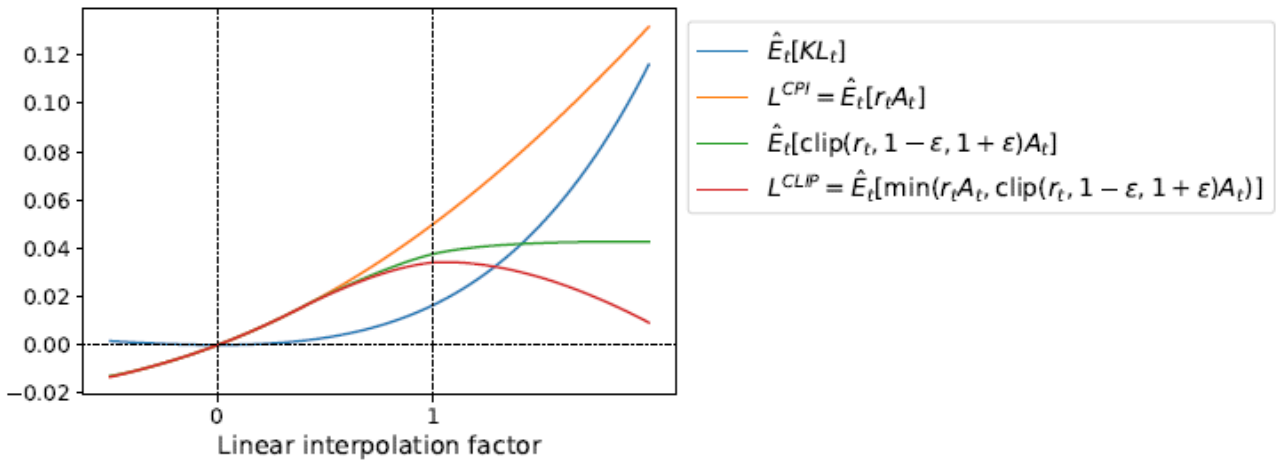


Figure 2.4: Shows the response of the different loss functions as policy θ is interpolated with θ_{old}. Notice how $L^{CLIP} \to 0$ the more θ deviates from θ_{old}.

current policy parameters towards the parameters of the previous policy, essentially reverting changes of the previous update. The authors of PPO also tried to use a KL-divergence penalty with the loss function instead, but found the clipped loss to give overall higher return in their experiments.

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

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

Algorithm 3 PPO, Actor-Critic Style

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

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

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

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

2.3.4 Reinforcement Learning for Self-Driving Vehicles

Most research in reinforcement learning we have looked at so far have primarily focused on solving video games and robotics' locomotion problems. Thus far there is quite limited research in use of deep reinforcement learning for autonomous driving, perhaps due to the fact that it is hard to safely train a reinforcement agent in the real world, since such an agent needs to explore the environment to learn. There is, however, recent notable work done by Kendall *et al.* in their 2018 paper titled *Learning to Drive in a Day.* In their work they were able to teach a full-scale autonomous car to reliably follow a country-side road with only a single monocular front-facing camera, and only needing to train it over a handful of epochs.

Applying a deep reinforcement learning model on a self-driving vehicle is not trivial. As we discussed in Section 2.3.1, reinforcement learning solves Markov decision processes. This entails that the task can be modeled as a process where we have an agent that takes actions in the

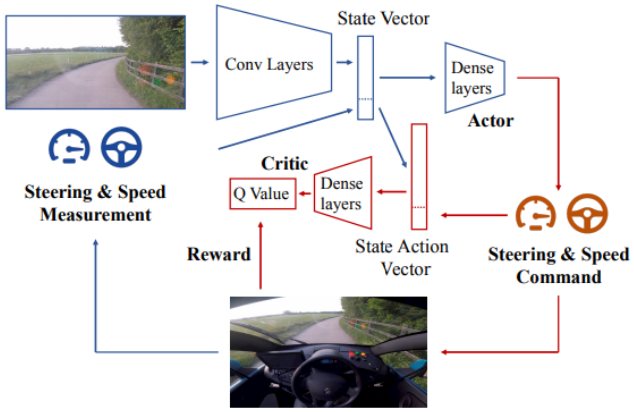


Figure 2.5: Kendall's *et al.* *Learning to Drive in a Day* model. This is an actor-critic based reinforcement learning model that learns to output steering and speed given a monocular input image, and given the vehicles current steering and speed measurements.

