

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

# Multi-Agent Deep Reinforcement Learning for autonomous driving

Gonçalo Moreno



Programa de Mestrado Integrado em Engenharia Informática

Supervisor: Pedro M. d'Orey

Co-Supervisor: Zafeiris Kokkinogenis

October 20, 2020



# **Multi-Agent Deep Reinforcement Learning for autonomous driving**

**Gonçalo Moreno**

Programa de Mestrado Integrado em Engenharia Informática

October 20, 2020



# Resumo

Este trabalho estuda diferentes métodos e técnicas de resolução do complexo problema de condução autónoma focando-se principalmente em aprendizagem por reforço, um método que consegue criar um agente com uma condução mais eficiente e segura. Isto é em contraste a aprendizagem por imitação muito usada em condução autónoma que traz desvantagens como a necessidade de grandes quantidades de dados. Com a análise de diferentes métodos feita propõe-se um novo método de aprendizagem, baseado em aprendizagem por reforço, mas com a adição de um componente de aprendizagem distribuída e outro de troca de mensagens que permite cooperação entre agentes. Com esta abordagem os agentes conseguem melhor desempenho que os métodos anteriores quando testados com uma bateria de testes. Todos os resultados são validados no simulador de condução autónoma chamado Carla.



# Abstract

Autonomous driving is the next revolution in transportation, promising to make road transportation safer and more efficient. Recently it has seen a big increase in research and development, this is partly due to the rise in processing power and advances in machine learning and deep learning. Typically a large part of self-driving is done using supervised learning or behaviour cloning, requiring massive datasets containing expert driving experiences. Reinforcement Learning provides a different approach to creating an autonomous driving agent, it still doesn't require the explicit programming of driving rules or law and due to its nature it doesn't require labelled expert data. Different methods of reinforcement learning and different configurations are studied and a final architecture is proposed and analysed. It consists of a ResNet encoder and a variant of a model-free algorithm, SAC (Soft-Actor-Critic). This approach additionally leverages a multi-agent architecture, adding the advantages of cooperation, faster sample collection and the ability to better utilize computational resources. The final agent is put through a battery of driving tests and also it is heavily scrutinized, by observing what parts of the input causes its action. The results are validated in a simulator called Carla.





# Acknowledgements

I would like to first thank my family for supporting me through college.

Next Richard S. Sutton and Andrew G. Barto for their incredible and insightful book on Reinforcement Learning.

The team of GPULab for helping this work with top of the line hardware resources. Amazon and Google for giving hundreds of dollars worth of cloud computing to this work.

Finally I would like to thank my supervisors for helping and guiding me throughout this entire project. I am also grateful to my host institution *Instituto de Telecomunicações* for the provided financial support and for granting access to powerful computing resources.

Gonçalo Moreno



*“Once you trust a self-driving car with your life, you pretty much will trust Artificial Intelligence with anything.”*

Dave Waters



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Context and Motivation . . . . .	1
1.2	Objectives . . . . .	2
1.3	Document Structure . . . . .	2
<b>2</b>	<b>Literature Review</b>	<b>3</b>
2.1	Autonomous Driving . . . . .	3
2.2	Preliminaries . . . . .	5
2.2.1	Machine Learning approaches . . . . .	5
2.2.2	Deep Learning with Artificial Neural Networks . . . . .	6
2.2.3	Batch Normalization and Dropout . . . . .	11
2.2.4	CNN Visualizations . . . . .	12
2.2.5	Recurrent Neural Networks . . . . .	12
2.2.6	Autoencoder . . . . .	13
2.2.7	Geometric Deep Learning . . . . .	15
2.3	End-to-End Training . . . . .	16
2.4	(Deep) Reinforcement Learning . . . . .	17
2.4.1	Markov Decision Processes . . . . .	18
2.4.2	Partial Observability . . . . .	19
2.4.3	Policy and value function. . . . .	20
2.4.4	Value Iteration and Policy Iteration . . . . .	22
2.5	Off-policy and On-policy . . . . .	24
2.5.1	Deep Reinforcement Learning . . . . .	26
2.5.2	DQN . . . . .	27
2.5.3	Policy Gradients . . . . .	28
2.5.4	Actor-Critic methods . . . . .	30
2.5.5	DDPG . . . . .	30
2.5.6	PPO . . . . .	31
2.5.7	SAC . . . . .	32
2.5.8	Curriculum Learning . . . . .	33
2.5.9	Transfer Learning and Domain Adaptation . . . . .	34
2.6	Multi Agent Reinforcement Learning . . . . .	34
2.6.1	A3C . . . . .	34
2.7	Self-Driving Cars . . . . .	35
2.7.1	Typical vehicle . . . . .	35
2.7.2	Training . . . . .	37
2.8	Related Work . . . . .	37

<b>3</b>	<b>Agent Framework</b>	<b>43</b>
3.1	Proposed Framework . . . . .	43
3.1.1	CARLA simulator . . . . .	43
3.1.2	Open AI Gym . . . . .	45
3.1.3	Main New Functionalities . . . . .	46
3.2	Training a RL Agent . . . . .	46
3.3	Method . . . . .	48
3.4	Evaluation framework . . . . .	49
3.4.1	Carla Scenarios . . . . .	49
3.4.2	Implementation . . . . .	50
3.4.3	Hardware Used . . . . .	50
3.4.4	VAE . . . . .	51
3.5	Results . . . . .	52
3.5.1	Hyperparameter Tuning . . . . .	52
3.5.2	VAE . . . . .	52
3.5.3	RL algorithms . . . . .	54
<b>4</b>	<b>Proposed Architecture</b>	<b>59</b>
4.1	Overview . . . . .	59
4.2	Distributed Reinforcement Learning . . . . .	59
4.2.1	Message Parsing . . . . .	60
4.3	Two-phase Multi-agent RL Pipeline . . . . .	60
4.3.1	Phase I - Behaviour Cloning . . . . .	60
4.3.2	Phase II - Reinforcement Learning . . . . .	61
4.4	Implementation of Phase I - Behaviour Cloning . . . . .	62
4.4.1	Data Collection . . . . .	62
4.5	Implementation Phase II - Multi Agent Reinforcement Learning . . . . .	63
4.5.1	Message Parsing . . . . .	63
4.5.2	Training Environment . . . . .	63
4.6	Experimental Setup and deployment . . . . .	65
4.6.1	Scenarios . . . . .	65
4.6.2	Inputs . . . . .	65
4.6.3	Encoder . . . . .	66
4.7	Results . . . . .	66
4.7.1	Deployment . . . . .	66
4.7.2	Hyperparameter Tuning . . . . .	67
4.7.3	Behaviour Cloning . . . . .	67
4.7.4	Multi Agent RL . . . . .	69
<b>5</b>	<b>Conclusions and Satisfaction of the objectives</b>	<b>71</b>
5.1	Future Work . . . . .	72
<b>A</b>	<b>Appendix</b>	<b>75</b>
A.1	Loss function for VAE . . . . .	75
A.2	Neural Network Parameters . . . . .	75
A.3	RL Algorithms Results . . . . .	77
A.3.1	PPO . . . . .	77
A.3.2	DQN . . . . .	78
A.3.3	Reinforce . . . . .	78

*CONTENTS*

xi

A.3.4 Scenarios . . . . .	79
A.4 Sensor processing . . . . .	80
A.5 Hyperparameters . . . . .	81
<b>References</b>	<b>83</b>





# List of Figures

2.1	Example of a Deep Neural Network. From [5]	7
2.2	CNN's convolution filter (middle matrix) computing its output (yellow matrix), filter slides through input data (blue matrix), first by the left image then right. From [127]	10
2.3	CNN's pooling layer, in this example we can see max pooling with a 2x2 filter and stride of 2. From [114]	10
2.4	Diagram of a CNN with an image input. From [70]	11
2.5	The loss surfaces of ResNet-56 with and without skip connections working with the CIFAR-10 dataset. From [77]	11
2.6	CNN's visualization, the network predicted the correct class and the right image shows where it "looked". From [97].	13
2.7	Diagram of a unfolded recurrent neural network. From [136]	14
2.8	Diagram of a autoencoder. From [133]	15
2.9	Diagram of a Variational Auto Encoder. From [62]	16
2.10	Diagram of the graphical model involve in VAE from [133].	17
2.11	Reparameterization trick.	18
2.12	End-to-end training. From [15]	19
2.13	Diagram of a Partial Observable Markov Decision Process.	20
2.14	Diagram of n-step methods, ranging from TD methods to MC ones. From [117]	26
2.15	Atari 2600 used to benchmark DQN and their performance comparison. From [90]	28
2.16	Effects of the baseline, first is updating with regular rewards and the next are different baselines with the third being the better one.	31
2.17	Intuition about SAC, policy network is projected to the orange function.	33
2.18	Diagram and table related to transfer learning. From [98]	35
2.19	Diagram of A3C.	36
2.20	Example of Tesla's input processing (top image) and its latent encoding (the representation of the world). From [119]	38
2.21	Diagram for the agent's input used in [36].	39
2.22	CIRL performance compared to other algorithms. From [79]	40
2.23	Sequential latent representation model diagram from [22]	41
2.24	Diagram of the architecture (top image) and results (bottom image) of [121].	42
2.25	Carla's Leaderboard (cropped) in September 2020. Leader is [21]	42
3.1	Diagram of the extensions done to Carla.	43
3.2	Carla Simulator. From [20].	44
3.3	Overview of a self driving car by Waymo. From [131]	45
3.4	View of the LIDAR data on Meshlab. From [20].	47

3.5	Final fused image of the cameras and LIDAR, from left to right, RGB front, RGB Back, LIDAR Projection. Note the text on the left corner is not part of the input. . .	48
3.6	Overview of training RL Agent with the 2 types of encoders. . . . .	49
3.7	Overview of training VAE . . . . .	51
3.8	Overview of training RL Agent with VAE. . . . .	51
3.9	Results from VAE, input is the top image, reconstruction is the bottom image, reconstructions are blurry and unusable. . . . .	53
3.10	SAC episode cumulative reward. . . . .	54
3.12	Example of SAC agent input, output action (top image) and corresponding integrated gradients. . . . .	55
3.11	SAC losses for all the networks. . . . .	57
4.1	Overview of Distributed RL. . . . .	60
4.2	Diagram of the two step RL algorithm. . . . .	65
4.3	Diagram of the encoder and policy network for the agent. . . . .	66
4.4	Training Loss for behaviour cloning . . . . .	67
4.5	Simple scenario screenshot, agent (blue car) has to travel to the roundabout. SAC+A3C agent wasn't able to reach its destination. . . . .	68
4.6	Example of Behaviour Cloning agent input, output action (top image) and corresponding integrated gradients. The bottom image is the multiplication of the top ones. . . . .	68
4.7	Final reward (top image) receive by one of the agents in multi RL up to 20k steps and policy loss (bottom image). . . . .	69
A.1	PPO rewards in each step. . . . .	77
A.2	DQN rewards in each step. . . . .	78
A.3	Reinforce rewards in each step. . . . .	78
A.4	PPO agent in Control Loss. The patches on the road are where the vehicle losses grip. . . . .	79
A.5	SAC agent in Cut In scenario. The right most vehicle is the agent and he has to keep in his lane while another vehicle (in the left most lane) cuts in front of him. . .	79
A.6	SAC agent in the Turn Right Scenario. The vehicle on the upper left part of the image is the agent and he has to drive through the junction. . . . .	79
A.7	SAC agent in the Turn Right Scenario. The vehicle on the upper left part of the image is the agent and he has to drive through the junction. . . . .	80
A.8	Example of a initial proposed architecture. . . . .	80

# List of Tables

3.1	Results of SAC RL agent. Best result from 5 runs through each scenario. . . . .	56
4.1	Results from the different scenarios tested with SAC+A3C. . . . .	70
A.1	Design of encoder and policy network. . . . .	76
A.2	Comparison of size between models. . . . .	77
A.3	Comparison of size between models. . . . .	81



# Acronyms

AI	Artificial Intelligence
AutoML	Automated Machine Learning
BC	Behaviour Cloning
CNN	Convolution Neural Network
CPU	Central Processing Unit
DL	Deep Learning
DDPG	Deep Deterministic Policy
DQN	Deep Q-Learning
FLOP	Floating Point Operation
GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit
IL	Imitation Learning
LSTM	Long Short Term Memory
MDP	Markov Decision Process
ML	Machine Learning
NN	Neural Network
POMDP	Partial Observable Markov Decision Process
PPO	Proximal Policy Optimization
ReLU	Rectifier Linear Unit
RL	Reinforcement Learning
RNN	Recurrent Neural Network
SAC	Soft Actor Critic
VAE	Variational Auto Encoder



# Chapter 1

## Introduction

### 1.1 Context and Motivation

During the last decades there has been a massive increase in computing power, closely related to the expansion of AI and machine learning techniques, past problems deemed impossible or extremely hard to work on are slowly being resolved. One of those problems is autonomous driving, specially in complex urban scenarios. Specifically, a complex problem to solve for an autonomous agent is to be able to process high dimensionality inputs, such as, RGB cameras or in-vehicle sensors (e.g. LIDAR or Radar) in a real time environment and follow all the driving laws, while being efficient with respect to energy consumption and time to destination. For context, the Driver and Vehicle Standards Agency (DVSA) indicates that the average person takes on average 65+ hours of practising to learn how to drive [81].

According to the Coalition for Future Mobility [35] autonomous vehicles have a wide range of benefits, they could save lives, change the lives of people with disabilities and increase productivity and reduce fuel use and carbon emissions. They are slated to be the next revolution in transportation [37].

In the last decade a number of self-driving startups have been founded. Although there exists little public information, the conventional approach to train an agent is to resort to behaviour cloning or supervised learning 2.8. Although this approach has its merits, it requires massive amounts of pre-labelled driving data from human drivers (which is costly) and might not perform well in specific scenarios with little or no available data. Reinforcement Learning (RL) is another approach to create an agent capable of driving. RL is an area of machine learning that recently has seen a lot of advances, mainly due to the advances in Deep Learning and interest by companies like OpenAI, developing agents that quickly can achieve superhuman performance in a variety of games and challenges [3]. Its one of the three basic machine learning paradigms along side supervised learning and unsupervised learning, it is concerned with creating an agent that chooses actions in order to maximize a reward and it doesn't need labelled data. Reinforcement learning has the big disadvantage of needing an advanced simulator (one that mimics real life challenges encountered by a car), or an environment where it isn't possible for the agent to do any harm.

Additional benefits related to more efficient transportation are possible when an agent is trained in conjunction with others, as a multi-agent system, where behaviours could emerge from the cooperation by the fleet of agents.

## 1.2 Objectives

The goals of this work is to develop an agent with the following characteristics:

- Capable of following basic driving laws.
- Trained with reinforcement learning and distributed.
- Validated and tested with real-life scenario examples.

## 1.3 Document Structure

First in [chapter 2](#) the foundations required to create an autonomous vehicle are studied, starting with the definition of autonomous driving, the concepts around it and related work, then the building blocks for creating a reinforcement learning agent, **Machine Learning**, then **Deep Learning** and finally **Reinforcement Learning**, in the end of this chapter we analyze implementations that are currently being used by companies. In [chapter 3](#) details of the framework, environment and the different implementations of each RL agent are presented and towards the end a novel architecture is showcased, results of each agent's performance are shown and discussed. In [chapter 4](#) a novel algorithm for multi-agent reinforcement Learning is presented and analysed. In the last chapter [chapter 5](#) conclusions and future work are presented.



## Chapter 2

# Literature Review

### 2.1 Autonomous Driving

Autonomous Driving is concerned with creating a vehicle capable of driving without any human supervision. The agent is required to sense the environment and safely reach a destination. Historically, many experiments since the beginning of the 20<sup>th</sup> century have been devised to assist a human driver to control a vehicle but only after the 1960's huge milestones have been achieved in order to properly and autonomously control a vehicle, [74]. More recently, in the last decade, with advances in machine learning, control theory and computation capability [42] the field of Autonomous Driving has exploded, with many startups appearing (e.g. Waymo, Zoox, Cruise, Embark), big investments by car autos such as Porsche, Tesla and also a increasing interest in this research area.

Automated transportation is not something new, Lille had Europe's first autonomous metro system in 1983 [101], they provide its users with better and more efficient commutes. Examples include the Sydney metro system, that became automated since 2019 and is already proving its advantages over having a human operator [73], such as lower costs and higher availability. By replacing the human with a computer, there is a big potential in making roads safer and automobile transport more efficient.

We need to differentiate between similar terms related to self driving and discuss the terminology adopted for this document.

**Self-driving vs Autonomous vs Driverless.** According to the Union of Concerned Scientists [123] self-driving cars are defined as:

Self-driving vehicles are cars or trucks in which human drivers are never required to take control to safely operate the vehicle. Also known as autonomous or "driverless" cars, they combine sensors and software to control, navigate, and drive the vehicle.

**Autonomous vs Automated** Autonomous is defined as being self-governing and the difference between it and automated is the level of human independence required. While according to many

authors these two terms may appear different or at least not the same, according to the Regulation (EU) 2019/2144 of the European Parliament and of the Council of 27 November 2019 these two terms are defined based on the autonomous capacity.

(21) ‘automated vehicle’ means a motor vehicle designed and constructed to move autonomously for certain periods of time without continuous driver supervision but in respect of which driver intervention is still expected or required;

(22) ‘fully automated vehicle’ means a motor vehicle that has been designed and constructed to move autonomously without any driver supervision;

For the rest of this document the terminology adopted is that autonomous is completely independent of any human interaction, i.e. **fully automated**. Also the distinction between fully automated and automated come in different levels, defined by the Society of automotive Engineers (SAE) [96] are the following:

- *Level 0 (No automation)*: all driving aspects are executed by a human, systems may momentarily intervene.
- *Level 1 (Driver assistance)*: both a human and an automated system control the vehicle, examples include Cruise Control (CC) and Parking Assistance.
- *Level 2 (Partial automation)*: the automated system is able to take control of steering, braking and accelerating but the human driver must be prepared to intervene promptly if the system requests it [ "hands off" ]
- *Level 3 (Conditional Automation)*: the automated system can control the vehicle without the full attention of the driver but it has to be able to intervene within a limited amount of time, the driver becomes a sort of co-driver [ "eyes off" ].
- *Level 4 (High Automation)*: no driver attention is needed for the system to be safely operated, the driver can even sleep, but the system is limited to only some areas, "geofenced", outside of these areas the system stops working in an orderly fashion [ "mind off" ].
- *Level 5 (Full Automation)*: The system is able to control the vehicle in all roads, areas and conditions, all year around, absolutely no human interaction is needed.

For the purpose of this work the objective of the final agent is to achieve fully autonomous vehicle, (i.e. **Level 4 or Level 5**) meaning that at any time no human intervention is required for the vehicle to operate safely. Next, the techniques and the fields required to build a self-driving vehicle are analyzed, followed by the description of historical and current approaches that have been taken to develop an autonomous driving agent.

## 2.2 Preliminaries

The review of previous works related to this field will follow a structure of first looking at machine learning, then some of its sub fields, then geometric deep learning and finally at reinforcement learning.

### 2.2.1 Machine Learning approaches

Machine Learning (ML) is a subarea of AI that focuses on algorithms that learn from data. Goodfellow et al. [39] describe ML as a form of applied statistics, to use a computer to estimate complicated functions and to make predictions. According to [88] the definition of learning is as follow:

“A computer program is said to learn from experience  $E$  with respect to some class of tasks  $T$  and performance measure  $P$ , if its performance at tasks in  $T$ , as measured by  $P$ , improves with experience  $E$ .”

Machine learning approaches fit into three main areas, namely supervised learning, unsupervised learning and reinforcement learning, which are briefly detailed in the following. This thesis will focus on reinforcement learning approaches for autonomous driving.

**Supervised Learning.** Training a computer program involves presenting it with labelled training data. The program will learn to map an input to an output based on that data. It will learn inference based on observations. The final quality/performance of the program is mainly dependent on the quality of data, any biases present on it will also show up in the final trained program. Typically, it involves performing the following tasks:

1. Collect a training dataset, ensure that it includes in similar amount all of the desired types of outputs. Ex: In recognition of handwritten digits, collect samples with similar amounts of 0's, 1's, 2's, ...
2. Select the input feature representation, features from the inputs have to be selected in order to accurately describe the object but to leave out unnecessary information. In the previous example it might be converting the images to black and white since the color of the image isn't important.
3. Select the learning algorithm, examples include support vector machines, decision trees or neural networks.
4. Run the algorithm on the training dataset, extracting the learned function.
5. Evaluate the accuracy of the resulting algorithm, on data not previously seen by the program and if it is unsatisfactory repeat training changing parameters or the algorithm or even the dataset.

Supervised learning is used regularly in autonomous driving, feeding systems massive amounts of driving experiences conducted by humans and having the program learn how to drive, this is the primary method used by companies such as Tesla and Waymo, the latter has even made part of its data publicly available [130].

**Unsupervised learning.** Unsupervised learning works with unlabelled and unclassified data. This method tries to infer a function that can describe a hidden structure of the data. The main methods used in unsupervised learning are principal component and cluster analysis. An example of such problem might be identifying best friends in graph representing social media connections. After training these algorithms can be used to make predictions on new data.

**Reinforcement learning.** Reinforcement Learning is about controlling an agent thorough an environment, receiving reward signals and having to maximize the cumulative reward. It is studied in greater detail in 2.4

## 2.2.2 Deep Learning with Artificial Neural Networks

### 2.2.2.1 Artificial Neural Networks

One important topic related to machine learning that in recent years has seen massive developments is artificial neural networks. They serve as important tools and are inspired in biology, more specifically how a human brain works. Composed of a collection of neurons in layers, they sequentially process the input given to it and pass it to the next layers until the output layer, finding complex patterns and making it possible to solve problems that would be otherwise very difficult. The reason that neural networks learn is that each node has a set of weights and biases, these are applied on the input and produce the output. The most common way to obtain a trained neural network is by training them using backpropagation (genetic algorithms are another way, such as NEAT [115]), where the error of each training step adjusts the connections (weights) of the network.

An example of a neural network can be seen in figure 2.1, it consists of an input layer, followed by two hidden layers and a output layer. Data is fed into the input layer, here it consists of a 16 dimensional vector, then passed to the neurons on the second layer, according to the weights and biases (in the picture its the lines), transforming into lower dimension (in the case of the image, 16 to 12 then 10 the finally 3), after each layer a required component is applying a non-linear function, the final output vector can be used for classifying or regression.

To train a neural network with backpropagation, a loss or error function needs to be defined, in the form  $E(\mathbf{y}, \mathbf{y}')$ , that takes the output of the network  $\mathbf{y}$  and the target label  $\mathbf{y}'$ , the network is initiated with the weights randomly (there are works where the weights might be initialized

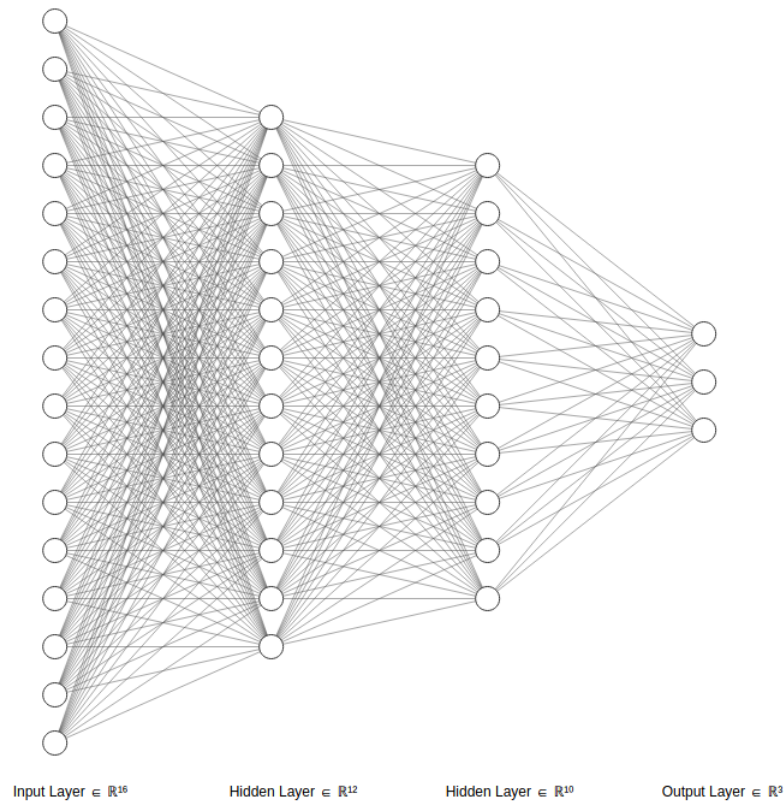


Figure 2.1: Example of a Deep Neural Network. From [5]

differently [87], [72]) and at each step the data is feedforwarded through the network explained by Equation 2.1 and Equation 2.2

$$a_j(l+1) = \sum_i o_i(l)w_{ij} + b_{ij} \quad (2.1)$$

$$o_j(l) = G(a_j(l)) \quad (2.2)$$

where:

- $a_j(l)$ : activation of the neuron  $a_j$  in layer  $l$
- $w_{ij}$ : weights of the connection from neuron  $i$  to  $j$
- $G$ : non-linear activation function
- $o_j$ : output of neuron  $j$

After obtaining the output the loss or error is computed ( $E$ ), then training with backpropagation can start. Starting from the output until the input the partial derivatives of the loss with respect to

the weights ( $w$ ) are computed, generically for each layer and weight using Equation 2.3.

$$\frac{\partial E}{\partial w_{ij}^k} = \delta_j^k o_i^{k-1} = G'(a_j^k) o_i^{k-1} \sum_{l=1}^{r^{k+1}} w_{jl}^{k+1} \delta_l^{k+1} \quad (2.3)$$

The error of the weights  $w$  at layer  $k$ ,  $\delta_i^k$  depends on the errors of next layer  $k+1$ ,  $\delta_l^{k+1}$ . For the last layer  $m$ , the partial derivative of error becomes

$$\delta_i^m = G'(a_i^m)(E) \quad (2.4)$$

Finally updating the weights becomes:

$$w_{ij}^k \leftarrow w_{ij}^k - \alpha \frac{\partial E(y, y')}{\partial w_{ij}^k} \quad (2.5)$$

where  $\alpha$  is the learning rate.

Usually training neural networks is abstracted from all of the mathematics involved by using libraries/programs that make use of automatic differentiation, it becomes as simple as writing only the feedforward part of the network (how the output is computed) and then the loss, a program keeps track of what the operations are done to the input and their adjoint  $\bar{w} = \frac{\delta E}{\delta w}$  in respect to the loss/error and when it comes to training, the program computes the change for the weights automatically. This is how popular machine learning libraries such as Pytorch [100], function.

Neural networks have been widely used today and have proven to be production ready, from performing translation to drawing realistic landscapes from 2d drawing [95]. Much of the ongoing research into machine learning involve neural networks and many approaches to autonomous driving have in their architectures one or more neural networks, they are important since they serve as a components capable of learning how to drive.

Generally, training a neural network involves choosing many hyperparameters such as:

- *Neural network architecture* (number of layers, neurons per layer and even the type of NN, such as CNN, RESNET): this is an extremely important step and the differences in accuracy between similar architectures can be massive. Most architectures are choose by experience or there are specific types that favour one type of task over other.
- *Learning Rate*: specifies how much the weights of the nets change at each training step, if chosen to high it could lead to not learning anything and having continuous increasing loss, if its too small then it takes a long time to converge. Typically its a value between 1e-3 and 1e-4 [45], [49].
- *Non linear activation function*: when passing outputs between neurons from a layer to a higher one a non linear activation function has to be used, otherwise the entire network would be equivalent to a single layer. There are many functions to choose, regularly RELU, leakyRELU, tanh and softmax, each useful in their own task.

- *Optimization algorithm*: there are some choices here, like simple stochastic gradient descent [16], RMSProp, AdaGrad but the most used and usually most effective is ADAM [64].

Choosing the correct hyperparameters is critical and although there is an entire field dedicated to automatically optimizing them, Automated machine learning and also Neural architecture search (NAS)[31], most hyperparameters are chosen by the programmers experience. This coupled with the fact that neural networks are in the end, black boxes, just pools of weights, that can't explain why they choose a particular output given an input leads to a reputability crisis [138].

Another important drawback of artificial neural networks is that they aren't exact, they can produce completely unexpected outputs and with confidence. This can be a problem for critical tasks like autonomous driving. There are variations and different neural network architectures that address this problem [14]. Nevertheless, neural networks are a very important component that is present in architectures used in researching autonomous vehicles.

### 2.2.2.2 Deep Learning with Convolution Neural Networks

Deep Learning is an area part of machine learning, the "deep" part refers to the use of neural networks or variations of it, that use of many "hidden" layers, between the input and output. Due to advances in computing power current architectures that leverage neural network use many and many hidden layers, with up to hundred of thousands of neurons, for example one of the best language models (able to perform tasks like translation, question-answering, and close tasks and tasks that require on-the-fly reasoning) has 175 billion parameters (parameters in general are the weights) [17].