environment, and receives rewards form its actions. In practice, this means that we need to design a reward signal that will teach the agent to solve the problem at hand. In the case of lane-following, we can imagine this reward signal representing the car's offset from the center of the lane. However, this approach is limited in scale, as the center point of the lane may not be obvious on various types of roads. Instead Kendall *et al.* define reward as the forward speed of the car, and terminate the episode when the car going off the road (termination signal is given by a human safety driver.) Defining the reward this way will encourage the car to attempt to cover as much distance as possible, because veloicy $*$ time $=$ distance. With this formulation of the reward signal in the lane-following problem, they apply the actor-critic method DDPG [LHP+15] out-of-the-box with no task-specific modifications.

Figure 2.5 shows the model they used. The action space, or output, of the model is a continuous two-dimensional vector representing the steering angle, and a setpoint speed in km/h. It is interesting to note that outputting the angle and speed directly, and letting the controller manage turning and the throttle, will most likely smooth out noise coming from the network. In their experiments they tried to use both regular convolutional layers and a Variational Autoencoder (VAE) to encode image information. They found that using the Variational Autoencoder to encode the images, vastly improved the performance of the model. Similar to earlier work, a simulator was adopted to tune the hyperparameters and to verify that their model works in a simulated environment. The final model was able to learn to follow a 250m

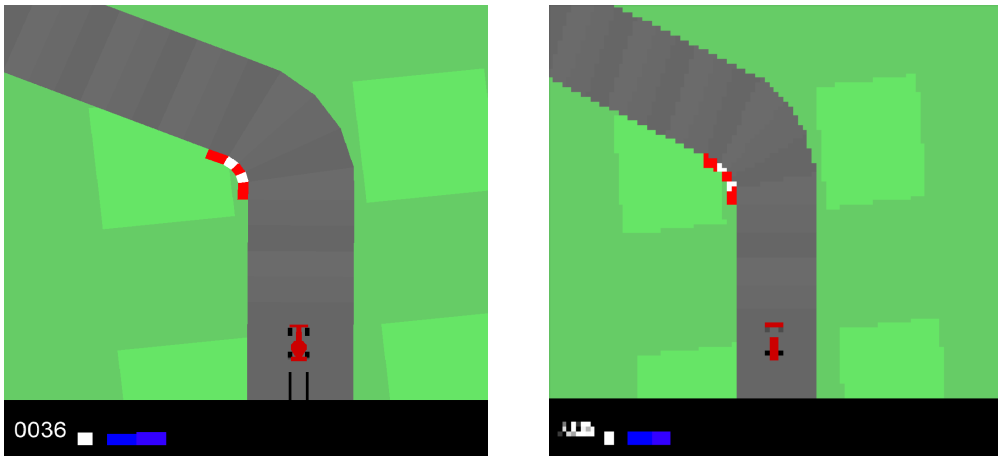
road in 11 epochs, or 15 minutes in real-time.

Chapter 3

Proximal Policy Optimization in Driving-Like Environment

3.1 Introduction

For the practical part of this report, we have implemented and evaluated the Proximal Policy Optimization (PPO) method in a driving-like environment. The authors demonstrated that PPO learns to solve a wide range of continuous control tasks in the Multi-Joint dynamics with Contact (MuJoCo) environment. They compared their algorithm to other state-of-the-art reinforcement learning algorithms in the continuous domain, such as A2C with and without a trust region (Section 2.3.3), Cross-Entropy Method (CEM), Policy Gradient (Section 2.3.2), and TPRO. In almost all their experiments, PPO showed to converge faster while also reaching a greater final score. PPO is therefore considered the present-day baseline for continuous control reinforcement learning problems, and appears to be the most suitable, general-purpose reinforcement learning algorithm that could learn to drive a car.



(a) (b)

Figure 3.1: (a) Screenshot of the CarRacing-v0 environment as rendered on the screen. (b) Example of the 96x96 state space as seen by the agent.