The standard architecture of a artificial neural network when applied to the task of computer vision doesn't work well, an image needs to be flatten and will lead to the first layer (the input one) having millions of neurons and it won't be able to capture many patterns specific to 2D data, that's where a variation of neural networks named Convolution Neural Networks (CNN) is used. While some variations exist to a regular CNN, this work is focused on ones applied to 2D Images. **Convolution Neural Networks** are an extension to the simple neural network architecture, it adds convolution layers between a simple neural network and its inputs. These new layers make it possible (or at the very least more feasible) to work with higher dimension *structured* data (e.g. images). These layers apply convolution to the inputs with a kernel, generating a representation with a different dimension (usually smaller), then pooling can also be used to further decrease the input dimension, after all the convolution and pooling layers transform the data (i.e. feature extraction), the output is then feed to linear layers (can be a deep neural network) which subsequently yield the output vector, exemplified in Figure 2.4. CNNs are an important for computer vision (they were even inspired by an animal's visual cortex) and have successfully been applied to many problems involving images and pattern recognition, in fields like medicine [82] to autonomous driving [139].

In the most basic form CNNs layers do two types of operations, namely **convolution** and **pooling**. Convolution works by having a  $n \times n$  filter (typically 3x3, 5x5 or 7x7) "slide" through

the input data computing the value using the filter's weights and the input data, example 2.2. The weights of the filters are trainable.

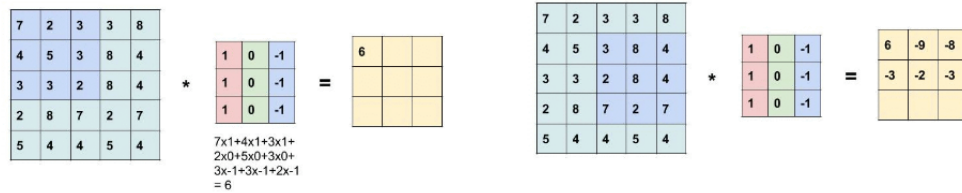


Figure 2.2: CNN's convolution filter (middle matrix) computing its output (yellow matrix), filter slides through input data (blue matrix), first by the left image then right. From [127]

Pooling layers serve to perform non-linear downsample to its input. There are several types of non-linear but the most commonly used is max-pooling. This method first partitions the image into  $k \times k$  blocks and for each partition it outputs its maximum as depicted in Figure 2.3. Pooling reduces the amount of memory required to work with the inputs, it also helps to prevent over-fitting.

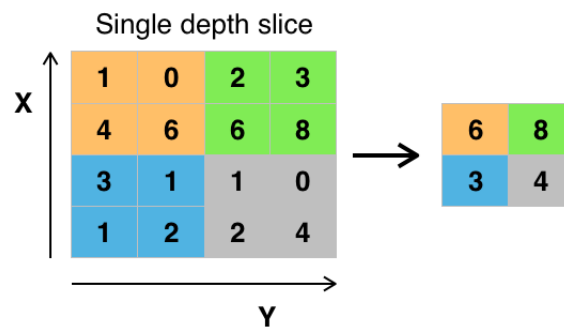


Figure 2.3: CNN's pooling layer, in this example we can see max pooling with a 2x2 filter and stride of 2. From [114]

For each one of these layers/filters the following hyperparameters exist:

- *Kernel size*: this is the size  $n \times n$  of the kernel, "window". Typically 2x2, 3x3, 5x5, 7x7, higher sizes are faster to compute but loose granularity, ability to detect small details.
- *Stride*: number of elements jumped between consecutive passes of the filter. Typically 1, 2 or 3, higher amounts reduce the amount of memory needed.
- *Non-linear activation function*: same hyperparameter as regular neural networks. Typically ReLU or tanh are used.

**ResNet.** As deep neural network models become more complex, with more layers, to attempt to solve more complex tasks, problems start to arise, namely vanishing gradients. When training these type of neural networks with backpropagation since the change in weights is proportional to the partial derivative of the the error with activation of the layer in front (see Equation 2.3), the



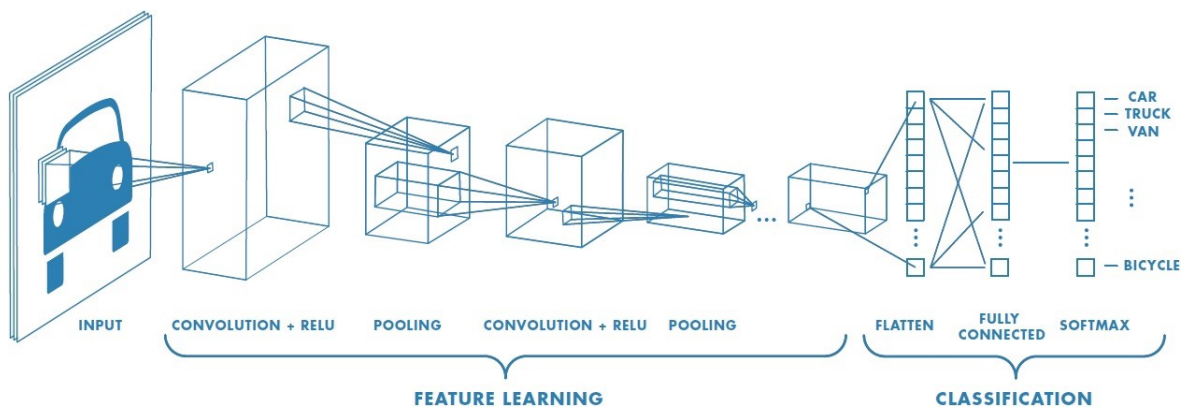


Figure 2.4: Diagram of a CNN with an image input. From [70]

gradient can become increasingly small until the weights no longer train and the networks fails to train further. ResNet [49] are a new architectural paradigm that use skip connections, in some layers the output simply skips one or more layers forward, with non-linear functions between the skips. During training layers can "opt" to skip or ignore previous layers until latter stages of training. ResNet models are able to be more "deep" than simple neural networks and achieve better accuracy. Another way to visualize the power of this architecture is by analyzing the loss function for a ResNet architecture with and without skip connections [77]. Training a network with skip connections becomes less prone to be stuck in local minimums, "valleys" in the loss function, and to train faster 2.5.

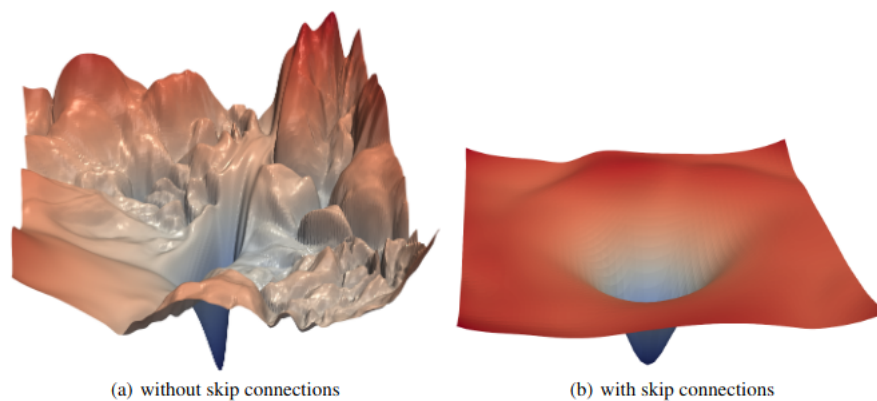


Figure 2.5: The loss surfaces of ResNet-56 with and without skip connections working with the CIFAR-10 dataset. From [77]

### 2.2.3 Batch Normalization and Dropout

Other ways to improve training, either making it faster or more stable, is to use Batch Normalization and/or Dropout. Batch Normalization (BN) [60] is transformation applied to data at each layer input. Normally training a neural network is done in batches, since the entire training dataset usually doesn't fit entirely into memory, for each batch its mean  $\mu$  and variance  $\sigma^2$  is calculated and

then each training input  $x_i$  is re-centered and finally scaled and shifted with two pairs of trainable parameters,  $\gamma$  and  $\beta$  along each of the inputs dimensions, according to Equation 2.6.

$$\begin{aligned}\mu_B &\leftarrow \frac{1}{m} \sum_{i=1}^m x_i \\ \sigma_B^2 &\leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2 \\ \hat{x}_i &\leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \\ y_i &\leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i)\end{aligned}\tag{2.6}$$

Since all of these operations are trainable, backpropagation can still be used for training. Although having a big performance impact, there still does not exist a consensus on the reasons for the improved performance. Some researchers say that its improvements come from smoothing the objective function [103].

Dropout [53] is a regularization another technique to increase a neural network performance, reducing overfitting. It works by randomly omitting the outputs of some neurons during training and thus preventing complex co-adaptions between neurons or group of neurons. Each neuron learns to detect a feature that is helpful for improving the performance of the neural network.

#### 2.2.4 CNN Visualizations

A disadvantage of using CNNs and NNs in general is their inherent black-box nature, i.e. no explanations to their output are produced, or no uncertainty over training data (works like Bayesian Neural Networks [14] attempt to address this) is given. This can be a problem for self-driving where at least some indication that the model isn't over-fitting is desirable and that it is correctly assessing its inputs. Several works exist to address this problem like [107], [111] and they revolve on creating an activation map that when overlayed on top of the input image. These techniques can be used for better assessment of a model's behaviour and its capabilities. For example, if a model decides to turn right and looking at where the activations for the model shows the sky, it must likely means that the model is overfitted and incorrectly trained. If on the other hand it decides to brake and activation maps show it was because it spotted object on the road it is a good indication that it is behaving like it should. One of the simplest techniques is *integrated gradients* they work by propagating the activations all the way to the input tensors, example of this technique 2.6.

#### 2.2.5 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a class/architecture of neural networks that use internal states, memory, this allows them to better process input data is a sequence, like time-series data or Natural Language Processing (NLP). They can be seen as a network where an internal state vector (hidden unit) is passed throughout the processing of a sequence to itself 2.7. They can process arbitrary length sequences, although long sequences see the problem of vanishing gradients, where activations from the processing of the initial inputs from a sequence have less effect on the output for latter inputs.



Figure 2.6: CNN's visualization, the network predicted the correct class and the right image shows where it "looked". From [97].

The most used and better performing RNNs architecture is Long Short Term Memory (LSTM) [54], it tries to mitigate some of the disadvantages of RNNs by the using multiple switch gates and units that are similar to ResNet skip connections.

Another similar RNNs architecture is Gated Recurrent Unit (GRU) [23], the only difference is that GRU's exposes the memory without having a separate update and forget gate, having fewer parameters and being computationally faster.

It should be noted that for tasks like Natural Language Processing, recurrent neural networks are considered outdated, being replaced by the Transformer [126], [29], but for other tasks they are still state of the art like lossless compression [9].

### 2.2.6 Autoencoder

Autoencoder (AE) is a type of artificial neural network, their usefulness comes from the fact that they can compress data to a much smaller dimension, learning a representation of the data 2.8.

Training an autoencoder is done in a unsupervised manner, the output of one is a reconstruction of the input data and in a middle hidden layer (called a bottleneck) lies a "code" that can accurately reconstruct the input.

They are typically used in dimensionality reduction and many variations exist, some can be used for denoising or reconstructing images images or even as a generative model.

In general autoencoders have a loss function that includes the reconstruction loss and also a regularizer term (that can be ommited) that prevents overfitting 2.7.

$$J = L(x, \hat{x}) + \text{Regularizer} \quad (2.7)$$

An advantage of using autoencoders in a autonomous driving agent is that unlike regular CNNs training the encoding part of the agent is decoupled from training the policy part and can lead to improvements in performance.

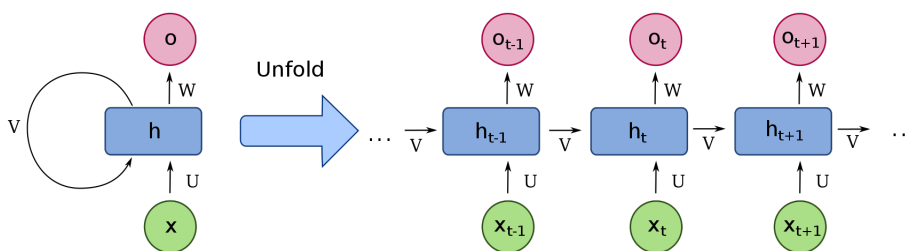


Figure 2.7: Diagram of a unfolded recurrent neural network. From [136]

Also using an autoencoder means that when the agent’s policy is training, it doesn’t have to process the input and with satisfactory performance it could be implemented by an Application-specific integrated circuit (ASIC) [94] granting efficiency gains and has the benefit that it could be reused for different training algorithms.

A big problem with this approach is that not only the autoencoder has to be trained to accurately reconstruct the images but the latent space (bottleneck) has to be somewhat interpretable, for example two very similar inputs must have small distance between their encoded representations (this is related to the regularizer term), this is addressed with a variant of the autoencoder, the **Variational Auto Encoder (VAE)**.

### 2.2.6.1 Variational Auto Encoder (VAE)

As a way to keep the latent space interpretable VAE’s introduce two variations to the regular autoencoder, one as a form of a regularizer term and other as its main architecture 2.9. It has different mathematical roots, namely **variational bayesian**. They are directed probabilistic graphical models (DPGM) that have a posterior approximated by a neural network. VAEs are also **generative models**, they model how the data generated and can even generate new data points [65].

The idea behind VAE is to encode input data  $x$  not into a fixed vector but to a distribution. It assumes the input data was generated by  $p_{\theta}(x|h)$  which is what the decoder will try to approximate, and the encoder part  $q_{\phi}(h|x)$  is learning an approximation the posterior  $p_{\theta}(h|x)$ ,  $\phi$  and  $\theta$  are parameters of the encoder and the decoder, which parametrized by neural networks it will be their weights. The prior of the latent variables is assumed to follow a multivariate Gaussian distribution  $p_{\theta}(\mathbf{z}) = \mathcal{N}(\mathbf{0}, \mathbf{I})$ . The reason for a Gaussian is that its easy to work with. Other works have worked with different distributions [27].

The loss function is presented in 2.8 is similar to 2.7, it has a reconstruction loss and a regularizer term that will penalize for encodings that don’t follow the Gaussian, this will in term make the reconstructions diverse and the latent space meaningful and interpretable, new samples can also be easily created.

$$\mathcal{L}(\phi, \theta, \mathbf{x}) = D_{\text{KL}}(q_{\phi}(\mathbf{z} | \mathbf{x}) || p_{\theta}(\mathbf{z})) - \mathbb{E}_{q_{\phi}(\mathbf{z} | \mathbf{x})}(\log p_{\theta}(\mathbf{x} | \mathbf{z})) \quad (2.8)$$

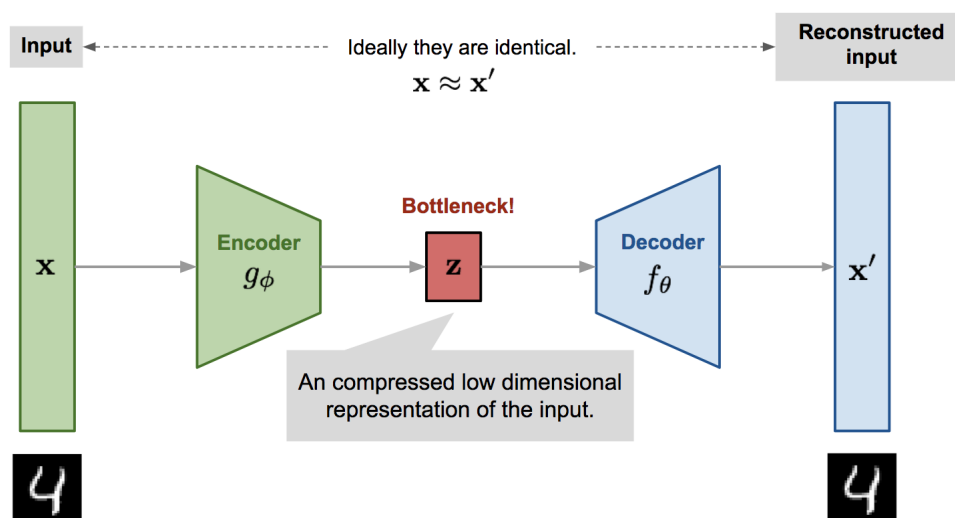


Figure 2.8: Diagram of a autoencoder. From [133]

One example of this meaningful latent space is for example training an autoencoder to reconstruct human faces, the latent space could be something like a 20 dimensional vector and the first dimension  $x_1$  could be the eye width and by adjusting this value we practically customize to our liking the reconstructed face. A more in depth study of the ability of VAEs to capture semantic information of face expression is explored in this work [57]. If a VAE is trained correctly and achieves a reasonable loss its latent space encoding could be used for the inputs of an reinforcement learning agent, close encodings (small distance) would be similar images and the agent could even learn what each dimension in the encoding vector could mean, all of these while providing a small observation space, which would make training an agent much faster and simpler.

As mentioned previously in 2.2.2.1 to train a neural network all operations from the input data to the calculation of the loss function need to be differentiable but in order to output a reconstruction we need to sample from a Gaussian distribution, something that isn't differentiable, to overcome this the reparameterization trick [66] needs to be used 2.11.

Note that while looking at diagram 2.10 we might say that the regularizer term off the loss function should be  $D_{\text{KL}}(q_\phi(\mathbf{z} | \mathbf{x}) || p_\theta(\mathbf{z} | \mathbf{x}))$  and would be correct but the posterior  $p_\theta(\mathbf{z} | \mathbf{x})$  is intractable and directly calculating the KL divergence between the two would take exponential time. The proof that minimizing 2.8 leads to the same result is in Appendix A.1.

### 2.2.7 Geometric Deep Learning

Geometric Deep Learning is a subarea of deep learning that studies how neural networks can be applied to data that doesn't follow a well defined structure like an image but instead graphs. Over the last two decades many algorithms have been proposed, but they have been shown to be similar and in the work [38], they were unified. They normally start by having nodes pass information between each other, such as messages, then each node computes an aggregation function and passes it to a differential function such as a neural network.

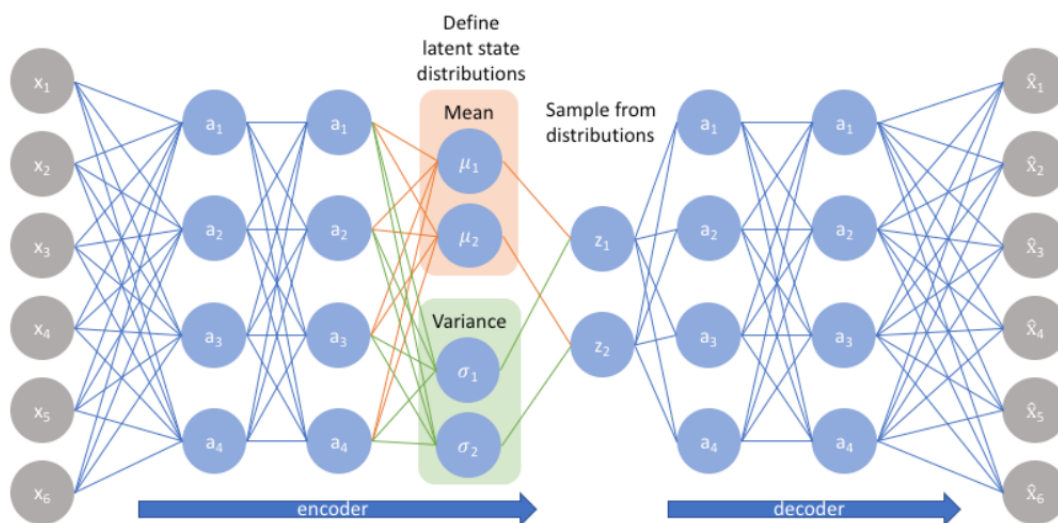


Figure 2.9: Diagram of a Variational Auto Encoder. From [62]

In general algorithms use the concept of message passing 2.9, from [33], where each node passes between each other messages (arrays of floats) and an aggregating function such as mean or max is used on all the messages a node receives.

$$\mathbf{x}_i^{(k)} = \gamma^{(k)} \left( \mathbf{x}_i^{(k-1)}, \square_{j \in \mathcal{N}(i)} \phi^{(k)} \left( \mathbf{x}_i^{(k-1)}, \mathbf{x}_j^{(k-1)}, \mathbf{e}_{j,i} \right) \right) \quad (2.9)$$

Algorithms like the Message Passing Algorithm have been applied with great success in areas like quantum chemistry.

In the context of Multi-agent Reinforcement Learning a message passing can be used for the information passed between the agents, for better cooperation. Messages are passed between the agents and their content changes as the agents are trained. The agents can form a complete graph if they are always communicating with each other or disjoint graphs for vehicles in range inferior to a certain amount.

## 2.3 End-to-End Training

A really important and studied work that combines machine learning and self-driving and that serves as a good starting point for creating an agent is a work done by Nvidia[15]. Its architecture consists of a neural network that is fed labelled data itself composed of expert driving experiences, it is reminiscent of training a neural network to recognize hand written digits but it is instead trained with driving data. Nvidia engineers collected data from driving near their work place, they used multiple cameras and a large part of the work is describing the setup and the hardware used as for the time (2013) it was computational expensive to not only gather this amount of data but to train NNs with images.

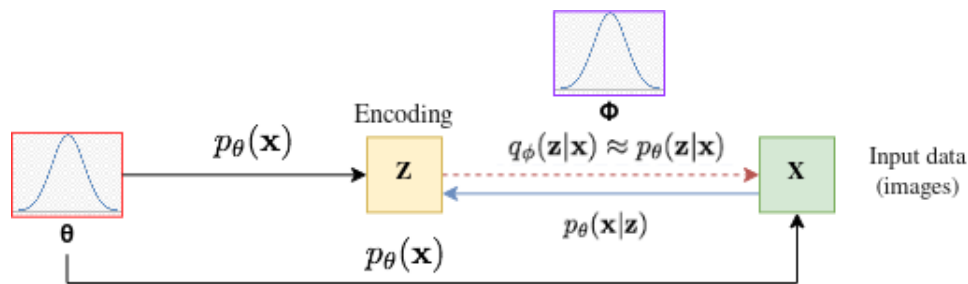


Figure 2.10: Diagram of the graphical model involve in VAE from [133].

It concludes the following, having multiple cameras in the bonnet of the car provide better results, making the agent better at driving at the center lanes and the final neural network does "look" at necessary features (like lanes) of the images when taking decisions of the output (such as turning left or right). This is very important because compared to traditional ways of attempting to create a self-driving agent, such as explicit programming all the rules to the agent and compiling all the rules and cases, neural networks are seen as unpredictable, prone to over-fitting and that don't really capture the rules of driving. While this work doesn't prove that they aren't, it also shows that NNs are likely to learn how to drive by looking at features that we humans look at too, such as lanes, signs and other cars.

Related to the work presented is the topic of deep learning called **End-to-End Training**.

Combining different components, namely architectures and neural networks can introduce stages where parts of its final architecture needs or must be hand tuned, for example if we had a self-driving pipeline that included a CNN for vision and a controller that takes as input the CNN's final layer and output a command which is the direction to travel, but doesn't directly control the vehicle then it isn't end-to-end.

End-to-End training refers to approach where an architecture is trained to directly learn the solution, it omits any hand-crafted intermediary components.

Looking at 2.12 the architecture itself has many components but it goes from sensor input to driving output, meaning its *end-to-end*.

This approach to training is important has it leads to less hyperparameters, less "guessing" and streamlines the training process.

## 2.4 (Deep) Reinforcement Learning

Reinforcement Learning (RL) is a machine learning area concerned with creating an agent that senses an environment and it takes actions in order to maximize a reward (given by the environment). Sutton in his book [117] depicts three threads in the history of reinforcement learning, going back to the 1960's and multiple researchers working in parallel on works and theory that would be eventually joined into the a field called Reinforcement Learning.

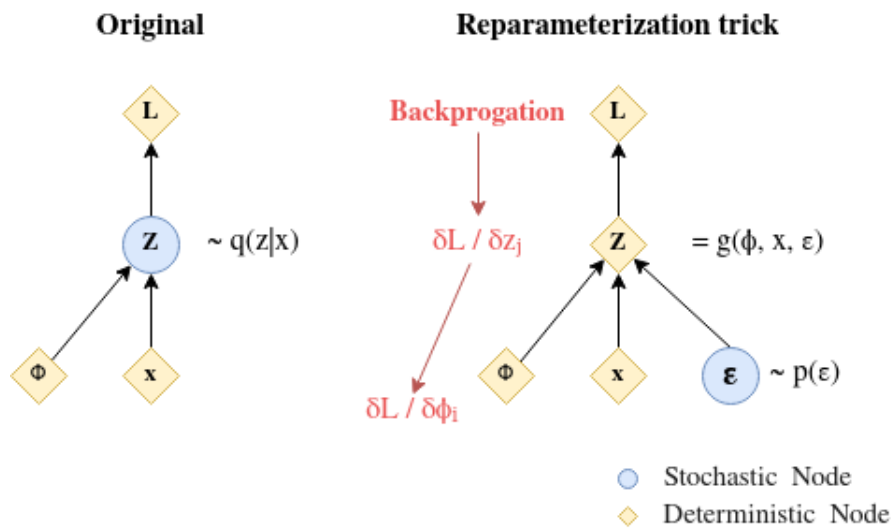
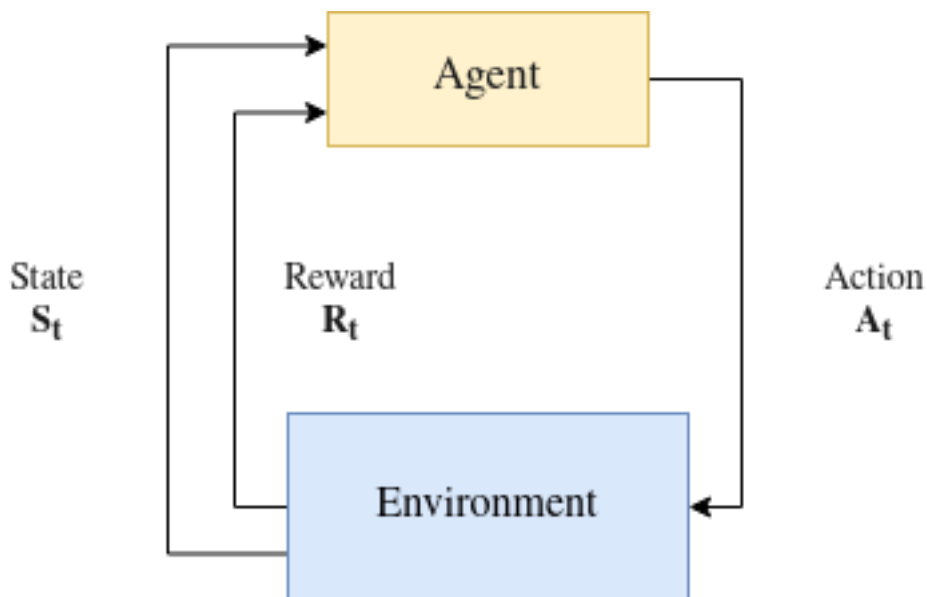


Figure 2.11: Reparameterization trick.

In order to study the algorithms and the architecture used for this work we first need to create a mathematical framework (Markov Decision Process) for our problem from which we can derive mathematical proofs and theorems that will show that in theory the algorithms will lead to an agent capable of controlling a vehicle.

### 2.4.1 Markov Decision Processes

Markov Decision Processes (MDP) serve a formalization for sequential decision making, it and captures how the agent moves and interacts with the environment. The general diagram of one in 2.4.1.



Reinforcement Learning is modelled by MDP and it has the following characteristics:



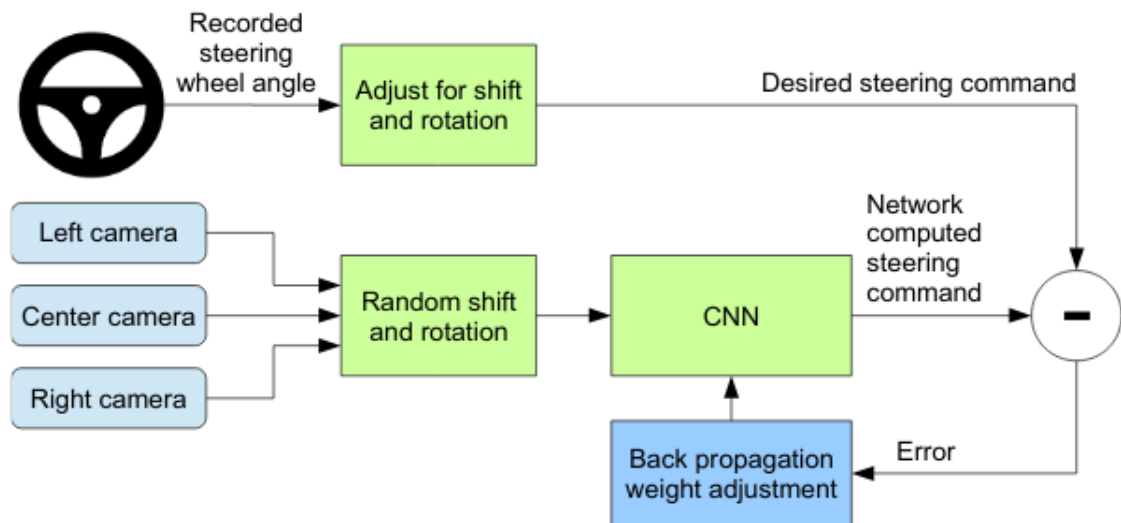


Figure 2.12: End-to-end training. From [15]

- A environment with states,  $S$ .
- A set of action that a agent can take,  $A$ .
- Model of the transitions of the environment,  $P_a(s, s') = \Pr(s_{t+1} = s' | s_t = s, a_t = a)$ , the probability of at time  $t$  transitioning from state  $s$  to state  $s'$  taking action  $a$ .
- A reward function  $R(s, s', a)$  that is the immediate reward from going to state  $s'$  from  $s$  by taking action  $a$ .