3.2 Environment

As discussed in Section 1.3, there exists a wide range of open source simulators ready to be used for the task of reinforcement learning. CARLA seems to be the most complete one in terms of emulating a real-life urban driving environment. However, it is not a very light weight environment due its focus on high-fidelity graphics, and for running in the somewhat expensive game engine of Unreal Engine 4. CARLA's environment is also pretty high in complexity; with traffic lights, lane lines and potentially other cars and pedestrians, making it harder to test simple hypotheses.

Instead, we opted for using OpenAI's *CarRacing-v0* to test the models. CarRacing-v0 is an environment available through OpenAI's *gym* – a suite of reinforcement learning environments.

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

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

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

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

Figure 3.1(b) show an example of the state space of CarRacing-v0. The state space of the environment consists of a 96x96 pixel RGB image, where the speedometer, ABS sensors, steering angle and break sensors are encoded into the image itself as vertical and horizontal bars in the lower 96x12 pixels of the image. The idea is that by encoding it in the image, we will only need to train a convolutional model, as the agent will learn to interpret these measurement directly from the image. However, this seems a bit limiting for some of the experiments we wanted to conduct, so we opted to create a modified version of the environment that we named *CarRacing-v1* that additionally outputs the following measurements:

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

3.3 Implementation Details

As discussed in Section 2.3.3, PPO is a model-free, policy gradient based reinforcement learning algorithm. In this section, we will lay out the details of our implementation of PPO for CarRacing-v0.

3.3.1 Setup

The implementation is written in Python 3.6 with TensorFlow 1.8. The simulations and the training were run on a single Nvidia GTX 970, which, admittedly, is a relatively old graphics card with only 4 GB of video memory. This directed some of the choices when it comes to choice of architecture and hyperparameters. It was also ran on a 4-core CPU, which is important for running environments in parallel efficiently. Complete code can be found at *https://github.com/bitsauce/CarRacing-v0-ppo*.

3.3.2 Algorithm

Algorithm 3 shows the general outline of the PPO algorithm. Recall that PPO works by optimizing current policy π_θ with respect to its deviation from the previous policy $\pi_{\theta_{old}}$. The training data for an optimization step is sampled by running a single or multiple environments in parallel with the old policy $\pi_{\theta_{old}}$. For running multiple environments, we used the OpenAI gym class *SubprocVecEnv*, which runs N environments in parallel, each on a separate thread. We use T-step temporal difference, where T denotes the number of steps we take along a trajectory before we calculate advantage estimates and update the policy. Advantage estimates are computed with the generalized advantage estimation equation (Equation 2.25), where λ is an interpolation factor that practically becomes a trade-off between bias and variance in the advantage estimates [SML+15], and γ is the reward discount factor. The last values used the TD-return calculations are bootstrapped with the old critic's prediction of the values in the last states of the trajectories, as shown in Equation 2.18. Once T steps have been computed for N