In reinforcement learning, an agent interacts with the environment in discrete time step,  $t = t_0, t_1, t_2, t_3 \dots$  receiving (at each time step  $t$ ) the state of the environment  $s_t$ , the agent then chooses an action  $a_t$  and the environment moves to a new state  $s_{t+1}$  with a probability given by  $\Pr(s_{t+1} | s_t, a_t)$ , the agent then receives the new state,  $s_{t+1}$  and a new reward  $r_{t+1}$ . An episode or trajectory  $\tau$  is composed of a set of transitions starting from the beginning state  $s_0$ , until the end state  $s_n$ ,  $(s_0, a_0, r_0, s_1, a_1, r_1, \dots, s_n, a_n, r_n)$ . The goal of the agent is to learn a **policy**  $\pi : A \times S \rightarrow [0, 1], \pi(a, s) = \Pr(a_t = a | s_t = s)$  that maximizes the expected cumulative reward  $R_{t_0} = \sum_{t=t_0}^{\infty} \gamma^{t-t_0} r_t$ , where  $\gamma$  is the discount factor.

In simpler terms the objective of reinforcement learning is to create a function (**policy**) that accepts a state  $s_t$  and outputs an action  $a_t$  maximizing the objective quantity, the cumulative reward  $R_t$ . The reason why it isn't just the immediate reward  $r_t$  is because what matters is all of the future rewards.

### 2.4.2 Partial Observability

In the last section MDP were introduced but one assumption was taken into consideration, the full observability of the environment at each step, this isn't the case with many environments or tasks,

in some cases the agent can only observe part of the environment states, it can only "see" part of what makes the environment function and transition. In this case the problem is formulated as a Partial Observable Markov Decision Process (POMDP) 2.13

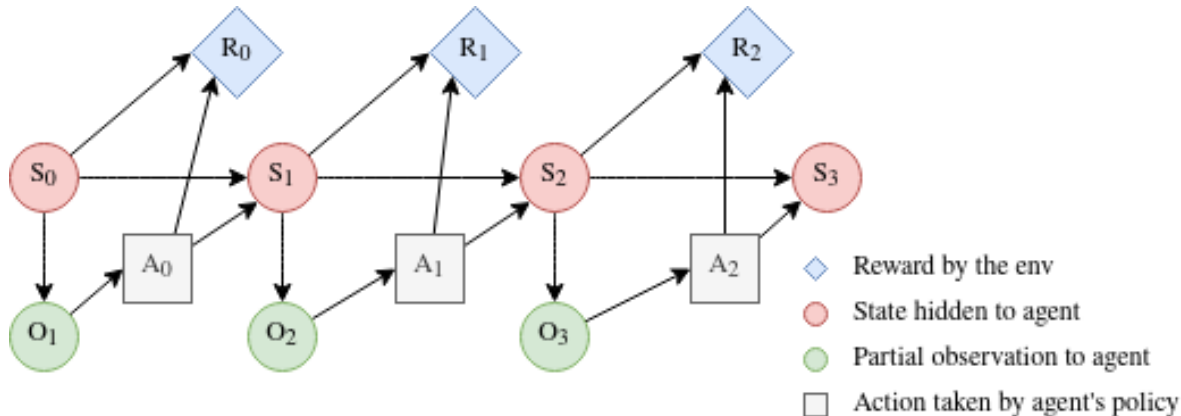


Figure 2.13: Diagram of a Partial Observable Markov Decision Process.

When working with POMDPs instead of the agent's policy taking as input a state  $s$  it takes an observation  $o$  and the set of action the agent can take can be restricted.

### 2.4.2.1 Stationary Distribution

A relevant concept for MDPs but more related to simple Markov Chains is the **stationary distribution**, its a row vector  $\pi$ , with the probability over all states that remains unchanged in the Markov Chain as time progresses. It indicates the probability of ending in one state after many transitions and is even independent from the starting state.

$$\pi = \pi \mathbb{P} \quad (2.10)$$

If the Markov Chain is ergodic that it is, it is expected to return to each state in a certain amount of steps and it is also aperiodic then it has a unique stationary distribution.

### 2.4.3 Policy and value function.

An agent's policy is a mapping of states to probabilities of actions. Reinforcement learning algorithms involve training the policy to maximize the discounted cumulative reward  $R_t$ , with this objective in mind we can obtain a function that gives a value of how good a given policy is starting from a state  $s_t$ , the value function  $v_\pi(s)$ , 2.11, it gives the expected value of the cumulative reward if the agent follows  $\pi$  policy starting from state  $s$ .

Related to the value function is the q-function or action-value function  $q_\pi(s, a)$  2.12 it is the expected cumulative reward if an agent follows policy  $\pi$  and at state  $s$  picks action  $a$ .

$$v_\pi(s) = \mathbb{E}_\pi[R_t] = \mathbb{E}_\pi \left[ \sum_{t=0}^{\infty} \gamma^t r_t \mid s_t = s \right] \quad (2.11)$$

$$q_{\pi}(s, a) = \mathbb{E}_{\pi}[R_t] = \mathbb{E}_{\pi} \left[ \sum_{t=0}^{\infty} \gamma^t r_t \mid s_t = s \mid a_t = a \right] \quad (2.12)$$

An important property of the value function is the following equality [2.13](#)

$$\begin{aligned} v_{\pi}(s) &= \mathbb{E}_{\pi}[R_t] = \mathbb{E}_{\pi} \left[ \sum_{t=0}^{\infty} \gamma^t r_t \mid s_t = s \right] \\ &= \mathbb{E}_{\pi} [r_{t+1} + \gamma \sum_{t=0}^{\infty} \gamma^t r_{t+1} \mid s_t = s + 1] \\ &= \mathbb{E}_{\pi} [r_{t+1}] + \gamma \mathbb{E}_{\pi} [R_{t+1}] \\ &= \mathbb{E}_{\pi} [r_{t+1}] + \gamma \sum_{s+1} P(s+1 \mid s, \pi(s)) v_{\pi}(s+1) \end{aligned} \quad (2.13)$$

This recursion of expected rewards is the Bellman Equation for  $v_{\pi}$ , it tells us how to find the value of a state and that its has a recursive dependency with the value of the next state. Bellman proved that the optimal value of a state must equal the action that give us the maximum expected immediate reward plus the maximum discounted long-term reward for the following state [\[12\]](#).

The action-value equation can also be described by the value function and vice-versa. [2.14](#)

$$\begin{aligned} q_{\pi}(s, a) &= \mathbb{E}_{\pi} [r_{t+1} + \gamma v_{\pi}(s_{t+1}) \mid s_t = s, a_t = a] \\ v_{\pi}(s) &= \sum_a \pi(a \mid s) q_{\pi}(s, a) \end{aligned} \quad (2.14)$$

This leads to the optimal policy, one that achieves the maximum value of the cumulative reward,  $v^*(s) = \max_{\pi} v_{\pi}(s)$  and reinforcement learning algorithms attempt to converge the agent's policy to the optimal one.

Given an optimal policy  $\pi^*$  the agent can act optimally by at each step choosing the action with the highest value from the optimal q-function  $q_{\pi}$ .

The optimal policy can also be expressed through the **Bellman optimality equation** [2.15](#)

$$v_*(s) = \sum_{s'} P_{ss'}^a (r(s, a) + \gamma v_*(s')) \quad (2.15)$$

These equations [2.15](#), [2.13](#) are these basis for optimization method known as **dynamic programming**. It works by breaking down a complicated problem into smaller ones and solving them recursively. If a problem can be solved optimally by this recursive dividing it into sub-problems and then solving those in a optimal manner that it is said to have **optimal substructure**.

If for a given task the complete knowledge of the MDP is known (subsequently the transition function) dynamic programming (DP) techniques can be used to obtain a optimal policy and two algorithms that can obtain it are **value iteration** and **policy iteration**. These algorithms and DP techniques also serve as an important foundation for more complex ones (even for ones without complete knowledge of the MDP).

### 2.4.4 Value Iteration and Policy Iteration

These two algorithms are very similar, starting with policy iteration, it has two steps, policy evaluation and policy improvement. First starting with a randomly initialized policy  $\pi(s)$  and value function  $v_\pi(s)$ , the value function for the policy is evaluated 2.16 for each step  $s \in \mathbb{S}$  and repeatedly until the value function converges (from one iteration to another has a small change).

$$v(s) \leftarrow \sum_{s'} p(s' | s, \pi(s)) [r(s, \pi(s), s') + \gamma v(s')] \quad (2.16)$$

After computing the value function  $v_\pi(s)$  we know which states are better, in a given step  $s$  we change the policy to take an action  $a_t$  that leads with higher probability to more valuable states  $s'$ , at each step  $s \in \mathbb{S}$  we update the policy according to 2.17.

$$\pi(s) \leftarrow \arg \max_a \sum_{s'} p(s' | s, a) [r(s, a, s') + \gamma v(s')] \quad (2.17)$$

Value iteration has a similar first step but instead of finding the value of the policy it attempts to find the optimal value policy, the difference is subtle, between 2.16 and 2.18.

$$v(s) \leftarrow \max_a \sum_{s'} p(s' | s, a) [r(s, a, s') + \gamma v(s')] \quad (2.18)$$

when the value function converges we extract the policy, in only one step 2.19.

$$\pi(s) = \arg \max_a \sum_{s'} p(s' | s, a) [r(s, a, s') + \gamma v(s')] \quad (2.19)$$

Both algorithms find the optimal policy,  $\pi^*$ , but according to [117] policy iteration converges faster.

Also both of these techniques can be seen under a more general method, **Generalized Policy Iteration**.

#### 2.4.4.1 Monte Carlo Methods

The previous methods required knowledge of the transition function  $P(s'|s, a)$  but if that's not the case than Monte Carlo (MC) techniques can be used to estimate the value function (and subsequently derive a better policy). MC techniques are widely used in statistics that rely on random sampling to obtain results.

Deriving the value of a state  $V(s)$  following a policy  $\pi$  starts with roll outs of episodes following the policy and at the end of each episode computing the average of the returns following visits to  $s$  1. MC algorithm is a unbiased estimator and converges to the value of the policy with error that has a standard deviation of  $1/\sqrt{n}$  with  $n$  as the number of returns averaged.

While it may seem that since we have evaluated the policy we can use policy iteration or value iteration since we don't have the transition function those algorithms can't be applied, to improve and converge to an optimal policy we need to estimate action-values  $q(s, a)$ . To estimate action

**Algorithm 1** Every Visit Monte Carlo policy estimation

---

```

1: Inputs: policy  $v_\pi$ 
2: Initialize:  $V(s)$ ,  $returns(s)$ ,  $history(s, a, r, t)$ ,  $N(s)$  for all  $s \in S$ ,  $a \in A$ ,  $r \in R$ 
3: for each episode do
4:   for each timestep in episode do
5:     Following policy  $v_\pi$  step through environment  $env_{step}(a)$ 
6:     Append experience  $history \leftarrow s, a, r, t$ 
7:   end for
8:    $R = 0$ 
9:   for each timestep in  $history$ ,  $t=T-1, T-2, \dots, 0$  do
10:     $R \leftarrow \gamma R + r_{t+1}$ 
11:     $returns(s_t) \leftarrow returns(s_t) + R$ 
12:     $N(s_t) \leftarrow N(s_t) + 1$ 
13:   end for
14:    $V(s) \leftarrow \frac{returns(s)}{N(s)}$ 
15: end for

```

---

values under a policy  $\pi$ ,  $q_\pi(s, a)$  we follow a similar method as 1 but instead of working with evaluations of a state we instead evaluate a state-action pair. One problem that is encountered is that if the policy is deterministic there are state-actions pairs that might never be visited (the estimation is biased) and the estimates will not improve with more samples. To overcome this problem we can either add exploration to the algorithm (in the form of choosing a random action either at start or with a certain chance) or we can choose to only work with policies that don't give zero probability to an action. This is referred to on-policy and off-policy training. On-policy training improve and evaluate the policy that is used to generate samples or the trajectory and off-policy training improves a policy with data generated from a different one.

After having estimated the  $q_\pi(s, a)$  then the policy can be improved by going adding more probability to the most valuable action, 2 shows the changes required to make to 1 to estimate state-action pairs and to improve a policy.

**Algorithm 2** MC changes for state-action values

---

```

1:  $q(s_t, a_t) \leftarrow average(returns(s_t, a_t))$ 
2:  $a \leftarrow argmax_a q(st, a)$ 
3: for action  $a \in \mathbb{A}(s_T)$  do
4:   if  $a \neq a$  then
5:      $\pi(a | s_t) \leftarrow \epsilon / |\mathbb{A}(s_T)|$   $\triangleright \epsilon$  is a hyperparameter and a constant probability
6:   else
7:      $\pi(a | s_t) \leftarrow 1 - \epsilon + \epsilon / |\mathbb{A}(s_T)|$ 
8:   end if
9: end for

```

---

## 2.5 Off-policy and On-policy

In off-policy training there is a target policy,  $\pi$  and a behaviour policy,  $b$ , the behaviour policy is used to generate samples and the target policy will improve from those samples. On-policy learning can be seen as a special case where the target and behaviour policy are the same. There are some complications and some requirements for off-policy training, for example to estimate the value of a target policy from a behaviour policy there is the requirement for *coverage*, namely that  $\pi(a|s) > 0 \rightarrow b(a|s) > 0$  this means that each action taken by  $\pi$  must be taken (even with a much smaller chance) by  $b$ . Training off-policy has several techniques such *importance sampling* which has the following form 2.20

$$\rho_{t:T-1} \doteq \frac{\prod_{k=t}^{T-1} \pi(a_k | s_k) p(s_{k+1} | s_k, a_k)}{\prod_{k=t}^{T-1} b(a_k | s_k) p(s_{k+1} | s_k, a_k)} = \prod_{k=t}^{T-1} \frac{\pi(a_k | s_k)}{b(a_k | s_k)} \quad (2.20)$$

$\rho_{t:T-1}$  is a ratio which adjust the expected return of the behaviour policy  $b$  and allows for the estimation of the value of the off-policy  $\pi$ .

$$\mathbb{E}[\rho_{t:T-1} r_t | s_t = s] = v_\pi(s), \quad \text{following policy } b \quad (2.21)$$

Similar to techniques presented in the last sections, to improve a policy we need to estimate action-values, so as extension to improve a policy, for each episode we need to calculate the importance sampling ratio and the resulting algorithm will be close to the ones 2 and 1.

### 2.5.0.1 Exploration vs Exploitation

Exploration is another important topic in reinforcement learning. In order to improve a policy we must make sure we know the each action-state but if following a behaviour policy that doesn't visit them we can be sure if their provide a better value than the ones currently been selected. On the other hand if the policy spends a lot of time exploring new state-actions it won't optimize their current decisions based on existing information (exploitation). There is a tradeoff balancing reward maximization based on what it already knows vs trying new actions to further increase its knowledge, *exploration vs exploitation*.

This problem is present in Reinforcement learning and there are many techniques to balance this, one of the most basic is  $\epsilon$  greedy, where the agent has a certain chance of just picking a random action, a probability for exploring instead of just exploiting. Several works attempt to tackle this issue [18] [26].

Using *off-policy* methods one could have the behaviour exploring policy,  $\epsilon$  greedy, and the target policy deterministic.

### 2.5.0.2 Temporal Difference

Temporal Difference (TD) combines Dynamic Programming methods and Monte Carlo ones, it doesn't need knowledge of the transition function (model-free) and it can learn from experience.

A key concept around TD is *bootstrapping*, which is a method where we will update our estimates to match later and more accurate predictions. We start with random estimates of, say the value function or action-state values, and then later update them to become more accurate. This is different than using MC techniques, with MC no initial estimation is done and we only adjust our estimates when the episode is over.

The update for a  $v_\pi(s_t)$  using all visit MC can be seen in the form 2.22

$$v_\pi(s_t) \leftarrow v_\pi(s_t) + \text{stepsize}[R_t - v_\pi(s_t)] \quad (2.22)$$

$R_t$  can only be known after stepping through the entire episode (it is the expect reward starting from  $t$ , see 2.11), in TD the update for  $v_\pi(s_t)$  it is in the form 2.23

$$v_\pi(s_t) \leftarrow v_\pi(s_t) + \text{stepsize}[r_t + \gamma v_\pi(s_{t+1}) - v_\pi(s_t)] \quad (2.23)$$

In order to update  $v_\pi(t)$  we need to be one step ahead, in  $t + 1$ , upon transition we can update the value. Looking at the general expression for updating a estimate 2.24

$$\text{NewEstimate} \leftarrow \text{OldEstimate} + \text{stepsize}[\text{Target} - \text{OldEstimate}] \quad (2.24)$$

$r_t + \gamma v_\pi(s_{t+1}) - v_\pi(s_t)$  is know as TD error and is important for studying reinforcement learning algorithms. Also the estimate update presented 2.23 is know as the simplest form of TD and since it only needs to be one step ahead to update it is known as TD(0). TD(0) is proven to converge to  $v_\pi$  and has several advantages when compared to MC methods, namely that it can be implemented in a online and incremental fashion and while it has not been mathematical proven to be faster than MC in practise it does [116].

### 2.5.0.3 n-step Bootstrapping

By allowing bootstrap to be done not on a single step like in TD(0) but for n-steps it allows for the update of an estimate to take into account more steps and to better perform credit assignment, that is to understand how each state influenced the reward, for example if we have an agent playing a football game with the reward given being the score of the game, many actions will get an immediate reward of 0 although they will have an influence on the final result, with n-step bootstrapping (and choosing a reasonable n) this problem is better addressed.

The general form of n-step bootstrapping applied to TD prediction is know as *n-step TD*.

Basically the update for time step  $t$ , weather it is a value prediction of action-value prediction, can only be done at step  $t + n$ , with the form 2.25.

$$v_{t+n}(s_t) = v_{t+n-1}(s_t) + \alpha [R_{t:t+n} - v_{t+n-1}(s_t)], \quad 0 \leq t < T \quad (2.25)$$

Note that  $v_{t+n}(s_t)$  is the value estimate at  $t + n$  iterations.

If the number of steps chosen to bootstrap is infinite (until episode ends) then it becomes Monte Carlo methods, as shown in figure 2.14.

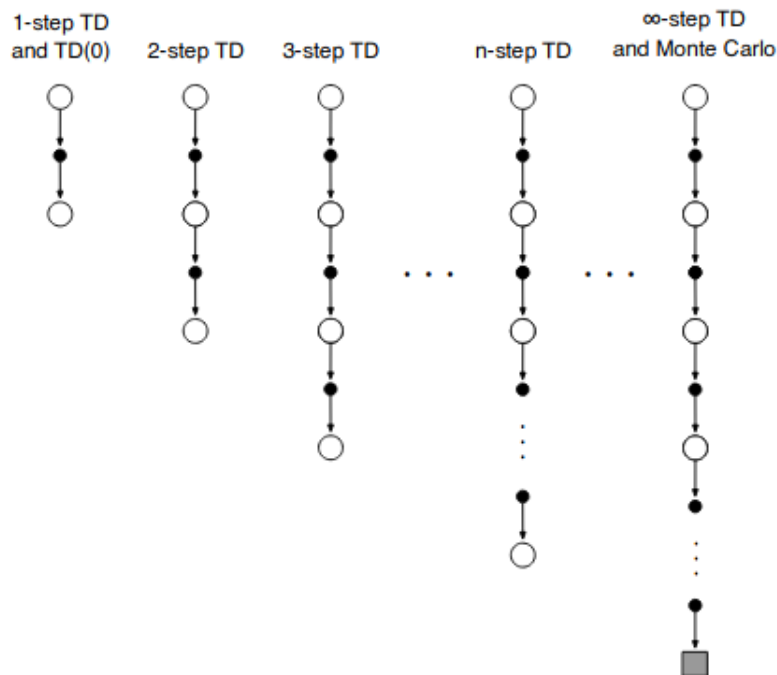


Figure 2.14: Diagram of n-step methods, ranging from TD methods to MC ones. From [117]

With all these concepts in mind a well known and used algorithm can be understood, it serves as a base for more complex ones, its called Q-learning [129] and its an off-policy, model-free temporal difference algorithm and the way it works is it updates the action-state function with the following relation

$$q^{new}(s_t, a_t) \leftarrow q(s_t, a_t) + stepsize \left( r_t + \gamma \max_a q(s_{t+1}, a) - q(s_t, a_t) \right) \quad (2.26)$$

In other notation *stepsize* can be  $\alpha$  or the learning rate.

The reason why Q-learning is off-policy is because the behaviour policy  $b$  has a random exploration factor,  $\epsilon$ , where it will choose a random action, while the target policy

The code for the final algorithm will be close to 3.

Q-learning even in its regular form has many practical applications, from finances [92], [75], to transportation [108] to playing games [19], to economics [63]. Its a robust algorithm, furthermore many variations of it exist, such as double Q-learning [47], Baysean Q-learning [28] and Nash Q-learning [59].

## 2.5.1 Deep Reinforcement Learning

The methods presented above are in tabular form, meaning that for every state or state-action pair we have a entry in a table, this works well in simple small cases and is easy to prove that it converges, but when the the state and action dimension increase it starts to require a lot of memory, this is known as *curse of dimensionality* and it was coined by Bellman in [13]. To overcome this



**Algorithm 3** Off policy Q-learning

---

```

1: Inputs: policy  $v_\pi$ 
2: Parameters:  $stepsize \in (0, 1], \epsilon \in (0, 1)$ 
3: Initialize:  $q(s, a), s_t$  for all  $s \in S, a \in A, r \in \mathbf{R}$ 
4: for each episode do
5:   for each step in episode do
6:     if  $\text{random}() < \epsilon$  then
7:        $a_t \leftarrow \text{random}(A)$ 
8:     else
9:        $a_t \leftarrow \max_a Q(s_t, a)$ 
10:    end if
11:     $s_{t+1}, r_t \leftarrow \text{env.step}(a_t | s_t)$ 
12:     $q^{new}(s_t, a_t) \leftarrow q(s_t, a_t) + stepsize \left( r_t + \gamma \max_a q(s_{t+1}, a) - q(s_t, a_t) \right)$ 
13:     $s_t \leftarrow s_{t+1}$ 
14:  end for
15: end for

```

---

problem and to be able to work in high dimension state/observations artificial neural networks can be used in conjunction with reinforcement learning. They will serve as function approximators for both the policy  $v_\pi$  and  $q_\pi(s, a)$ . The next assortment of algorithms are going to be presented with the intention of being used with NNs or are made from the ground up with them.

### 2.5.2 DQN


DQN [90] is a variant of Q-learning coupled with CNNs, it is able to sense raw pixel data and learn how to play Atari 2600 games, one of the first successful deep learning models that can work with such high dimension data.

A deep CNN is used to approximate the q-function and to train the loss for the weights of net,  $\theta$  is the TD error, explained in 2.5.0.2.

$$L_i(\theta_i) = \mathbb{E}_{s_t, a_t \sim b(\cdot), s_{t+1} \sim \text{env}} \left( r_t + \gamma \max_a q(s_{t+1}, a; \theta_{i-1}) - q(s_t, a_t; \theta_i) \right)^2, \quad b \text{ is the behaviour policy} \quad (2.27)$$

The behaviour policy  $b$  is an  $\epsilon$  – greedy, selecting a random action with a  $\epsilon$  probability. After the error has been calculated it is propagated using backpropagation 2.5.

DQN when it was released was a breakthrough in terms of performance 2.15, beating other reinforcement learning algorithms and even beating humans at the games. It also suffered from a variety of problems which were addressed in many works presenting variations on this simple algorithm, such as Double Deep Q-learning [125] that addresses the problem of overestimation of action-values, especially in noisy environments, extending DQN to continuous actions with an algorithm called DDPG [80], or soft Q-learning [44].



	B. Rider	Breakout	Enduro	Pong	Q <sup>*bert</sup>	Seaquest	S. Invaders
Random	354	1.2	0	-20.4	157	110	179
Sarsa [10]	996	5.2	129	-19	614	665	271
Contingency [11]	1743	6	159	-17	960	723	268
DQN	<b>4092</b>	<b>168</b>	<b>470</b>	<b>20</b>	<b>1952</b>	<b>1705</b>	<b>581</b>
Human	7456	31	368	-3	18900	28010	3690
HNeat Best [48]	3616	52	106	19	1800	920	<b>1720</b>
HNeat Pixel [48]	1332	4	91	-16	1325	800	1145
DQN Best	<b>5184</b>	<b>225</b>	<b>661</b>	<b>21</b>	<b>4500</b>	<b>1740</b>	1075

Figure 2.15: Atari 2600 used to benchmark DQN and their performance comparison. From [90]

### 2.5.3 Policy Gradients

With Q-learning and value based methods we get as an output the value of state-action pairs from then we have a deterministic policy which just selects the max action in a given state. Sometimes its useful to have a stochastic policy and also to handle continuous actions unlike simple Q-learning. Policy Gradients is a class of algorithms that acts on the policy itself, searching directly in the policy space and constantly improving it.

A parameterized policy is used and training consists of adjusting the policy parameter vector,  $\theta$ . It doesn't need to know the state-action values. The policy is defined as  $\pi(a|s, \theta) \rightarrow P(a|s, \theta)$ , the probability of selecting action  $a$  in state  $s$  with the parameters  $\theta$ , one requirement for the policy is that it needs to be differentiable with respect to its parameters.

To train the policy a quantity dependent on the parameters  $\theta$  needs to be defined (this is the loss function or the opposite of it)  $\mathbb{J}(\theta)$ . Defining it as the state values by following the policy we can then use gradient ascend to maximize the expected reward following the policy 2.28, 2.29.

$$\mathbb{J}(\theta_t) = v_{\pi_{\theta_t}}(s_0) = \mathbb{E}_{\pi_{\theta_t}}[v_0], \quad s_0 \text{ is the starting state} \quad (2.28)$$

$$\theta_{t+1} = \theta_t + \alpha \nabla_{\theta} \mathbb{J}(\theta_t) \quad (2.29)$$

Following the relation of 2.14 we can define the loss as 2.30.

$$\mathbb{J}(\theta_t) = \mathbb{E}_{\pi_{\theta_t}}[v_0] = \sum_{s \in S} \mu_{\pi_{\theta_t}}(s) \left[ \sum_{a \in A} \pi_{\theta_t}(a|s) q_{\pi_{\theta_t}}(s, a) \right] \quad (2.30)$$

$\mu$  is the stationary distribution for  $\pi_{\theta}$

To compute the gradient of 2.30 it is required to compute the derivative of the stationary distribution with respect to  $\theta$  this is typically impossible since the model of the environment is

unknown, fortunately the *policy gradient theorem* by [117] proves that 2.31.

$$\nabla_{\theta} \mathbb{J}(\theta_t) = \nabla_{\theta} \sum_{s \in \mathcal{S}} \mu_{\pi_{\theta}}(s) \left[ \sum_{a \in \mathcal{A}} \pi_{\theta}(a|s) q_{\pi_{\theta}}(s, a) \right] \propto \sum_{s \in \mathcal{S}} \mu_{\pi_{\theta}}(s) \left[ \sum_{a \in \mathcal{A}} \nabla_{\theta} \pi_{\theta}(a|s) q_{\pi_{\theta}}(s, a) \right] \quad (2.31)$$

With some tricks detailed here A.1 we can express the gradient of the loss as

$$\nabla_{\theta} \mathbb{J}(\theta_t) = \mathbb{E}_{\pi_{\theta}} [\nabla \ln \pi_{\theta}(a|s) q_{\pi_{\theta}}(s, a)] \quad (2.32)$$

And in [105] the authors go a step forward and define a general relation that policy gradient methods follow:

$$\nabla \mathbb{J}(\theta) = \mathbb{E} \left[ \sum_{t=0}^{\infty} \Psi_t \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \right] \quad (2.33)$$

where  $\Psi_t$  may be one of the following:

1.  $\sum_{t=0}^{\infty} r_t$  : total reward of the trajectory.
4.  $q_{\pi}(s_t, a_t)$  : state-action value function.
2.  $\sum_{t'=t}^{\infty} r_{t'}$  : reward following action  $a_t$
5.  $A_{\pi}(s_t, a_t)$  : advantage function.
3.  $\sum_{t'=t}^{\infty} r_{t'} - b(s_t)$  : baselined version of previous formula.
6.  $r_t + v_{\pi}(s_{t+1}) - v_{\pi}(s_t)$  : TD residual. The latter formulas use the definitions

$$A_{\pi}(s_t, a_t) := q_{\pi}(s_t, a_t) - v_{\pi}(s_t), \text{ (Advantage function)} \quad (2.34)$$

### 2.5.3.1 Reinforce

With the mathematical foundations set a widely used and relative simple policy gradient algorithm can be introduced, REINFORCE [137]. The parameters of the policy are updated with the following relation 2.35

$$\theta_{t+1} = \theta_t + \alpha R_t \nabla_{\theta} \ln \pi_{\theta}(a_t | s_t) \quad (2.35)$$

The pseudocode of the algorithm becomes 4

REINFORCE works by adjusting the parameters of the policy to give higher probabilities to actions that have higher future cumulative returns.