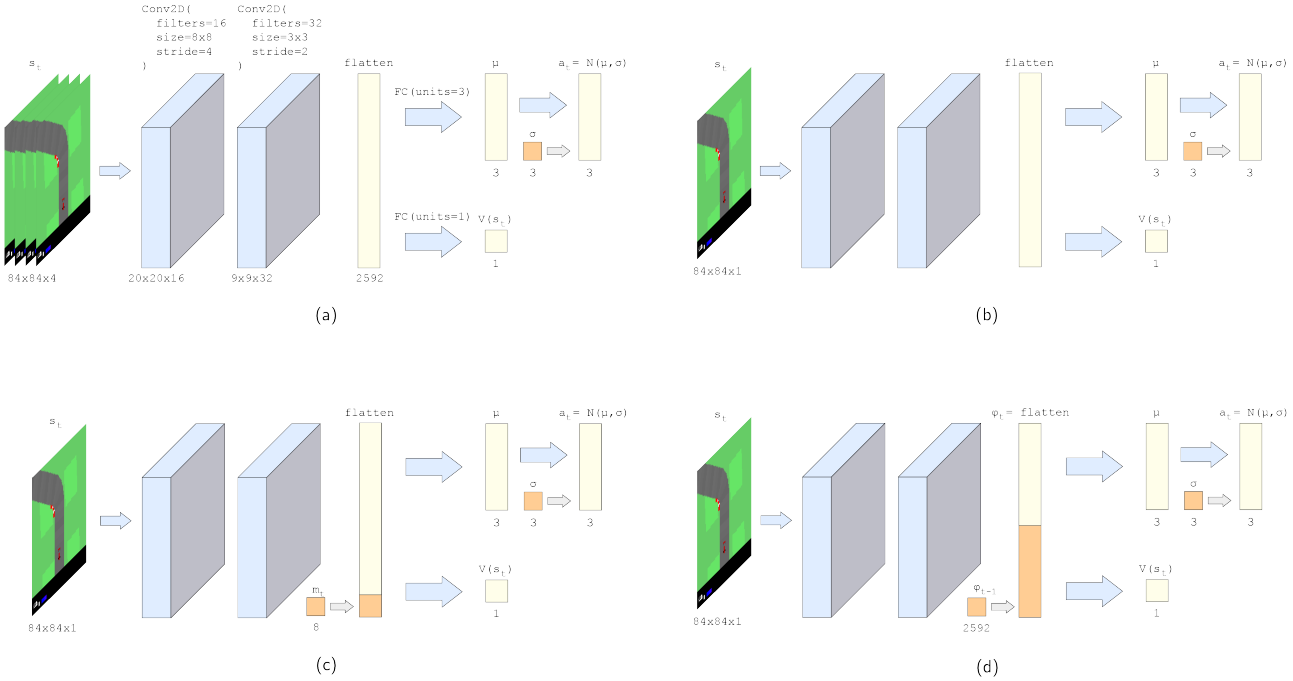


Figure 3.2: Shows the 4 CarRacing-v0 models that were tested in the experiments. (a) Is the typical frame stack model with 4 stacked frames. (b) Same as (a) but with only one frame as input. (c) Car measurements are concatenated with the latent features ϕ. (d) Recurent model where previous step's latent features, ϕ_{t-1}, are concatenated with the current latent features, ϕ_t. Note that convolutional and fully-connected layers have the same kernel sizes and units in every model.

environments, we end up with $N*T$ samples which we stochastically sample minibatches of size $M \leq NT$ from, for K number of epochs. These minibatches of $(s_{n,t}, a_{n,t}, R_{n,t}, \hat{A}_{n,t})$-tuples are feed through an actor-critic network that optimizes the parameters θ with the *Adam* gradient decent optimizer according to the $L^{CLIP+VF+S}$ loss function as shown in Equation 2.24. Recall that the $L^{CLIP+VF+S}$ introduces a clipping parameter ϵ for the L^{CLIP} loss, and a value loss scaling factor α and entropy loss scaling factor β. In the following section, we will go into further details on how the various models that we tested were constructed.

3.4 Experiments

Frame stack model

Stacking sequential frames is the most straight-forward and most common way to represent the state space for a reinforcement learning agent that take images as input. As we discussed

in Section 2.3.3, Mnih *et al.* found it to be essential to stack 4 sequential to solve Atari-based environments with Deep Q-learning; because a stack of sequential frames store temporal information. In this model we also opted to create a frame stack of 4 sequential frames, and we use this as the input.

Image Preprocessing: When we stack 4 96x96 RGB images on top of each other, the resulting state space becomes quite high dimensional ($96 \times 96 \times 3 \times 4 = 110,592$ state values). In order to reduce the size of the state space, we opted to crop and convert the 96x96 RGB frames into 84x84 greyscale images, giving us a state space of $84 \times 84 \times 3 = 27,648$ state values. The cropping removes 6 pixels from the left and right sides because these pixels typically contain information about objects that are far away from the agent and are therefore not relevant to the agent at the moment, and we also crop the lower 12 pixels to from the image because these pixels encode the dashboard parameters, and should not be necessary when these parameters can be inferred directly from the sequential stack of frames.

Network Architecture: The network is a typical convolutional neural network with a fully-connected layer for the critic output and a fully-connected layer for the actor output. The convolutional part of the network architecture was inspired by a project report titled *Reinforcement Car Racing with A3C* [JML] by students at Stanford University's course CS234. In the report, they claim to have tested networks with 2 up to 7 convolutional layers of varying filter sizes and strides. In the end they found that deeper networks did not help the agent learn. This is probably because state space is not very complex to begin with – it consists of mostly straight lines and simple solid-colored geometric shapes as shown in Figure 3.1(b) – so a deeper network will provide little benefit. Another consideration is that deeper networks are harder to train and take longer to converge. Since is seems shallower architectures are faster to train and will preform better or similarly to deep architectures, we opted for the 2-layer architecture shown in Figure 3.2(a). The first convolutional layer has 16 8x8 filters with stride 4, while the second layer has 32 3x3 filters with a stride of 2. They both use *leaky ReLU*, $f(x) = x$ if $x > 0$ else $0.01x$, as their activation functions. Note that the filters are quite big compared to the filters in deep architectures, such as ResNet [HZRS15]. This is due to the fact that shallow convolutional networks need bigger filters to increase their receptive fields.

After the convolutions, the resulting feature maps are flattened and shared between two different fully-connected layers. The first fully-connected layer represents the agent's actor. It has 3 output units, each of which represents the mean of one of the agents 3 actions, (γ, a, b). Since CarRacing-v0 is an environment with a continuous action space, we represent the actions by a multivariate normal distribution, $a_i \sim \mathcal{N}(\mu_i, \sigma_i)$, where i represent the i$^{\text{th}}$ action in the action space. Recall also from Section 3.2 that each of the agent's actions are limited to a predetermined range of valid values. As a result, it seems reasonable to scale these action means to their range of possible values. This is done through the following transformation:

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

Where o_i is the raw output value of the i$^{\text{th}}$ neuron. By passing the raw outputs of the fully-connected layer through the hyperbolic tangent function (tanh) we end up with values in the range $[-1, 1]$. By adding one and dividing by two our values are now in $[0, 1]$ range. Finally, we do a linear interpolation between the i$^{\text{th}}$ actions min and max value, resulting in $a_i^{(min)} \leq \mu_i \leq a_i^{(max)}$. The standard deviations for each action, σ_i, are trainable parameters in the network. In A3C they predicted the standard deviation alongside the mean through another fully-connected layer. However, we found this formulation to produce erratic agent behaviour, and opted for having a trainable variable for each action's standard deviation instead. Note that these trainable variables, which we will call $\sigma^{(\log)}$, actually represent the logarithm of σ, so to retrieve σ we do $e^{\sigma^{(\log)}} = e^{\log(\sigma)} = \sigma$. This is done to ensure that the standard deviation is never negative, and that the standard deviation will increase faster when more exploration is necessary. Using this formulation, we initialize $\sigma_i^{(\log)} = 0$ for each action i, meaning $\sigma_i = e^0 = 1$ at the start. The weights for the action layer are initialized with variance scaling [GB10] with a factor of 0.1 to lower the chances of starting with some weights that influence the policy too much.

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

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

Table 3.1: Hyperparameters used in the experiments.

represent any possible value that R_t might have, in other words $-\infty \leq V(s) \leq \infty$.

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

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

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

Single frame model

The single frame model is the same as the frame stack model except that the state is represented by the current frame only (see Figure 3.2(b)). We wanted to train this model to have a baseline to compare the rest of the models to.

Single frame with measurements model

This model is also a single frame model, however, the speed, steering angle, ABS and angular velocity measurement of the vehicle, m_t, are concatenated with the latent space vector ϕ, as shown in Figure 3.2(c). It is reasonable to believe that this state representation should encode

as much information as the frame stack variant, since stacking frames simply aims to let the network infer temporal properties such as the aforementioned ones from the differences in the frames.

Recurrent model

For the recurrent model, we wanted to see if the agent performs better if it can utilize temporal information beyond 4 frames. In this model, we concatenated the latent space vector ϕ_{t-1} from the previous step to the latent space vector ϕ_t from the current step, as seen in Figure 3.2(d).

Non-scaled actions, frame stack model

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

Chapter 4

Results

4.1 Model Comparisons

Figure 4.1 shows the cumulative reward of each of the models over number of epochs of training. The evaluation step was run every 200 epoch during training. Evaluation involves using the current policy over a complete run from start to end on one agent and recording its cumulative reward. Note that these measurements are quite noisy because we are only evaluating the performance over a single trajectory. Because of this, a sliding window of ± 5 was applied to calculate the means and standard deviations shown in the figure.