One problem with simple REINFORCE and other Policy gradients is having high variance, for example in one episode the policy can take a very similar actions and receive vastly different rewards or just one of the actions yields a very negative reward, all of this makes training harder so introducing a baseline helps training converge. Baseline "calibrate" the rewards to the average action that could be taken in a given state 2.16, 2.33.

The baseline has to be unbiased and not dependent on the policy parameters, a simple one and commonly used and simple baseline is a estimate of the value function of a state,  $\hat{v}(s_t, w)$ ,

---

**Algorithm 4** REINFORCE

---

```

1: Inputs: Differential policy  $\pi_\theta$ 
2: Initialize:  $history(s, a, r, t)$ , for all  $s \in S, a \in A, r \in R$ 
3: for each episode do
4:   for each timestep in episode do
5:     Following policy  $\pi$  step through environment  $env_{step}(a)$ 
6:     Append experience  $history \leftarrow s, a, r, t$ 
7:   end for
8:   for each timestep in  $history$ ,  $t=0, 1, 2, \dots, T - 1$  do
9:      $R \leftarrow \sum_{k=t+1}^T \gamma^{k-t-1} r_k$ 
10:     $\theta \leftarrow \theta + \alpha \gamma^t R \nabla \ln \pi_\theta(a_t | s_t)$ 
11:   end for
12: end for

```

---

where  $w$  are the learnable parameters for this estimate value function, this estimate would be subtracted from the action-value function and form the advantage function, seen here 2.33,  $A(s, a) = q(s, a) - v(s)$

### 2.5.4 Actor-Critic methods

Actor-Critic methods are related to policy gradients, it involves learning the policy and also learning the value function, the value function helps updating the policy. Just like the policy function, the value function approximator needs to be a differentiable. The previous algorithm works in a Monte-Carlo fashion but Actor-Critic methods can be used with Temporal-Difference (TD). An example algorithm is presented in In general Actor-Critic work similar to 5, from [109].

---

**Algorithm 5** Actor-Critic TD

---

```

1: Inputs: Differential policy  $\pi_\theta$  and Differential value function  $\hat{v}_\phi$ 
2: for each step do
3:    $s_{t+1}, r_t \leftarrow env.step(a_t | s_t)$ 
4:    $\phi \leftarrow \alpha_\phi \nabla [\hat{v}_\phi(s_t) - (r_t + \hat{v}_\phi(s_{t+1}))]^2$ 
5:    $A_\pi = r_t + \hat{v}_\phi(s_{t+1}) - \hat{v}_\phi(s_t)$ 
6:    $\theta \leftarrow \theta + \alpha \gamma^t A_\pi \nabla \ln \pi_\theta(a_t | s_t)$ 
7: end for

```

---

### 2.5.5 DDPG

Deep Deterministic Policy Gradients [80] is an off-policy algorithm similar to DQN but that is only used for continuous action spaced environments.

DDPG uses four differential neural networks, one for the Q network  $q_\phi$ , one as the target  $q_{\phi'}$ , a policy network  $\pi_\theta$  and its target network  $\pi_{\theta'}$ . The target networks are used to stabilize training, so that at a given step the parameters of a network aren't been updated with the result from its output.

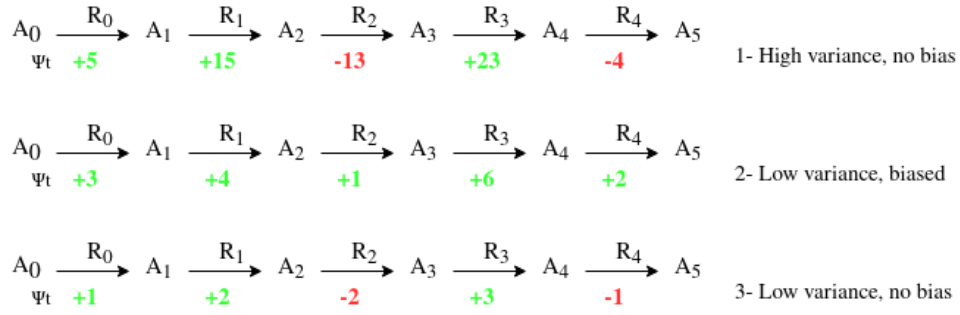


Figure 2.16: Effects of the baseline, first is updating with regular rewards and the next are different baselines with the third being the better one.

A network and its target are updated with the following rule:

$$Target \leftarrow \tau Network + (1 - \tau) Target \quad (2.36)$$

It uses a replay buffer  $D$  that stores experiences or samples,  $D \leftarrow (s, a, r, s', d)$  where  $d$  is if the state is terminal or not (1 for true, 0 otherwise).

Like DQN it's loss is related to the TD error 2.27, but since its in continuous actions it isn't possible to compute the max of the action, to do this a we use the stochastic policy target network 2.37.

$$L(\phi) = \mathbb{E}_{(s,a,r,s',d) \sim D} \left[ \left( q_\phi(s,a) - (r + \gamma(1-d)q_{\phi'}(s', \pi_{\theta'}(s'))) \right)^2 \right] \quad (2.37)$$

When collecting samples to improve exploration the output given by the policy network is added with some noise from a Normal distribution.

$$a_t = clip(\pi_\theta(s_t) + \varepsilon \sim \mathcal{N}, a_{min}, a_{max}), \text{ action for sample collection} \quad (2.38)$$

The policy parameters  $\theta$  are updated with the following rule:

$$\theta \leftarrow \alpha \nabla_{\theta} q_\phi(s_t, \mu_\theta(s_t)) \quad (2.39)$$

## 2.5.6 PPO

Proximal Policy Gradients [106] builds upon existing policy gradients algorithms but by simplifying on the way the policy is updated, doing it in a conservatively way. Algorithms like PPO and Trust Region Policy Optimization (TRPO) [104] restrict updates on policy by making sure the new policy isn't as different as the old one but PPO has the advantage of being simpler.

Starting with 2.35, as pointed by the authors of PPO it would be beneficial if with only one trajectory worth of samples it was feasible to optimize the policy many times, but this leads to

large policy updates that end up destroying it. TRPO addresses this by proposing minimizing 2.40

$$\begin{aligned} & \underset{\theta}{\text{maximize}} && \hat{\mathbb{E}}_t \left[ \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \hat{A}_t \right] \\ & \text{subject to} && \mathbb{E}_t [\text{KL}[\pi_{\theta_{\text{old}}}(\cdot|s_t), \pi_\theta(\cdot|s_t)]] \leq \delta \end{aligned} \quad (2.40)$$

where  $\theta_{\text{old}}$  the last updated policy. This has the problem of being computationally expensive and PPO proposes an improvement, minimizing instead 2.41.

$$L^{\text{CLIP}}(\theta) = \hat{\mathbb{E}}_t [\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1 - \varepsilon, 1 + \varepsilon)\hat{A}_t)] \quad (2.41)$$

where  $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$  and  $\varepsilon$  is a hyperparameter.

Minimizing this objective with a clipped term means that the updated policy doesn't move too far from the old policy.

PPO at the time of its release was consistently one of the best performing model-free RL algorithms and was extensively used.

### 2.5.7 SAC

Soft-actor Critic [45] is currently one of the state of art reinforcement learning algorithms. Its an actor-critic, off-policy and based on the policy iteration [58]. It first defines soft-policy iteration, like in regular policy iteration it starts with a policy evaluation step which computes the value of a policy based on a maximum entropy objective. This leads to the modified Bellman backup operator  $T^\pi$  that computes the soft Q-value 2.42. The value function is defined by 2.43

$$T^\pi q(\mathbf{s}_t, \mathbf{a}_t) \triangleq r(\mathbf{s}_t, \mathbf{a}_t) + \gamma \mathbb{E}_{\mathbf{s}_{t+1} \sim p} [V(\mathbf{s}_{t+1})] \quad (2.42)$$

$$V(\mathbf{s}_t) = \mathbb{E}_{\mathbf{a}_t \sim \pi} [q(\mathbf{s}_t, \mathbf{a}_t) - \alpha \log \pi(\mathbf{a}_t | \mathbf{s}_t)] \quad (2.43)$$

In the paper it is proven that repeatedly applying the modified Bellman backup operator will converge and obtain the soft Q-function of for any policy.

In the policy improvement step the policy first is made to be tractable and is restricted a Gaussian, it then gets updated towards the exponential of the new soft Q-function, it is also proven that this leads always to an improved policy. Although the soft Q-function needs to be normalized, since the output of the policy for each state is a probability, the partition function would be impossible to compute and isn't required for computing the gradients. Kullback-Leibler is used to measure how the policy function differs from the soft Q-function.

The soft Bellman residual defined in 2.44 is used to train the critics parameters.

$$J_Q(\theta) = \mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t) \sim \mathcal{D}} \left[ \frac{1}{2} (q_\theta(\mathbf{s}_t, \mathbf{a}_t) - (r(\mathbf{s}_t, \mathbf{a}_t) + \gamma \mathbb{E}_{\mathbf{s}_{t+1} \sim p} [V_\theta(\mathbf{s}_{t+1})]))^2 \right] \quad (2.44)$$

Due to the nature of the policy network, being a neural network with differential parameters an outputting a Gaussian the loss function for it parameters becomes 2.45

$$J_{\pi}(\phi) = \mathbb{E}_{\mathbf{s}_t \sim \mathcal{D}, \epsilon_t \sim \mathcal{N}} [\alpha \log \pi_{\phi}(f_{\phi}(\epsilon_t; \mathbf{s}_t) | \mathbf{s}_t) - q_{\theta}(\mathbf{s}_t, f_{\phi}(\epsilon_t; \mathbf{s}_t))] \quad (2.45)$$

In simple terms SAC has the two steps in the same fashion as policy iteration, for the policy evaluation step it uses a modified operator that takes into account the entropy of the policy. For the policy improvement, the policy function is projected to the soft Q-function. Looking in more of a graphic representation, in 2.17 there is a random soft Q-function and its exponential plotted, SAC attempts to project the policy network into the exponential (the orange line), by doing it, the policy will output low probability on actions that have low soft Q-values.

SAC has proven to be good enough to train robots to navigate an environment and has shown to have better performance in a variety of environment and tasks compared to other reinforcement learning algorithms.

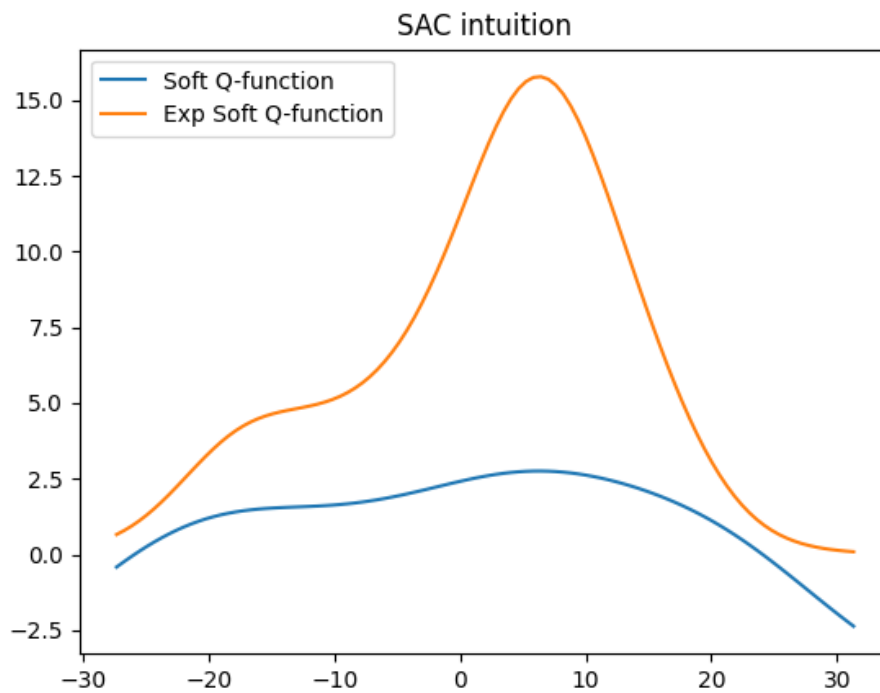


Figure 2.17: Intuition about SAC, policy network is projected to the orange function.

### 2.5.8 Curriculum Learning

In deep learning an optimization technique that improves its performance is having the algorithm solve similar but simpler tasks, introducing harder tasks as time goes by and as the algorithm converges on a solution on the simpler ones.

This approach has been used extensively with reinforcement learning with the example of [84], where it was used to make an agent solve mazes in Minecraft.

Curriculum learning takes inspiration from how humans learn and it can be easily integrated into autonomous driving, in the same way as humans when learning to drive first start with the basics, like how to accelerate, brake, park and then go for harder tasks, like driving in highways.

One problem with this type of approach is that if the tasks are too different, share too little in common, due to neural networks being prone to **Catastrophic Interference** as explained by [85], in this work it was found that a neural network trained with backpropagation to add numbers with one digit would forget how to do it after learning how to add with two digits.

A procedure to mitigate this effect is to find tasks that require the same skills but are simpler, such as learning to drive in smaller chunks of a city, rather than for example to learn to drive in an empty plot and then move to a populated city.

### 2.5.9 Transfer Learning and Domain Adaptation

To improve learning and to be able to reuse training from other tasks, models can have their knowledge transferred from one task to another different but related. There is some disagreement and inconsistency between *Transfer learning* and *Domain adaptation*, according to [78]

The notion of domain adaptation is closely related to transfer learning. Transfer learning is a general term that refers to a class of machine learning problems that involve different tasks or domains. In the literature, there isn't yet a standard definition of transfer learning. In some papers it's interchangeable with domain adaptation.

In the work [98] a more detailed explanation of the different concepts involved in transfer learning and domain adaptation is provided, it is summed in 2.18.

Transfer learning is used heavily in reinforcement learning, it allows to use neural networks trained in related (and even unrelated tasks) and improve tune them for the RL task.

## 2.6 Multi Agent Reinforcement Learning

The past algorithms presented were only applicable to a single agent or intended to. One of the most successful works in presenting a **multi agent deep reinforcement learning** method, it is scalable and easy to extend its multi-agent framework to other algorithms.

### 2.6.1 A3C

Asynchronous Advantage Actor-Critic [89] is a policy gradient algorithm that is focused on distributed training. It consists of multiple actor-critic agents, many workers interacting with their own environment, collecting samples and themselves training their actor and critic networks, then after a certain amount of time or steps a worker synchronizes their model with a global model, by pushing the gradients accumulated in training and by pulling the network weights 2.19.



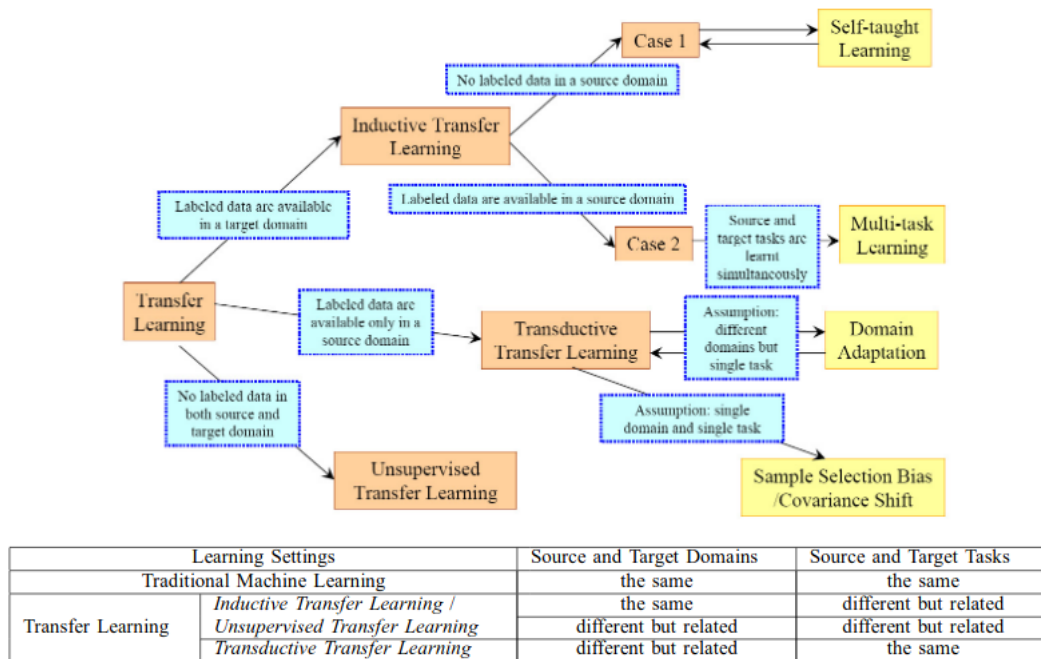


Figure 2.18: Diagram and table related to transfer learning. From [98]

## 2.7 Self-Driving Cars

Currently there are many companies working on creating a completely autonomous vehicle, level 4 or 5, the most well known are Waymo (a spin-off of Google), Tesla, GM Cruise, from General Motors, Argo AI (from Ford), Uber, Zoox, Autox, Nuro.

Most of these companies have a proprietary architecture whose details are either unknown to the public or scarce. Nevertheless companies like Waymo regularly publish articles or research works with details of their vehicles inner workings and others like Tesla post job openings and also with interviews where it is possible to gather some insights.

It should also be noted that currently there isn't a commercially available autonomous vehicle, the closest one is probably Tesla, it even claims that level 4 or 5 autonomy will be available this year (2020) as an update to current Tesla vehicles with no additional hardware [8].

### 2.7.1 Typical vehicle

With all of the information a typically autonomous research vehicle and how it is trained can be estimated (this will also be a prediction off how the first autonomous driving vehicle will look like).

#### 2.7.1.1 Hardware and Sensors

Most testing vehicles seen on the road are modified pre-existing ones, like Uber uses modified Volvo SUVs [122], Waymo uses vehicles like Toyotas, Lexus but they also have a custom built

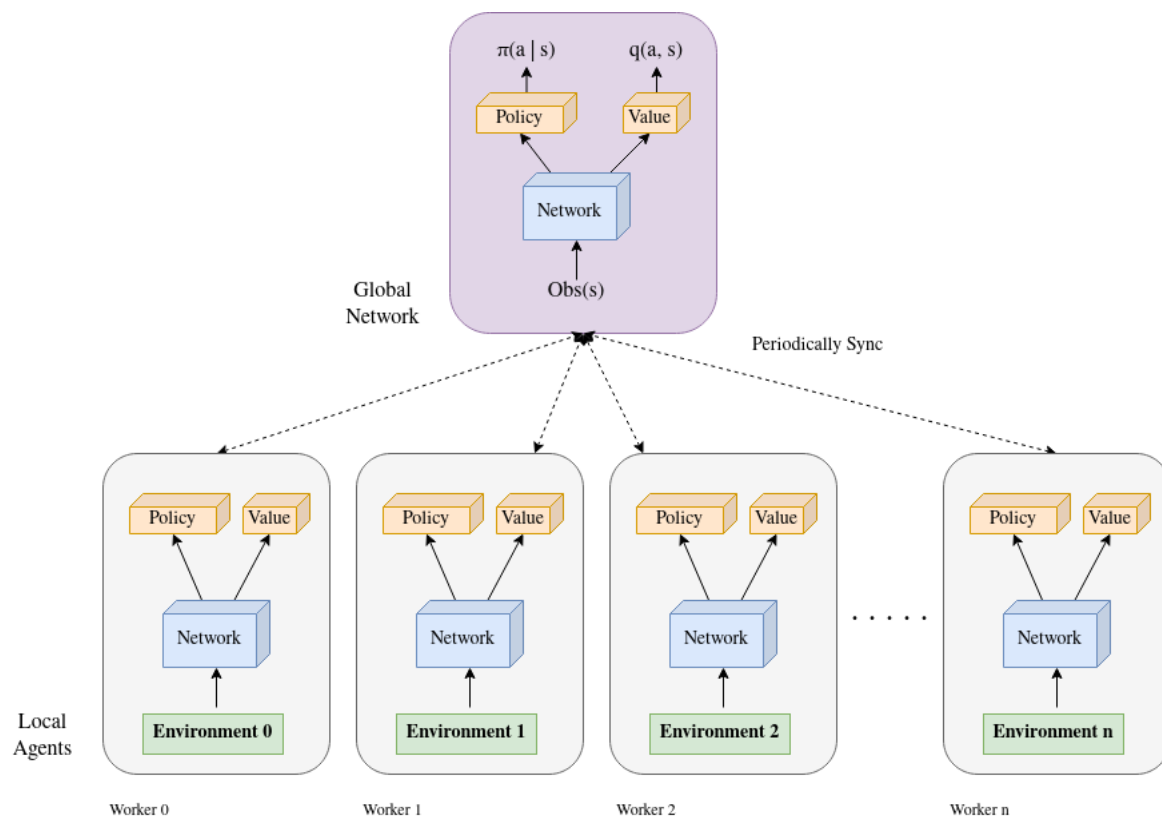


Figure 2.19: Diagram of A3C.

prototype called "Firefly" [132], it's safe to assume that modifying a commercially available non-autonomous vehicle is a practical solution and things like sensors, computers, and controllers (like servos for the steering wheel and pedals) can be bolted onto them.

For sensors, most use the following: GPS, LIDAR, RADAR, and RGB cameras, although there is a debate especially with Tesla and LIDAR, while many say that LIDAR is crucial for self-driving, Elon Musk (CEO of Tesla) declares that LIDAR is unnecessary [76], and so a Tesla vehicle ships with radar, sonar, and cameras. It should also be noted that Tesla also designs their processors, using ARM licensed CPU cores and custom-designed Neural Processing Units. Tesla claims that this custom design chip is capable of 21x improvement in frame rate compared to older hardware used (previously it was running Nvidia GPUs) while consuming 25% more power [135].

Uber also uses a hardware stack similar to Waymo, with RGB cameras, LIDAR [112].

So with all of this information, the vehicle will have LIDAR, RADAR, 360° degree RGB cameras (or at least many RGB cameras facing different directions) and also antennas for GPS, for processing, custom-designed chips are used for processing or even a custom Application-specific integrated circuit (ASIC) for better efficiency.

### 2.7.2 Training

For any self-driving vehicle the entire driving architecture is complex with many components, but most of the self-driving companies use Machine learning, Deep Learning and Neural Networks. Waymo, Tesla [119] and Uber seem to use many different neural networks models to process each frame of the input and also to create a world from the inputs and consequently optimizes and predicts the agent's next move 2.20. Waymo [6] in particular discloses more details about its development. One example of a model's architecture that they developed is *ChauffeurNet* and from what can be gathered from it shares some paradigms with others. The first one is the use of *supervised learning* or more specifically *behaviour cloning* or *Imitation learning*. This can be seen by for example Waymo open-sourcing its training dataset [130] or Tesla talking about its advantage [93]. Tesla in particular has their regular vehicles with human drivers constantly recording so that it can create a massive expert driving dataset which will in turn use supervised learning to train the autopilot AI. Tesla has stated in their job listings that they were interested in researchers with knowledge in reinforcement learning [120]. Although works like [67] highlight advantages of the use of RL, it seems like supervised learning is the preferred paradigm to train an agent.

Another important aspect common to the works presented is to forgo end-to-end learning, that is a supervised learning that gets its input from the sensors and outputs directly driving commands, the usual approach taken is to instead leverage mid-level input and output representations. In the more specific case of Waymo's *ChauffeurNet*, it has a model (a convolutions neural network) digest the sensors input, produce a intermediate representation that consists of a bird-eye view with vehicles as 2D boxes which then gets "fed" to another model (a recurrent neural network) that outputs driving trajectories which in turn are consumed by a driving controller.

Looking at Uber and Tesla descriptions on their website it seems that this pipeline of using deep learning models to process the input and then outputs a world view subsequently using another deep learning model to process it, is used, and it makes sense, it allows the agent to achieve higher sample efficiency by not having to have to implicitly learn to create an intermediate world representation (readily available models for lane detection or depth estimation can be used for this effect).

## 2.8 Related Work

In this section a variety of works are presented, from RL, to autonomous driving and also ones related to the driving simulators.

Starting with self driving [128] presents a simple actor-critic framework with DDPG. The problems faced with autonomous driving are explained (mainly the complex observation space) and also their differences compared to challenges like Atari games, tasks where RL can easily achieve superhuman performance. The effectiveness of this framework and the final model trained is shown in a simulator called The Open Racing Car Simulator (TORCS).

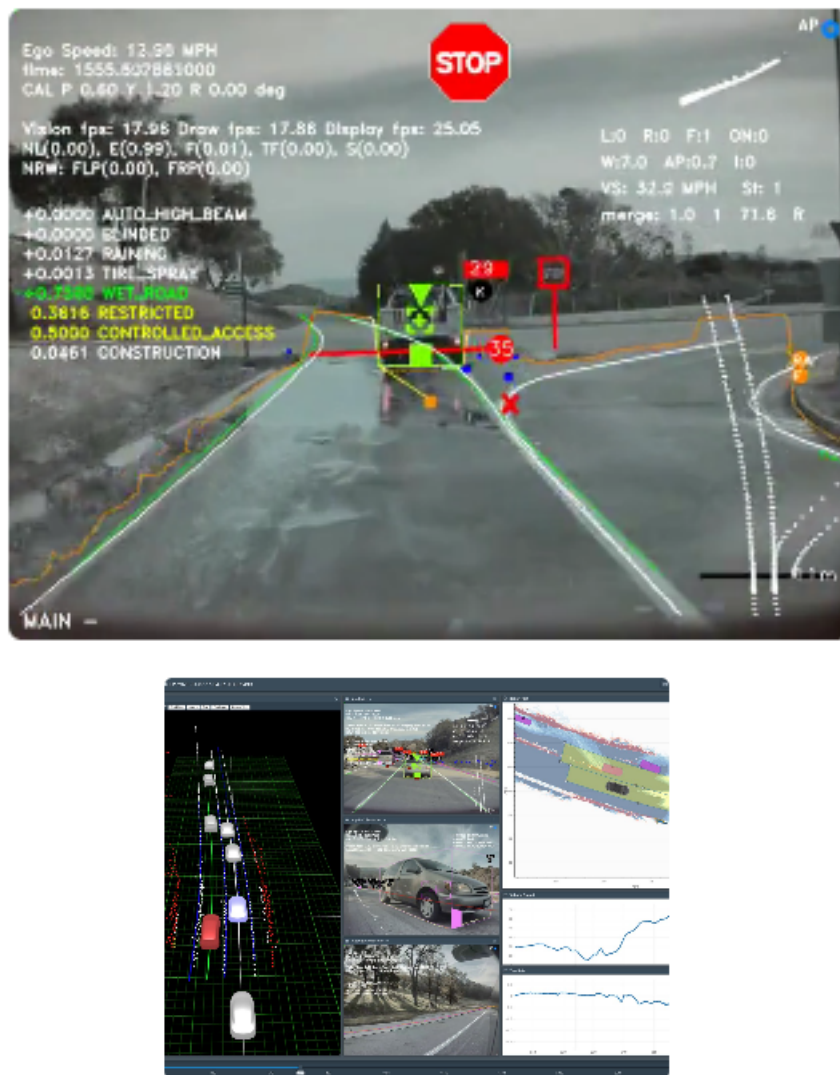


Figure 2.20: Example of Tesla's input processing (top image) and its latent encoding (the representation of the world). From [119]

The previous framework explained and also the RL algorithms studied in the last chapter are model-free and as such they don't do explicit planning or learn the environment dynamics but using a framework that uses concepts like [110] where the model is given can lead to much improved sample-efficiency. In [55] the authors create such model and apply it with good results in a scenario of highway merging and it serves as a work that successfully applies model-based techniques to autonomous driving.

An example of deep RL achieving superhuman performance is [36] where the authors train using multiple agents in the video game *Gran Turismo Sport* in a parallel fashion. Using 80 agents at once, all collecting samples and storing it in a replay buffer, another computer using SAC trains using this buffer and the final policy network is then pushed to the agents. The final network took about 73 hours of training and in the end it was able to lap a given racetrack faster than human

players. While it does achieve really good performance one aspect that needs to be mentioned is the agent's input, in this case it used simple "rays" that measure distance until the edge of the track 2.21, this means that the observation space is much simpler than using RGB cameras and other "real life" sensors.

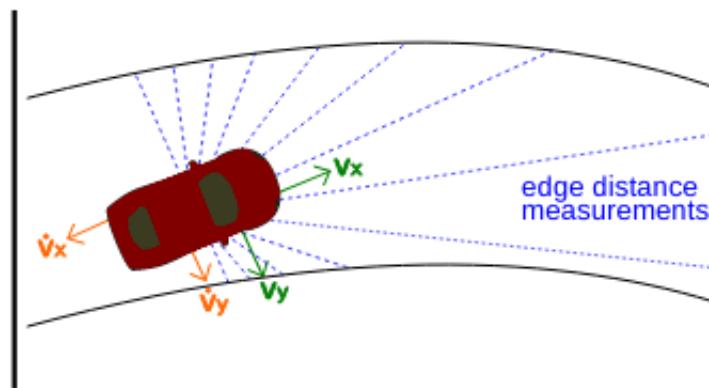


Figure 2.21: Diagram for the agent's input used in [36].

One of the first works in autonomous driving on Carla environment is presented by Codevilla et al. [24] where a range of Imitation learning methods are considered. The approach consisted of expert images derived from a front facing camera, a Modular Pipeline, that featured a PID for low level control, a planner that functions as a state machine and generates way-points that will coordinate the PID controller and a semantic segmentation network that segments each pixel into categories such as road, sidewalk, static or dynamic object and Reinforcement Learning (A3C), their results showed that RL had the worst results, it failed to avoid collision with cars and other static objects, the other two methods provided much better results but overall they all failed the task of driving on a straight empty road. The authors attempt to explain the poor performance of A3C with the following:

Why does RL underperform, despite strong results on tasks such as Atari games [91], and maze navigation [89] [69]? One reason is that RL is known to be brittle [50], and it is common to perform extensive task-specific hyperparameter search, such as 50 trials per environment as reported by Mnih et al. [89]. When using a realistic simulator, such extensive hyperparameter search becomes infeasible. We selected hyperparameters based on evidence from the literature and exploratory experiments with maze navigation. Another explanation is that urban driving is more difficult than most tasks previously addressed with RL. For instance, compared to maze navigation, in a driving scenario the agent has to deal with vehicle dynamics and more complex visual perception in a cluttered dynamic environment.

Challenges with training a reinforcement learning algorithm in Carla are studied in [113], where the authors note that training a DQN model to drive a vehicle is not only very resource intensive but yields poor generalization, instead it recommends behaviour cloning approaches noting their lightweight final models and their need for less computing resources.

In [25] an in depth study of behaviour cloning is conducted and a dataset called *CARLA100* is presented, consisting of hundreds of hours of driving experiences done by an AI that leverages privileged information (Carla API calls that don't have a translation to real life). It demonstrated good performance in a variety of scenarios.

A work that gets good better performance with a RL algorithm is [79], it acknowledges that conventional end-to-end reinforcement learning while it has benefits such as long-term decision making it spends a lot of time doing meaningless exploration and therefore takes a long time to converge, to address this the training of an agent is divided into two parts one where standard Imitation learning occurs with a deep neural network, then this network (or more specifically its weights) are transferred into a reinforcement learning (DDPG) agent, this is similar to transfer learning. This algorithm named Controllable Imitative Reinforcement Learning (CIRL) achieves the best performance when compared to others like the ones in [30] 2.22.

Task	Training conditions				New town				New weather				New town/weather			
	MP	IL	RL	CIRL	MP	IL	RL	CIRL	MP	IL	RL	CIRL	MP	IL	RL	CIRL
Straight	98	95	89	<b>98</b>	92	97	74	<b>100</b>	100	98	86	<b>100</b>	50	80	68	<b>98</b>
One turn	82	89	34	<b>97</b>	61	59	12	<b>71</b>	<b>95</b>	90	16	94	50	48	20	<b>82</b>
Navigation	80	86	14	<b>93</b>	24	40	3	<b>53</b>	<b>94</b>	84	2	86	47	44	6	<b>68</b>
Nav. dynamic	77	<b>83</b>	7	82	24	38	2	<b>41</b>	<b>89</b>	82	2	80	44	42	4	<b>62</b>

Figure 2.22: CIRL performance compared to other algorithms. From [79]

Another work that achieves good performance is End-to-end Autonomous Driving Perception with Sequential Latent Representation Learning [22], its a end-to-end approach to autonomous driving that while attempts to have a similar architecture to approaches to real life, like using a perception or decision making system but it does away with many human engineering features/heuristics or tuning of these features, since training is end-to-end. The agent learns a sequential latent representation model 2.23 and from the sensors outputs and their previous outputs it provides information about neighbouring vehicles and a semantic mask of the road. After training this latent architecture reinforcement learning algorithms can subsequently trained with it as an input. This method has both an advantage it terms of both performance and also one for providing interpretability on how the agent encodes the world. Its a complex architecture but it provides good results.

Another successful reinforcement learning work [121] uses an encoder (ResNet) first trained in a supervised manner and then the encoder is used with reinforcement learning. This architecture achieves goods results 2.24 compared to the ones in [30].

The best performing algorithm on Carla's leaderboard 2.25 is [21], its a imitation learning agent that trains in two stages, one where a privileged agent gets access to a map with all the

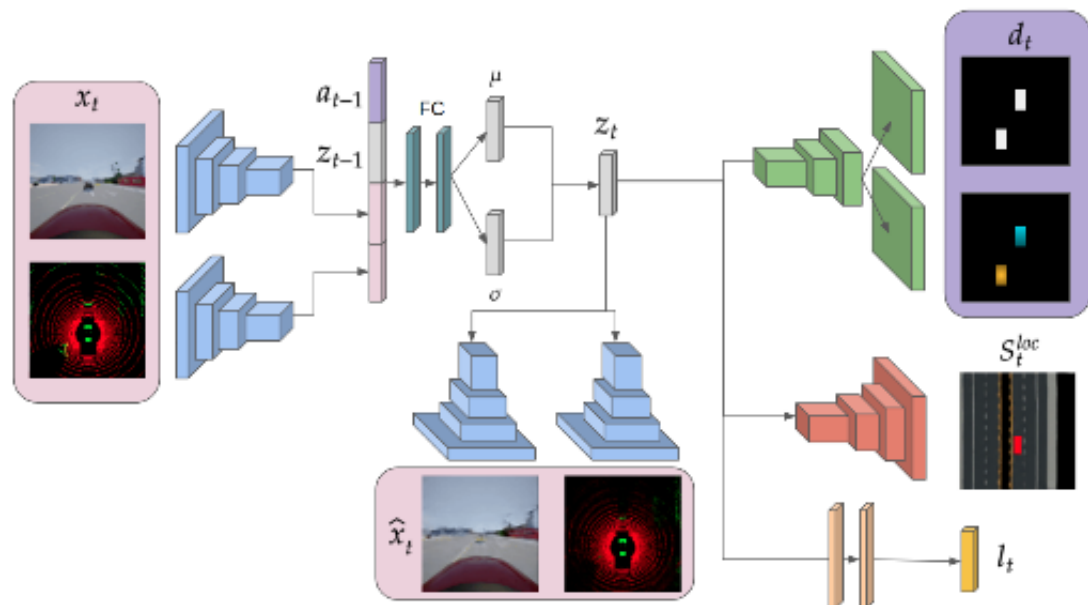
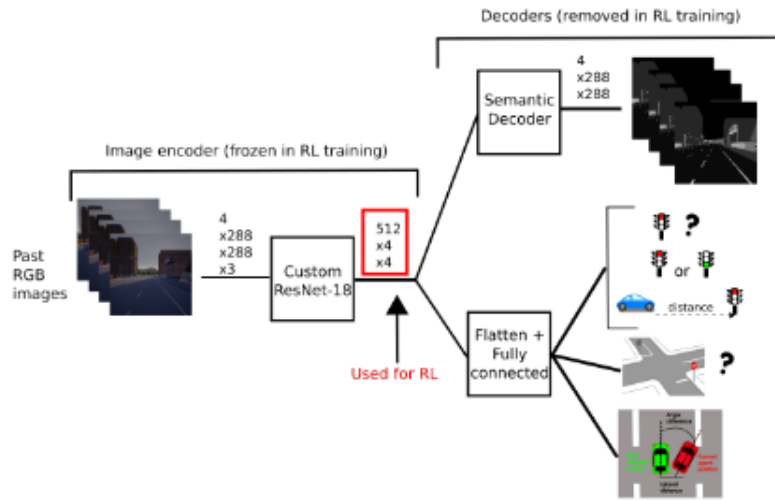


Figure 2.23: Sequential latent representation model diagram from [22]

information about traffic lights, vehicles, pedestrians, an unrealistic observation (its even called learning by cheating) and is trained with supervised learning with expert data. After this agent is done training another agent is trained to imitate the privileged agent, this time its input instead of a map is a RGB image from a front facing camera.

All of the works previously presented trained an agent in Carla and the resulting policy would be validated and only used in it. The authors of [61] attempt to transfer the resulting policy, trained by a RL algorithm in a simulator into a real life scaled model. The task attempted to solve was traversing a roundabout with other vehicles presented and the transfer from simulator to real life proved to yield good results and it shows that policies trained in simulators like Carla (although this wasn't the simulator used) could be used to, or at the very least help, train an agent capable of self-driving in real life.

A different approach to the problem of training an agent in a simulator and applying it to real life was studied in [99] where a Generative Adversarial Network (GAN) architecture learns to convert a synthetic image rendered by a simulator into a image close to a real life one, which subsequently is fed to a reinforcement learning agent. This method allows the use of any simulator and any RL algorithm and for the policy network it thinks that it is training in a real life scenario.



Task	RL	CoRL2017 (train town)				NoCrash (train town)		
		CAL	CILRS	LBC	Ours	Task	LBC	Ours
Straight	89	100	96	100	100	Empty	97	100
One turn	34	97	92	100	100	Regular	93	96
Navigation	14	92	95	100	100	Dense	71	70
Nav. dynamic	7	83	92	100	100			

Task	RL	CoRL2017 (test town)				NoCrash (test town)		
		CAL	CILRS	LBC	Ours	Task	LBC	Ours
Straight	74	93	96	100	100	Empty	100	99
One turn	12	82	84	100	100	Regular	94	87
Navigation	3	70	69	98	100	Dense	51	42
Nav. dynamic	2	64	66	99	98			

Figure 2.24: Diagram of the architecture (top image) and results (bottom image) of [121].

## Leaderboard

### Sensors Track

Copy CSV

Team	Submission	Driving score	Route completion	Infraction penalty	Collisions pedestrians	Collisions vehicles	Collisions layout	Red light infractions	Stop sign infractions
	Units	%	%	[0, 1]	Infractions/Km	Infractions/Km	Infractions/Km	Infractions/Km	Infractions/Km
+	LBC Learning by Cheating (CoRL 2019)	10.86	21.32	0.55	0.00	0.48	0.03	0.85	0.20
+	MaRLn MaRLn	6.50	23.64	0.35	0.25	0.71	1.48	0.33	0.12

Showing 1 to 2 of 2 entries

Figure 2.25: Carla’s Leaderboard (cropped) in September 2020. Leader is [21]



## Chapter 3

# Agent Framework

The main goal of this section is to present Carla, the simulator that is going to be used to develop the autonomous agent, to showcase the modifications done to Carla in order to run a reinforcement learning agent and also to benchmark approaches to training an autonomous driving agent.

The objective of this chapter is to see how well the algorithms and methods presented in the last chapter work to train a realistic autonomous driving agent (meaning using sensors and observations that have a real life counterpart).

### 3.1 Proposed Framework

The proposed architecture consists of two main modules, namely the CARLA simulator and the OpenAiGym toolkit, as presented in Figure 3.1. These modules communicate with each other and make it possible to easily code a RL agent. In the following, we briefly describe each of this modules.

#### 3.1.1 CARLA simulator

**CARLA** is a open-source simulator for autonomous driving research. The simulator was built on top of one of the most popular game developing software (Unreal Engine 4). It serves as a realistic dynamic world for developing and validating autonomous driving systems. It is designed as a

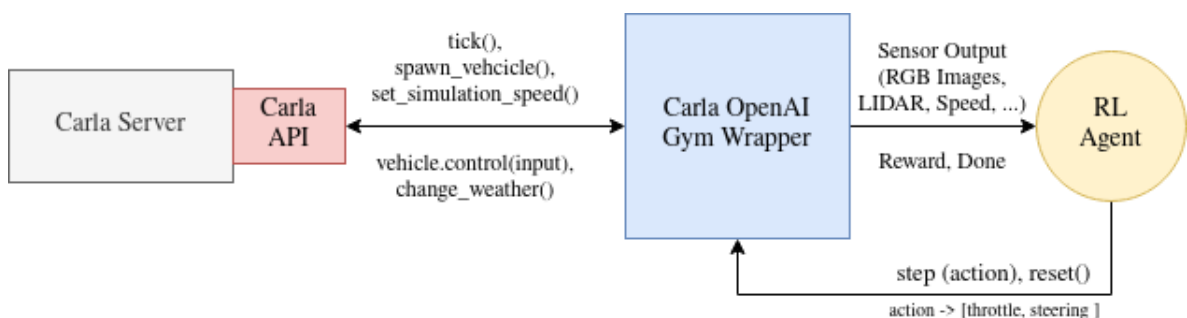


Figure 3.1: Diagram of the extensions done to Carla.



Figure 3.2: Carla Simulator. From [20].

server-client system where the server (built on C++) runs the simulation and a Python API serves as the client.

The client first sets up the simulation, it has a wide choice of settings, from the many maps/towns available (some imitating US towns other European ones), to the agent's vehicle configuration (it's type and model and also sensors attached to it, like cameras or even RADAR), pedestrians behaviour, weather patterns. During the simulation the client controls the vehicles throttle and steering and it receives from the server the sensor's output.

The server is the environment from where all the RL agents implementations will be developed and tested. Although Carla is built for this purpose it still needs to be modified in order to provide the correct functionality for the RL agents.

During the configuration of the server, one of the most important settings for Carla is the simulation speed and synchronous or asynchronous mode. Each configuration has its advantages and disadvantages, the following table describes them.

	Fixed time-step	Variable time-step
Synchronous Mode	Client has control over the simulation and its information.	Risk of non reliable simulations.
Asynchronous Mode	Good time references for information. Server runs as fast as possible.	Non easily repeatable simulations.

As per the Carla documentation for most cases the best option is to use Synchronous Mode and a fixed time step. The choice of time step is 0.1 seconds as it is the biggest without breaking the physics, also because the system used is has enough computing power the wall time and the simulation differ, meaning that when an amount of seconds pass in real life about 5 times that pass in the simulation, this is important as it shows that as computing power improves more and more samples can be obtained per time and potentially agents can achieve better performance. Note that because of frame skipping the agents see the environment 0.2 seconds forward in time every step. This time-step was chosen as in manual testing it was seen as a good trade-off between responsiveness, the time that the agent has to react to the environment and time spent processing, as decreasing it would mean less samples per second.

**Sensors.** Carla has a variety of sensors that can serve as input for the agent, such as LIDAR, RGB Camera, RADAR, GPS sensor, collision sensors.

As mentioned before the sensors chosen have to have a real life counter-part and will be similar to the ones used by works described in 2.8. The agent's real life equivalent would look close to 3.3.

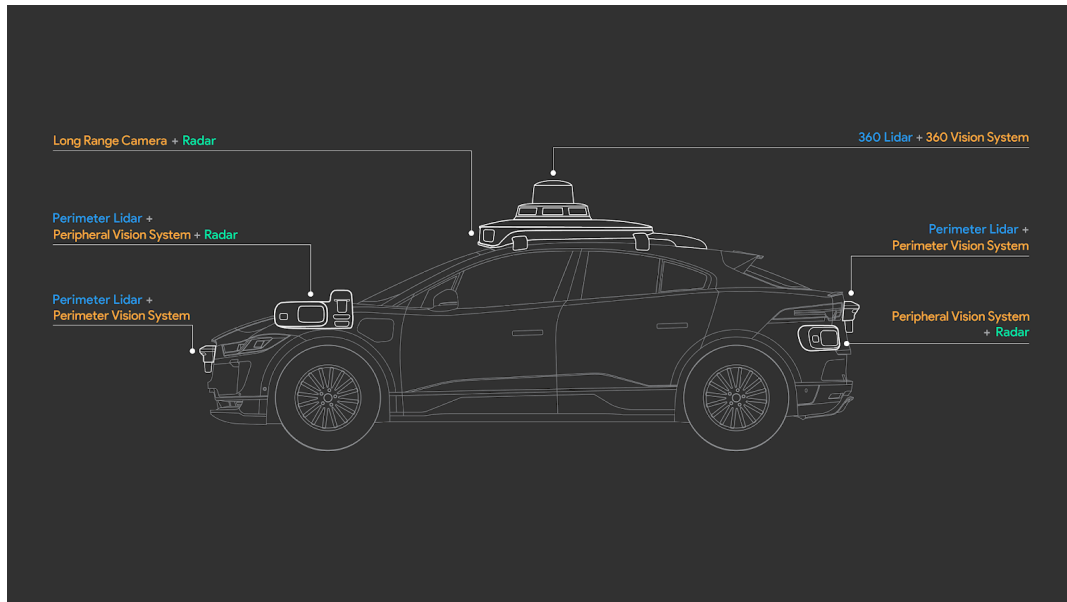


Figure 3.3: Overview of a self driving car by Waymo. From [131]

### 3.1.2 Open AI Gym

**OpenAI Gym** [4] is the simplest and most used platform/interface for developing and testing reinforcement learning tasks. It is extensively used by researchers and other works and allows different implementations to be rapidly tested and compared. It serves as a simple realization of the Markov Decision Process.

Its used for testing and comparing different RL implementations, on their official website many environments, from Atari games, to text adventures are available for rapid testing.

An OpenAI Gym conforming environment needs to implement the following functions:

- `reset()` : sets up the environment for another episode.
- `step(action)` : given an action it does a simulation step of the environment. Returns the observation, reward and if the episode is done.
- `render()` : displays a frame of the environment.
- `sample()` : returns a random action.

For the final framework of the environment a Wrapper for Carla is developed that interfaces between the RL agent and the Carla API.

### 3.1.3 Main New Functionalities

Implementing Carla as a OpenAIgym environment required writing a wrapper for Carla's API, on top of that functionally, the following new improvements were added:

- **Multiple Agent support:** Locking with a mutex is added to the world tick as otherwise if two agents ticked at almost the same time Carla would crash.
- **2D Mini-map:** The mini-map function from Carla's Scenario Runner is modified and is provided as a image tensor.
- **Sensor fusion:** Multiple sensor types such as RGB cameras and LIDAR can be combined in a single image tensor and also GPS coordinates and velocity.
- **Automatic weather:** at every 100 steps the weather changes to a new one. This is required so that the agent doesn't become overfitted or only knows how to act in a single weather pattern.
- **Route Indication:** To better help the agents navigate, arrows are drawn to indicate where it has to go, indications are also supplied in the additional observation.

In the following the two main new functionalities are described in more detail.

**Sensor Fusion.** The input of each RL agent will be composed of two RGB cameras (front facing and back facing), a LIDAR sensor, a speedometer, and the GPS coordinates. In order to simplify the processing of different sensors and dimensions (RGB Cameras output 2D data and LIDAR 3D), the data from the LIDAR is put through a 2d orthogonal projection, collapsing the Z axis, see [3.5](#). This allows the LIDAR and Camera to be fused together into a single image. If the LIDAR data was kept in 3D this would increase by a lot the complexity of the agents encoder as it would need not only to do 2D convolution but also 3D convolution, unless a completely different method of image processing was used (different from CNNs).

**Route Indication.** Treated as a regular sensor but as a way for the agent to have a simple planner and a navigation system a route is calculated using Carla's API (which in turn uses an A\* algorithm), this route is conveyed through the additional observation vector, as a one-hot encoded vector with 6 positions, they mean [Left, Right, Straight, Lane Follow, Change lane left, Change lane right]. Also when the agent gets to one of the waypoints indicated it gets a additional positive reward.

## 3.2 Training a RL Agent

To completely train a reinforcement learning agent a number of methods/architectures were explored and compared and ultimately one chosen. The first way of training an agent was by using

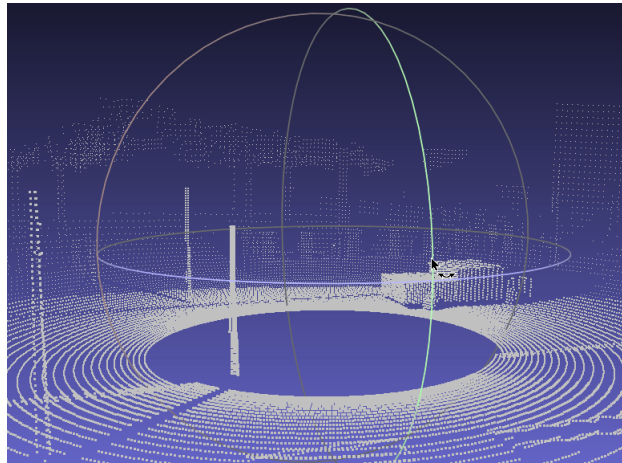


Figure 3.4: View of the LIDAR data on Meshlab. From [20].

a Variational auto encoder (VAE) 2.2.6.1, using it to reconstruct images from the sensors. The VAE is used is to decrease the dimensions of the inputs, going from an image (300x900x3) to a flat list of floats (in this case 412 floats). As previously mentioned, this provides two advantages: i) decouples the encoder from each RL agent and ii) allows the encoder to train in a supervised manner, to ensure that it can compress its input into the bottleneck. The VAE is trained with about 50k images of 300x900x3, and its bottleneck is 400 floats. All choice of hyper-parameters are explained in this section 3.5.1. After the VAE is trained each of the only RL agents looks at the bottleneck of the VAE.

**Reward Function.** When training RL agents one important aspect is having a proper reward function. Setting a reward function is a complicated task despite usually being done manually. A reward function for this environment was developed and tweaked over the course of 500+ simulations mostly using a human agent (i.e. by controlling the car using the keyboard) and seeing if the positive rewards correspond to the agent progressing towards its target goal (i.e. the coordinate (0,0,0) that corresponds to a roundabout in the center of the map) or negative rewards if the agent is doing poorly. Special attention was also given to situations where an agent performs "reward hacks". This occurs when an agent finds a set of actions that it shouldn't be doing but it gets a positive reward (e.g. the reward function needs to take into account the agent turning in circles and if does, penalize it).

$$\begin{aligned} \text{Reward} = & \alpha \text{Velocity} + \beta \text{DistanceToObjective} - \sigma \text{TimeSpent} \\ & + \rho \text{FollowingGPSRoute} - \gamma \text{CollisionImpulse} - \tau \text{Lane} + \phi \text{Success} \end{aligned} \quad (3.1)$$

- **Velocity:** this value is negative if the agent is travelling at a speed below 5 km/h and  $\frac{\text{speed in km/h}}{\text{cruise velocity}}$  where the cruise velocity is 40 km/h, if the agent is travelling at more than 50 km/h it gets a negative reward and also if it has a big angular velocity (to stop it from pulling really tight corners).



Figure 3.5: Final fused image of the cameras and LIDAR, from left to right, RGB front, RGB Back, LIDAR Projection. Note the text on the left corner is not part of the input.

- Distance To Objective: its destination is the point (0,0,0) and it gets a positive reward if since the start of the episode it approaches that position *lastdistance to destination – new distance to destination*, if on the other hand it gets further away from it, it gets a negative reward.
- Time Spent: this is a simple negative reward of -0.1 at every step, this is so the agent looks for positive rewards.
- Following GPS Route: as mentioned a route (set of waypoints) is calculated from its starting location until the destination, every time the agent reaches a waypoint it gets a positive reward, +10.
- Collision Impulse: a vector that quantifies the force applied to the vehicle by a collision, if it has a magnitude greater than a predefined value (300) or the agent breaks a basic driving law he gets a massive negative reward and the episode ends and it gets a negative reward of -100.
- Success: if the agent gets to the destination, it gets +200.

This reward function is used for the main part of training the agent. It runs for about 24 hours and the networks are saved and used for the next stage of the training. In the second stage of learning the agents is put through a set of scenarios and ran, its performance in the scenario is then used for as a sparse reward for the agent (see 3.4). While it trains while running the scenarios, it takes a long time to run through a scenario so it won't update much of its networks from scenario runs.

### 3.3 Method

As previously mentioned two architectures for the encoder were used, one with a VAE and then one consisting of a ResNet encoder.

Both types of encoder (VAE and ResNet) use Convolution neural Networks (studied here [2.2.2.2](#)), they are designed to work with images, but images aren't the entire agent's input, there is a special vector that contains information such as the speed, GPS coordinates and also the route indication, this is called the *additional observation* and it needs to skip the processing done by the convolution layers and filters and straight into the policy network.

Also since the environment is a partial observable Markov decision process, recurrent neural networks are used in the policy's and critic's network.

Figure 3.6 describes the entire pipeline and how the additional observation vector is used for training.

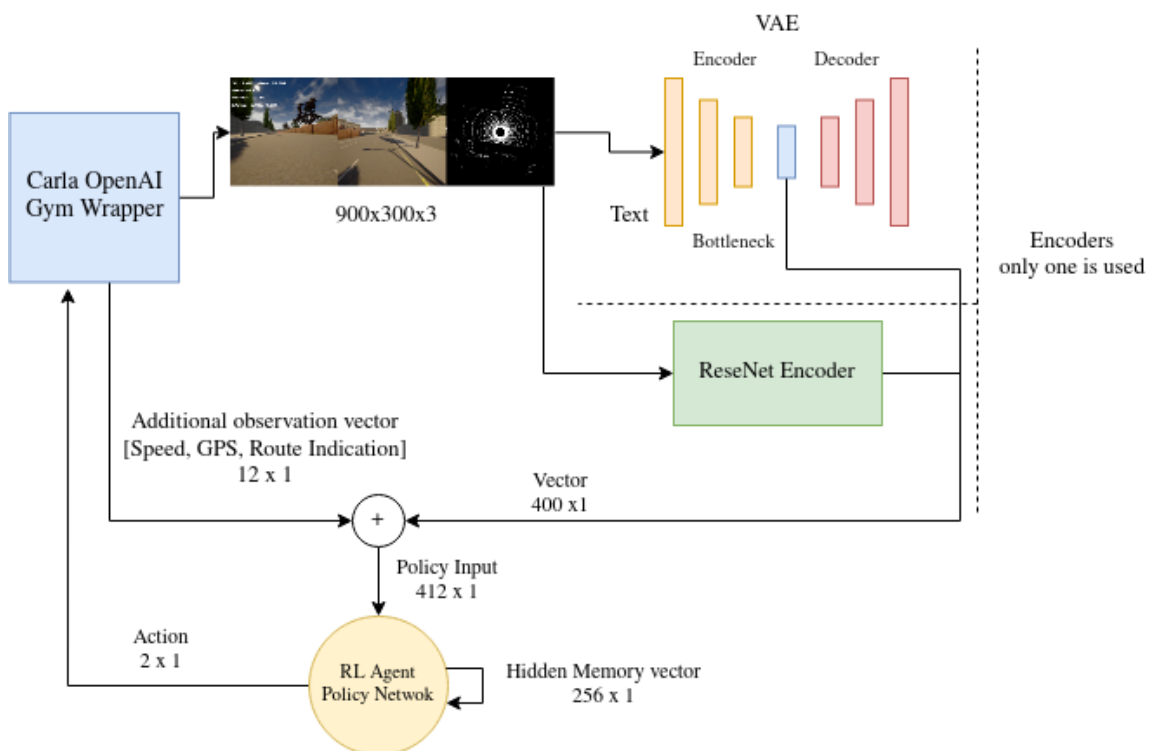


Figure 3.6: Overview of training RL Agent with the 2 types of encoders.

## 3.4 Evaluation framework

For evaluating the performance of each RL agent, after training the agent was put in a scenario called "simple benchmark" where it had to control a vehicle 100m in a straight line, on top of that it also was bench marked in the following scenarios provided by Carla's team.

### 3.4.1 Carla Scenarios

- Control Loss: Measures the ability of the agent to retain the control of the vehicle. In its lane there are patches or bumps that when driven over make the vehicle loose its grip.

- **Following Leading Vehicle:** Measures the ability of the agent to following another vehicle through an urban environment.
- **Cut in:** Measures the if the agent can correctly handle when a vehicle is gets in front of him in the highway.
- **Traffic Light:** Measures the ability of the agent to correctly pass an intersection with a traffic light with another vehicle.

Screenshots of these scenarios can be seen in [A.3](#).

### 3.4.2 Implementation

Implementation of the different reinforcement learning algorithm followed their respective original works without major modifications. The same encoders architectures were used for all the algorithms.

When it came for the amount of steps ran, they weren't the same for all, instead what was the same was the wall time, this way its possible to compare just how better more complex methods are.

When it come to policy gradients and actor-critic methods two separate networks were used for the value estimation and the policy.

With SAC and PPO a total of 5 networks were needed, 2 for the critics, 2 for their targets and a policy network.

Looking at works like SAC [45] the size of the memory buffer is in the range of millions, this can't naively be applied to the inputs of the agent. The ones in the SAC work are small, consisting of a low dimension vector of floats and could be stored easily in RAM. Initially the same approach was used but this limited the max size of the memory buffer to below 100.

The solution developed to address this problem was to develop a buffer that in a transparent would automatically compress a observation sample (composed of the sensor data, the hidden vector and additional observation in a step), store it in disk and when training the neural networks load it, decompress it and send it to the GPU. This adds overhead compared to a regular implementation and makes the amount of samples obtainable per second lower and also more dependent on hardware like storage speed, PCI Express speed and also CPU speed but theses disadvantages are necessary to obtain a reasonable memory buffer size.

In the future this implementation could benefit from technologies like GPUDirect [1].

All the algorithms are also TD(0).

### 3.4.3 Hardware Used

Training the different agents was done in GPU Lab <https://doc.ilabt.imec.be/ilabt/gpulab/>, a distributed system for running GPU-enabled docker instances, the hardware used consisted of 8 CPU threads, 10 GB of RAM and a Nvidia Tesla V100 32 GB.