Frame stack model

Frame stacking is a tried and true method in deep reinforcement learning, and using frame stacking in CarRacing-v0 appears to be no exception. This model achieved the highest score out of all the model – a score of 896.6 in its best run – and got an average score of 726.4 over the last 100 runs, as seen in Table 4.1. While the model did not "solve" the CarRacing-v0 environment as far as OpenAI's criteria of achieving an average of 900 reward over 100 trials is concerned, it does reliably stay on the road and is able to make sharp turns while maintaining a constant speed throughout the run without any erratic behaviour. This, and

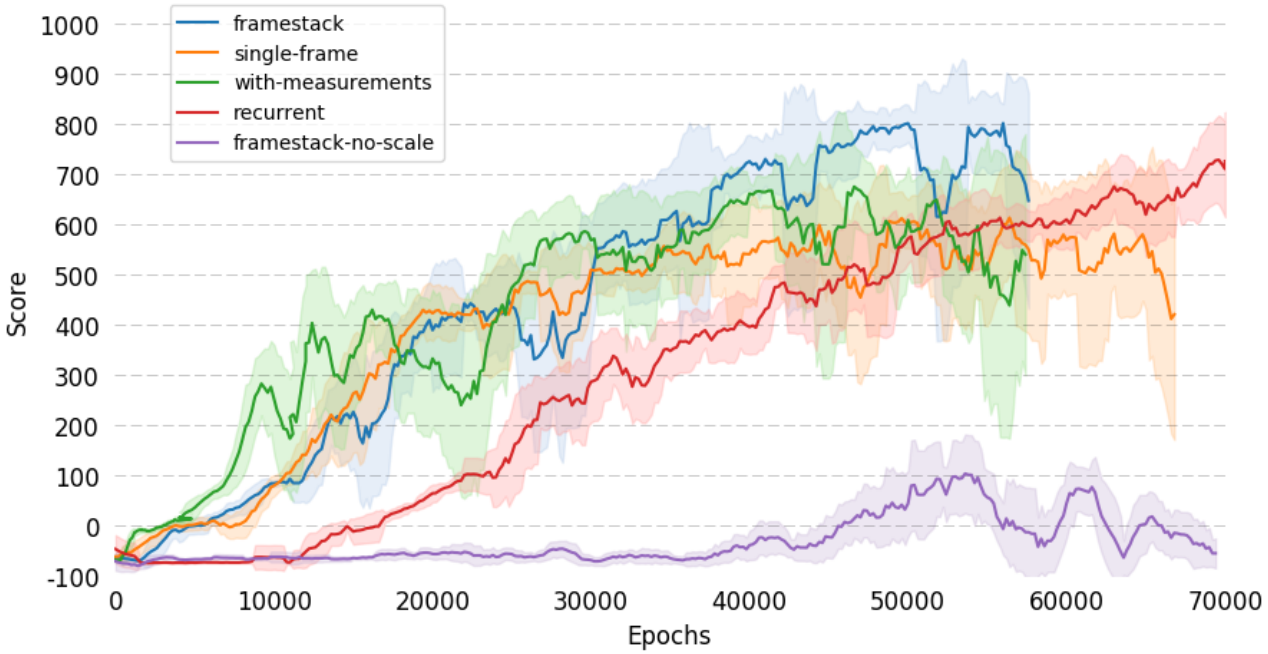


Figure 4.1: Shows the cumulative reward of the models over number of epochs. Note that a sliding window of ±5 was applied to smooth out the graph.

most of the other models, were even able to *recover* from bad states, which is a property that is very relevant to self-driving vehicles. We believe that this model could learn to solve the environment if we make it a bit deeper by adding more convolutional or fully-connected layers; the current network is relatively shallow to favour training time. A video of the results can be found at *https://youtu.be/8X_LSy4TF84*, demonstrating the best run of this agent, in addition to showing runs of all the 5 models over 50k epochs, and an example of how the agent is able to recover from bad states.

Single frame model

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

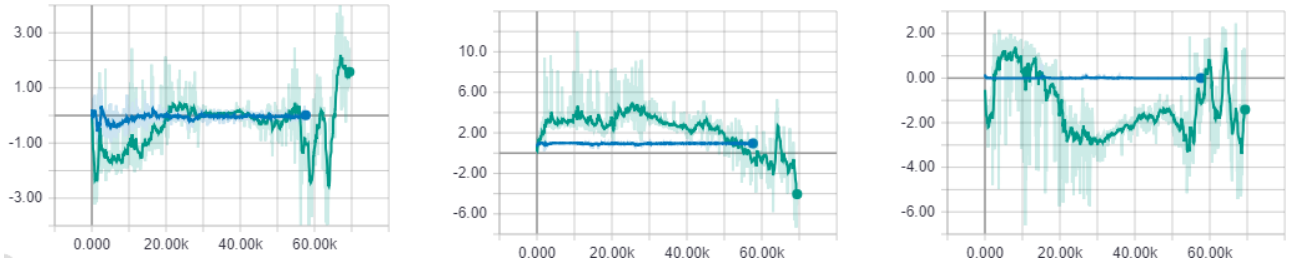
Single frame with measurements model

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

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

Recurrent model

The recurrent model had a slow start, but eventually achieved similar scores to that of the frame stack model. Looking at Figure 4.1 and Table 4.1, we may observe that this model has substantially less variance in its performance measures compared to the other models. Having low variance during training and testing is a very desirable property, and a particularly important feature when it comes to training self-driving vehicles. This difference may originate from the fact that we have doubled the number of parameters in our fully connected layers, because the size of the shared feature vector is doubled after concatenation. Regardless, we find this model to be the most interesting model to investigate further.



(a) (b) (c)

Figure 4.2: Shows the action means for (γ, a, b) over number of epochs. The blue graph is the frame stack model and green graph is the non-scaled frame stack model. (a) shows the steering angle γ, (b) shows the acceleration a, and (c) shows the breaking b.

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

Table 4.1: Mean and standard deviation of the cumulative rewards obtained by 100 evaluation runs after convergence.

Non-scaled actions, frame stack model

The non-scaled action mean version performed substantially worse than the scaled one given the same amount of training time. The agent did not learn anything for the first 40,000 epochs because the action means, particularly the steering angle, diverged too far away from the space of valid actions as seen in Figure 4.2. We believe that the agent would reach the same performance as the scaled variant over time, but it is obviously slower to train and is therefore not favorable.

OpenAI baseline comparison

OpenAI's PPO implementation was ran for comparison, with default parameters except for the network model, where we chose to use a convolutional neural network instead of a multi-layered perceptron for a more accurate comparison. The parameters are, by default, set to the same parameters that they used to solve continuous control problems in the MuJoCo environment (Table 3 from [SWD+17].) If these parameters can solve MuJoCo control problems,

it is reasonable to assume that they should work for CarRacing-v0 as well. To summarize the most important differences between our models and theirs: their model only runs a single environment over 1 million frames before terminating, the horizon is $T = 2048$ and number of epochs is $K = 10$ with minibatch size $M = 64$, meaning that we ended up running $\lfloor \frac{\#\text{steps}}{T} \rfloor * K = \lfloor \frac{10^6}{2048} \rfloor * 10 = 4880$ epochs in total.

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

Chapter 5

Conclusion

In this report, we have looked at where the current state-of-the-art is in reinforcement learning, and how reinforcement learning is currently used for autonomous driving. Imitation learning currently stands as the most prevalent approach when it comes to training autonomous vehicles to drive. However, with recent advancements in reinforcement learning such as PPO and Kendall's *et al.* paper *Learning to Drive in a Day*, it seems that reinforcement learning is starting to become more viable in the context of autonomous driving.

5.1 Contributions

The contribution in this precursory study comes in the form of experimenting and comparing different neural network architectures for a PPO-driven agent. In these experiments, we showed, first and foremost, that scaling the action mean values to their respective action spaces before sampling the normal distribution is a good idea, as this makes it much more likely for the agent to pick action that lie inside the action space. When an agent picks actions outside the action space, those action will be clipped back into the of range the action space, making it so that policies that tend to pick actions outside this space will end up exploring very few unique trajectories. This makes the training go much slower. Note that the action means are not scaled in OpenAI's official implementation of PPO [DHK+17] to the best of our knowledge.