### 3.4.4 VAE

To train a complete reinforcement learning agent first a Variational auto encoder (VAE) is trained. The VAE is trained to reconstruct images from the sensors. The reason why a VAE is used is to decrease the dimensions of the inputs, going from an image (300x900x3) to a flat list of floats (in this case 412 floats), the VAE also decouples the encoding of the sensor data from the Reinforcement Learning agent, meaning that it can be reused between different RL algorithms. Due to the loss function of VAEs the encoded vector is interpretable unlike regular autoencoders.

The VAE is trained with 50k images of 300x900x3, and its bottleneck is 400 variables, all hyper-parameters are in [subsection 3.5.1](#).

After the VAE is trained each of the only RL agents looks at the bottleneck of the VAE.

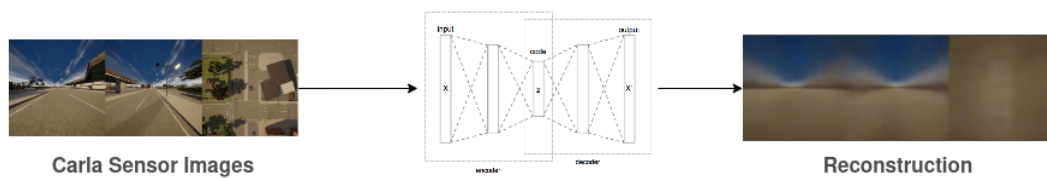


Figure 3.7: Overview of training VAE

To train a RL agent its observation consists of the bottleneck of the VAE (array of 400 floats) and an additional array of 12 floats consisting of more sensors (such as speed, GPS coordinates and route planning info), the CARLA environment also supplies the reward and if the episode is done.

All the different RL algorithms trained in this work will output a Neural Network that will be policy.

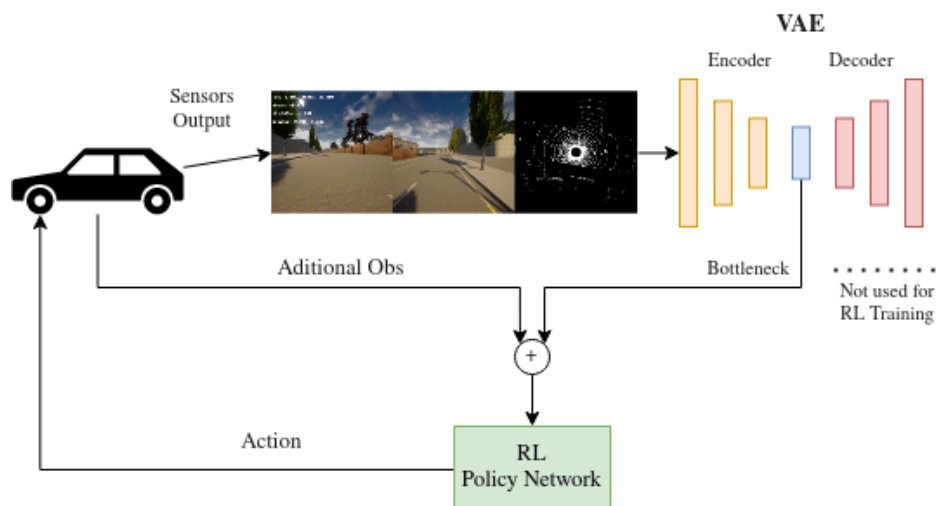


Figure 3.8: Overview of training RL Agent with VAE.

The training occurs in different stages, first a "free-roam" stage, the agent is spawned (randomly from a different set of possible points) into a town in the CARLA simulator, with no other

vehicles (unless the algorithm works distributively, like A3C). In this stage the agent doesn't know nothing about controlling the vehicle and it is meant to learn the basics about driving, such as learning not to crash and to obey to basic driving rules (traffic lights, driving between lanes). It has a simple objective of reaching the center of the map from its spawn point.

Additional details of training each of the Reinforcement learning algorithms are as follow:

- DQN: Similar to the way [90] trained, one deep neural network was used as the approximation of the value function, no additional target network was used.
- A3C: multiple agents (3 in this case) were trained simultaneous and the networks were synced at the end of the episode.
- PPO and SAC: These algorithms used in total 5 networks, 2 for the critics, 2 for the critics target and 1 for the policy network.

## 3.5 Results

### 3.5.1 Hyperparameter Tuning

Both the Carla environment itself and each RL algorithm had a set of tunable hyperparameters, like the amount of frames to skip or the size of the CNNs kernel filters. Techniques like automated machine learning [32] were studied and pondered (more specifically this library [86]) but due to the fact that they were still in early stages and also not easily incorporated to a reinforcement learning environment they weren't used, instead they were tuned in a manual manner. In total about 200 simulation runs were executed just for tuning hyperparameters, most of which were for the reward function weights. All runs and the different hyperparameters tested can be seen here <https://app.wandb.ai/gonvas/gpulab> and <https://app.wandb.ai/gonvas/carlaFinal>.

The list with all hyperparameters can be seen here [A.5](#).

### 3.5.2 VAE

The VAE was trained for approximately 24 hours, its sole purpose as mentioned was to encode images into a bottleneck that could not only make training faster for RL algorithms but also to decouple training the encoder part to training the policy network.

As seen by image 3.9 the trained VAE is not capable of generating images with much detail, it can be seen that it can only generate shapes of buildings and a road, no markings, no signs and a very blurry image overall. Training RL algorithms using the VAE as an encoder becomes impossible, different observations that would require completely different action by the agent would have near identical bottleneck encoding as for the agent would essentially be the same image. This is also the case when trying with different hyperparameters, such as the bottleneck size (trained from 50-400 size), different filter sizes (for the convolution layers), different network architectures (number of layers and total parameters) and other hyperparameters (such as learning rate and activation functions). A total of about 20 different trained networks were tested and

almost all produced similar results, although some didn't even manage to train as the loss started increasing exponential and the resulting reconstructions were noise.

Other works that use VAEs and Carla produce better results, but one major difference is the input/sensors used, in this work a realistic approach to how an agent can sense the world was taken while in those works the sensor used is a 2D "Cartoon" mini map drawn using Carla's API which doesn't have a real life counterpart.

Still the advantages for having a VAE are clear, it leads to much faster training and could even be used for model base reinforcement learning [43]. Several approaches could be used to improve performance for the VAE, such as using improved architectures/variants of a regular VAE [51] [41] [102] [124]

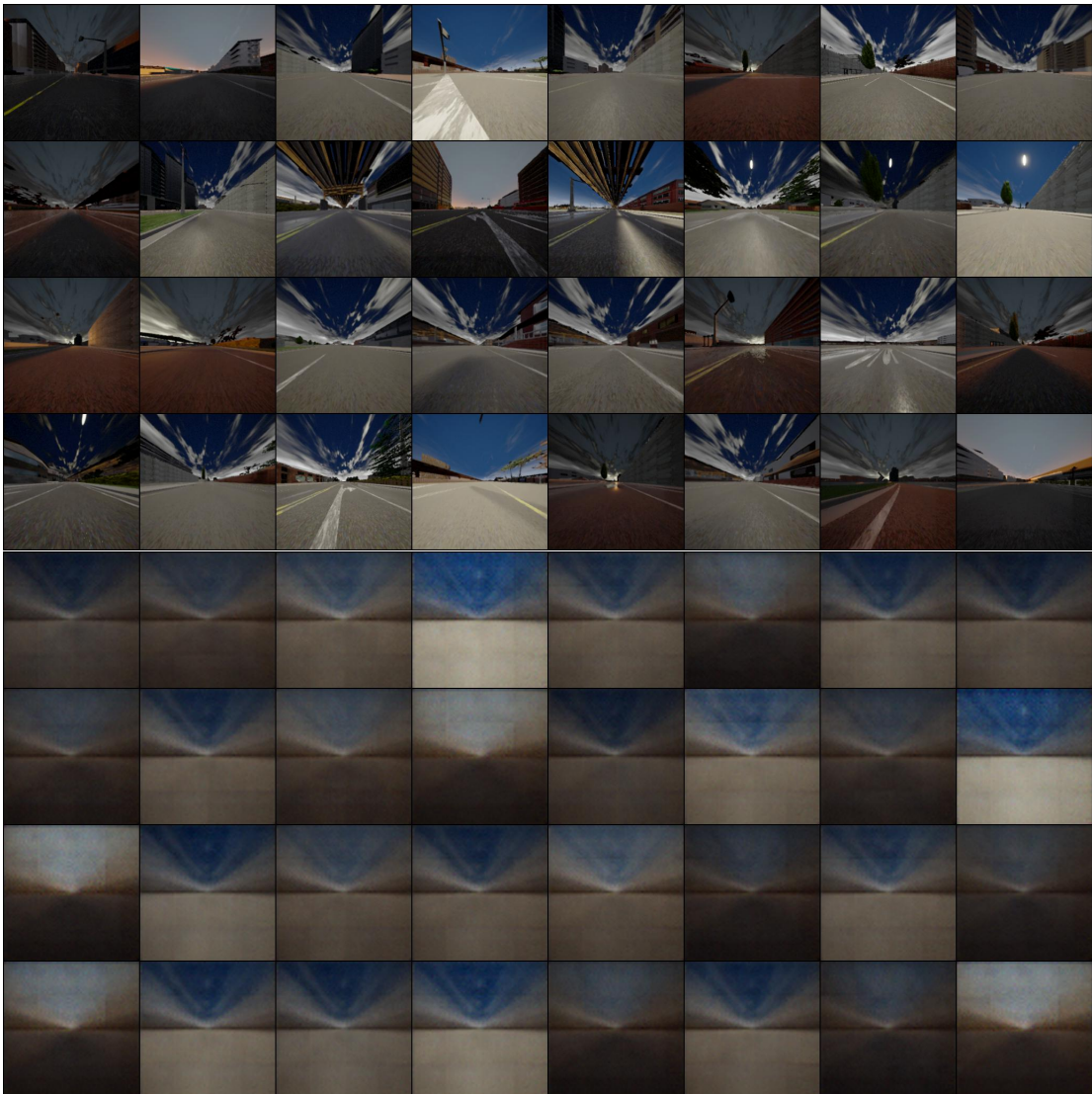


Figure 3.9: Results from VAE, input is the top image, reconstruction is the bottom image, reconstructions are blurry and unusable.

### 3.5.3 RL algorithms

Due to the failure of the VAE the architecture of the reinforcement learning agents was switched to a ResNet Encoder coupled with a recurrent neural network, depicted here 4.3. The sensors used were still the same.

From analysing the resulting policy network the best performing algorithm was SAC (although they all had similarly bad performance.) and it is the agent whose results are going to be analyzed 1, the other RL algorithms are presented here A.3.

1

The agent was trained for about 400k steps on a GPULab instance (Tesla V100) and looking at the cumulative episode reward 3.10 it is clear that while the agent improved in the first 20k steps it stagnated and it still couldn't learn a policy that could achieve positive rewards.



Figure 3.10: SAC episode cumulative reward.

Looking at the network losses it's clear the model couldn't generalize and learn, as training goes on the loss doesn't decrease and converges at a high value 3.11. The fact that the critics have a high loss might be the reason why the policy has one too, since in SAC the policy is approximated to the action-value function (critics).

Benchmarking the agent through Carla's scenarios yields bad results 3.1. The agent isn't able to finish a single one with success. The observation given to the agent is of very high dimension so the agent might have difficulty attributing a value. The problem might be too complex and the agent can't map what it sees to a representative value and subsequently can't learn a policy, it keeps getting negative rewards, outputting distance wrong action-state values resulting in big losses that are propagated through the networks but still those networks can't decide what parts of the input too look at to determine the action-state value. Another reason that learning might not be happening is that the networks are either too complex (making it stuck on a local minima 2.5) or too simple, the choice of encoder architecture is discussed in A.2 and other simpler architectures

<sup>1</sup>The training run for SAC, with the code and trained models, can be seen here <https://app.wandb.ai/gonvas/gpulab/runs/2a27ikz7>.

were also tested but they still result in this type of behaviour (see runs under the same project as 1). It is apparent why other works (discussed here 2.8) either don't use end-to-end RL (building intermediate representations) or use different simpler inputs (like Carla's 2D cartoon minimap), the input space might be too large.

Analysing the saliency maps also shows a agent not looking at anywhere relevant like lanes, signs or marking, an indication that it fails "understand" its input 3.12.

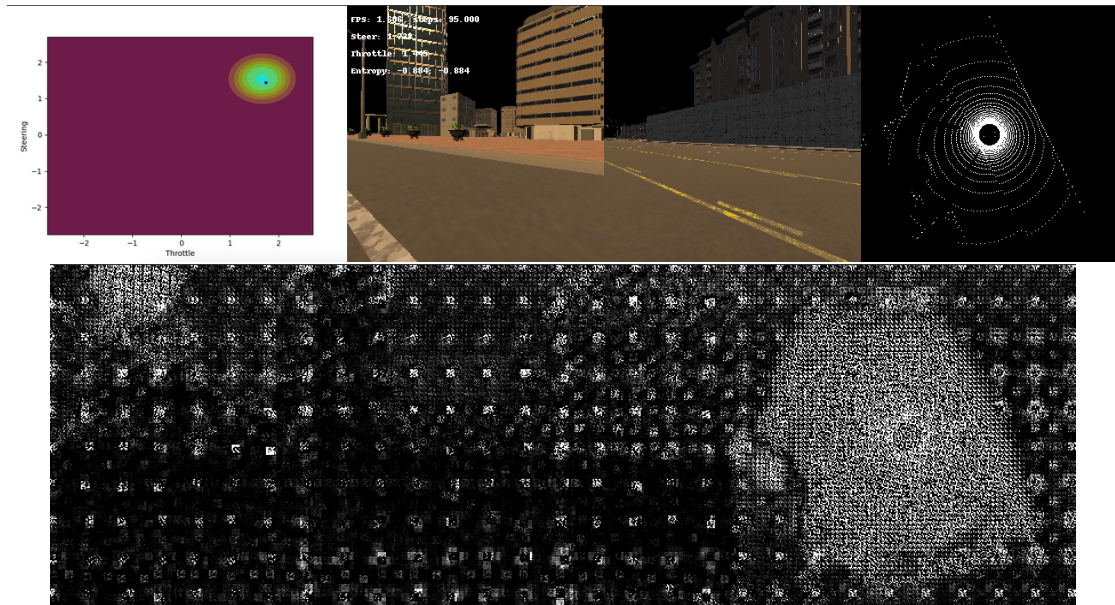


Figure 3.12: Example of SAC agent input, output action (top image) and corresponding integrated gradients.

On the top figure the first image is the output of the agent, in the case of SAC since it is continuous control, it outputs a mean and deviation from which a multivariate Gaussian distribution is created and an action is sampled from, in the case of this input, it outputted  $[1.4, 1.7]$  and the bottom image shows what parts of the input mattered for the output (this is excluding the additional observation vector and memory vector).

Looking at the integrated gradients it's apparent the agent gives a lot of importance to the LIDAR sensor, also there seems to be artifacts of tiny squares, but ultimately the agent isn't focused on more important features like the road and the lane. The images shown correspond to an episode where the agent spawned and quickly accelerated and stepped on a solid white line (it can be seen on the lower left part of the first image) which ended the episode with a very negative reward. For comparison figure 4.6 the agent seems to actually take lanes into account (an outline of a lane marking can be seen in the integrated gradients).

Scenario	Criteria	Result	Actual Value	Expect Value	Observations
Simple Benchmark	Check Collision	Failure	1.00	0.00	After taking command of the vehicle it went outside its lane.
	Distance Objective (m)	Failure	4.5	93.2	
Control Loss	Check Collision	Failure	0.00	0.00	Got to the first bump but stepped outside its lane.
	Duration (s)	Failure	60.04	60.00	
Signal Junction (Turn Right)	Check Collision	Failure	1.00	0.00	Collided with the other vehicle in the junction.
	Duration	Failure	80.04	80.00	
Follow Lead	Check Collision	Failure	1.00	0.00	Stepped outside its lane.
	Duration (s)	Failure	60.03	60.00	
Cut In	Check Collision	Failure	1.00	0.00	Stepped outside its lane.
	Duration (s)	Success	7.7	600.00	

Table 3.1: Results of SAC RL agent. Best result from 5 runs through each scenario.

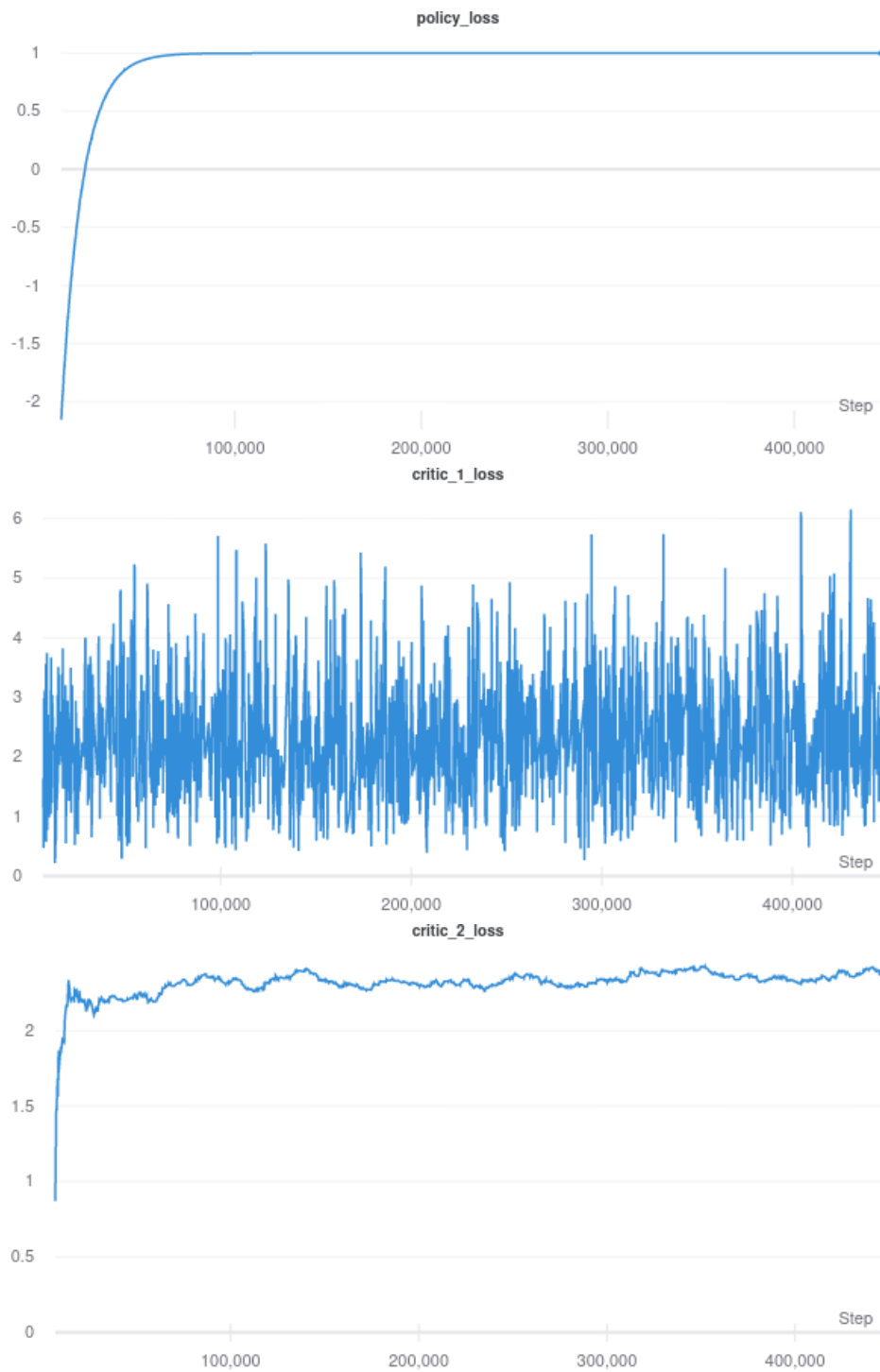


Figure 3.11: SAC losses for all the networks.





## Chapter 4

# Proposed Architecture

In the following chapter a final Multi-Agent Reinforcement Learning architecture is proposed, analyzed and tested. It leverages concepts and algorithms from the chapters previously presented and its the main contribution of this work. In the next chapters details about its choices of architectures, inputs and algorithms are detailed.

### 4.1 Overview

Training is divided into two phases, first a policy network is trained with behaviour cloning using data collected from a human driver and then a reinforcement learning algorithm, SAC, is used in conjunction with multiple agents, all cooperating with each other.

### 4.2 Distributed Reinforcement Learning

In order to leverage more computing resources and to make sure the final driving agent is able to navigate an environment there are multiple agents training at once, all working together to improve a shared model. The final training algorithm is similar to A3C and uses its distributed training concepts but the base algorithm is SAC. It works with a shared model and the trainer agents periodically pull from it, when training the agents send their gradients to the shared model, all of this is done asynchronous. The advantages of being distributed is that multiple agents are on the environment interacting with it, making possible use of multiple CPUs/GPUs, obtaining more samples and allowing for agents to cooperate. Figure 4.1 depicts the architecture.

More complex approaches to multi-agent reinforcement learning could be used like the ones in COMMA [34] where a centralized critic is used, but for this work an extension to SAC inspired by A3C was used. It creates multiple agent each with a policy network and two critic networks (as a normal SAC agent) that train in their own environment collecting samples and learning. When they update their networks the gradients are accumulated and subsequently applied to a shared model. This shared model is synced with each of the agents policy at certain intervals, similiar to A3C.

### 4.2.1 Message Parsing

Treating each agent as a node in a complete graph, before the agents networks process the inputs and return an action, messages are passed between agents asynchronous and stored in a shared buffer. Each message consists of a vector of 32 floats with the first part of the vector is hard coded with the agent speed and GPS indication. The contents of the vector are the result of a neural network layer and are trainable. The approach taken is similar to the work on Message Parsing Networks and explained in 2. The advantages of this approach is the combination of both a fixed amount of information on each message that will help the agents achieve better performance but also with a trainable component.

Each of the agents message is computed by a linear layer in the policy network after the processing of the inputs by the RES-NET blocks. At each time step an aggregate function (average in this case) is performed on all messages from the other vehicles.

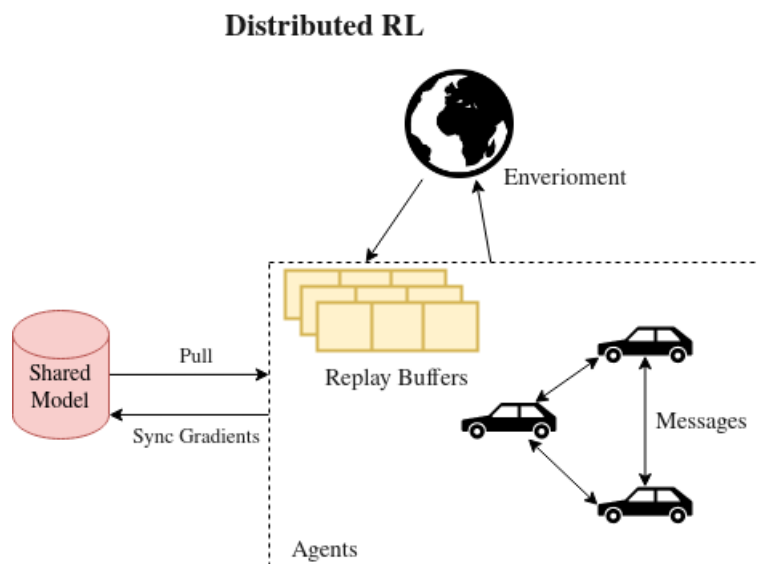


Figure 4.1: Overview of Distributed RL.

## 4.3 Two-phase Multi-agent RL Pipeline

### 4.3.1 Phase I - Behaviour Cloning

Results from many attempts with different algorithms, architectures, sensors one thing was clear, while using reinforcement learning has its advantages, in the initial parts of training the agent behaves poorly, completely randomly. This is normal as the agent initially starts with a random policy and needs to collect samples to train and obtain a better policy. The biggest problem was the speed that it was converging, usually very slowly. This can be improved, by using a more "gradual" reward function, i.e one that gives positive rewards for closing the distance to the objective, or for following the GPS route, but as Sutton says, the reward function specifies the objective not how it

is done. This can lead to reward hacking. Another way to improve this convergence is by using curriculum learning, but ultimately it takes a long time for the initially random agent to obtain positive and to converge. A better solution and already used [79] is to use expert samples as a primer for the agent (both the policy and the critic networks).

For the behaviour cloning the loss function has two components, a MSE (Mean Squared Error) 4.2 and one to keep the agent with a good amount of exploration 4.3. The final action of an agent is sampled from a Gaussian. The policy outputs a mean and deviation from which a Gaussian is constructed and sampled 4.1. If after the behaviour cloning part the policy network outputs small deviations then the agent won't explore as much, to fix this the loss function penalizes deviations not near 0.2.

$$Action_t \sim \mathcal{N}(Policy_t^1, Policy_t^2). \quad (4.1)$$

$$J_{mse} = \frac{1}{N} \sum_{i=0}^{BatchSize} (expert_i - Policy_i^2)^2 \quad (4.2)$$

$$J_{exp} = \frac{1}{N} \sum_{i=0}^{BatchSize} (|0.2 - Policy_i^2|) \quad (4.3)$$

where:

$Policy_t^1$  First component of the output from policy network at time step  $t$  used as mean

$Policy_t^2$  Second component of the output from policy network at time step  $t$ , used as deviation

### 4.3.2 Phase II - Reinforcement Learning

Once we have a policy network and a critic capable of basic navigation and control of a vehicle, a reinforcement learning phase is executed. If the initial behaviour cloning wasn't performed the agents would be acting randomly for a long time and not collecting samples that would allow them to converge, 3.

As mentioned learning is done with a modified hybrid SAC + A3C algorithm with an added message parsing step to ensure agents communicate to each other, they are all running in the same town and using the same car, sensors and networks structure.

To ensure correct learning from the agents throughout the training previous unseen expert samples are added to the sample buffer of the agents, this type of approach is part of the sub-field, in RL, offline reinforcement learning. SAC is an off-policy learning algorithm so it can learn from samples from different policies. As Sutton states in his book,

Off-policy methods also have a variety of additional uses in applications. For example, they can often be applied to learn from data generated by a conventional non-learning controller, or from a human expert.

SAC is not the best approach for learning only from offline data but its use is going to be a mixture of data derived from its policy and expert data. This concept is similar to one taken from

curriculum learning [83] where at certain times samples from when the policy had learnt to do a task, in this case it is expert data.

The reasons for adding these modifications to SAC is that from our testing the RL algorithms (DQN, REINFORCE, A3C, PPO, SAC) tested none were able to train an agent to perform basic tasks like driving in a straight line and between lanes.

#### 4.3.2.1 Offline Reinforcement Learning

SAC is the RL algorithm used, being an off-policy algorithm is able to learn from samples not taken by its current policy, such as expert data. The sub field of reinforcement learning that covers this type of learning is covered here 2. To help the agent achieve better performance, in the initial part of RL training its sample buffer is going to have some expert data samples. Note that this is only in the beginning, as SAC is not a good performing fully offline reinforcement learning algorithm, although for it can benefit from some expert samples, [71].

## 4.4 Implementation of Phase I - Behaviour Cloning

The following sections describe how different aspects of behaviour cloning were implemented and done. The pseudocode for the algorithm is presented in 6.

---

### Algorithm 6 Behaviour Cloning

---

**Inputs:** expert data samples buffer  $\hat{D}$   
Initialize policy network  $\phi$   
Initialize critic network  $\theta$   
**for** each epoch **do**  
  **for** training *step* **do**  
     $\hat{o}_t, \hat{\mathbf{a}}_t, \hat{\mathbf{r}}_t \sim \hat{D}$  ▷ Sample from expert buffer  
     $\mu_t, \sigma_t \sim \pi_\phi(a_t, o_t)$  ▷ Sample mean and deviation from policy  
     $\mathbf{a}_t \sim \mathcal{N}(\mu_t, \sigma_t^2)$  ▷ Construct Gaussian and sample action  
     $J_\pi(\phi) = MSE(\mathbf{a}_t, \hat{\mathbf{a}}_t) + |\sigma_t^2 - 0.2|$  ▷ Compute MSE error  
     $\phi \leftarrow \phi - \lambda_\pi \hat{\Delta}_\phi J_\pi(\phi)$  ▷ Update policy network  
     $\theta \leftarrow \theta - \lambda_Q \hat{\Delta}_\theta J_Q(\theta)$  ▷ Update critic network  
  **end for**  
**end for**  
**Outputs:**  $\phi, \theta$

---

#### 4.4.1 Data Collection

First the agent will be trained using supervised learning on expert data, collected from a human driver. It consists of driving experiences in Carla each is a vector with the following components:

$$Sample_t = [Obs_t, Action_t, Reward_t, Done_t] \quad (4.4)$$

The reward used is the same as the one explained in 3.

Note that the *Reward* and *Done* boolean are needed to train the critic network.

The data was collected in the *Town01* and many other vehicles (using CARLA's built in AI) and pedestrians were spawned in. The weather was also randomly changed every 100 samples.

In total about 3 hours of driving were collected totalling 17.5k samples, they were subsequently compressed using DEFLATE compression format and they take 30GB of space. <sup>1</sup>

## 4.5 Implementation Phase II - Multi Agent Reinforcement Learning

Each step in the environment is divided into two sub steps, one for processing and aggregating the messages sent by the vehicles and a another one that has the agent stepping through the environment and training its policy and critic networks. This two step method is illustrated in 4.2.

### 4.5.1 Message Parsing

Due to the nature of the agents they are all running asynchronous so while one might be running its training another might have finished and is requesting the environment to step. The way that the messages are passed between each other is using a shared message buffer, when an agent is finished and ready to output its message it simply puts in its last message in its corresponding index on the buffer. When its time to process the input it gathers all the messages and computes an average. It is possible for the agent to read the same message twice (an old message), it has to learn in training that the communications aren't always 100% reliable.

The final message that each agent processes is explained in 4.5 and the final implementation can be seen in 7

$$MsgIn = \gamma \left( \sum_{k=0}^N \frac{Msg_k}{N} \right) \quad (4.5)$$

where:

$\gamma$  = differential linear layer in a neural network

$N$  = number of agents

### 4.5.2 Training Environment

During training the max quality settings for Carla were used and the training environment of the agents consists of driving freely through Town03. Similar to curriculum learning, where harder to solve tasks are introduced in later stages of training, three different tasks are introduced in latter parts of training, pedestrians, other uncontrollable vehicles and penalties by driving trough red lights, this is to ensure that the vehicle focuses on learning the basics first (such as maintaining the vehicle between lanes).

---

<sup>1</sup>The data used in behaviour cloning is available at the URL <https://drive.google.com/file/d/1ShBVqCqrBnezCj16ZKEEzpAkhPkl-caq/>.

---

**Algorithm 7** Asynchronous SAC with message parsing for each agent process.
 

---

**Inputs:** policy network  $\phi$ , critic networks  $\theta_1, \theta_2$ , shared message buffer  $MsgBuf$  and sample Buffer  $\hat{D}$  with some expert samples, shared model  $\Pi$

Initialize target networks  $\bar{\theta}_1, \bar{\theta}_2 \leftarrow \theta_1, \theta_2$

Initialize network gradients  $d\Theta$

**for** each episode **do**

**for** each environment *step* **do**

$Msg_t = \frac{1}{N} \sum_{k=0}^N \frac{Msg_k}{N}$  ▷ Compute message from neighbour agents

$\mathbf{a}_t \sim \pi_\phi(\mathbf{a}_t, \mathbf{o}_t \cup Msg_t)$  ▷ Sample action from policy

$\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1} | \mathbf{s}_t, \mathbf{a}_t)$  ▷ Sample transition from environment

$\hat{D} \leftarrow \hat{D} \cup \{\mathbf{o}_t \cup Msg_t, \mathbf{a}_t, r_t, done_t\}$  ▷ Store transition in experiences buffer

**end for**

**for** each gradient *step* **do**

$\theta_i \leftarrow \theta_i - \lambda_Q \hat{\Delta}_{\theta_i} J_Q(\theta_i)$  for  $i \in \{1, 2\}$  ▷ Update critics networks

$d\Theta \leftarrow d\Theta + \lambda_\pi \hat{\Delta}_\phi J_\pi(\phi)$  ▷ Store Policy gradients

$\phi \leftarrow \phi - \lambda_\pi \hat{\Delta}_\phi J_\pi(\phi)$  ▷ Update local Policy network

$\alpha \leftarrow \alpha - \lambda \hat{\Delta}_\alpha J_\alpha(\alpha)$  ▷ Update temperature (for auto entropy adjustment)

$\bar{\theta}_i \leftarrow \tau \theta_i + (1 - \tau) \bar{\theta}_i$  for  $i \in \{1, 2\}$  ▷ Update target critics networks

**if** *step* mod *SyncUpdate* **then**

$\Pi \leftarrow \Pi + d\Theta$  ▷ Update shared model

$d\Theta \leftarrow 0$

**end if**

**end for**

**end for**

**Outputs:**  $\phi_1, \phi_2$

---

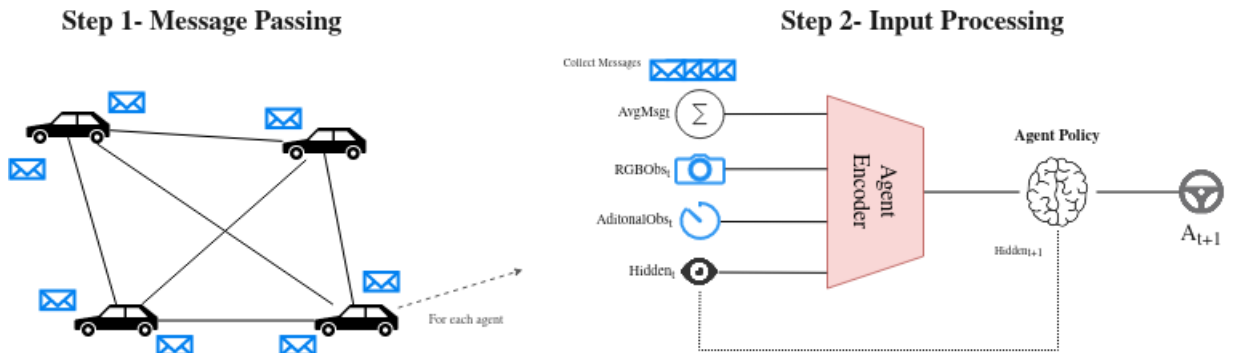


Figure 4.2: Diagram of the two step RL algorithm.

## 4.6 Experimental Setup and deployment

### 4.6.1 Scenarios

Similar to last chapter's testing, the list of scenarios used for testing and evaluating the performance of the agents are the ones designed by the CARLA team. They are traffic situations are based on the NHTSA (National Highway Traffic Safety Administration) typology.

The list are:

- Simple Lane Follow: The ego vehicle has to travel about 100 meters in a straight one way road.
- Control Loss: The ego vehicle has to travel a road that has a patch on it that makes the vehicle loose grip.
- Follow Lead Vehicle: The ego vehicle must follow another vehicle that goes along a predefined path.
- Junction without signal and with signals: The ego vehicle has to successfully navigate a junction with lights and another vehicle also approaching the junction.
- Cut In: The ego vehicle has to keep moving forward in a highway while another vehicle cuts in.

The performance metrics are given by CARLA's Scenario Runner, it indicates if the ego vehicle was successful and how much time it took.

More details about the scenarios is available in [A.3](#).

### 4.6.2 Inputs

For the inputs a realistic approach was taken, so the 2D cartoon bird eye camera and the top down RGB Camera were excluded and instead the inputs consist of two RGB cameras 300x300x3 each, one front facing and other back facing. LIDAR is also used, creating an orthogonal projection

on the Z axis, leaving a black and white 2D image. Additionally GPS indications and speed are passed to the agent. It uses the same input as described in 3 and looks like 3.5.

### 4.6.3 Encoder

Due to the poor performance of VAEs, the encoder architecture used to process the images is based around ResNet, figure 4.3 is diagram of the encoder and policy network, further details and characteristics are in A.2.

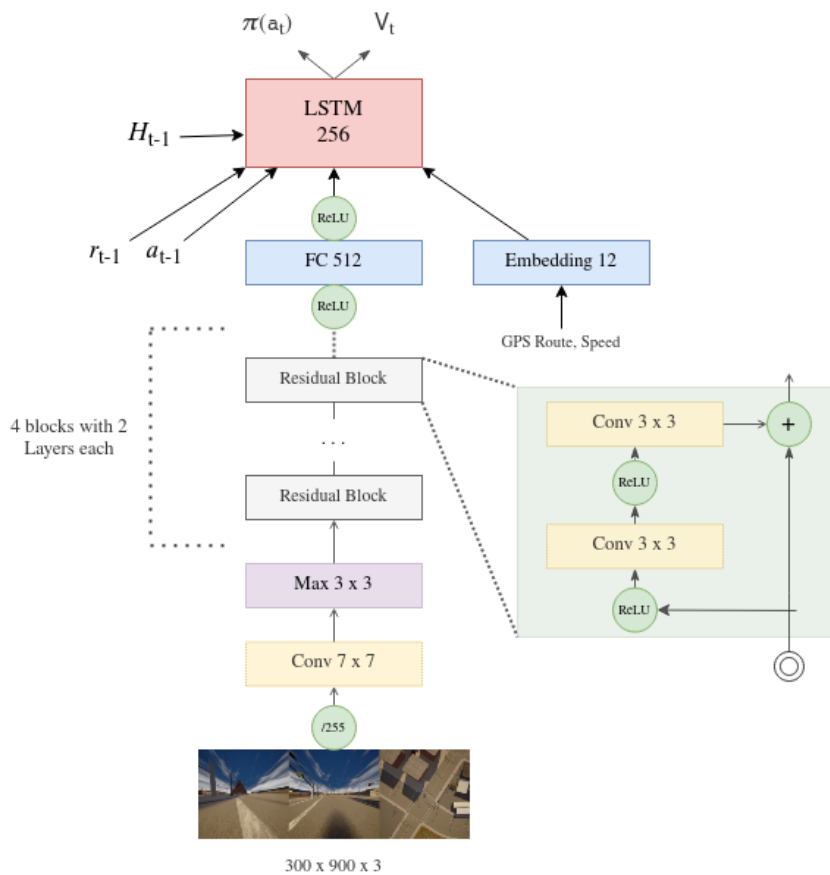


Figure 4.3: Diagram of the encoder and policy network for the agent.

## 4.7 Results

### 4.7.1 Deployment

To train the agents an instance from GPUlab was used, 3 agents in total were trained and the hardware used was a Tesla V100 with 32GB of VRAM, 64GB of RAM and 8 CPU threads. In total training took 5 days, used about 1TB of NVME storage and 80-90% of the hardware requested. As expected training takes a lot of RAM, VRAM and storage, it should be noted that fast storage



is almost a requirement for training as a large amount of data is loaded straight from storage to the GPU.

### 4.7.2 Hyperparameter Tuning

Almost all hyperparameters of this algorithm were the same as the ones in SAC and they were either tuned or just copied from the last chapter 3.5.1. Some hyperparameters are introduced by the multi-agent addition to the SAC, like the number of agents and how many steps to sync the shared model. For the number of agents three were used since it was already using a lot of hardware (64+GB of RAM) and for syncing the shared model it was done at the end of each agent's episode.

### 4.7.3 Behaviour Cloning



Figure 4.4: Training Loss for behaviour cloning

While training the policy network to on expert data it managed to converge 4.4, looking at the results it seems to have captured the "jumpiness" of keyboard controls, quickly changing from not accelerating to accelerating, it manages to keep the vehicle inside the lanes, it managed after many tries to complete the simple scenario of driving, consisting of driving about 30 meters, but it fails on all other scenarios. This is expected as it is only a first phase of training and the final policy outputs a high deviation so the agent still explores in the reinforcement learning phase. 4.5

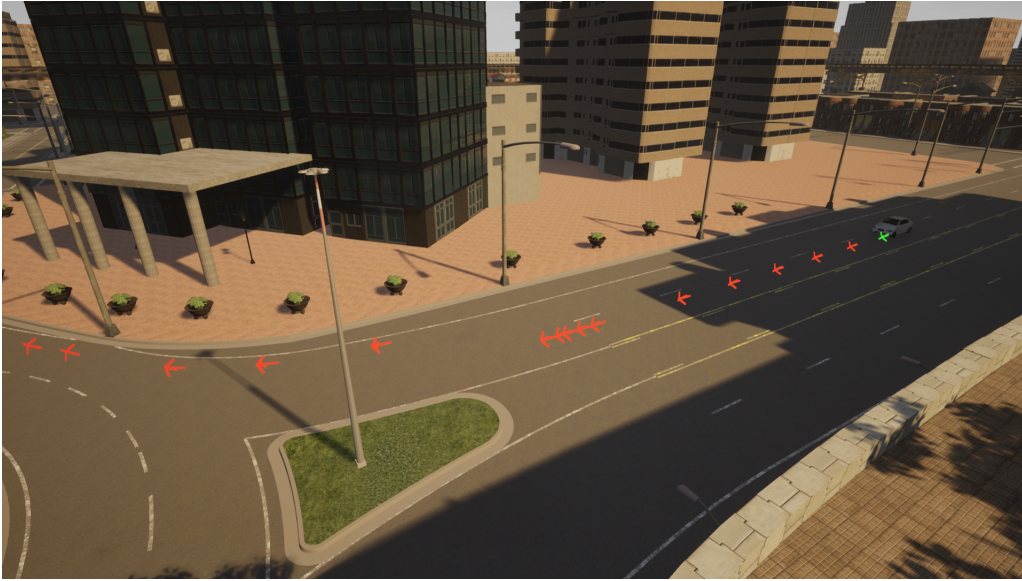


Figure 4.5: Simple scenario screenshot, agent (blue car) has to travel to the roundabout. SAC+A3C agent wasn't able to reach its destination.

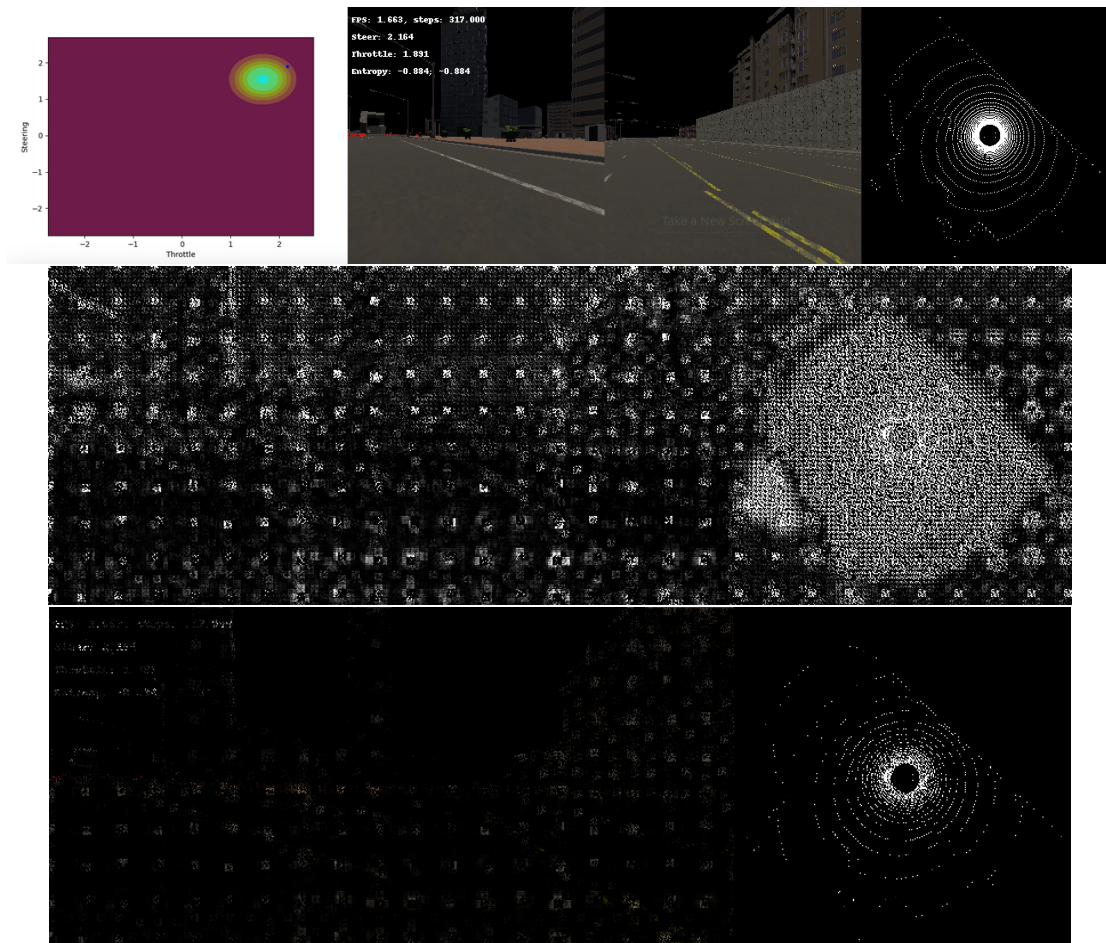


Figure 4.6: Example of Behaviour Cloning agent input, output action (top image) and corresponding integrated gradients. The bottom image is the multiplication of the top ones.

#### 4.7.4 Multi Agent RL

Looking at the final reward it looks like the agent once again wasn't able to learn, even with a decreased learning rate the policy and critics loss exploded after the 40k steps [4.7](#)

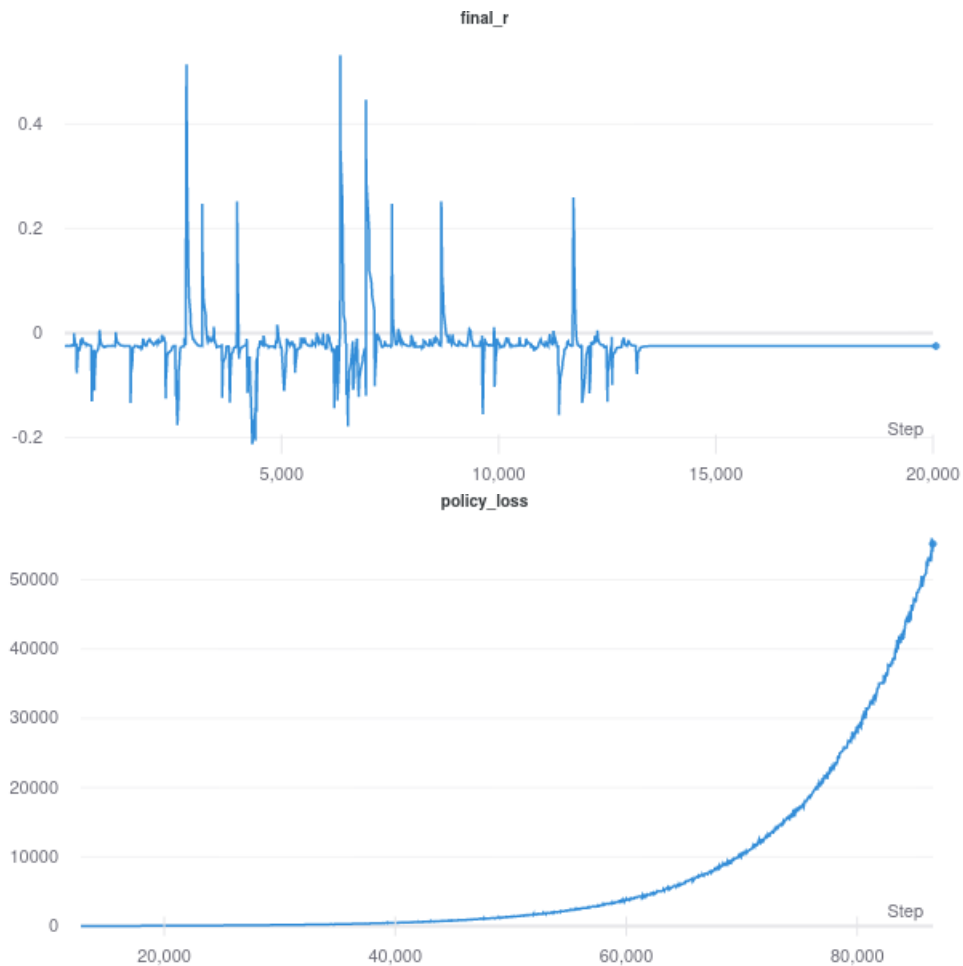


Figure 4.7: Final reward (top image) receive by one of the agents in multi RL up to 20k steps and policy loss (bottom image).

Due to the loss increase in the next result only the policy network trained for 45k steps was used. But looking at its performance on control loss scenario, it wasn't all a failure, it managed to move forward in the lane without colliding, it achieved the best result in this scenario.

The results for the scenarios were captured by running the agent on them for 5 times and analyzing the best run. The table [4.1](#) sums up all the results.

For most scenarios the agent didn't do so well, except for *Control Loss* where it managed to move forward pass a couple of bumps where the agent would loose grip, it didn't collide with anything but ultimately it wasn't able to reach the end of the road in time so it got a partial success.

The most surprising result was the highway one, the *Cut In* scenario, in one of the runs it was able to do it perfectly, move forward on the highway and not loose control when the other vehicle got in front of it. On the other runs, it didn't move in a straight line for long enough and crashed

Scenario	Criteria	Result	Actual Value	Expect Value	Observations
Simple Benchmark	Check Collision	Failure	1.00	0.00	Traveled about 15m but stepped into its right yellow lane marking.
	Distance Objective (m)	Failure	15.3	93.2	
Control Loss	Check Collision	Success	0.00	0.00	Got through some bumps but didn't make it to the end in time.
	Duration (s)	Failure	60.04	60.00	
Signal Junction (Turn Right)	Check Collision	Success	0.00	0.00	Didn't collide with anything but didn't get through the junction.
	Duration	Failure	80.04	80.00	
Follow Lead	Check Collision	Failure	1.00	0.00	Travelled for a bit but then it crashed into a pole.
	Duration (s)	Failure	60.03	60.00	
Cut In	Check Collision	Success	0.00	0.00	Vehicle kept in its lane without been disturbed by the other vehicle.
	Duration (s)	Success	22.23	600.00	

Table 4.1: Results from the different scenarios tested with SAC+A3C.

onto the other barriers. In general in this scenarios (the only one in a highway) the agent managed to move better in a straight line compared to urban ones. One possible reason might be that a highway has a "simpler" observation meaning it has less visual clutter, no buildings, no signs, no lights, no junctions just roads.

## Chapter 5

# Conclusions and Satisfaction of the objectives

In this work various reinforcement learning algorithms (including multi-agent one) were studied and tested also related techniques to autonomous driving such as using a VAE. In the end a novel multi-agent reinforcement learning algorithm is presented, consisting of a combination of two methods of training a autonomous driving agent, imitation learning and model-free reinforcement learning. A innovation is also introduced in the form of using an idea from Geometric Deep Learning, which involves treating the different vehicles as a graph and applying a Message Passing algorithm. The novel algorithm presented was used in a setting where only realistic inputs could be used, such as the RGB cameras and LIDAR, this makes the observation space really complex and it most likely contributes to its poor performance. While some works managed to obtain acceptable performance using RL some used inputs that do not have translation to real life, like Carla's 2D minimap. The massive observation space, the complex environment dynamics and the sensitivity of the many hyperparameters involved (topic mentioned here [2.8](#)) are most likely the reasons why it doesn't perform well, when compared to the same algorithms in settings like Atari games.

While end-to-end learning has managed to obtain good performance, better than hand-crafted feature engineering and extraction, in areas such as image classification while reducing the need for expert domain expertise, autonomous driving is a completely different challenge, a much more complex one. Its evident from looking at related work [section 2.8](#) that a complex architecture built with features designed by human and even building an entire perception model that explicitly condenses the information of the sensors is required. From the results presented in [chapter 3](#) [chapter 4](#) a naive architecture consisting of a neural network with an encoder and another for the policy is only capable of guiding a vehicle through a straight line in between lanes and not much more, it can't and won't solve the challenge of self-driving but it does serve as a starting architecture for architecture that might.

## 5.1 Future Work

The algorithms presented are all model-free, in recent years model-based deep reinforcement learning has been making lots of progress and it could be a massive improvement, works like [46] have an agent construct a model of the world and enable it to plan ahead and predict. This planning and encoding of the world allows it to not only achieve excellent sample efficiency but also to enable some interpretability to its policy.

When it comes to the VAE that was tested it didn't achieve acceptable performance and maybe if more advanced methods (mentioned here 3.5.2) it could have been used with reinforcement learning.

As mentioned, a major component of an agent is his encoder and a big improvement compared to the work presented here would be better encoding architecture, a VAE was tried but something more complex would probably yield much better results, like building a perception system like the one used in [22]. In the end a CNN architecture was used, more specifically a ResNet one, but even this could be improved by using more state of the art architectures like EfficientNet [118]. This encoder performance could also be improved with self supervised learning, using techniques like the ones in explored in [134] could make the entire agent more sample efficient. Memory is used so that the agent can tackle the problem of partial visibility better, it was done using GRU neural networks and while they are still used in state of the art architectures (even in the context of autonomous driving) other methods like attention [126], Hopfield networks [56], Boltzmann machines [52] or even Neural Turing Machines [40] could be studied.

Still on the topic of the encoder, one method explored but ultimately abandoned (this is due to the severe lack of VRAM on the local machine), it would involve using pre-trained models readily available for tasks like lane-detection (this work was analysed [68] A.4) that would process the sensor's output and then send it to the encoder.

Hierarchical reinforcement learning [7] could also lead to an improvement in performance, although probably not as much as the other methods proposed.

The LIDAR system used gets projected into a 2D black and white image, while this is common among other works 2.8 and makes it simple to be fused with the RGB cameras, the better way of handling this input would be using 3D convolution networks.

Also more tweaking to the hyperparameters could be made (although this is true for every problem in machine learning).

A big problem with any Reinforcement learning environment and especially this one is creating a good reward function, one that makes it impossible for the agent to reward hack but gradually gives good rewards when it starts solving the final objective, its generally done manually and there isn't a heuristic (although there are tips and tricks) or algorithm to create one except inverse reinforcement learning [2] and its an area that could be explored that could possibly extract a better reward function.

One final topic that could be investigated would be using an agent trained in Carla and then study how it performs when used in a real life, either in a scaled down environment (maybe with

RC cars) or even in a real-life vehicle.





# Appendix A

## Appendix

The following sections are supplementary material for the main thesis.

### A.1 Loss function for VAE

Proof for the loss function in the VAE.

$$\begin{aligned} & D_{\text{KL}}(q_\phi(\mathbf{z} | \mathbf{x}) \| p_\theta(\mathbf{z} | \mathbf{x})) \\ &= \int q_\phi(\mathbf{z} | \mathbf{x}) \log \frac{q_\phi(\mathbf{z} | \mathbf{x})}{p_\theta(\mathbf{z} | \mathbf{x})} d\mathbf{z} \\ &= \int q_\phi(\mathbf{z} | \mathbf{x}) \log \frac{q_\phi(\mathbf{z} | \mathbf{x}) p_\theta(\mathbf{x})}{p_\theta(\mathbf{z}, \mathbf{x})} d\mathbf{z} \\ &= \int q_\phi(\mathbf{z} | \mathbf{x}) \left( \log p_\theta(\mathbf{x}) + \log \frac{q_\phi(\mathbf{z} | \mathbf{x})}{p_\theta(\mathbf{z}, \mathbf{x})} \right) d\mathbf{z} \\ &= \log p_\theta(\mathbf{x}) + \int q_\phi(\mathbf{z} | \mathbf{x}) \log \frac{q_\phi(\mathbf{z} | \mathbf{x})}{p_\theta(\mathbf{z}, \mathbf{x})} d\mathbf{z} \\ &= \log p_\theta(\mathbf{x}) + \int q_\phi(\mathbf{z} | \mathbf{x}) \log \frac{q_\phi(\mathbf{z} | \mathbf{x})}{p_\theta(\mathbf{x} | \mathbf{z}) p_\theta(\mathbf{z})} d\mathbf{z} \\ &= \log p_\theta(\mathbf{x}) + \mathbb{E}_{\mathbf{z} \sim q_\phi(\mathbf{z} | \mathbf{x})} \left[ \log \frac{q_\phi(\mathbf{z} | \mathbf{x})}{p_\theta(\mathbf{z})} - \log p_\theta(\mathbf{x} | \mathbf{z}) \right] \\ &= \log p_\theta(\mathbf{x}) + D_{\text{KL}}(q_\phi(\mathbf{z} | \mathbf{x}) \| p_\theta(\mathbf{z})) - \mathbb{E}_{\mathbf{z} \sim q_\phi(\mathbf{z} | \mathbf{x})} \log p_\theta(\mathbf{x} | \mathbf{z}) \end{aligned} \tag{A.1}$$

### A.2 Neural Network Parameters

The complete architecture from encoder to policy network is shown in table [A.1](#) and it has the following parameters and characteristics:

- Total params: 170,625,316
- Input size (MB): 3.09
- Forward/backward pass size (MB): 163.84
- Params size (MB): 650.88
- Estimated Total Size (MB): 817.81

The table [A.2](#) has a comparison of the size (with number of parameters) of known models.