Another contribution comes in the form of a short comparison study of the effect of using different kinds of architectures for the environment CarRacing-v0. In this study, I found that the frame stack model is the model that achieves the best cumulative rewards the fastest. Adding the car's measurements to a single frame model without any additional layers or without any attention mechanism turned out to be unsuccessful despite initial predictions. The recurrent model is probably the most interesting model to explore further, as it achieved similar scores to the frame stack model while also having substantially less variance than any of the other models.

5.2 Future Work

There is a lot of interest and ongoing research in creating general-purpose reinforcement learning algorithms. Some specialized methods are used in robotics, however there has been very limited research in the use of reinforcement learning for autonomous vehicles. Here are the areas of future work that I identified throughout this project:

Reinforcement Learning for Autonomous Driving

As discussed in Section 2.3.3, PPO is currently considered the state-of-the-art baseline for general purpose reinforcement learning. In the CARLA benchmark by Dosovitskiy *et al.* [DRC+17] they compared three different approaches, one of which was imitation learning based, and another one reinforcement learning based. In their reinforcement learning benchmark they used A3C, and found it to do fairly poorly compared to the other two. There could be several reasons for this, *e.g.* using a too shallow state space of 84x84 pixels; so it would be worth to try and improve this metric by evaluating various models and different reward formulations. Given the recent successes in reinforcement learning for autonomous vehicles by Kendall *et al.*, it would be interesting to explore if taking some ideas from Kendall, such as their reward function, would yield better results in CARLA. Additionally, it would be interesting to see if PPO can perform better than DDPG in this scenario. Taking a cue from Codevilla's *et al.* conditional imitation

learning, it would also be interesting to explore PPO with conditional sub-policies for different types of maneuvers.

Environments for Reinforcement Learning-Based Autonomous Driving

One problem with reinforcement learning research for autonomous vehicles is the lack of simple and varied environments. CARLA, along with AirSim, are fully-fledged car simulators that simulate real-world physics and appearance to a great extent, while simultaneously trying to maintain real-time frame rates. A consequence of this is that these environments are somewhat expensive in their computational loads, and it can be hard to run multiple such environments in parallel. OpenAI's CarRacing-v0 provides some of the same challenges we encounter in real-world driving, such as road following and acceleration management, however, it is nowhere near complex enough to be a valid environment to verify reinforcement learning algorithms for real-world driving. CARLA is a lot better in this regard, as it features two-way traffic with turns, traffic lights, speed limits, and more. However, the road scenarios we find in CARLA are still limited compared to what we might see in the real-world. There are no highways, no multi-lane roads, every intersection is a T-section with traffic lights, and there is a lack of complicated signage such as STOP and YIELD, etc. The lack of diversity makes it difficult to test certain ideas, such as reinforcement learning agents with several sub-policies.

Therefore, I propose that an interesting project would be to extend CARLA, by either creating procedural generation algorithms for cities, or to create algorithms that recreate roads from the real-world map data. The latter idea sounds more interesting to me, as there already exists publicly available datasets with highly detailed road data, such as OpenDRIVE [DG06]. It would also be possible to alternatively use this data to make a light-weight simulator with complex road scenarios that can be used for reinforcement learning.

Expand on General-Purpose Reinforcement Learning Methods

My final suggestion is simply a question that sprung up from curiosity; would it be fruitful to combine policy gradient based agents with genetic algorithms to make the training converge faster? Genetic algorithms are great at finding a local optimum quickly, while reinforcement learning tends to find better optimums. An approach that combines both of these approaches seems like it could result in agents that learn faster while also learning policies that are as good as typical policy gradient.

References

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

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

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

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

[DG06] Marius Dupuis and Han Grezlikowski. OpenDRIVE®-an open standard for the description of roads in driving simulations. In *Proceedings of the Driving Simulation Conference*, pages 25–36, 2006.

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

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

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

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

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

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

[Int14] SAE International. Automated driving levels of driving automation are defined in new sae international standard J3016. 2014.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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