Layer (type)	Output Shape	# Param
Conv2d-1	[-1, 32, 150, 450]	4,704
BatchNorm2d-2	[-1, 32, 150, 450]	64
ReLU-3	[-1, 32, 150, 450]	0
MaxPool2d-4	[-1, 32, 75, 225]	0
Conv2dAuto-5	[-1, 32, 75, 225]	9,216
BatchNorm2d-6	[-1, 32, 75, 225]	64
ReLU-7	[-1, 32, 75, 225]	0
Conv2dAuto-8	[-1, 32, 75, 225]	9,216
BatchNorm2d-9	[-1, 32, 75, 225]	64
ResNetBasicBlock-10	[-1, 32, 75, 225]	0
Conv2dAuto-11	[-1, 32, 75, 225]	9,216
BatchNorm2d-12	[-1, 32, 75, 225]	64
ReLU-13	[-1, 32, 75, 225]	0
Conv2dAuto-14	[-1, 32, 75, 225]	9,216
BatchNorm2d-15	[-1, 32, 75, 225]	64
ResNetBasicBlock-16	[-1, 32, 75, 225]	0
ResNetLayer-17	[-1, 32, 75, 225]	0
Conv2d-18	[-1, 64, 38, 113]	2,048
BatchNorm2d-19	[-1, 64, 38, 113]	128
Conv2dAuto-20	[-1, 64, 38, 113]	18,432
BatchNorm2d-21	[-1, 64, 38, 113]	128
ReLU-22	[-1, 64, 38, 113]	0
Conv2dAuto-23	[-1, 64, 38, 113]	36,864
BatchNorm2d-24	[-1, 64, 38, 113]	128
ResNetBasicBlock-25	[-1, 64, 38, 113]	0
Conv2dAuto-26	[-1, 64, 38, 113]	36,864
BatchNorm2d-27	[-1, 64, 38, 113]	128
ReLU-28	[-1, 64, 38, 113]	0
Conv2dAuto-29	[-1, 64, 38, 113]	36,864
BatchNorm2d-30	[-1, 64, 38, 113]	128
ResNetBasicBlock-31	[-1, 64, 38, 113]	0
ResNetLayer-32	[-1, 64, 38, 113]	0
Conv2d-33	[-1, 128, 19, 57]	8,192
BatchNorm2d-34	[-1, 128, 19, 57]	256
Conv2dAuto-35	[-1, 128, 19, 57]	73,728
BatchNorm2d-36	[-1, 128, 19, 57]	256
ReLU-37	[-1, 128, 19, 57]	0
Conv2dAuto-38	[-1, 128, 19, 57]	147,456
BatchNorm2d-39	[-1, 128, 19, 57]	256
ResNetBasicBlock-40	[-1, 128, 19, 57]	0
Conv2dAuto-41	[-1, 128, 19, 57]	147,456
BatchNorm2d-42	[-1, 128, 19, 57]	256
ReLU-43	[-1, 128, 19, 57]	0
Conv2dAuto-44	[-1, 128, 19, 57]	147,456
BatchNorm2d-45	[-1, 128, 19, 57]	256
ResNetBasicBlock-46	[-1, 128, 19, 57]	0
ResNetLayer-47	[-1, 128, 19, 57]	0
Conv2d-48	[-1, 256, 10, 29]	32,768
BatchNorm2d-49	[-1, 256, 10, 29]	512
Conv2dAuto-50	[-1, 256, 10, 29]	294,912
BatchNorm2d-51	[-1, 256, 10, 29]	512
ReLU-52	[-1, 256, 10, 29]	0
Conv2dAuto-53	[-1, 256, 10, 29]	589,824
BatchNorm2d-54	[-1, 256, 10, 29]	512
ResNetBasicBlock-55	[-1, 256, 10, 29]	0
Conv2dAuto-56	[-1, 256, 10, 29]	589,824
BatchNorm2d-57	[-1, 256, 10, 29]	512
ReLU-58	[-1, 256, 10, 29]	0
Conv2dAuto-59	[-1, 256, 10, 29]	589,824
BatchNorm2d-60	[-1, 256, 10, 29]	512
ResNetBasicBlock-61	[-1, 256, 10, 29]	0
ResNetLayer-62	[-1, 256, 10, 29]	0
ResNetEncoder-63	[-1, 256, 10, 29]	0
AdaptiveAvgPool2d-64	[-1, 256, 10, 16]	0
Linear-65	[-1, 4096]	167,825,408
GRUCell-66	[-1, 256]	0
Linear-67	[-1, 2]	514
Linear-68	[-1, 2]	514

Table A.1: Design of encoder and policy network.

Name	# Param	Details
Inception-V3	24M	2017, for image classification, ResNet\Inception
AlexNet	60M	2017, for image classification, CNN
GPT-1	110M	2018, for NLP, Transformer
VGG-19	143M	2018, for image classification, ResNet
MyModel	170M	2020, Model presented above <a href="#">A.1</a>
Bert-large	340M	2018, for NLP, Transformer
GPT-2	1.5B	2019, for NLP, Transformer
GPT-3	175B	2020, for NLP, Transformer

Table A.2: Comparison of size between models.

### A.3 RL Algorithms Results

Here the results of the training of the other RL algorithms are presented here. None of them managed complete Carla's scenarios, or the resulting policy was significantly differently from each other.

The positive spikes on the graphs happen when the car gets to a speed above 10 km/h and are all below 0.5.

#### A.3.1 PPO



Figure A.1: PPO rewards in each step.

### A.3.2 DQN

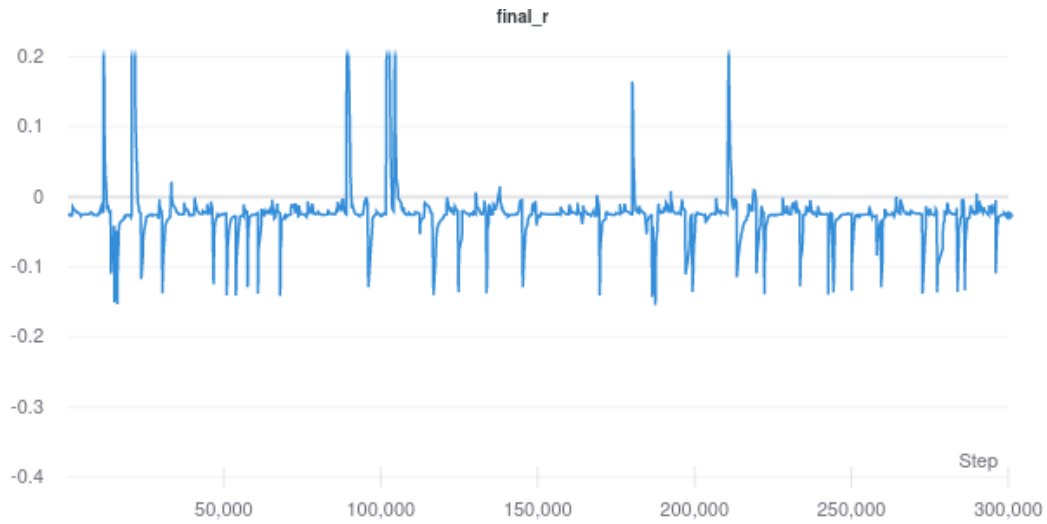


Figure A.2: DQN rewards in each step.

### A.3.3 Reinforce

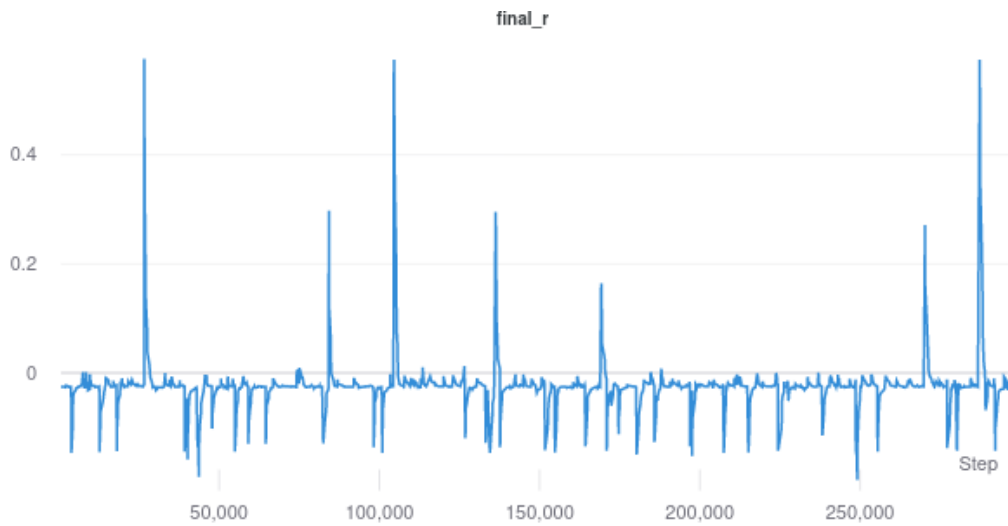


Figure A.3: Reinforce rewards in each step.

To better understand the scenarios ran in the next subsection are some screenshots of the agents running through Carla's scenarios.

### A.3.4 Scenarios



Figure A.4: PPO agent in Control Loss. The patches on the road are where the vehicle loses grip.



Figure A.5: SAC agent in Cut In scenario. The right most vehicle is the agent and he has to keep in his lane while another vehicle (in the left most lane) cuts in front of him.



Figure A.6: SAC agent in the Turn Right Scenario. The vehicle on the upper left part of the image is the agent and he has to drive through the junction.



Figure A.7: SAC agent in the Turn Right Scenario. The vehicle on the upper left part of the image is the agent and he has to drive through the junction.

## A.4 Sensor processing

While not used in the end, an addition to the encoder that could provide more performance would be to have pre-trained models for useful tasks like lane detection which then would be added to the encoder's input [A.8](#), in theory the agent's policy or even the encoder should be able to detect lanes, but with pre-trained ones, it could boost its sample efficiency. It was dropped due to the lack of VRAM available on local testing (4GB available and 1.5GB are already used for Carla and almost the rest are for the policy and critic networks).

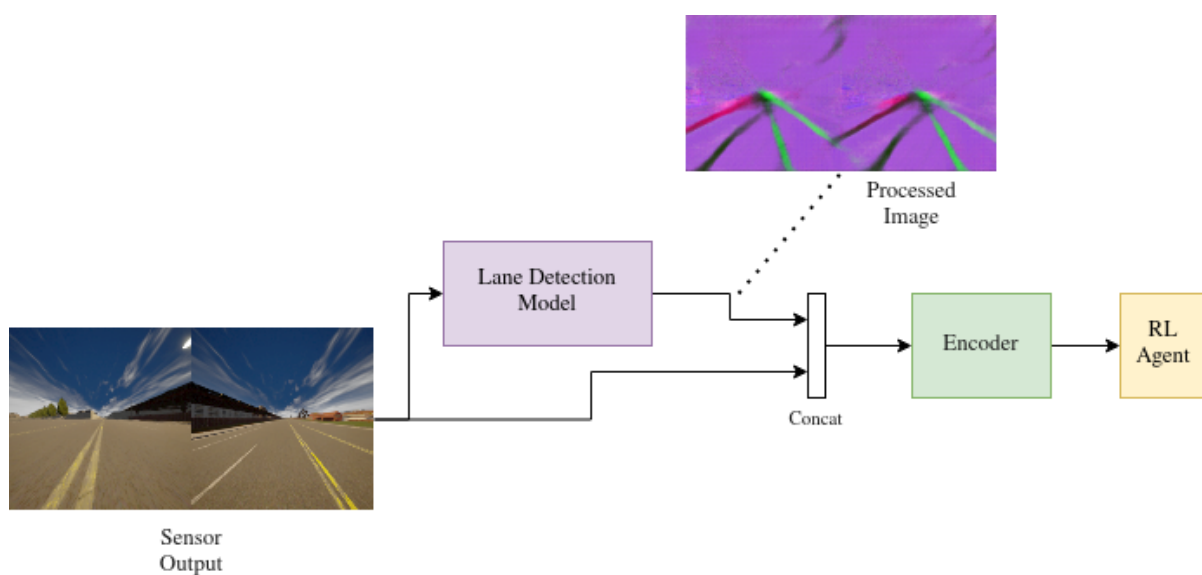


Figure A.8: Example of a initial proposed architecture.

## A.5 Hyperparameters

As mentioned before all hyperparameters were tuned manually over many different simulation runs, the following table list all of the best hyperparameters found and consequently used:

<b>Hypeparameter</b>	<b>Value</b>
Actor learning rate	3e-4
Critic learning rate	3e-4
Action Scale	2.5
Action Bias	[1, 0]
Size of Hidden vector	256
Tau	0.005
Gamma	0.99
Alpha	0.1
Epsilon	1e-6
Log Std Max	0.1
Log Std Min	0.01
Memory Start Threshold	0.1
Max Secs per episode	30
Updates per Step	1
Bootstrapping steps	1

Table A.3: Comparison of size between models.

The batch size depended on the algorithm and the amount of networks used, but it generally it would be an amount that would use 80-90% of the GPU's (Tesla V100 32GB) VRAM.

The memory buffer size was done in a way that would use up to 1 TB of storage per agent. Usually in the range of 100k.





# References

- [1] Gpudirect, Aug 2020.
- [2] Pieter Abbeel and Andrew Y Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, page 1, 2004.
- [3] Open AI. Openai baselines: Dqn, May 2014. Open AI solving games. Available in <https://openai.com/blog/openai-baselines-dqn/>.
- [4] Open AI. Openai gym, May 2020. Open AI gym. Available in <https://gym.openai.com/>.
- [5] Pierre Auger and Olivier Pironneau. Parameter identification by statistical learning of a stochastic dynamical system modelling a fishery with price variation. *Comptes Rendus. Mathématique*, 358:245–253, 07 2020.
- [6] Mayank Bansal, Alex Krizhevsky, and Abhijit S. Ogale. Chauffeurnet: Learning to drive by imitating the best and synthesizing the worst. *CoRR*, abs/1812.03079, 2018.
- [7] Andrew G Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete event dynamic systems*, 13(1-2):41–77, 2003.
- [8] BBC. Elon musk says full self-driving tesla tech 'very close', September 2020. Tesla claims of completely autonomous driving update. Available at <https://www.bbc.com/news/technology-53349313>.
- [9] Fabrice Bellard. Lossless data compression with neural networks, 2019.
- [10] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, Jun 2013.
- [11] Marc G Bellemare, Joel Veness, and Michael Bowling. Investigating contingency awareness using atari 2600 games. In *AAAI*. Citeseer, 2012.
- [12] Richard Bellman. Some applications of the theory of dynamic programming—a review. *Journal of the Operations Research Society of America*, 2(3):275–288, 1954.
- [13] Richard Bellman. Dynamic programming. *Science*, 153(3731):34–37, 1966.
- [14] Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. Weight uncertainty in neural networks, 2015.

- [15] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.
- [16] Léon Bottou. Stochastic gradient descent tricks. In *Neural networks: Tricks of the trade*, pages 421–436. Springer, 2012.
- [17] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.
- [18] Apostolos N Burnetas and Michael N Katehakis. Optimal adaptive policies for markov decision processes. *Mathematics of Operations Research*, 22(1):222–255, 1997.
- [19] Chenghui Cai and Silvia Ferrari. A q-learning approach to developing an automated neural computer player for the board game of clue®. In *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, pages 2346–2352. IEEE, 2008.
- [20] Carla. Carla documentation.
- [21] Dian Chen, Brady Zhou, Vladlen Koltun, and Philipp Krähenbühl. Learning by cheating. volume 100 of *Proceedings of Machine Learning Research*, pages 66–75. PMLR, 30 Oct–01 Nov 2020.
- [22] Jianyu Chen, Shengbo Eben Li, and Masayoshi Tomizuka. Interpretable end-to-end urban autonomous driving with latent deep reinforcement learning. *arXiv preprint arXiv:2001.08726*, 2020.
- [23] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation, 2014.
- [24] Felipe Codevilla, Matthias Müller, Antonio López, Vladlen Koltun, and Alexey Dosovitskiy. End-to-end driving via conditional imitation learning. In *International Conference on Robotics and Automation (ICRA)*, 2018.
- [25] Felipe Codevilla, Eder Santana, Antonio M. Lopez, and Adrien Gaidon. Exploring the limitations of behavior cloning for autonomous driving. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019.
- [26] Melanie Coggan. Exploration and exploitation in reinforcement learning. *Research supervised by Prof. Doina Precup, CRA-W DMP Project at McGill University*, 2004.
- [27] Tim R. Davidson, Luca Falorsi, Nicola De Cao, Thomas Kipf, and Jakub M. Tomczak. Hyperspherical variational auto-encoders, 2018.
- [28] Richard Dearden, Nir Friedman, and Stuart Russell. Bayesian q-learning. In *Aaai/iaai*, pages 761–768, 1998.

- [29] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018.
- [30] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. Carla: An open urban driving simulator. *arXiv preprint arXiv:1711.03938*, 2017.
- [31] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *arXiv preprint arXiv:1808.05377*, 2018.
- [32] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in neural information processing systems*, pages 2962–2970, 2015.
- [33] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.
- [34] Jakob Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. Counterfactual multi-agent policy gradients, 2017.
- [35] Coalition for Future Mobility. Highly automated technologies, often called self-driving cars, promise a range of potential benefits., September 2020. Information provided by Coalition for Future Mobility. Available in <https://coalitionforfuturemobility.com/benefits-of-self-driving-vehicles/>.
- [36] Florian Fuchs, Yunlong Song, Elia Kaufmann, Davide Scaramuzza, and Peter Duerr. Superhuman performance in gran turismo sport using deep reinforcement learning. *arXiv preprint arXiv:2008.07971*, 2020.
- [37] Richard Wallace Gary Silberg. Self-driving cars: The next revolution., September 2020. Whitepaper. Available in <https://www.cargroup.org/publication/self-driving-cars-the-next-revolution/>.
- [38] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry, 2017.
- [39] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. <http://www.deeplearningbook.org>.
- [40] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines, 2014.
- [41] Karol Gregor, George Papamakarios, Frederic Besse, Lars Buesing, and Theophane Weber. Temporal difference variational auto-encoder, 2018.
- [42] Bonnie Gringer. History of the autonomous car, Apr 2020.
- [43] David Ha and Jürgen Schmidhuber. Recurrent world models facilitate policy evolution. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 2450–2462. Curran Associates, Inc., 2018.
- [44] Tuomas Haarnoja, Haoran Tang, Pieter Abbeel, and Sergey Levine. Reinforcement learning with deep energy-based policies, 2017.

- [45] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018.
- [46] Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination, 2019.
- [47] Hado V Hasselt. Double q-learning. In *Advances in neural information processing systems*, pages 2613–2621, 2010.
- [48] Matthew Hausknecht, Joel Lehman, Risto Miikkulainen, and Peter Stone. A neuroevolution approach to general atari game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(4):355–366, 2014.
- [49] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.
- [50] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters, 2017.
- [51] Irina Higgins, Loic Matthey, Arka Pal, Christopher Burgess, Xavier Glorot, Matthew Botvinick, Shakir Mohamed, and Alexander Lerchner. beta-vae: Learning basic visual concepts with a constrained variational framework. 2016.
- [52] G. E. Hinton. Boltzmann machine. *Scholarpedia*, 2(5):1668, 2007. revision #91076.
- [53] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors, 2012.
- [54] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.
- [55] Carl-Johan Hoel, Katherine Driggs-Campbell, Krister Wolff, Leo Laine, and Mykel J Kochenderfer. Combining planning and deep reinforcement learning in tactical decision making for autonomous driving. *IEEE Transactions on Intelligent Vehicles*, 5(2):294–305, 2019.
- [56] J. J. Hopfield. Hopfield network. *Scholarpedia*, 2(5):1977, 2007. revision #91363.
- [57] Xianxu Hou, Linlin Shen, Ke Sun, and Guoping Qiu. Deep feature consistent variational autoencoder, 2016.
- [58] Ronald A Howard. Dynamic programming and markov processes. 1960.
- [59] Junling Hu and Michael P Wellman. Nash q-learning for general-sum stochastic games. *Journal of machine learning research*, 4(Nov):1039–1069, 2003.
- [60] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.
- [61] Kathy Jang, Eugene Vinitzky, Behdad Chalaki, Ben Remer, Logan Beaver, Andreas Malikopoulos, and Alexandre Bayen. Simulation to scaled city: zero-shot policy transfer for traffic control via autonomous vehicles. pages 291–300, 04 2019.

- [62] Jeremy. Variational autoencoders., Mar 2018.
- [63] Byung-Gook Kim, Yu Zhang, Mihaela Van Der Schaar, and Jang-Won Lee. Dynamic pricing and energy consumption scheduling with reinforcement learning. *IEEE Transactions on Smart Grid*, 7(5):2187–2198, 2015.
- [64] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.
- [65] Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2013.
- [66] Durk P Kingma, Tim Salimans, and Max Welling. Variational dropout and the local reparameterization trick. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2575–2583. Curran Associates, Inc., 2015.
- [67] B Ravi Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A. Al Sallab, Senthil Yogamani, and Patrick Pérez. Deep reinforcement learning for autonomous driving: A survey, 2020.
- [68] Yeongmin Ko, Younkwan Lee, Shoaib Azam, Farzeen Munir, Moongu Jeon, and Witold Pedrycz. Key points estimation and point instance segmentation approach for lane detection, 2020.
- [69] A Dosovitskiy V Koltun. Learning to act by predicting the future. *ICLR 2017*, 2017.
- [70] Uday Kulkarni, Meena S M, Sunil V Gurlahosur, and M. Uma. Classification of cultural heritage sites using transfer learning. pages 391–397, 09 2019.
- [71] Aviral Kumar, Justin Fu, George Tucker, and Sergey Levine. Stabilizing off-policy q-learning via bootstrapping error reduction, 2019.
- [72] Siddharth Krishna Kumar. On weight initialization in deep neural networks, 2017.
- [73] Jon Lamonte. Sydney metro: a game changer for passengers, December 2019. Article by Intelligent Transport. Available in <https://www.intelligenttransport.com/transport-articles/91304/sydney-metro-a-game-changer-for-passengers/>.
- [74] Wired Brand Land. A brief history of autonomous vehicle technology, Aug 2016.
- [75] Jae Won Lee, Jonghun Park, O Jangmin, Jongwoo Lee, and Euuseok Hong. A multiagent approach to  $q$ -learning for daily stock trading. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 37(6):864–877, 2007.
- [76] Timothy B. Lee. Elon musk: “anyone relying on lidar is doomed.” experts: Maybe not, August 2019. Article by Arstechnica. Available in <https://arstechnica.com/cars/2019/08/elon-musk-says-driverless-cars-dont-need-lidar-experts-arent-so-sure/>.
- [77] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. Visualizing the loss landscape of neural nets, 2017.
- [78] Qi Li. Literature survey: domain adaptation algorithms for natural language processing. 2012.

- [79] Xiaodan Liang, Tairui Wang, Luona Yang, and Eric Xing. Cirl: Controllable imitative reinforcement learning for vision-based self-driving. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 584–599, 2018.
- [80] 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, 2015.
- [81] Direct Line. How long does it take to learn to drive?, September 2020. Article by Direct Line. Available in <https://www.directline.com/car-cover/how-long-does-it-take-to-learn-to-drive>.
- [82] Shih-Chung B Lo, Heang-Ping Chan, Jyh-Shyan Lin, Huai Li, Matthew T Freedman, and Seong K Mun. Artificial convolution neural network for medical image pattern recognition. *Neural networks*, 8(7-8):1201–1214, 1995.
- [83] Tabet Matiisen, Avital Oliver, Taco Cohen, and John Schulman. Teacher-student curriculum learning, 2017.
- [84] Benjamin Rosman Herman A. Engelbrecht Matthew Reynard, Herman Kamper. Combining primitive dqns for improved reinforcement learning in minecraft. *Conference: 2020 International SAUPEC/RobMech/PRASA Conference*, 2020.
- [85] Michael McCloskey and Neal J Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*, volume 24, pages 109–165. Elsevier, 1989.
- [86] Hector Mendoza, Aaron Klein, Matthias Feurer, Jost Tobias Springenberg, Matthias Urban, Michael Burkart, Max Dippel, Marius Lindauer, and Frank Hutter. Towards automatically-tuned deep neural networks. In Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren, editors, *AutoML: Methods, Systems, Challenges*, chapter 7, pages 141–156. Springer, December 2018. To appear.
- [87] Dmytro Mishkin and Jiri Matas. All you need is a good init, 2015.
- [88] Tom M. Mitchell. *Machine Learning*. Carnegie Mellon University, 1997. <http://www.deeplearningbook.org>.
- [89] 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, 2016.
- [90] 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.
- [91] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.
- [92] Ralph Neuneier. Enhancing q-learning for optimal asset allocation. In *Advances in neural information processing systems*, pages 936–942, 1998.

- [93] Applico Nicholas L. Johnson. Tesla's 1.3 billion mile advantage, September 2020. Tesla's data advantage againts other competitors. Available here <https://www.applicoinc.com/blog/teslas-1-3-billion-mile-advantage/>.
- [94] Jari Nurmi. *Processor design: system-on-chip computing for ASICs and FPGAs*. Springer Science & Business Media, 2007.
- [95] Nvidia. Landsape ai, July 2020. Article by NVIDIA. Available in <http://nvidia-research-mingyuliu.com/gaugan/>.
- [96] Society of Automotive Engineers. Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles, September 2020. Available in [https://www.sae.org/standards/content/j3016\\_201806/](https://www.sae.org/standards/content/j3016_201806/).
- [97] Utku Ozbulak. Pytorch cnn visualizations. <https://github.com/utkuozbulak/pytorch-cnn-visualizations>, 2019.
- [98] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2009.
- [99] Xinlei Pan, Yurong You, Ziyang Wang, and Cewu Lu. Virtual to real reinforcement learning for autonomous driving. *arXiv preprint arXiv:1704.03952*, 2017.
- [100] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.
- [101] City Metric Rachel Holdsworth. Lille had europe's first fully automated metro system. it opened in 1983, September 2020. Article by City Metric. Available in <https://www.citymetric.com/transport/lille-had-europe-s-first-fully-automated-metro-system-it-opened-1983-3856>.
- [102] Ali Razavi, Aaron van den Oord, and Oriol Vinyals. Generating diverse high-fidelity images with vq-vae-2, 2019.
- [103] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization?, 2018.
- [104] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization, 2015.
- [105] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2015.
- [106] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.
- [107] Ramprasaath R. Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. *International Journal of Computer Vision*, 128(2):336–359, Oct 2019.
- [108] Darja Šemrov, Rok Marsetič, Marijan Žura, Ljupčo Todorovski, and Aleksander Srdic. Reinforcement learning approach for train rescheduling on a single-track railway. *Transportation Research Part B: Methodological*, 86:250–267, 2016.

- [109] Sergey. Actor-critic algorithms, 2017.
- [110] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dhharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017.
- [111] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps, 2013.
- [112] Danielle Muoio Skye Gould, Yu Han. Here’s the tech that lets uber’s self-driving cars see the world, September 2020. Details about Uber’s self driving vehicles. Available at <https://www.businessinsider.com/how-ubers-driverless-cars-work-2016-9>.
- [113] Bharathwaj Krishnaswami Sreedhar and Nagarajan Shunmugam. Deep learning for hardware-constrained driverless cars. In *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 26–29. IEEE, 2020.
- [114] Stanford Stanford. Cs231n: Convolutional neural networks for visual recognition.
- [115] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.
- [116] Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.
- [117] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.
- [118] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks, 2020.
- [119] Tesla. Autopilot, September 2020. Tesla’s technologies. Available at [https://www.tesla.com/pt\\_PT/autopilotAI](https://www.tesla.com/pt_PT/autopilotAI).
- [120] Tesla. Autopilot internship/co-op (summer 2019), September 2020. Tesla’s reinforcement learning internship. Available here <https://web.archive.org/web/20190209090339/https://www.tesla.com/careers/job/autopilot-internship-co-opsummer2019-37950?redirect=no>.
- [121] Marin Toromanoff, Emilie Wirbel, and Fabien Moutarde. End-to-end model-free reinforcement learning for urban driving using implicit affordances. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7153–7162, 2020.
- [122] Uber. We believe in the power of technology, September 2020. Website of UBER <https://www.uber.com/us/en/atg/technology/>.
- [123] ucsusa. Self-driving cars explained, February 2018. Article by Union of Concerned Scientists. Available in <https://www.ucsusa.org/resources/self-driving-cars-101>.
- [124] Aaron Van Den Oord, Oriol Vinyals, et al. Neural discrete representation learning. In *Advances in Neural Information Processing Systems*, pages 6306–6315, 2017.



- [125] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015.
- [126] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.
- [127] Rijul Vohra, Sep 2019.
- [128] Sen Wang, Daoyuan Jia, and Xinshuo Weng. Deep reinforcement learning for autonomous driving, 2019.
- [129] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.
- [130] Waymo. Waymo training data, May 2014. Training data available here <https://waymo.com/open/>.
- [131] Waymo. Introducing the 5th-generation waymo driver: Informed by experience, designed for scale, engineered to tackle more environments, September 2020. Article by Waymo. Available in <https://blog.waymo.com/2020/03/introducing-5th-generation-waymo-driver.html>.
- [132] Waymo. Waymo’s faq, September 2020. Waymo’s details about its vehicles <https://waymo.com/faq/>.
- [133] Lilian Weng. From autoencoder to beta-vae. *lilianweng.github.io/lil-log*, 2018.
- [134] Lilian Weng. Self-supervised representation learning. *lilianweng.github.io/lil-log*, 2019.
- [135] WikiChip. Fsd chip - tesla, September 2020. Custom designed and built Tesla hardware, information available in [https://en.wikichip.org/wiki/tesla\\_\(car\\_company\)/fsd\\_chip](https://en.wikichip.org/wiki/tesla_(car_company)/fsd_chip).
- [136] Wikipedia contributors. Recurrent neural network — Wikipedia, the free encyclopedia, 2020. [Online; accessed 18-September-2020].
- [137] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.
- [138] Wired. Artificial intelligence confronts a ‘reproducibility’ crisis, July 2020. Article by Wired. Available in <https://www.wired.com/story/artificial-intelligence-confronts-reproducibility-crisis/>.
- [139] Qin Zou, Hanwen Jiang, Qiyu Dai, Yuanhao Yue, Long Chen, and Qian Wang. Robust lane detection from continuous driving scenes using deep neural networks. *IEEE Transactions on Vehicular Technology*, 69(1):41–54, Jan 2020.